Justifying The Strong Memory Semantics of Concurrent High-Level Programming Languages for System Programming

Dissertation zur Erlangung des Grades des Doktors der Ingenieurswissenschaften der Fakultät für Mathematik und Informatik an der Universität des Saarlandes

Jonas Oberhauser

Saarbrücken, Oktober 2017

Datum des Kolloquiums:	7. Februar 2018
Dekan:	Prof. Dr. Frank-Olaf Schreyer
Vorsitzender des Prüfungsausschusses:	Prof. Dr. Gert Smolka
Berichterstatter:	Prof. Dr Wolfgang Paul
	Dr. Ernie Cohen
Akademischer Mitarbeiter:	Dr. Marvin Künnemann

Zusammenfassung

In dieser Dissertation rechtfertigen wir die sequentiell konsistente Semantik von parallelen Hochebenensprachen wie C11 oder Java, wenn diese für x86-ähnliche Architekturen übersetzt werden. Sprachen wie C11 garantieren dass Programme, welche eine bestimmte Software-Disziplin befolgen, sich so verhalten als würden sie auf einer einfachen kohärenten Mehrkernprozessor mit gemeinsamem Speicher ausgeführt werden. In echten x86 Maschinen ist der gemeinsame Speicher hingegen nicht kohärent, und Übersetzer fügen langsame Synchronisationsinstruktionen ein, um Kohärenz vorzugaukeln. Wir zeigen dass eine Software-Disziplin die sich stark an Software-Disziplinen solcher Programmiersprachen wie C11 und Java anlehnt — im Grunde, dass Speicherzugriffe in einer Wettlaufsituation vom Programmierer annotiert werden — und eine bestimmte Methode, relativ wenige Synchronisationsinstruktionen einzufügen — im Grunde zwischen einem annotierten Schreibzugriff und einem annotierten Lesezugriff des gleichen Fadens — ausreichen, um diese Illusion zu gewährleisten. Die Software-Disziplin müssen bei individuellen Programmen nur in der sequentiell konsistenten Semantik der Hochebenensprache geprüft werden, so dass die gesamte Verifikationsarbeit in der sequentiell konsistenten Semantik der Hochebenensprache stattfinden kann. Wir behandeln eine Maschine mit Betriebssystemunterstützung, und unsere Theoreme können entsprechend auch zur Verifikation unterbrechbarer Mehrkern-Betriebssysteme verwendet werden, einschliesslich solcher, bei denen Benutzer unverifizierte Programme laufen lassen können, die nicht die Software-Disziplin befolgen.

Abstract

In this thesis we justify the strong memory semantics of high-level concurrent languages, such as C11 or Java, when compiled to x86-like machines. Languages such as C11 guarantee that programs that obey a specific software discipline behave as if they were executed on a simple coherent shared memory multi-processor. Real x86 machines, on the other hand, do not provide a coherent shared memory, and compilers add slow synchronizing instructions to provide the illusion of coherency. We show that one software discipline that closely matches software disciplines of languages such as C11 and Java — in a nutshell, that racing memory accesses are annotated as such by the programmer — and one particular way to add relatively few synchronization instructions — in a nutshell, between an annotated store and an annotated load in the same thread — suffice to create this illusion. The software discipline has to be obeyed by the program in the semantics of the high-level language, therefore allowing the verification effort to take place completely in the strong memory semantics of the high-level language. We treat a machine with operating system support, and accordingly our theorems can be used to verify interruptible multi-core operating systems, including those where users can run unverified programs that do not obey the software discipline.

Acknowledgments

I thank Prof. Dr. Wolfgang J. Paul for his patient counsel in all matters, the freedom I enjoyed while working at his chair, his guidance in how to present scientific results, and especially all the errors he has spotted in my work. I could not have imagined a better doctoral advisor.

I also thank Dr. Ernie Cohen, Prof. Dr. Wolfgang J. Paul, and my colleagues for many deeply engaging discussions, both scientific and non-scientific, which have kept me motivated to complete this document.

I also thank my wife for her unfailing support and the sunshine she brings into my life each day, and my friends and family who have brought me countless hours of joy.

Finally, I thank Dr. Chad E. Brown, Henning Diedrich, Susanne Oberhauser-Hirschoff and many others to whom I have looked up as mentors and from whom I have learned many of the skills I used throughout my PhD.

Contents

Chapter 1

Introduction

When programming a software system, there are huge productivity gains by using high-level languages, which a) abstract away from complicated details of the target machine, b) allow development to be easily ported to other target machines, and c) allow actions that require many instructions in the target machine to be expressed within a single, expressive statement. One obtains similar gains in productivity when verifying the program in its high-level semantics. However, then one also has to show that the compiled program, when run on the target machine, does not exhibit new behaviors. State-of-the-art methods, such as translation validation [PSS98,CV16], are incomplete and have to be repeated for each program and whenever any element of the tool chain, such as the compiler, compiler flags, linker, any line in the source program, or the target architecture, change. Automation is normally employed to help tackle this tedious and effortful work, but may suddenly stop working due to subtle changes in the code or tool chain. A more fundamental approach is to verify the correctness of the linker and the compiler, i.e., to show once and for all that all programs in the source language are compiled correctly. For single-core machines with a strong memory model (unlike Intel, AMD, or ARM processors, which all have weak memory models) such compiler correctness proofs exist (e.g., [CA], [PBLS16, Chapter 12]), but no complete proofs exist for multi-core processors with weak memory models. We are aware of two results from the literature that tackle compiler correctness on multi-core processors, but the first [BSDA14] ignores the problem, and the second [VN] simply replicates the interleaving and memory semantics of the target architecture. There are two main challenges to be overcome:

- High-level languages have to provide a single memory model (typically strong memory consistency), while the architectures targeted by the language can have very diverse and very weak memory models. For example, C11 offers a strong memory model, but can be compiled to Intel x86 and to ARM Armv8-A processors, which each have their own distinct weak memory model.
- High-level languages have a much coarser interleaving model than their target architectures as what is usually considered a single unit of execution in the high-level language may be translated into multiple steps of execution in the target machine, such as the well-known `i++`.

Due to these challenges, compilers can not be expected to compile all programs "correctly" w.r.t. the intuitive semantics. Instead, languages require help from the

programmer, and leave the semantics of programs undefined when the programmer does not do her part. For example, C11 provides program annotations to flag memory accesses as racing, and defines no semantics for programs unless all racing accesses are flagged by the programmer. Obviously, this property is verified in the semantics of the high-level language, i.e., a verification engineer has to show that all races in all coarsely interleaved, sequentially consistent executions of the program, are flagged correctly; only then can she expect that the program has no new behaviors when compiled to a finely interleaved, weak memory machine. However, this raises one big question: how can one make assumptions about a thing to prove that the thing exists? Is this not like the famous Baron Münchhausen, who pulled himself from mire by his own hair (Figure 1.1)?

Figure 1.1: Using properties of a program in a semantics of which we do not know it exists to show that it exists.

In this thesis we address this question for x86-like multi-core processors with operating system support, and show a set of sufficient high-level conditions under which sequential consistency can be regained on such an architecture. The thesis is split into three parts.

1. In the first part (Chapter 2) we define an extremely powerful and generic, yet compact machine model that has all support for all of the features we expect from a state-of-the-art processor architectures, such as:

   (a) Autonomous devices, which can register read- and write accesses and change their state both immediately as a deterministic side-effect (e.g., a device that accumulates all values stored into one register, and clears that register when writing to a second register) and eventually in a non-deterministic step (e.g., a disk that responds to a load command eventually),

   (b) Inter-processor-interrupts (IPIs) that are delivered directly into the core of other processors, thereby interrupting them immediately,

   (c) Asynchronous write buffers, which act as a buffer between each processor and the shared memory system, and are the source of the weak memory model of x86-like processors,

(d) Memory accesses that bypass the write buffers, e.g., when memory management units (MMUs) read and modify page table entries without snooping or modifying the write buffers,

(e) Heterogeneous memory accesses, such as byte and word accesses and misaligned accesses,

(f) Read-modify-writes (RMW), such as compare-and-swap and fetch-and-add,

(g) Complex core components, such as translation look-aside buffers (TLBs) and floating point units,

(h) Single-cycle fetch and execute,

(i) Writable code regions, as is necessary for just-in-time compilation and page faults on fetch,

(j) Guard conditions, which keep the computations sane by disabling steps that make no sense, such as extending translations that do not exist or for which there are no rights.

This machine model has two semantics: a low-level semantics where all steps use the write buffer, and a high-level semantics where only unverified code uses the write buffer.

2. In the second part (Chapter 3) we instantiate our model with a variant of MIPS ISA known as MIPS86, in order to show how to use the model to implement several of the components mentioned above. This MIPS ISA supports IPIs, MMUs, and a simple device. That this ISA can actually be implemented is demonstrated in [PLO], which gives a gate level construction and correctness proof of a MIPS86 multi-core processor.

3. In the third part (Chapter 4), we give a sufficient set of conditions such that any program that satisfies the conditions in the sequentially consistent variant of the ISA will not exhibit new behaviors when executed on the weak memory ISA. Our conditions make no assumption about the behavior of (potentially malicious) user code, which therefore still exhibits weak memory behaviors in our sequentially consistent ISA.

   Our conditions are based on an efficient software discipline of Cohen and Schirmer [CS10], which has been proven correct for a much simpler machine. The gist of the conditions is that rather than using inefficient synchronization instructions behind every shared write (as is the case in state-of-the-art compilers), a synchronization instruction is only needed between a racing write and a racing read on the same thread.

   The correctness of our conditions is proven in a write buffer reduction theorem, which shows that all finite weak memory machine executions simulate a sequentially consistent execution. Finally, we extend the theorem to infinite executions.

   Similar theorems are proven in [CCK14, CS10], which make safety assumptions about user code, in particular, that user programs strictly obey the flushing discipline, a condition that can not be enforced by hardware. Furthermore, they do not deal with interrupt controllers or mixed-size and misaligned accesses, which cause significant difficulties. In the thesis of Chen [Che16], mixed-size (but not misaligned) accesses are considered, but only under the assumption of a stronger

memory model than that of x86. On the other hand, their work can be applied to situations where a page fault handler races with memory management units, while in our work, page fault handlers can only be active while the memory management unit on the same core is disabled.

We are aware of one more efficient software discipline [BDM13] for a simple machine. We deem this software discipline too complicated to be applied by humans, and unlike the disciplines based on Cohen and Schirmer's work, can not be automatically applied to practical programs yet (even if the programmer correctly flags all races).

This thesis therefore reduces compiler correctness proofs for x86-like multi-core processors to sequentially consistent reasoning. Informally speaking, it shows that any reasonable[1] high-level language that enforces the conditions given in this thesis (e.g., has undefined semantics for programs that do not satisfy the conditions) can be compiled correctly and straightforwardly to x86-like machines.

Our write buffer reduction theorem is the first (non-trivial) to apply to operating systems with untrusted user programs. Since user code is not assumed to obey any programming discipline, write buffers can not be completely abstracted away, and even in our higher level of abstraction remain visible for user code. A similar problem exists with processor correctness theorems, which so far (e.g., [KMP14]) assume conditions for all code on the machine, in particular, lack of unsynchronized self-modifying code. Since we can not in good faith assume this property for user code, our MIPS instantiation employs a non-deterministic instruction buffer for users, which in case of an unsynchronized self-modification may non-deterministically either provide the old or the new instruction.

1.1 Overview

1.1.1 Model

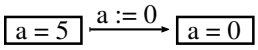
We define a non-deterministic computational model in which multiple active *units*, such as processors, APICs, and devices, share memory. The shared memory is split into normal read-writable memory, and reactive device memory. While the normal memory simply accepts modifications, the device memory can use hardwired logic to make a complex transition based on incoming writes. For example, a device register might count incoming writes by incrementing its value every time a write is sent (cf Fig. 1.2). This allows us to implement complex write-triggered (also known as transport-triggered) devices.

Each unit is provided with local storage, called the *core*. The core is separated into two parts: normal registers, which are completely local to the unit, and interrupt registers, which are used as-if local to the unit but can be modified by other units. When the interrupt registers of a unit i are modified by another unit, we say that unit i is a *victim* of the other unit.

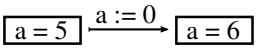
Each unit can access its core, the interrupt registers of other units, and the shared memory. A full map of the memory is given in Fig. 1.3.

Each unit is split into two active, almost asynchronous *objects*: a "smart" object, called the *processor*, and a "dumb" object, called the *write buffer*. The processor can

[1] Under reasonable we understand languages in the class of C11 or Java that behave like turing machines, do not have a memory model stronger than sequential consistency, etc.



(a) Writing to a normal memory address: the value is taken verbatim from the write.



(b) Writing to a write-triggered device register: value of write and current state are used to compute next state, in this case by incrementing the current state.

Figure 1.2: The effect of writing value 0 to shared memory address a depends on whether the address is a device address or a normal memory address.

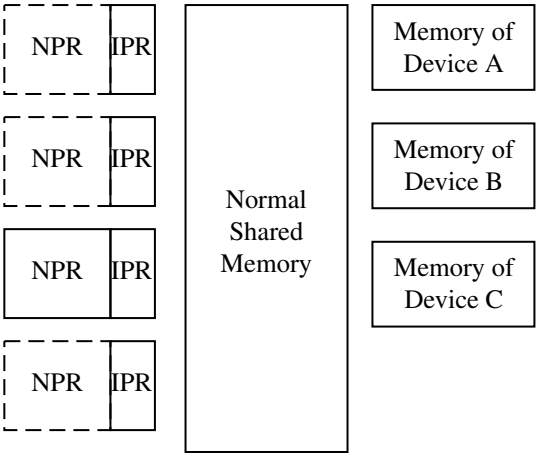


Figure 1.3: The memory layout for a design with four units and three device memories. The third unit can access its own NPR, all the IPRs, the shared memory and devices (represented by boxes with solid lines in the figure), but not the normal processor registers of other units (represented by boxes with dashed lines in the figure).

non-deterministically chose between a variety of steps which can do complex memory accesses and buffer writes, i.e., assign them to the write buffer for later execution. The write buffer can only do one type of step, which commits the oldest buffered write to memory.

In each step, the processor has full access to its core, then computes a memory region to fetch from. After having a fetch result, the processor computes two memory region to read from: one that prefers more up-to-date values from its own write buffer to the current state of the shared memory, and one which simply loads from the shared memory and completely ignores the write buffer. Both ignore the content of write buffers of other units. After collecting the content of this memory region from the memory system (own write buffer + current state of shared memory and interrupt registers), the processor computes two writes: one write to be executed immediately, and one write to be buffered[2].

This distinction between memory accesses that ignore the buffer (called bypassing accesses) and those that go through the buffer (called buffered accesses) is necessary for

[2]If the latter is empty, nothing is added to the buffer to prevent unnecessary write buffer steps.

three reasons. Firstly, core registers such as the program counters or the GPR are close to the CPU and are modified directly, rather than through the write buffer. Secondly, processors often provide special atomic operations, called read-modify-writes (RMW), which in a single step load values, perform some computations on them, and then write values back to the memory system. These do not appear to be atomic if the modification goes into the write buffer. Imagine, for example, two locking operations, where the second locking operation succeeds because the first locking operation is still in the write buffer (but has already signaled to its processor that the lock was successfully acquired). Thirdly and most importantly, modern processors have special components that create additional memory accesses, such as the memory management unit (MMU) when it walks page tables. These components share central data structures with the processor, such as the translation look-aside buffer (TLB) in case of an MMU, which therefore have to be part of the core. At the same time, some of these components act like independent units in that they completely ignore the write buffer. To model such components with a single, uniform model, we need the full power of bypassing memory accesses.

Since for technical reasons all units have a processor *and* a write buffer, in particular APICs and devices have a write buffer. By instantiating the processors for APICs and devices in such a way that they never compute a write to be buffered, the write buffer of these unit stays empty and effectively disabled. Similarly, APICs for technical reasons have interrupt registers and normal registers, which they never need. Most of the state of APICs and devices is stored in the shared memory, which is easily accessible for other units. For example, the interrupt command register of an x86 APIC can be modeled as a normal memory register, which is accessed via normal store and load instructions of an x86 processor.

It is equally necessary to point out that when the word "processor" is normally used (i.e., outside of this thesis), it denotes only the programmable MIPS processor and not APICs or devices. In this document, however, processors are not necessarily programmable; each processor can have its own logic that makes it tick. This allows us to model APICs as just another type of processor which fetches "instructions" from the interrupt command register, and similarly, devices as just another type of processor which fetches "instructions" from, e.g., the command and status register, or in case of a sensor, updates certain memory registers based on the environment (which can be modeled as non-deterministic input to the device processor).

The technical goal here is to use a uniform model for all imaginable types of active units that can make steps that are completely asynchronous from the normal programmable processor. As a simple example, consider the ability of APICs to deliver interrupts to normal processors. In this document, this ability is not a magic hard-coded ability of APICs, which therefore would need to be considered as a special case in each step of the proof; but rather, all units have the power to modify the interrupt registers of all other units. In our instantiation of MIPS86, we will of course only allow the APICs do this. This uniform model also allows us to instantiate virtually the complete MIPS86 semantics in a mere 30 pages (Chapter 3).

The write buffer of unit i is actively drained on three occasions.

1. When the processor of unit i explicitly requests a write buffer drain, e.g., when executing an `mfence` instruction.

2. When the processor of unit i tries to load a value that is partially but not fully served by the most up-to-date write in its write buffer. This situation is called a

partial hit. The write buffer is only partially drained, namely until those writes leave the write buffer.

3. When unit i is a victim of an interrupt. This is necessary because the interrupt immediately moves control of the processor to the interrupt handler. But if there are still writes in the buffer, the interrupted program will have an afterglow: the buffered writes will still be executed and it looks as if the unit had not been interrupted.

We define two semantics for our model: low-level semantics and high-level semantics. In low-level semantics, a processor that decides to buffer a write will buffer the write and not execute it, and a write buffer step will execute the oldest buffered write. In high-level semantics, the write that the processor planned to buffer will instead be executed immediately, and a write buffer step has no effect (for technical reasons, we still keep the buffer as a ghost register) (cf Fig. 1.4).

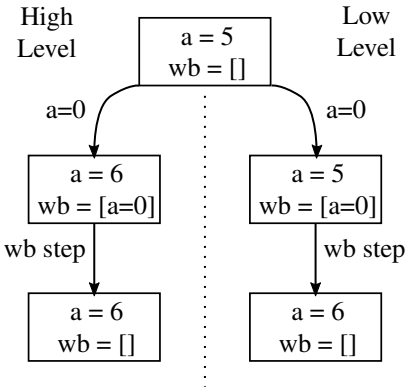


Figure 1.4: Using again the example of a device that counts the number of writes, we show the difference between high-level semantics on the left and low-level semantics on the right. Execution starts at the top with the device having counted five writes, and an empty write buffer. We then let the processor buffer a write to the device, and then step the write buffer. In high-level semantics, the write immediately reaches the device and the count is incremented immediately, whereas the write buffer step does not execute the buffered write. In low-level semantics, the processor step has no effect on the device, and the write only reaches the device when the write buffer step executes it.

We introduce two machines $M \in \{\downarrow, \uparrow\}$: the low-level machine $\downarrow$ and the high-level machine $\uparrow$. The low-level machine always has low-level semantics. The high-level machine sometimes has low-level semantics and sometimes has high-level semantics. Which of the two it is is completely deterministic and depends on whether the processor is currently running trusted (operating system) code which obeys the software discipline, or untrusted (user) code which does not obey the discipline. To distinguish between the two modes we introduce for each unit memory mode registers $A_{SC,i}$ — ghost or otherwise — in the normal registers, the value of which is used as an input to predicate SC_i for *sequentially consistent* mode of unit i

$$SC_i(c.m\big|_{A_{SC,i}}).$$

The meaning of this predicate is simple: $SC_i(c.m\big|_{A_{SC,i}}) = 1$ means the unit is in configuration c in the middle of trusted code, must obey the software discipline, and will use high-level semantics in the high-level machine. $SC_i(c.m\big|_{A_{SC,i}}) = 0$ means that the unit is in configuration c in the middle of untrusted code, ignores the software discipline, and will use low-level semantics in the high-level machine.

The non-determinism in our model is made precise by so called oracle input. The oracle input is used in each step to a) uniquely identify an object (and unit) making the step and b) to provide external inputs to processors when they make steps. This gives us one transition operator $\blacksquare_M$ for each machine M, which moves us from one configuration c to the next configuration c' using an oracle input x

$$c' = c \blacksquare_M x.$$

A sequence of such oracle inputs is called a schedule. Each schedule s, when starting from an original configuration, induces on each machine a computation $c[s]_M$, which is a sequence of configurations where the $t+1$-th configuration is obtained by applying the transition operator to the t-th configuration and the t-th oracle input

$$c[s]_M^{t+1} = c[s]_M^t \blacksquare_M s(t).$$

Our definitions are such that we can run the high-level machine and the low-level machine with the same program in parallel using the same schedule. When in each step of the two computations made by a processor the processor 1) has the same core configuration, 2) fetches the same instructions, and 3) obtains the same results from the memory system, we say that the computation is reduced. In this case the unit making the step will make the same decisions in both machines.

Not all computations are reduced, as the following example shows. We consider a program with two threads; the first thread modifies a variable `x`, the second thread loads from that variable. We use the symbol = for assignment through the write buffer and the symbol `:=` for an assignment that goes directly to the memory. We write `-> BUF` behind an assignment that enters the write buffer and does not change memory, and `->` before an assignment that comes from the buffer.

```
x = 1; || y := x;
```

We now consider the schedule where Thread 1 steps first, and then Thread 2, and finally the write buffer of Thread 1. In the low-level machine computation, the write to `x` is buffered, and Thread 2 sees the old value from the shared memory.

```
                                  ||          x  y
x = 1; -> BUF                     ||          0  0
                                  || y := x;  0  0
-> x = 1;                         ||          1  0
```

In the high-level machine computation (using high-level semantics), the write is executed immediately and the write buffer step has no effect (`NOOP`). Consequently, Thread 2 sees the new value written by Thread 1.

```
                                  ||          x  y
x = 1;                            ||          1  0
                                  || y := x;  1  1
-> NOOP                           ||          1  1
```

In one machine, Thread 2 reads a 0, in the other machine it reads a 1, and thus the computation is not reduced.

In fact, almost all programs have non-reduced schedules. It is thus not valuable to prove a theorem showing that all programs satisfying some condition have only reduced schedules, since the conditions would have to be of such a kind that only very trivial programs satisfy them. Instead, we prove that for each schedule *s*, there is another schedule *r* such that

- the low-level machine computations of the two schedules are equivalent, i.e., when a processor makes its *n*-th step in the low-level computations induced by schedules *s* and *r*, it 1) has the same core configuration, 2) fetches the same instructions, and 3) obtains the same results from the memory system,
- and schedule *r* is reduced.

One can now use transitivity to obtain that the original low-level machine computation always has the same core, fetched instructions, and memory results, as the new high-level computation. Therefore any sequence of values observed by processors in a low-level machine computation can be explained by a sequence of values observed by processors in a high-level machine computation.

Considering again the schedule from above, we can easily define an equivalent, reduced schedule, namely by stepping Thread 2 first. We check that this schedule is indeed equivalent by comparing the value read by Thread 2 in the low-level computation, which as in the original low-level computation is zero.

```
                          ||            x  y
                          || y := x;    0  0
x = 1; -> BUF             ||            0  0
-> x = 1;                 ||            1  0
```

Furthermore, Thread 2 also reads zero in the high-level computation of this schedule.

```
                          ||            x  y
                          || y := x;    0  0
x = 1;                    ||            1  0
-> NOOP                   ||            1  0
```

It is easy to check that for each schedule of the program above there is such an equivalent, reduced schedule. Schedules where Thread 2 is scheduled before the write buffer step are all equivalent to the reduced schedule above; schedules where the write buffer step is stepped before Thread 2 are already reduced.

In this thesis, we consider a set of sufficient conditions of all *high-level machine* computations which imply that all schedules of the *low-level machine* have an equivalent, reduced schedule. We call such programs sequentially consistent.

1.1.2 Programming Discipline

The programming discipline is formulated as a set of properties of high-level computations. A program where all possible high-level computations have those properties is called safe. The properties are defined in such a way that they always hold for untrusted code, either because they explicitly are only checked for trusted code, or because they trivially hold for trusted code.

To distinguish between trusted code and untrusted code, we have introduced registers $A_{SC,i}$ — ghost or otherwise — that indicate whether a processor is currently executing trusted code or untrusted code. The proof obligation for these registers is that changing the register content is only allowed when the write buffer is empty[3].

Most of the properties are defined in terms of races between memory accesses. A race occurs when two memory accesses in a computation affect the same memory region, and at least one of them is modifying that region. For our discipline, however, we only consider those races where the memory accesses can occur directly after each other, i.e., at steps t and $t+1$ in the computation. This restriction at first glance seems to exclude many races that could potentially be harmful for our program, but we show in Sections 4.8 to 4.10 that this restricted set of races is sufficient to find all relevant races in the program. One way to find such races is by an ownership discipline such as the one of Cohen and Schirmer [CL98], where memory regions are either owned by a thread or shared, and are either write-able or not; depending on the ownership state, a memory access is classified as potentially racing or not; or the one of Oberhauser [Obe16] in which threads exclude other threads from reading or from writing to memory regions, and an access to a memory region is potentially racing when other threads are not excluded from accessing that region.

Like Cohen and Schirmer, we introduce a ghost annotation of *shared* memory access. This annotation comes with a simple proof obligation: all memory accesses involved in a race where the accesses occur directly after each other have to be annotated as shared. This immediately corresponds to the requirement of Java that variables that are used by multiple threads concurrently have to be annotated as "volatile", and of C11/C++11 that all memory accesses involved in a race have to be annotated as "atomic". Furthermore, memory accesses to 1) device memory, 2) interrupt registers (except for the implicit snoop of the interrupt registers at the beginning of each processor step) have to be shared. Because we can not expect the untrusted code to be annotated correctly, we simply define that all steps of untrusted code are shared. Similarly, write buffer steps (which are asynchronous and not under directly control of the system programmer) are defined to be shared.

We then take the central proof obligation of the software discipline from Cohen and Schirmer, i.e., that no shared read is ever executed by a processor that is buffering a shared write. Of course in our software discipline, this is only required when the processor is executing trusted code (i.e., $SC_i(c.m\big|_{A_{SC,i}}) = 1$).

These are the main proof obligations.

As a concrete example, we consider again the simple program from before.

```
x = 1; || y := x;
```

Since the assignment to `x` by Thread 1 and the load of `x` by Thread 2 can be executed directly after each other in the high-level machine, they need to be annotated as shared. We do so by using `x.store(1)` instead of the simple assignment operator =, and `x.load()` instead of simply the expression `x`

```
x.store(1); || y := x.load();
```

Since no single thread executes both shared writes and shared reads, the Cohen-Schirmer condition is trivially satisfied; the program is now safe, without having to insert instructions that would drain the write buffer.

[3]Strictly speaking, it is likely harmless to change the content of the registers as long as this does not change the result of SC_i; we leave this as minor future work.

The Cohen-Schirmer condition is simple and efficient, but is insufficient in the context of heterogeneous memory accesses, i.e., memory accesses that are not aligned and of the same size. Dealing with heterogeneous memory accesses turns out to be difficult, mainly because compare-and-swap accesses can in the high-level machine be prevented from racing with read accesses by a buffered write (which has been executed in the high-level machine). We add a very technical condition which is efficient but not simple, and suggest a few potential simplifications. Proving that these simplifications are indeed correct and determining their impact on performance is left as future work. Similarly, since we allow dynamic code, we have to add a technical condition that prevents programs from hiding races by modifying the code of a racing access, e.g., by turning a racing read instruction into an instruction that loads a constant value. We call this Section 4.5.

The conditions are as follows. If there is a race between steps k' and $k'+1$ such that step $k'+1$ modifies some values used by step k', and does so using a buffered write, then until another object makes a step, i.e., for all $l+1 > k'+1$ such that all steps $t' \in (k'+1 : l+1)$ are made by the same object, all of the following must hold:

1. step $l+1$ only reads a memory region modified by step k' if the values have since then been overwritten (i.e., the values written by step k' are not visible to step $l+1$),

2. step $l+1$ only modifies a memory region accessed by step k' if step $l+1$ is annotated as shared.

Finally, we need to restrict our model somewhat when running trusted code: the bypassing memory accesses are too powerful and can easily be used to break sequential consistency, and similarly, buffering writes to interrupt registers is extremely harmful since the interrupt will be triggered instantaneously in the high-level machine but only once the write leaves the write buffer in the low-level machine. In more detail, a processor that is currently executing trusted code may never

1. Buffer writes to interrupt registers.

2. Use a bypassing reading access (fetch or read) when more up-to-date values are in the ghost buffer of the same unit (in which case an outdated value can be read in the low-level machine).

3. Overwrite a buffered value in its ghost buffer with a bypassing write (in which case the new value will be overwritten by the older, buffered value once it leaves the buffer).

4. Have a shared write be buffered *and* bypassing at the same time (in which case other processors may see the bypassing half of the write but not the buffered half).

5. Use a shared bypassing write while it has writes in its ghost buffer (in which case a lock might be released with the shared write before the unshared memory accesses to the locked data structure are visible to the other threads).

6. Do a shared RMW where the modification is being buffered (in which case the RMW is atomic in the high-level machine but absolutely not atomic in the low-level machine).

However, we explicitly allow non-shared bypassing writes, in particular even if there are writes in the buffer (shared or otherwise). This suggests that the order of non-shared writes among another is irrelevant and that a non-shared write can overtake a shared write (but not the other way around) without breaking sequential consistency. If that is true, the discipline in this paper would also work for architectures with write-combine buffers as long as no shared write may ever overtake a non-shared write due to write-combining. The details are future work.

To show how the power of bypassing memory accesses can be easily used to break sequential consistency, we consider the MIPS86 MMU in more detail. The MIPS processor and the MMU share four normal registers: 1) the OS mode register in the SPR, since translations are only done in user mode, 2) the page table origin, where the MMU begins walking, 3) the address space identifier, which is a tag associated with each translation to separate translations of different user programs, and 4) the TLB, where translations are buffered. All of these registers are part of the core and can be changed by the MIPS processor with instructions such as `flush` (for flushing all translations out of the TLB) or `movg2s` (for the changing the other three registers). Furthermore, the TLB is accessed at the beginning of each MIPS processor step before even fetching an instruction. Similarly, the TLB is accessed by the MMU in virtually all of its steps, to add new translations. That is why we have to make the MMU and the MIPS processor a single unit, so that they can both access these normal core registers. On the other hand, a MIPS86 MMU does not drain the write buffer nor does it consider more up-to-date writes in the write buffer: its memory accesses, e.g., when loading a page table entry, completely bypass the write buffer. Imagine now a single-core MIPS86 machine where a single operating system thread has just swapped out a page, and is now attempting to prevent the MMU from using stale translations to that page. This is done in two simple steps: 1) lower the present bit `PTE.p` of the page table entry that maps to the swapped out page, and 2) flush all the stale translations from the TLB that were created using the no longer present page table entry. Let us now assume that while this is happening[4], the MMU autonomously performs translations on present page table entries, filling the TLB and setting accessed bits `.a` on those page table entries (all of this is an atomic operation). In code, the situation can be represented like this, using one thread for the CPU and one thread for the MMU:

```
CPU          ‖ MMU
PTE.p = 0;   ‖ if (PTE.p) PTE.a := 1;
flush;       ‖
```

Note that it makes sense to use two threads `CPU` and `MMU` even though the steps are made by the same processor for two reasons: 1) the MMU non-deterministically translates page table entries, and thus the translation attempt for `PTE` can be scheduled at any point, and 2) the MMU bypasses the write buffer, and thus acts like a separate thread with no access to the write buffer. One easily checks that in high-level machine computations, after flushing the TLB, the MMU will not add a new translation for that page table entry, since the present bit has been set to zero.

We also easily find a schedule in which the high-level computation violates the conditions of the discipline. The computation looks as follows (we focus on the ghost write buffer, rather than the values of the variables `PTE.p` and `PTE.a`).

[4]In MIPS86, the MMU is disabled in operating system mode and the problem below can not normally occur.

```
CPU                 ‖ MMU                          ghost wb
PTE.p = 0;          ‖                            [PTE.p = 0;]
flush;              ‖                            [PTE.p = 0;]
                    ‖ if (PTE.p) PTE.a := 1;     [PTE.p = 0;]
-> NOOP             ‖                                 []
```

The processor in the MMU step uses a bypassing reading access to read the present bit, for which a more up-to-date write exists in its own ghost write buffer.

Running the same schedule in the low-level machine shows how this bypassing read destroys sequential consistency.

```
CPU                 ‖ MMU                          PTE.p  PTE.a
PTE.p = 0; -> BUF   ‖                                1      0
flush;              ‖                                1      0
                    ‖ if (PTE.p) PTE.a := 1;         1      1
-> PTE.p = 0;       ‖                                0      1
```

Note that the flush of the TLB is executed after the assignment to PTE has entered the buffer, but before it is committed to memory. As a consequence, the MMU can set the accessed bit and put a translation of the no longer present page table entry into the TLB.

1.1.3 Proof

As mentioned above, the main result of this paper is that programs that obey the software discipline in all computations of the high-level machine are sequentially consistent, i.e., for each schedule s there is an equivalent (in the low-level machine) schedule r which is reduced.

The proof is very technical but follows a relatively simple plan.

1. Sections 4.8 to 4.10: We define a synchronization relation $\blacktriangleright$ between steps. The central idea is that when a shared read sees the value written by a shared write, the two steps are synchronized. Additionally, steps of the same object are usually synchronized, a step that interrupts another step is synchronized with that step, and a shared RMW that does not see the value written by a later write is synchronized with that write (since it essentially witnesses that the write has not occurred yet). We show with Lemma 258 that even though our discipline only considers races where the memory accesses occur directly after one another, all races are actually synchronized, i.e., there is a sequence of shared reads and shared writes, steps of the same unit, interrupts, and shared RMWs that do not see the value of a later write, which effectively prevent us from executing the two memory accesses next to each other.

   This result only depends on the high-level machine and does not use any software condition except for the condition that races where the accesses occur directly after one another are annotated as shared.

   Interestingly, one often finds the opposite approach in language specifications: two memory accesses are considered "concurrent" if they are not synchronized by such a sequence of synchronization steps, and have to be annotated as shared when they are concurrent.

2. Section 4.11: We show several key properties of reduced schedules, such as that the write buffers and cores in the machines are kept in sync. We then introduce

a key weapon in our arsenal, which is a class of schedules where no unit makes *global* steps — shared writes that modify memory or shared reads — while a shared write is being buffered by a unit that is executing trusted code. We show that such schedules are always reduced with Lemma 305. The proof has three central steps:

- Lemma 281 shows that a buffered write is never synchronized with other units, because none of the synchronization rules apply:

  shared write/shared read: per discipline, a shared bypassing write is not allowed while a write is being buffered; so the shared write would have to be buffered as well. A shared read would be a global step, which can not occur while a shared write is buffered.

  interrupt: an interrupt would drain the buffer, but the buffer is not empty.

  a shared RMW: per discipline, a shared RMW would have to be bypassing. But per discipline, a shared bypassing write is not allowed while a write is being buffered.

- Because races require synchronization (Proof Step 1.), and the buffered write is not synchronized with other units, there is no race. This is stated jointly by Lemmas 282 and 283.
- Thus a unit never modifies or reads from a memory region to which another unit is buffering writes. As a consequence, the order of writes and of reads and writes is never changed. The memory systems (but not memories) are kept in sync, i.e., during the execution of trusted code, buffer + memory in the low-level machine correspond to memory in the high-level machine, and during execution of untrusted code the memories are the same. This is stated in Lemma 295. Furthermore, since a unit only reads a memory region after all writes to that region have left the buffer (shown in Lemma 301), reads always see the most up-to-date value. This is stated jointly in Lemmas 298 and 304.
- Because reads see the same writes in both machines, they see the same values, i.e., the schedule is reduced. This is Lemma 305.

Because the order of writes and of reads and writes is the same in both machines, we call these schedules *ordered* schedules.

3. Section 4.12: We show how to reorder any schedule into an ordered schedule. The basic idea is simple: unroll the schedule in the order of global steps. We do this iteratively. After t iterations, we have some reordering O_t that created an ordered prefix $sO_t[0:t-1]$ of length t. We look at the next global step g_t. Steps $l \in [t:g_t)$ are thus by definition local. We try to create an ordered prefix by moving this step forward to position t, but we might get stuck because one of the steps $l \in [t:g_t)$ is a step of the same unit, or is interrupted by step g_t. We thus move instead step k_t, which is the first step made by the same unit or interrupted by g_t. For the sake of illustration, we run the reordering strategy on a schedule of the simple program from before

   ```
   x.store(1); ‖ y := x.load();
   ```

   As shown in Fig. 1.5, we begin with the non-reduced schedule sO_0 where Thread 1 is stepped first. Since in the low-level machine the shared write goes into the

buffer and does not change memory, the step is considered local. Thread 2 makes the second step, which is a shared read. The last step is made by the write buffer of Thread 1, which is a shared write and thus global.

```
x.store(1); -> BUF  ||                    x  y
                    ||                    0  0
                    || y := x.load();     0  0
-> x.store(1);      ||                    1  0
```

The next global step is thus the step made by Thread 2 ($g_0 = 1$). Since that step is the only step of that unit, it is also the first step of that unit ($k_0 = g_0 = 1$). We move k_0 to position 0, which moves step 0 to the back. After one iteration we are now in schedule sO_1. The next global step g_1 is now step 2, which is made by the write buffer of Thread 1. Since step 1 is made by the same unit, we can not simply reorder step 2 to position 1; instead, we move step $k_1 = 1$. This clearly does not change the schedule. After two iterations we are now in schedule sO_2. The next global step g_2 is still step 2, which is now also the first step of that unit outside of the ordered prefix. We thus move $k_2 = g_2 = 2$ to position 2, which again does not change the schedule. After three iterations we are now in schedule sO_3. Since there are no unordered steps remaining, we are done. Note that the resulting schedule is exactly the schedule described on page 13 in Section 1.1.1:

```
                    ||                    x  y
                    || y := x.load();     0  0
x.store(1); -> BUF  ||                    0  0
-> x.store(1);      ||                    1  0
```

This demonstrates the basic principles behind the reordering strategy. To show that the reordering strategy works for all programs that obey the software discipline and all schedules is considerably more involved.

When run on more complex programs and schedules, such as in Fig. 1.6, it can sometimes take multiple iterations before a global step can be moved.

4. Sections 4.12.1 to 4.12.5: To show that this reordering really works, we need to show 1) that the schedule sO_{t+1} is ordered until t and 2) that the schedule sO_{t+1} is equivalent with schedule sO_t.

   We observe first that all intermediate steps $l \in [t : g_t)$ are local. We say that the schedule in iteration t has a *local tail* from t to g_t, and write $\mathcal{L}O_t(t, g_t)$. Since the steps are all local, we obtain with Lemma 320 that the schedule is not only ordered before t, but actually before g_t. Since ordered schedules are reduced, we can run the program in the high-level machine and low-level machine in parallel with the schedule sO_t and obtain that all steps before g_t see the same values and make the same decisions in the two machines, and that the write buffer is the same. This allows us to use the software discipline — which only speaks about the high-level computation — to reason about those steps in the low-level computation.

   We distinguish at this point between two cases. When step k_t is a write buffer step of a unit currently executing trusted code (and because it is global, also

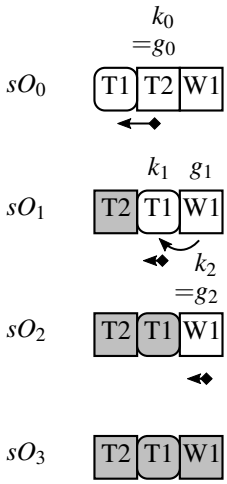


Figure 1.5: A complete run of the reordering strategy on the non-reduced schedule of the simple program from page 12. Steps are lined up like pearls on a string, with step zero at the left. Local steps are represented by squares with round corners, global steps simply by squares. The object making the step is inscribed in the square representing the step, either Thread 1 (T1), Thread 2 (T2), or the write buffer of Thread 1 (W1). The ordered prefix is shaded gray. The bent arrow indicates steps of the same unit or which are interrupted, and thus have to be moved first. The straight arrow indicates a reordering and begins at the step to be moved and ends at the target step. In this case, we move first step 1 to position 0, then step 1 to position 1, and finally step 2 to position 2. The latter two reorderings are obviously no-ops.

the next global step $g_t = k_t$), we say that the schedule has a local tail that commits a write buffer step and write $\mathcal{L}^W O_t(t, g_t)$. This case (handled completely in Section 4.12.4) is relatively simple, because write buffer steps do not have read results and the intermediate steps do not race with the buffered write because the schedule is ordered until g_t.

When step k_t is not a write buffer step of a unit currently executing trusted code, we observe that at least it is not made by any of the units making steps $l \in [t : k_t)$ and does not interrupt any of them. In this case we say that the schedule has a local tail with an independent end and write $\mathcal{L}^I O_t(t, k_t)$. We can easily show with Lemma 332 that in the low-level machine none of these steps modifies the instructions fetched by step k_t. In the high-level machine that is not the case. Consider, for example, a situation where a thread uses a buffered shared write to change an instruction. In the low-level machine, the step is local, because the buffered write has no effect. In the high-level machine, the step is executed immediately, and step k_t in the high-level machine will execute the new instruction. So clearly we can not transfer all of the properties of the software discipline for step k_t, in particular none of the properties that talk about decisions made after fetching, such as the read and written addresses.

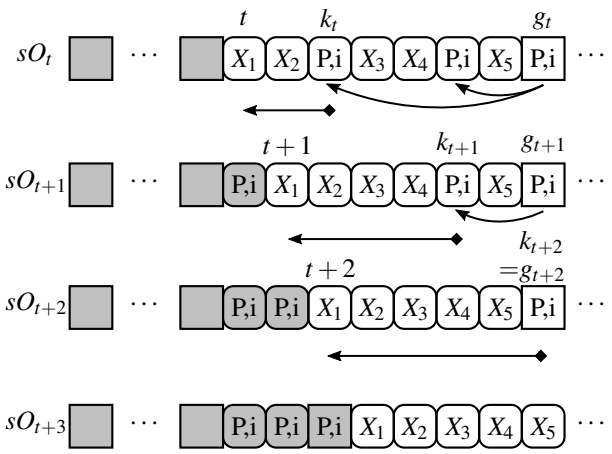


Figure 1.6: Four iterations of the reordering strategy. In iteration t, there are two steps made by the same unit as step g_t — unit i — which therefore have to be moved first. After two iterations, the path is clear, and the global step can be added to the ordered prefix. As can be seen, once a step is added to the ordered prefix, it is never moved again. This allows us to diagonalize and obtain an infinite ordered schedule.

We move step k_t to position t. Since none of the steps change the instruction fetched by step k_t, this move does not change the instructions fetched by former step k_t. This is stated by Lemma 334. To actually justify the reordering, however, we also need to show that 1) the read results are not changed, 2) the inputs of all the intermediate steps are not changed, and 3) the configurations after step k_t are not changed.

We obtain with Lemma 336 that after being moved to position t, the step sees the same values in the high-level and low-level computations, i.e., sO_{t+1} is reduced until t. We can now begin to use the software conditions also for step k_t at its new position.

A naive (and incorrect) plan for proving that the values seen by step k_t and all of the intermediate steps l are unchanged by this reordering proceeds as follows. We push step k_t (now at position t and reduced) back until it is at l and the former step l is at position $l+1$ (cf. Fig. 1.7). Since the two steps occur directly after one another, we obtain for the high-level computation that any race between k_t and l must be annotated as shared. Since step l is local in the low-level computation, it could not be a shared read, thus eliminating races where step k_t is the modifying step, and it could also not be a shared write that has an effect, and thus (in the low-level computation) there would not be a race where step l is the modifying step either. Thus there would be no race between l and k_t in the low-level machine.

This plan fails because, as mentioned before, there could be a malicious step $k' \in [t : l)$ which changes the sets of accessed addresses of step k_t, in one of the following two ways: step k' changes the instruction executed by step l, e.g., by changing it to a NOOP; or step k_t is an RMW instruction, such as a test-and-set, and the modification made by step k' changes the value of the test variable so that the test-and-set no longer makes a modification. Recall that this is possible

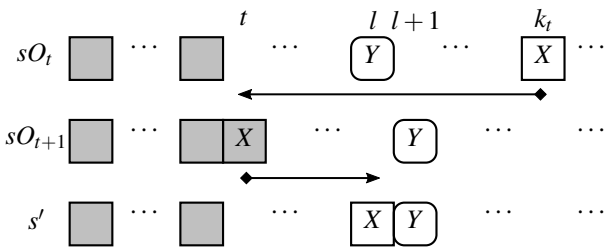


Figure 1.7: A naive plan for proving that sO_{t+1} is equivalent to sO_t moves the former step k_t (now step t) back next to each l (reordering depicted as schedule s' in the figure) to show that there is no race between k_t and l in the low-level machine.

even though step k' is a local step in the low-level machine because step k' could be buffering a shared write, and thus only change the behavior of step k_t in the high-level machine. Therefore step l in the schedule s' from Fig. 1.7 is no longer reduced, and even if there is no race in the high-level computation, a race might exist in the low-level computation.

We instead with Lemma 344 only move the former step k_t as far as possible, which is right before such a step k'. The resulting schedule, denoted as s'' in Fig. 1.8, has a step at $k'+1$ racing with step k', and as shown in Lemma 342 must thus be shared. Because it is shared but local in the low-level machine, the step must be buffering a write which is executed in the high-level computation, and is thus executing trusted code (since untrusted code uses low-level semantics). We want to show that there is no race with step $l+1$ (which was step l in the original schedule). This *almost* corresponds to Section 4.5, the technical software condition introduced above, except for one crucial detail: the steps between $k'+1$ and $l+1$ might be by different objects.

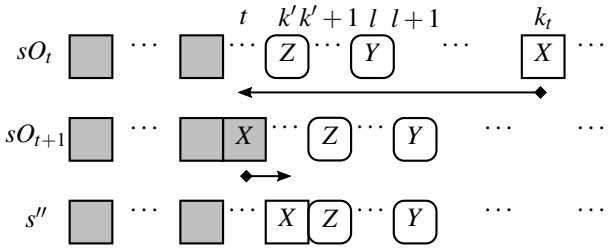


Figure 1.8: Instead of moving until position l, we move only until position k'.

We apply one last crucial trick: before moving step k_t at all, we sort with Lemma 319 the local tail of the schedule, moving all steps of the object that makes step l to the very front. In Fig. 1.8, this is object Y. In the resulting schedule r, shown in Fig. 1.9, step l is moved to position $l' \leq l$ and all steps between t and l' are made by the same object, namely Y. It is easy to see that schedule r is equivalent to the original schedule, since only local steps occurring next to each other have to be reordered (we use bubblesort [Knu98] to sort the schedule) and in a reduced schedule, two local steps that occur directly next to each other never have a race.

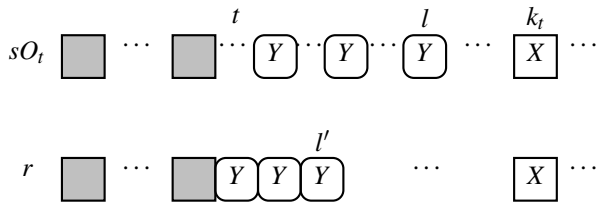


Figure 1.9: Before moving step k_t to the front, we sort the local tail.

Furthermore, if we would now try to move step k_t to position t and then back to position l', there might still be a step k' that prevents us from doing so, but in this case all steps between $k'+1$ and $l'+1$ would obviously have to be made by the same object (cf. Fig. 1.10). For this case, we show with Lemma 352 that the technical software condition from above applies. By Lemma 348 we can

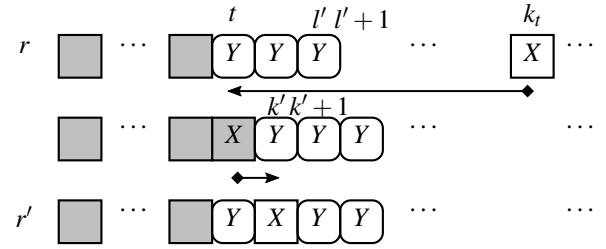


Figure 1.10: In the sorted schedule, a race between a malicious step $k'+1$ and step k' can only occur if all steps between $k'+1$ and $l'+1$ are made by the same object.

now immediately use Section 4.5 to show that there may not be a race between step $l'+1$ and step k' in schedule r' of Fig. 1.10, which is stated jointly by Lemmas 357, 360 and 364. It is at this point an easy book keeping exercise to show that there can also be no race between step l and step k_t in schedule sO_t (stated jointly by Lemmas 362, 365 and 368.

Thus in both cases the schedule after iteration $t+1$ is equivalent to that after iteration t (this is Theorem 1).

5. Section 4.13: We now have that after t iterations of the reordering, a prefix of length t is ordered (Theorem 1). This works great for finite schedules such as the one of the simple program from before, but in case of, e.g., an operating system which hopefully does not crash after a finite amount of steps but can run indefinitely, may not be enough. To show that for infinite schedules there is also an equivalent, reduced schedule, we observe that steps in the ordered prefix of the reordered schedule are never moved. We thus obtain a growing, finite prefix. By diagonalizing through the reorderings and defining

$$s_\infty(t) = sO_{t+1}(t),$$

we therefore obtain a schedule that is ordered at every point and thus reduced, as stated by Theorem 4. It is also easy to show that every step in $s_\infty(t)$ sees the same values as the step in sO_{t+1}, and since that schedule is equivalent to s, also as the original step in that schedule.

However it is not as easy to show that every step from s eventually ends up in the growing prefix, and therefore in s_∞. A glance at Fig. 1.6 shows that step t in schedule sO_t of object X_1 is delayed several times, and it is not clear why it should not be delayed like this indefinitely and thus never enter the growing finite ordered prefix. In fact, the reordering strategy described above is slightly incomplete and might delay steps indefinitely in that manner. The actual reordering strategy is sometimes greedy and does not always choose g_t to be the next global step; it sometimes chooses local steps that are not buffering anything, and which might therefore be pushed into nirvana if the unit that makes the step never again makes a global step. We call these steps *pushable*, and define g_t to be the next step which is global *or pushable*. That this does not harm the correctness proof of the reordering strategy is easy to see: a local step is moved easily from k_t to position t without breaking equivalence, and because the step does not buffer a write, it can never buffer a shared write that would make the schedule not ordered.

That this suffices rests on one central observation. If a step is not pushable, it is either a) global and will be picked up as g_t eventually, or b) it is buffering a write which has to be committed at some point, and that commit will be picked up as g_t eventually and pick up the step in question as k_t (since it is made by the same unit). These observations are combined in Theorem 5, which states that all schedules can be reordered into an equivalent, reduced schedule.

Chapter 2

Machine Model

In this chapter we introduce the model and semantics of our machine, as well as the general notation used through the thesis.

2.1 Notation

We introduce some shared notation and language for the following chapters. The set of booleans is

$$\mathbb{B} = \{ 0, 1 \}.$$

The set of sequences of X of length n is denoted by

$$X^n,$$

e.g., for 32-bit strings we use

$$\mathbb{B}^{32}.$$

Finite sequences are indexed in the subscript, starting from zero:

$$l \in X^n \to l = [l_0, \ldots, l_{n_1}] \wedge l_i \in X.$$

For concatenation of sequences we use the binary infix symbol $\circ$, where the lower indices are on the left and the higher indices are on the right:

$$l \in X^n \wedge l' \in X^k \wedge i < n+k \to (l \circ l')_i = \begin{cases} l_i & i < n \\ l'_{i-n} & \text{o.w.} \end{cases}$$

A slice of a sequence is obtained by

$$l[n:k] = [l_n, \ldots, l_k].$$

The head is the zero-indexed element, and the tail are the remaining elements:

$$l \in X^{n+1} \to hd(x) = l_0 \wedge tl(x) = l[1:n].$$

For the empty list, we define head and tail to be some fresh symbol $\perp \notin X$

$$hd(\varepsilon) = tl(\varepsilon) = \perp.$$

The set of finite sequences is denoted by

$$X^*.$$

Analogous to set comprehension we employ a list comprehension

$$[\, f(e) \mid e \in l \wedge P(e) \,],$$

which preserves the order of the elements in x and is defined recursively by the following

$$[\, f(e) \mid e \in \varepsilon \wedge P(e) \,] = \varepsilon,$$
$$[\, f(e) \mid e \in (h \circ t) \wedge P(e) \,] = \begin{cases} f(h) \circ [\, f(e) \mid e \in t \wedge P(e) \,] & P(h) \\ [\, f(e) \mid e \in t \wedge P(e) \,] & \text{o.w.}, \end{cases}$$

If $P(e)$ is always equivalent to 1, we drop the conjunct.

One easily shows that the list comprehension distributes over concatenation; we do not show a proof.

Lemma 1.

$$\left[\, f(e) \;\middle|\; e \in (l \circ l') \wedge P(e) \,\right] = [\, f(e) \mid e \in l \wedge P(e) \,] \circ \left[\, f(e) \;\middle|\; e \in l' \wedge P(e) \,\right].$$

We say *N and M intersect* when their intersection is non-empty. Formally we write

$$N \barwedge M \equiv N \cap M \neq \emptyset.$$

This relation binds weaker than set operators.

For intervals, we use a bracket when the endpoint is included and a parenthesis when the endpoint is excluded

$$[t : k) = \{\, t, \ldots, k-1 \,\}.$$

We similarly use the words "until k" and "before k" to denote inclusion resp. exclusion when k is the higher endpoint and (somewhat inconsistently) "from t" and "after t" to denote inclusion resp. exclusion when t is the lower endpoint.

A dependent product, written

$$\Pi x \in X.Y(x),$$

is a function type where the co-domain depends on the particular input. A formal definition is given by

$$\Pi x \in X.Y(x) = \left\{ f : X \to \bigcup_x Y(x) \;\middle|\; \forall x \in X.f(x) \in Y(x) \right\}.$$

Just as in other quantifiers, the '.' in the definition binds weaker than other connectives. See the following example.

$$\Pi x \in X.Y(x) \times Z(x) \to \bigcup_y T(y,x) \;=\; \Pi x \in X.(Y(x) \times Z(x) \to \bigcup_y T(y,x)).$$

Just as in other quantifiers, we sometimes bind multiple variables with a single Π

$$\Pi x_1 \in X_1, \ldots, x_n \in X_n.Y(x_1, \ldots, x_n) = \Pi(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n.Y(x_1, \ldots, x_n).$$

In general we do not distinguish between cascaded function types[1]

$$f : X \to Y \to Z,$$

and cartesian function types

$$f : X \times Y \to Z,$$

but we prefer the cartesian style over the cascaded style. Therefore the following two are considered equal, while the second notation is preferred

$$f(x)(y) = f(x,y).$$

Dependent products are well-known to type theorists, but are considered strange and complicated by some people working with set theory. This does not have to be the case as dependent products are used a lot by people who work with set theory, and we give some examples. The first example are parametric functions where the type of the function depends on the parameter, like a device transition function of device d that maps from the cartesian product of the set of device configurations of that device $K_{DEV,d}$ and set of oracle inputs of that device $\Sigma_{DEV,d}$ into the set of device configurations of that device

$$\delta_{DEV,d} : K_{DEV,d} \times \Sigma_{DEV,d} \to K_{DEV,d}.$$

If one defines instead a device transition function δ'_{DEV} of which d is a proper argument, one has to do so with a dependent product

$$\delta'_{DEV} : \Pi d \in Devices.K_{DEV,d} \times \Sigma_{DEV,d} \to K_{DEV,d}.$$

It can be read as "given a device d, a device configuration of d and an oracle input of d, we obtain a new device configuration of d." Let now d be a device, $c \in K_{DEV,d}$ a device configuration of that device and $x \in \Sigma_{DEV,d}$ a corresponding oracle input. One obtains the equality

$$\delta_{DEV,d}(c,x) = \delta'_{DEV}(d,c,x).$$

For configurations c and components x we use the notation

$$c.x.$$

Components can be functions in which case we use the notation

$$c.f(a).$$

In our simulation proofs we will be talking about a low-level machine which simulates a high-level machine. In such cases, we use a machine-type index $M \in \{\downarrow, \uparrow\}$ in the subscript to distinguish between definitions for the high-level machine and definitions for the low-level machine. For example, we will later define a buffered write BW, and distinguish between the write buffered in the low-level machine

$$BW_{\downarrow}$$

and the write buffered in the high-level machine

$$BW_{\uparrow}.$$

[1] As is usual, function (and implication) arrows are right-associative and bind weaker than other connectives except for '.'.

We model the non-determinism in our machine by oracle inputs in our transition operator. From each configuration, multiple transitions may be enabled; the oracle input determines which of these transitions to take. Given a sequence s of oracle inputs this allows us to define a sequence of configurations, where configuration $n+1$ is the result of applying the transition operator with configuration n and oracle input n. We call such a sequence of configurations *a computation*.

The same sequence s of oracle inputs can, however, yield different configurations in the different machines, and again we distinguish a computation in the low-level machine from computations in the high-level machine with a machine-type index M

$$c_M[s].$$

A list of commonly used symbols and notations and their general meaning is given below.

- The set of all addresses $\mathcal{A}$
- Sets of addresses $A \subseteq \mathcal{A}$
- Each address $a \in \mathcal{A}$ has a range $V(a)$
- Configurations c are labeled tuples
- Components $c.x$ are projections of configurations where x is the label of the projection
- We often consider an original configuration c^0
- The set of units U
- Units i, j
- The set of oracle step inputs Σ
- Oracle step inputs $x \in \Sigma$
- Steps c, x and the resulting configuration $c\,.\,x$. We call $.$ the transition operator.
- Schedules s are sequences of oracle step inputs.

2.2 The Programming Language

We sometimes provide example programs. The programs use a C-like pseudo syntax. Unless stated otherwise, variables are stored in shared memory, have integer value, and are initially zero. A shared store x`.store(`v`)` updates the value of variable x to v. A shared load x`.load()` is an expression that evaluates to the current value of variable x. An atomic compare-and-swap operation x`.cas(`$cmp \rightarrow new$`)` atomically changes the value of variable x to *new* if the old value is equal to *cmp*, and leaves it unchanged otherwise; in each case it evaluates to the old value of x. It is always a shared operation. The assignment operator =, on the other hand, is a local operation. Loads and stores and the normal assignment operator use the buffer, while an atomic compare-and-swap operation flushes the buffer and modifies the memory directly. Finally we allow a local bypassing assignment operator `:=`, which is a local operation that modifies the memory directly but does not drain the write buffer (and therefore bypasses the stores

in the write buffer). Parallel threads are denoted by vertical bars `||`, and statement boundaries are denoted by `;`.

We create sub-word accesses with `x[`a`:`b`]`, e.g., `x[0:7] = x[8:15];` copies the second byte of `x` to the first byte of `x`.

Furthermore, we give threads access to a thread-local struct `ICR` with the following fields.

`ICR.target:` A thread which is to be interrupted.

`ICR.status:` A boolean flag with values `PENDING` and `IDLE`. An interrupt is eventually sent when the status is `PENDING`. The interrupt also sets the status back to `IDLE`.

Finally, we give threads access to a thread-local variable `IRR` (for interrupt request register), which is a boolean flag that records whether an inter-processor interrupt has been received.

Our programming language has two semantics: a high-level semantics which is sequentially consistent and where all writes immediately change the shared memory (and ICR/IRR), and a low-level semantics where only bypassing assignment and compare-and-swap operations immediately change the shared memory, and all other assignments can be delayed by the write buffer.

2.3 The Machine Model

We consider a machine with units U and addresses $\mathcal{A}$. The exact values of U and $\mathcal{A}$ are left uninterpreted and are open to the instantiation of the model, but we make some assumptions.

Addresses $\mathcal{A}$ can be partitioned into processor registers for each unit $i \in U$,

$$A_{PR,i},$$

also known as core registers; registers of write-triggered devices $d \in D$

$$A_{DEV,d},$$

and main memory registers

$$A_{MEM}.$$

Processor registers are partitioned further into interrupt and normal registers

$$A_{PR,i} = A_{IPR,i} \uplus A_{NPR,i}.$$

The difference between these is that the interrupt registers can be modified and read by other processors in order to send inter processor interrupts, while the normal processor registers are completely local for the processor. In the case of MIPS, the APIC registers that are used to compute the interrupt vector — i.e., the interrupt request register IRR and the in-service register ISR — will be interrupt registers, whereas the remaining registers such as the ICR will not be. The distinction is important as we will not allow interrupt registers to be modified by buffered writes in strong memory mode, as the write would immediately trigger an interrupt in a sequentially consistent model but will not trigger the interrupt in a weak memory model until the write leaves the buffer,

possibly many instructions later. The remaining APIC registers, however, can be modified by buffered writes; in fact, since the APIC likely is not integrated into the cache system, this may be the only way to modify those APIC registers at all.

We define shorthands for the set of all device addresses

$$A_{DEV} = \bigcup_d A_{DEV,d}$$

and for the set of all interrupt registers

$$A_{IPR} = \bigcup_i A_{IPR,i}.$$

Visible to processor i are only interrupt registers, its own normal registers, device registers, and memory registers

$$ACC_i = A_{NPR,i} \cup A_{IPR} \cup A_{DEV} \cup A_M.$$

Processors can buffer writes to the interrupt registers, device registers, and memory registers, but never to the normal processor registers[2]. For example, the write buffer can not be used to buffer modifications of the program counter or the GPR. We define the set of *bufferable addresses*

$$BA = A_{IPR} \cup A_{DEV} \cup A_M.$$

Furthermore, each oracle input x may cause direct-to-core IPIs under certain circumstances. These circumstances are entirely defined by the content of a memory region $A_{IPI}(x)$, which belongs to interrupt and device registers[3], but not memory registers

$$A_{IPI}(x) \subseteq A_{IPR} \cup A_{DEV}.$$

We call these registers the *IPI-relevant registers for x*. In the case of MIPS86, they are the entries of the redirection table for the IO APIC delivery step, and the command and status registers for the APIC delivery steps. We do not need to assume anything about their structure or semantics, merely that they exist; the aim is to prevent processor steps from disabling interrupts that target them, in which case interrupts might be disabled in the low-level semantics by seemingly local steps. Since interrupts are a form of synchronization, programs that disable interrupts that target them using bypassing local writes seem well-synchronized, as the following example shows.

```
ICR.target.store(Thread1);  ||  while (!IRR);
ICR.status.store(PENDING);  ||  assert(false);
ICR.status.store(IDLE);     ||
ICR.target := Thread2;      ||
```

Clearly, in sequentially consistent executions, Thread 2 can only be a target when the status register has already been set to IDLE, and thus Thread 2 will never be interrupted.

It is also clear that this is not true in the low-level machine

[2]This is not technically necessary for the proof to go through, but it simplifies the proof and makes sense.

[3]This makes sure that accesses to these registers are marked as shared.

```
ICR.target.store(Thread1); -> BUF  ||
BUF -> ICR.target.store(Thread1);  ||
ICR.status.store(PENDING); -> BUF  ||
BUF -> ICR.status.store(PENDING);  ||
ICR.status.store(IDLE); -> BUF     ||
ICR.target := Thread2;             ||
IPI: Thread2.IRR = 1;              ||
                                   || while (!IRR);
                                   || assert(false);
```

The program still seems well-synchronized because the APIC is never active at the time when the target is changed.

Each address $a \in \mathcal{A}$ has a specific value domain $V(a)$. For example, an instruction buffer can be modeled as a relation between physical instruction addresses and instruction words. In an instantiation, one could model the instruction buffer of processor i as a processor register IB_i, with all relations from instruction addresses to instruction words

$$V(\mathrm{IB}_i) \subseteq 2^{32} \times 2^{32},$$

while an addressable MIPS memory cell $m[x]$ has an 8 bit value

$$V(\mathrm{m}[x]) = \mathbb{B}^8,\ x \in \mathbb{B}^{32}.$$

For more examples, see our instantiation of the model for MIPS86 in Chapter 3.

A configuration of each machine is simply a function that maps each address to a value from its domain. We generalize this concept to sets of addresses $A \subseteq \mathcal{A}$ and introduce the set of valuations over A

$$Val(A) = \Pi a \in A.V(a).$$

We denote by $\emptyset$ the empty valuation. If address a is not in the domain of valuation ν, we use the notation

$$\nu(a) = \perp,$$

where $\perp$ is a fresh symbol not used in the domain of any address

$$\perp \notin \bigcup_a V(a).$$

Therefore, two valuations agree on the value of an address iff either a is in the domain of both and both valuations yield the same value, or if a is in the domain of neither valuation.

We use valuations also for modifications to the memory, assigning to each address a new value. Since modifications only change a subset of addresses, we introduce partial valuations of $A \subseteq \mathcal{A}$ as total valuations of subsets of A by

$$PVal(A) = \bigcup_{A' \subseteq A} Val(A').$$

A configuration for our write buffer- and sequentially consistent machines has two components, a memory

$$c.m \in Val(\mathcal{A})$$

and for each unit $i \in U$ a write buffer, which is a sequence of writes to the bufferable addresses

$$c.wb(i) \in PVal(BA)^*.$$

Note that the component $c.m$ contains the state not only of the shared memory, but also of processors and devices.

We also consider an initial configuration

$$c^0,$$

about which we only make the assumption that write buffers are empty:

$$c^0.wb(i) = \varepsilon.$$

In what follows we will run both the low-level and the high-level machine starting with this configuration.

2.4 Memory Update and Semantics of Devices

A device configuration is a valuation of device addresses

$$K_{DEV,d} = Val(A_{DEV,d}),$$

and a device oracle input is a partial valuation of device addresses

$$\Sigma_{DEV,d} = PVal(A_{DEV,d}).$$

In other words, the state of a device is the content of the registers of the device, while a device oracle input is an update of the device state. We assume a *device transition function*

$$\delta_{DEV,d} : K_{DEV,d} \times \Sigma_{DEV,d} \to K_{DEV,d}.$$

We say v *and* v' *agree on addresses* A if the value of the addresses are equal between the two configurations. Formally, we define

$$v =_A v' \equiv \forall a \in A.v(a) = v'(a).$$

We define a memory update operator which updates a valuation v by a more up-to-date well-formed write $w \in Val(A)$. Since a device modification depends on the current state of the device, the whole device state becomes an input to steps that modify the device. Similarly a modification of a single address in the device can trigger a change of all addresses in the device. We define the *device closure* of a set of addresses A by including all addresses of devices which have addresses in A:

$$dc(A) = \bigcup \left\{ A_{DEV,d} \;\middle|\; A_{DEV,d} \mathbin{\dot{\cap}} A \right\}.$$

The set of device addresses contained in A is a subset of the device closure of A.

Lemma 2.

$$A_{DEV} \cap A \subseteq dc(A).$$

Proof.

$$\begin{aligned} A_{DEV} \cap A &= \left\{ a \;\middle|\; a \in A_{DEV,d} \cap A \right\} \\ &= \left\{ a \in A_{DEV,d} \;\middle|\; a \in A_{DEV,d} \cap A \right\} \\ &\subseteq \left\{ a \in A_{DEV,d} \;\middle|\; A_{DEV,d} \mathbin{\dot{\cap}} A \right\} \\ &= \bigcup \left\{ A_{DEV,d} \;\middle|\; A_{DEV,d} \mathbin{\dot{\cap}} A \right\} \\ &= dc(A). \end{aligned}$$

□

We define the *inclusive device closure* of A as the union of A and its device closure

$$idc(A) = A \cup dc(A).$$

We say that a set of addresses A is closed under devices if it is equal to its inclusive device closure

$$closed(A) \equiv A = idc(A).$$

This is the case iff it contains its device closure.

Lemma 3.

$$closed(A) \equiv dc(A) \subseteq A.$$

Proof. $closed(A) \rightarrow dc(A) \subseteq A$**:** Assume A is equal to its its inclusive device closure

$$A = idc(A).$$

We rewrite this equality on the right-hand-side of the claim and reduce it to the following

$$dc(A) \subseteq idc(A).$$

We unfold the definition of *idc*

$$dc(A) \subseteq A \cup dc(A),$$

which is trivially true.

$dc(A) \subseteq A \rightarrow closed(A)$**:** Assume A is contained in its device closure

$$dc(A) \subseteq A.$$

Unfolding the definition of *closed* and *idc* we reduce the claim to the following

$$A = A \cup dc(A).$$

We show that both sides of the equality are contained in each other.

$A \subseteq A \cup dc(A)$**:** This is trivially true.

$A \cup dc(A) \subseteq A$**:** Follows directly with the assumption

$$A \cup dc(A) \subseteq A \cup A = A.$$

□

All of the fixed sets of addresses are closed. We do not present proofs.

Lemma 4. *Let A be one of the major sets of addresses*

$$A \in \left\{ A_{MEM}, A_{PR,i}, A_{NPR,i}, A_{IPR,i}, A_{IPR}, A_{DEV,d}, A_{DEV}, BA, ACC_i \right\},$$

then A is closed

$$closed(A).$$

We define now ν updated by $w \in Val(A)$ as

$$\nu \circledast w(a) = \begin{cases} \delta_{DEV,d}(\nu|_{A_{DEV,d}}, w|_{A_{DEV,d}})(a) & a \in dc(A) \cap A_{DEV,d} \\ w(a) & a \in A \setminus dc(A) \\ \nu(a) & \text{o.w.} \end{cases}$$

This operator is left-associative

$$\nu \circledast w \circledast w' = (\nu \circledast w) \circledast w'.$$

Agreement under device closure is preserved by updates.

Lemma 5.

$$A \subseteq idc(Dom(w)) \wedge closed(A) \wedge \nu =_{dc(A)} \nu' \rightarrow \nu \circledast w =_A \nu' \circledast w.$$

Proof. We have to show that for addresses $a \in A$ the two valuations agree

$$\nu \circledast w(a) = \nu' \circledast w(a).$$

The proof is only difficult for addresses in the device closure

$$a \in dc(Dom(w)) \cap A.$$

This is only the case for device addresses

$$a \in A_{DEV,d},$$

which are consequently also in the device closure of A

$$A_{DEV,d} \subseteq dc(A).$$

Consequently the valuations agree on the state of the device d

$$\nu|_{A_{DEV,d}} = \nu'|_{A_{DEV,d}}$$

and we easily conclude

$$\begin{aligned} \nu \circledast w(a) &= \delta_{DEV,d}(\nu|_{A_{DEV,d}}, w|_{A_{DEV,d}})(a) \\ &= \delta_{DEV,d}(\nu'|_{A_{DEV,d}}, w|_{A_{DEV,d}})(a) = \nu' \circledast w(a). \end{aligned}$$

□

A set is closed under devices iff it either contains all registers of a device, or none of them.

Lemma 6.

$$closed(B) \equiv \forall d.A_{DEV,d} \subseteq B \vee A_{DEV,d} \mathbin{\dot{\not\cap}} B.$$

Proof. We show the equivalence first from left to right and then from right to left.

$\Longrightarrow$**:** It suffices to show that if the device registers intersect with B, they are all contained in B. Assume thus

$$A_{DEV,d} \mathbin{\dot{\cap}} B.$$

By definition the device registers are in the device closure, and since B is closed, they must also be in B

$$A_{DEV,d} \subseteq dc(B) \subseteq B.$$

$\Longleftarrow$**:** Assume that the registers of each device are either contained in B or disjoint from it

$$\forall d.A_{DEV,d} \subseteq B \vee A_{DEV,d} \mathbin{\dot{\not\cap}} B.$$

We apply Lemma 3 and reduce the claim to showing that the device closure of B is subsumed by it

$$dc(B) \overset{!}{\subseteq} B.$$

Let $b \in dc(B)$ one of the addresses, and it suffices to show that b is contained in B

$$b \overset{!}{\in} B.$$

Clearly b is a register of some device d

$$b \in A_{DEV,d},$$

the addresses of which therefore intersect with B

$$A_{DEV,d} \mathbin{\dot{\cap}} B.$$

By assumption, all addresses of the device must be contained in B,including b

$$b \in A_{DEV,d} \subseteq B,$$

which is the claim.

□

The device closure distributes over union.

Lemma 7.

$$dc(A \cup B) = dc(A) \cup dc(B).$$

Proof.

$$\begin{aligned} dc(A \cup B) &= \bigcup \left\{ A_{DEV,d} \;\middle|\; A_{DEV,d} \mathbin{\dot{\cap}} A \cup B \right\} \\ &= \bigcup \left\{ A_{DEV,d} \;\middle|\; A_{DEV,d} \mathbin{\dot{\cap}} A \vee A_{DEV,d} \mathbin{\dot{\cap}} B \right\} \\ &= dc(A) \cup dc(B). \end{aligned}$$

□

The device closure of an intersection is a subset of the intersection of the device closures.

Lemma 8.

$$dc(A \cap B) \subseteq dc(A) \cap dc(B).$$

Proof.

$$\begin{aligned} dc(A \cap B) &= \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot{\cap}} A \cap B \} \\ &\subseteq \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot{\cap}} A \wedge A_{DEV,d} \mathbin{\dot{\cap}} B \} \\ &= dc(A) \cap dc(B). \end{aligned}$$

□

The device closure is monotone.

Lemma 9.

$$A \subseteq B \rightarrow dc(A) \subseteq dc(B).$$

Proof.

$$\begin{aligned} dc(A) &= \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot{\cap}} A \} \\ &\subseteq \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot{\cap}} B \} \\ &= dc(B). \end{aligned}$$

□

The device closure distributes over subtraction with closed sets.

Lemma 10.

$$closed(B) \rightarrow dc(A \setminus B) = dc(A) \setminus dc(B).$$

Proof. We show first that the intersection between the registers of a device and $A \setminus B$ is either the intersection between A and the registers of the device, or empty.

$$\begin{aligned} A_{DEV,d} \cap (A \setminus B) &= (A_{DEV,d} \setminus B) \cap A \\ &= \begin{cases} A_{DEV,d} \cap A & A_{DEV,d} \not\mathbin{\dot{\cap}} B \\ \emptyset & A_{DEV,d} \subseteq B. \end{cases} \end{aligned}$$

The case distinction is exhaustive due to Lemma 6.

Consequently, the registers of a device intersect $A \setminus B$ iff they intersect A but not B

$$\begin{aligned} A_{DEV,d} \mathbin{\dot{\cap}} (A \setminus B) &\iff A_{DEV,d} \cap (A \setminus B) \neq \emptyset \\ &\iff \begin{cases} A_{DEV,d} \cap A \neq \emptyset & A_{DEV,d} \not\mathbin{\dot{\cap}} B \\ \emptyset \neq \emptyset & A_{DEV,d} \subseteq B \end{cases} \\ &\iff A_{DEV,d} \cap A \neq \emptyset \wedge A_{DEV,d} \not\mathbin{\dot{\cap}} B \\ &\iff A_{DEV,d} \mathbin{\dot{\cap}} A \wedge A_{DEV,d} \not\mathbin{\dot{\cap}} B. \end{aligned}$$

The claim follows:

$$dc(A \setminus B) = \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot{\cap}} (A \setminus B) \}$$

$$
\begin{aligned}
&= \bigcup \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\cap} A \wedge A_{DEV,d} \mathbin{\dot\not\cap} B \,\} \\
&= \bigcup \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\cap} A \,\} \cap \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\not\cap} B \,\} \\
&= \bigcup \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\cap} A \,\} \setminus \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\cap} B \,\} \\
&= \left(\bigcup \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\cap} A \,\} \right) \setminus \bigcup \{\, A_{DEV,d} \mid A_{DEV,d} \mathbin{\dot\cap} B \,\} \\
&= dc(A) \setminus dc(B).
\end{aligned}
$$

□

The complement of a closed set is also closed.

Lemma 11.

$$closed(A) \rightarrow closed(\mathcal{A} \setminus A)$$

Proof. Assume A is closed. We apply Lemma 6, which reduces the claim to showing that the registers of a device are either contained in or disjoint from the complement of A

$$A_{DEV,d} \subseteq \mathcal{A} \setminus A \overset{!}{\vee} A_{DEV,d} \mathbin{\dot\not\cap} \mathcal{A} \setminus A.$$

A set is a subset of another set iff it does not intersect its complement. We apply this twice and reduce the claim to showing that the device addresses are either disjoint from or contained in the complement of the complement of A

$$A_{DEV,d} \mathbin{\dot\not\cap} \mathcal{A} \setminus (\mathcal{A} \setminus A) \overset{!}{\vee} A_{DEV,d} \subseteq \mathcal{A} \setminus (\mathcal{A} \setminus A).$$

The complement of the complement of a set is the set, and thus it suffices to show that the device addresses are either disjoint from or contained in A

$$A_{DEV,d} \mathbin{\dot\not\cap} A \overset{!}{\vee} A_{DEV,d} \subseteq A.$$

By Lemma 6 this is equivalent to showing that A is closed, which we have by assumption. □

The intersection of a closed set is also closed.

Lemma 12.

$$closed(A) \wedge closed(B) \rightarrow closed(A \cap B)$$

Proof. With Lemmas 3 and 10 we obtain that the device closure of the intersection is a subset of the intersection

$$
\begin{aligned}
dc(A \cap B) &\subseteq dc(A) \cap dc(B) && \text{L 10} \\
&\subseteq A \cap B. && \text{L 3}
\end{aligned}
$$

The claim is now Lemma 3. □

The device closure can be used to identify intersections between a set and device registers.

Lemma 13.

$$dc(A) \mathbin{\dot\cap} A_{DEV,d} \iff A \mathbin{\dot\cap} A_{DEV,d}.$$

Proof. Follows by definition of dc and the disjointness of device registers

$$\begin{aligned} dc(A) \mathbin{\dot\cap} A_{DEV,d} &\iff \bigcup \{\, A_{DEV,d'} \mid A \mathbin{\dot\cap} A_{DEV,d'} \,\} \mathbin{\dot\cap} A_{DEV,d} \\ &\iff \exists d'.A_{DEV,d'} \mathbin{\dot\cap} A_{DEV,d} \wedge A \mathbin{\dot\cap} A_{DEV,d'} \\ &\iff \exists d'.d = d' \wedge A \mathbin{\dot\cap} A_{DEV,d'} \\ &\iff A \mathbin{\dot\cap} A_{DEV,d}. \end{aligned}$$

□

Since device closures are just a union of device registers, we obtain a similar result for those.

Lemma 14.

$$dc(A) \mathbin{\dot\cap} dc(B) \iff A \mathbin{\dot\cap} dc(B).$$

Proof. Follows with Lemma 13

$$\begin{aligned} dc(A) \mathbin{\dot\cap} dc(B) &\iff dc(A) \mathbin{\dot\cap} \bigcup \{\, A_{DEV,d} \mid B \mathbin{\dot\cap} A_{DEV,d} \,\} \\ &\iff \exists d.dc(A) \mathbin{\dot\cap} A_{DEV,d} \wedge B \mathbin{\dot\cap} A_{DEV,d} \\ &\iff \exists d.A \mathbin{\dot\cap} A_{DEV,d} \wedge B \mathbin{\dot\cap} A_{DEV,d} && \text{L 13} \\ &\iff A \mathbin{\dot\cap} \bigcup \{\, A_{DEV,d} \mid B \mathbin{\dot\cap} A_{DEV,d} \,\} \\ &\iff A \mathbin{\dot\cap} dc(B). \end{aligned}$$

□

If a set is closed under devices, it intersects with another set iff it intersects with the inclusive device closure of that set.

Lemma 15.

$$closed(A) \to A \mathbin{\dot\cap} B \iff A \mathbin{\dot\cap} idc(B).$$

Proof. Because A is closed, it is equal to its inclusive device closure

$$A = idc(A) = A \cup dc(A).$$

The claim now follows with Lemma 14

$$\begin{aligned} A \mathbin{\dot\cap} B &\iff (A \cup dc(A)) \mathbin{\dot\cap} B \\ &\iff A \mathbin{\dot\cap} B \vee dc(A) \mathbin{\dot\cap} B \\ &\iff A \mathbin{\dot\cap} B \vee dc(A) \mathbin{\dot\cap} dc(B) && \text{L 14} \\ &\iff A \mathbin{\dot\cap} B \vee A \mathbin{\dot\cap} dc(B) && \text{L 14} \\ &\iff A \mathbin{\dot\cap} (B \cup dc(B)) \\ &\iff A \mathbin{\dot\cap} idc(B). \end{aligned}$$

□

Complementarily, a set is a subset of a closed set iff its inclusive device closure is a subset of that set.

Lemma 16.

$$closed(A) \to B \subseteq A \iff idc(B) \subseteq A.$$

Proof. We negate both sides

$$B \not\subseteq A \overset{!}{\iff} idc(B) \not\subseteq A.$$

Clearly a set is not a subset of another set iff it intersects with the complement of that set. We can thus change the claim to the following

$$B \dot{\cap} (\mathcal{A} \setminus A) \overset{!}{\iff} idc(B) \dot{\cap} (\mathcal{A} \setminus A).$$

We apply Lemma 15, which reduces the claim to showing that the complement of A is closed

$$closed(\mathcal{A} \setminus A),$$

which is Lemma 11. □

Inclusive device closures intersect iff the original sets or their device closures intersect

Lemma 17.

$$idc(A) \dot{\cap} idc(B) \iff A \dot{\cap} B \vee dc(A) \dot{\cap} dc(B).$$

Proof. Follows by applying twice Lemma 14

$$\begin{aligned} idc(A) \dot{\cap} idc(B) &\iff (A \cup dc(A)) \dot{\cap} (B \cup dc(B)) \\ &\iff A \dot{\cap} B \vee dc(A) \dot{\cap} B \vee A \dot{\cap} dc(B) \vee dc(A) \dot{\cap} dc(B) \\ &\iff A \dot{\cap} B \vee dc(A) \dot{\cap} dc(B) \vee dc(A) \dot{\cap} dc(B) \vee dc(A) \dot{\cap} dc(B) && \text{L 14} \\ &\iff A \dot{\cap} B \vee dc(A) \dot{\cap} dc(B). \end{aligned}$$

□

The device closure is idempotent.

Lemma 18.

$$dc(B) = dc(dc(B)).$$

Proof. The claim easily follows with Lemma 13:

$$\begin{aligned} dc(B) &= \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \dot{\cap} B \} \\ &= \bigcup \{ A_{DEV,d} \mid A_{DEV,d} \dot{\cap} dc(B) \} \\ &= dc(dc(B)). \end{aligned}$$

□

The device closure of the inclusive device closure is the device closure of the original set.

Lemma 19.

$$dc(idc(B)) = dc(B).$$

Proof. With Lemmas 7 and 18

$$\begin{aligned} dc(idc(B)) &= dc(B \cup dc(B)) \\ &= dc(B) \cup dc(dc(B)) && \text{L 7} \\ &= dc(B) \cup dc(B) && \text{L 18} \\ &= dc(B). \end{aligned}$$

□

The device closure is closed.

Lemma 20.

$$closed(dc(B)).$$

Proof. Follows with Lemma 18

$$dc(B) = dc(B) \cup dc(B) = dc(B) \cup dc(dc(B)).$$

□

Inclusive device closures are always closed.

Lemma 21.

$$closed(idc(A)).$$

Proof. We obtain first with Lemma 19 that the device closure of the inclusive device closure is a subset of the inclusive device closure

$$\begin{aligned} dc(idc(A)) &= dc(A) && \text{L 19} \\ &\subseteq A \cup dc(A) \\ &= idc(A). \end{aligned}$$

The claim follows with Lemma 3. □

Lemma 22.

$$idc(A) \mathbin{\hat{\cap}} B \iff A \mathbin{\hat{\cap}} idc(B).$$

Proof. By Lemma 21, both $idc(A)$ and $idc(B)$ are closed

$$closed(idc(A)),\ closed(idc(B)).$$

The claim follows with Lemma 15.

$$\begin{aligned} idc(A) \mathbin{\hat{\cap}} B &\iff idc(A) \mathbin{\hat{\cap}} idc(B) && \text{L 15} \\ &\iff A \mathbin{\hat{\cap}} idc(B). && \text{L 15} \end{aligned}$$

□

One can subtract any device closure from a closed set and the resulting set is still closed.

Lemma 23.

$$closed(A) \to closed(A \setminus dc(B)).$$

Proof. It suffices to show that the device closure of $A \setminus dc(B)$ is a subset of $A \setminus dc(B)$, which follows with Lemmas 10 and 18.

$$\begin{aligned} dc(A \setminus dc(B)) &= dc(A) \setminus dc(dc(B)) && \text{L 10} \\ &= dc(A) \setminus dc(B) && \text{L 18} \\ &\subseteq A \setminus dc(B). \end{aligned}$$

□

For some of the lemmas above, we obtain variants for the inclusive device closure. We state the ones we need without giving proofs.

Lemma 24.

$$idc(A \cup B) = idc(A) \cup idc(B).$$

Lemma 25.

$$idc(idc(A)) = idc(A).$$

Lemma 26.

$$A \subseteq B \to idc(A) \subseteq idc(B).$$

Lemma 27.

$$idc(A) \mathbin{\hat{\cap}} A_{DEV,d} \iff A \mathbin{\hat{\cap}} A_{DEV,d}.$$

A write only modifies the inclusive device closure of its domain

Lemma 28.

$$A \mathbin{\hat{\not\cap}} idc(Dom(w)) \to \nu \circledast w =_A \nu.$$

Since addresses outside of the domain of the write are unchanged, we can obtain a lemma similar to Lemma 5 for sets that are not subsumed in the inclusive device closure of the write if we assume that the memories also agree on those portions not subsumed by the device closure.

Lemma 29.

$$closed(A) \wedge \nu =_A \nu' \to \nu \circledast w =_A \nu' \circledast w.$$

Proof. Clearly the intersection between A and the inclusive device closure of the domain of the write is a subset of the inclusive device closure of the domain of the write

$$A \cap idc(Dom(w)) \subseteq idc(Dom(w)).$$

The inclusive domain closure is closed by Lemma 21

$$closed(idc(Dom(w)))$$

and since A is closed, so is their intersection by Lemma 12

$$closed(A \cap idc(Dom(w))),$$

and by Lemma 3 we conclude that the device closure of their intersection is contained in A

$$\begin{aligned} dc(A \cap idc(Dom(w))) &\subseteq A \cap idc(Dom(w)) && \text{L 3} \\ &\subseteq A. \end{aligned}$$

We conclude first that the memories agree on that region of memory

$$\nu =_{dc(A \cap idc(Dom(w)))} \nu'$$

and then with Lemma 5 that the updated memories agree on that set

$$\nu \circledast w =_{A \cap idc(Dom(w))} \nu' \circledast w.$$

On the other hand the remaining part does not intersect with the inclusive device closure of the domain of the write

$$A \setminus idc(Dom(w)) \not\dot{\cap} idc(Dom(w))$$

and with Lemma 28 we immediately obtain that the updated memories also still agree on that portion

$$\begin{aligned} \nu \circledast w &=_{A \setminus idc(Dom(w))} \nu && \text{L 28} \\ &=_{A \setminus idc(Dom(w))} \nu' \\ &=_{A \setminus idc(Dom(w))} \nu' \circledast w && \text{L 28.} \end{aligned}$$

The claim follows. □

If two writes do not interact, their order is irrelevant.

Lemma 30. *For writes*

$$w \in Val(A), w' \in Val(A')$$

which do not interact

$$idc(A) \not\dot{\cap} idc(A'),$$

the order is irrelevant

$$\nu \circledast (w \cup w') = \nu \circledast w \circledast w'.$$

Proof. Note that for devices the lack of intersection of the device closures implies that each device completely belongs to at most one write

$$(w \cup w')\big|_{A_{DEV,d}} = \begin{cases} w\big|_{A_{DEV,d}} & A_{DEV,d} \dot{\cap} A \\ w'\big|_{A_{DEV,d}} & A_{DEV,d} \dot{\cap} A'. \end{cases}$$

We fold and unfold the definitions and ultimately obtain three cases

$$\begin{aligned} \nu \circledast (w \cup w')(a) &= \begin{cases} \delta_{DEV,d}(\nu\big|_{A_{DEV,d}}, w \cup w'\big|_{A_{DEV,d}})(a) & a \in dc(A \cup A') \cap A_{DEV,d} \\ (w \cup w')(a) & a \in A \cup A' \setminus dc(A \cup A') \\ \nu(a) & \text{o.w.} \end{cases} \\ &= \begin{cases} \delta_{DEV,d}(\nu\big|_{A_{DEV,d}}, w\big|_{A_{DEV,d}})(a) & a \in dc(A) \cap A_{DEV,d} \\ \delta_{DEV,d}(w\big|_{A_{DEV,d}}, w'\big|_{A_{DEV,d}})(a) & a \in dc(A') \cap A_{DEV,d} \\ w(a) & a \in A \setminus dc(A) \\ w'(a) & a \in A' \setminus dc(A') \\ \nu(a) & \text{o.w.} \end{cases} \\ &= \begin{cases} (\nu \circledast w)(a) & a \in dc(A) \cap A_{DEV,d} \\ (\nu \circledast w')(a) & a \in dc(A') \cap A_{DEV,d} \\ (\nu \circledast w)(a) & a \in A \setminus dc(A) \\ ((\nu \circledast w) \circledast w')(a) & a \in A' \setminus dc(A') \\ (\nu \circledast w)(a) & \text{o.w.} \end{cases} \end{aligned}$$

$$= \begin{cases} (v \circledast w')(a) & a \in dc(A') \cap A_{DEV,d} \\ ((v \circledast w) \circledast w')(a) & a \in A' \setminus dc(A') \\ (v \circledast w)(a) & \text{o.w.} \end{cases}$$

$$= \begin{cases} (v \circledast w')(a) & a \in dc(A') \\ ((v \circledast w) \circledast w')(a) & a \in A' \setminus dc(A') \\ (v \circledast w)(a) & a \notin idc(A'). \end{cases}$$

We complete the three cases as follows.

$a \in dc(A')$**:** With Lemma 28 we obtain that w does not change the device closure of A'

$$v \circledast w =_{dc(A')} v.$$

With Lemma 20 we obtain that the device closure is closed

$$closed(dc(A')),$$

and thus with Lemma 29 the device closure is still unchanged after applying w'

$$v \circledast w \circledast w' =_{dc(A')} v \circledast w'.$$

The claim follows with $a \in dc(A')$

$$(v \circledast w \circledast w')(a) = (v \circledast w')(a) = v \circledast (w \cup w')(a).$$

$a \in A' \wedge a \notin dc(A')$**:** This is already the claim

$$v \circledast (w \cup w')(a) = v \circledast w \circledast w'(a).$$

$a \notin idc(A')$**:** With Lemma 28 we obtain the following

$$v \circledast w \circledast w' =_{\mathcal{A} \setminus idc(A')} v \circledast w.$$

The claim follows with $a \in \mathcal{A} \setminus idc(A')$

$$(v \circledast w \circledast w')(a) = (v \circledast w)(a) = v \circledast (w \cup w')(a).$$

□

If we are only interested in a small closed portion of memory, we can drop all other parts of a memory update.

Lemma 31.

$$closed(A) \rightarrow v \circledast w =_A v \circledast w\big|_A.$$

Proof. Let B be the addresses in the domain of w not included in A

$$B = Dom(w) \setminus A.$$

Clearly we can split w between A and B

$$w = w\big|_A \cup w\big|_B.$$

Because A is closed, it equals its own inclusive device closure

$$idc(A) = A.$$

We obtain with Lemma 15 that the inclusive device closures of A and B do not intersect

$$\begin{aligned}
& idc(A) \mathbin{\dot\cap} idc(B) \\
\iff & A \mathbin{\dot\cap} idc(B) \\
\iff & A \mathbin{\dot\cap} B && \text{L 15} \\
\iff & A \mathbin{\dot\cap} Dom(w) \setminus A \\
\iff & 0.
\end{aligned}$$

The claim now follows with Lemmas 30 and 28

$$\begin{aligned}
\nu \circledast w &= \nu \circledast (w\big|_A \cup w\big|_B) \\
&= \nu \circledast w\big|_A \circledast w\big|_B && \text{L 30} \\
&=_A \nu \circledast w\big|_A. && \text{L 28}
\end{aligned}$$

□

If two memory updates agree on the portion we are interested in, we can swap them.

Lemma 32.

$$closed(A) \wedge w =_A w' \to \nu \circledast w =_A \nu \circledast w'.$$

Proof. We simply apply Lemma 31 twice

$$\begin{aligned}
\nu \circledast w &=_A \nu \circledast w\big|_A && \text{L 31} \\
&= \nu \circledast w'\big|_A \\
&=_A \nu \circledast w'. && \text{L 31}
\end{aligned}$$

□

2.5 The Semantic Framework

We define our non-deterministic semantics in terms of an oracle step input $x \in \Sigma$. We distinguish for each unit $i \in U$ between oracle step inputs for processor steps and write buffer steps

$$\Sigma_{P,i}, \Sigma_{WB,i}.$$

These sets of oracle inputs are pairwise disjoint, which allows us to define $u(x)$ to be *the unit to which x belongs*

$$u(x) = i \text{ where } x \in \Sigma_{WB,i} \cup \Sigma_{P,i}.$$

For the sake of precise language, we use the term *object* for processors and write buffers. Analogously to the unit we define $o(x)$ to be the object to which an oracle input x belongs

$$o(x) = X \text{ where } x \in \Sigma_X,\ X \in \{ P,i \mid i \in U \} \cup \{ WB,i \mid i \in U \}.$$

Each processor step has three transitions, shown also in Fig. 2.1

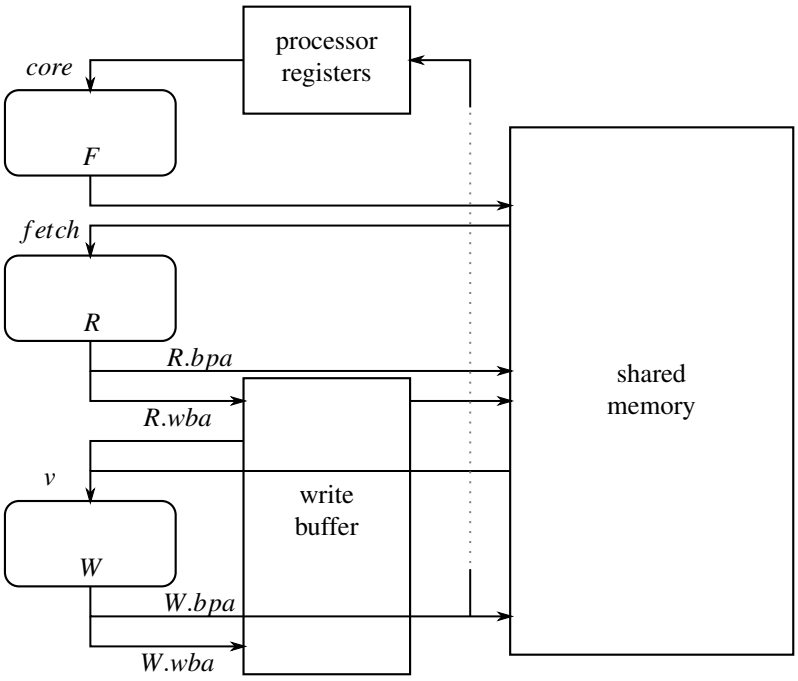


Figure 2.1: The three transitions in the processor step are represented as rounded boxes on the left. First, the processor only sees the core configuration $core$ and uses it to compute F, the set of addresses to fetch from. Then the memory returns the fetched instructions $fetch$, and the processor uses core configuration and fetched addresses to compute $R.wba$, the set of addresses to be read where more up-to-date values from the buffer are preferred over the shared memory, and $R.bpa$, the set of addresses to be read where the buffer is completely ignored. As a result, the processor returns a memory view v which combines the results for $R.wba$ (from memory and buffer) and of $R.bpa$ (only from memory). Using the memory view v and the previous results $(core, fetch)$, the processor computes two writes: $W.bpa$ which immediately modifies memory, and $W.wba$ which is entered into the buffer and only modifies memory later. The portion $W.bpa$ is also used to update the core configuration.

Fetch: The processor first considers its local state and decides which addresses to fetch from.

Read: After fetching the values from the memory, the processor decides which addresses to read from. It distinguishes for each address whether it wants to snoop its own write buffers for more up-to-date values or not.

Write: After receiving the results from the memory system, the processor now decides where and what to write. This includes writing to its local state, e.g., to increment the program counter or to store the read results.

Since we are in a setting of write buffers, read and write accesses to the memory system may go through the write buffer or bypass it. In case of a read, going through the write buffer means forwarding of more up-to-date values if they are present in the

buffer; in case of a write, it means inserting a new write at the end of the buffer. In fact, the same step may do some part of the access through the buffer (such as a store operation) and bypass it for other parts of the access (such as incrementing the program counters, etc.). To formalize this, we define a *disjoint split* of a set A as the disjoint pairs of subsets of A

$$D(A) = \{ (wba, bpa) \mid wba, bpa \subseteq A \wedge wba \not\cap bpa \}.$$

Note that we use the acronyms *wba* for write buffer addresses and *bpa* for bypassing addresses, respectively.

We overload notation to allow a disjoint pair of sets of addresses as a parameter of *PVal*. We simply distribute the sets of addresses over the partial valuations, i.e., the partial valuation of a pair is the cross product of the partial valuations of the components of the pair

$$PVal((wba, bpa)) = PVal(wba) \times PVal(bpa).$$

In the high-level machine, write buffers are still sometimes used, e.g., in untrusted code of users and in the code that switches between the untrusted code and the trusted code. To distinguish between those two types of code, each processor records its current memory mode in *memory mode registers* $A_{SC,i}$, which are normal processor registers

$$A_{SC,i} \subseteq A_{NPR,i};$$

the current memory mode can be extracted using a function SC_i

$$SC_i : Val(A_{SC,i}) \to \mathbb{B}.$$

In the low-level machine, the function does not matter; but in the high-level machine, a unit in sequentially consistent (or *strong memory*) mode essentially ignores the write buffer: writes that should go through the buffer are executed immediately, the write buffer is not snooped for more up-to-date values, and write buffer steps have no effect. The write buffer is kept, however, as a ghost component, and is filled with the writes that the machine would have buffered if it had not been in strong memory mode.

A unit is in strong memory mode in configuration c when its memory mode registers indicate that it is[4]

$$SC_{iM}(c) = SC_i(c.m|_{A_{SC,i}}).$$

In order to easily define forwarding, we overload $\circledast$ by allowing the write to be in the form of a sequence:

$$\nu \circledast wb,\ wb \in PVal(\mathcal{A})^*.$$

In this case, the entries of the sequence are simply applied from left to right:

$$\nu \circledast w \circ wb = \nu \circledast w \circledast wb,\ \nu \circledast \varepsilon = \nu.$$

We now formalize forwarding. We define *perfect world forwarding memory system* by simply applying all outstanding writes to the memory:

$$pfms_{iM}(c) = c.m \circledast c.wb(i).$$

This perfect world forwarding memory system predicts the value of device transitions correctly, i.e., if a write in the write buffer is going to update a write-triggered device,

[4] We later define for functions f_M with a machine-type index notation for schedules $f_M[s](t)$. To make use of this notation, we sometimes add a machine-type index where it is not necessary, as here.

the device state is correctly computed and forwarded. Therefore, a write buffer step never changes the perfect world forwarding memory system of the processor that issued the write (cf. Lemma 68, page 62).

It is easy to see that this perfect world forwarding can not be implemented in hardware, as the forwarding hardware can not possibly predict the behavior of devices. Since we are creating a processor model for which we claim that it is implementable, we have to use a different function for forwarding in our model. We define an *uninformed update* operator which treats device modifications like normal writes, i.e., does not apply the device transition

$$v \circledcirc v' = \begin{cases} v'(a) & a \in Dom(v') \\ v(a) & \text{o.w.}, \end{cases}$$

and define the *forwarding memory system* by

$$fms_{iM}(c) = c.m \circledcirc c.wb(i).$$

This forwarding memory system does not correctly predict results for devices correctly, and so a program that first modifies and then reads from a write-triggered device might not behave sequentially consistent. We state without proof variants of Lemmas 28 and 29 for the uninformed update.

Lemma 33.

$$v =_A v' \rightarrow v \circledcirc w =_A v' \circledcirc w.$$

Lemma 34.

$$w \in Val(B) \wedge B' \not\cap B \rightarrow v \circledcirc w =_{B'} v.$$

Furthermore, the uninformed update and the update agree on everything except devices.

Lemma 35.

$$dc(A) \not\cap Dom(w) \rightarrow v \circledast w =_A v \circledcirc w.$$

Proof. With Lemma 14 we obtain that the device closure of the domain of the write does not intersect A

$$\begin{aligned} & dc(A) \not\cap Dom(w) \\ \iff & dc(A) \not\cap dc(Dom(w)) && \text{L 14} \\ \iff & A \not\cap dc(Dom(w)) && \text{L 14.} \end{aligned}$$

Let $a \in A$ be an address for which we have to show the equality

$$v \circledast w(a) \stackrel{!}{=} v \circledcirc w(a).$$

Clearly a can not be in be in the device closure of the domain of the write

$$a \notin dc(Dom(w)).$$

The claim follows now by simply unfolding the definitions on each side.

$$(v \circledast w)(a) = \begin{cases} \ldots & a \in dc(Dom(w)) \cap A_{DEV,d} \\ w(a) & a \in A \setminus dc(A) \\ v(a) & \text{o.w.} \end{cases}$$

$$= \begin{cases} w(a) & a \in A \setminus dc(A) \\ \nu(a) & \text{o.w.} \end{cases}$$
$$= (\nu \circledcirc w)(a).$$

□

We consider three list operations. Each of those list operations is a partial function that maps a list and one list element into the set of lists

$$\delta : X^* \times X \to X^*.$$

In this context, $l, p, q \in X^*$ are usually lists and $e, x \in X$ are usually single elements.
The first function, *push*, adds the element to the end of of the list

$$push(l, e) = l \circ e.$$

The second function, *pop*, drops the head of the list

$$pop(l, e) = \begin{cases} \varepsilon & l = \varepsilon \\ tl(l) & \text{o.w.} \end{cases}$$

The third function, *noop*, just leaves the list as-is

$$noop(l, e) = l.$$

We will apply these transition functions frequently for write buffers, which are lists of partial valuations.

We show that the list comprehension and the list operations commute.

Lemma 36.

$$\delta \in \{ push, pop, noop \} \to \delta([\, f(e) \mid e \in l \,], f(x)) = [\, f(e) \mid e \in \delta(l, x) \,].$$

Proof. By case distinction on δ. We only show the case for the push operation

$$\delta = push,$$

the other cases are simpler. The proof easily follows with Lemma 1

$$\begin{aligned} \delta([\, f(e) \mid e \in l \,], f(x)) &= push([\, f(e) \mid e \in l \,], f(x)) \\ &= [\, f(e) \mid e \in l \,] \circ f(x) \\ &= [\, f(e) \mid e \in l \,] \circ [\, f(e) \mid e \in [x] \,] \\ &= [\, f(e) \mid e \in l \circ x \,] && \text{L 1} \\ &= [\, f(e) \mid e \in push(l, x) \,] \\ &= [\, f(e) \mid e \in \delta(l, x) \,]. \end{aligned}$$

□

We state a few simple lemmas about lists without proof.

Lemma 37. *If an element x is in the list l but is not the head, then it must be in the tail.*

$$x \neq hd(l) \wedge x \in l \rightarrow x \in tl(l).$$

Lemma 38. *If an element x is in the list l but is not the head, then the tail can not be empty.*

$$x \neq hd(l) \wedge x \in l \rightarrow tl(l) \neq \varepsilon.$$

Lemma 39. *If the tail of a list l is not empty, and that list is split into a prefix p and a single-element suffix x, then the prefix must also be non-empty.*

$$tl(l) \neq \varepsilon \wedge l = p \circ x \rightarrow p \neq \varepsilon.$$

Lemma 40. *If a list l is split into a prefix p and a suffix q, and the prefix is non-empty, the list and its prefix have the same head.*

$$p \neq \varepsilon \wedge l = p \circ q \rightarrow hd(l) = hd(p).$$

Lemma 41. *If a prefix p is not empty, tail commutes with concatenation.*

$$p \neq \varepsilon \rightarrow tl(p \circ q) = tl(p) \circ q.$$

Push and noop only add elements at the tail.

Lemma 42.

$$\delta \in \{ push, noop \} \rightarrow \exists q. \delta(l,e) = l \circ q.$$

Proof. By case distinction on δ.

$\delta = push$**:** Claim follows with $q := e$

$$push(l,e) = l \circ e.$$

$\delta = noop$**:** Claim follows with $q := \varepsilon$

$$noop(l,e) = l = l \circ \varepsilon.$$

□

If a list is non-empty, push and noop do not change the head element.

Lemma 43.

$$\delta \in \{ push, noop \} \wedge l \neq \varepsilon \rightarrow hd(\delta(l,e)) = hd(l).$$

Proof. By Lemma 42 we obtain a suffix q that is added to l

$$\delta(l,e) = l \circ q.$$

The claim is just Lemma 40

$$hd(\delta(l,e)) = hd(l \circ q) = hd(l).$$

□

Pushing or doing nothing does not make the list empty; we do not show a proof.

Lemma 44.

$$\delta \in \{ push, noop \} \wedge l \neq \varepsilon \rightarrow \delta(l,e) \neq \varepsilon.$$

If a list is non-empty during push, all operations commute with the tail operation.

Lemma 45.

$$(\delta \neq push \vee p \neq \varepsilon) \rightarrow tl(\delta(p,e)) = \delta(tl(p),e).$$

Proof. By case distinction on δ.

$\delta = push$**:** The claim is proven by Lemma 41

$$tl(push(p,e)) = tl(p \circ e) = tl(p) \circ e = push(tl(p),e).$$

$\delta = pop$**:** Claim follows

$$tl(pop(p,e)) = tl(tl(p)) = pop(tl(p),e).$$

$\delta = noop$**:** Claim follows

$$tl(noop(p,e)) = tl(p) = noop(tl(p),e).$$

□

Using the fact that a *pop* applies a tail operation, we obtain that *pop* commutes with other operators in the same manner.

Lemma 46.

$$(\delta \neq push \vee p \neq \varepsilon) \wedge \delta' = pop \rightarrow \delta'(\delta(p,e),e') = \delta(\delta'(p,e'),e).$$

Proof. Follows directly from Lemma 45

$$\begin{aligned} \delta'(\delta(p,e),e') &= pop(\delta(p,e),e') \\ &= tl(\delta(p,e)) \\ &= \delta(tl(p),e) \\ &= \delta(\delta'(p,e'),e). \end{aligned}$$

□

Recall the three transitions from before (fetch, read, write). We specify uninterpreted functions for these transitions, which again are defined only in the instantiation. In what follows, we bind the big union operator weaker than all other connectors, in particular weaker than the function arrow.

For the fetch transition, we specify an uninterpreted function *fetch-set* for each processor. We are only allowed to fetch from accessible registers

$$F : \bigcup_i Val(A_{PR,i}) \times \Sigma_{P,i} \rightarrow 2^{ACC_i}.$$

Note that we have specified the function type as union over processor-specific domains. One of the parameters is an oracle input of the processor $\Sigma_{P,i}$, which serves to distinguish the individual function domains; as a consequence, given a configuration

of processor registers $core \in Val(A_{PR,i})$ and a oracle step input $x \in \Sigma_{P,i}$, the fetched addresses are always accessible to unit $i = u(x)$

$$F(core,x) \subseteq ACC_{u(x)}.$$

For the read transition, we have the value of the processor registers and the value of the fetched addresses. Since the fetched addresses depend on the state of the processor registers, we have to use a dependent product.

$$R : \bigcup_i \Pi core \in Val(A_{PR,i}), x \in \Sigma_{P,i}.Val(F(core,x)) \to D(ACC_i).$$

Given the state of the processor registers $core \in Val(A_{PR,i})$, the oracle input $x \in \Sigma_{P,i}$, and the value of the fetched addresses $fetch \in Val(F(core,x))$, we thus obtain two disjoint sets of addresses

$$R(core,x,fetch).wba \subseteq ACC_{u(x)},$$
$$R(core,x,fetch).bpa \subseteq ACC_{u(x)},$$

each of which is accessible to the unit $i = u(x)$ to which the oracle input belongs.

For the write transition, we specify a function *prepared writes* for each processor, using the dependent product to make explicit the dependency on the results from memory after fetching and reading:

$$PW : \bigcup_i \Pi core \in Val(A_{PR,i}), x \in \Sigma_{P,i}, fetch \in Val(F(core,x)).Val(R(core,x,fetch)) \to \bigcup_{d \in D(ACC_{u(x)})} PVal(d).$$

Given the state of the processor registers $core \in Val(A_{PR,i})$, the oracle input $x \in \Sigma_{P,i}$, the value of the fetched addresses $fetch \in Val(F(core,x))$, and the memory view (results of the read) $v \in Val(R(core,x,fetch))$, we obtain a disjoint pair of sets of addresses to be written

$$(wba,bpa) \in D(ACC_{u(x)}),$$

and partial valuations of each component of the pair

$$PW(core,x,fetch,v).wba \in PVal(wba),$$
$$PW(core,x,fetch,v).bpa \in PVal(bpa).$$

We restrict the domain of the buffered prepared write further to buffered addresses

$$PW(core,x,fetch,v).wba \subseteq BA. \tag{2.1}$$

Unlike processor steps, which are defined in terms of uninterpreted functions like PW, the semantics of write buffer steps are completely defined in our semantics; we give the definitions later.

Not all steps are meaningful. Most noticeably, write buffers can not make steps while the buffer is empty, and memory management units can not create translations for pages which are not present. We use *guard conditions*, which in general are predicates on the inputs of a step, to formally exclude such steps. In order to make this possible we distinguish further between instance guard conditions, which are defined in the

instantiation and may indeed only depend on normal inputs of a step (e.g., the read-set for a processor step), and *buffer conditions*, which in addition may depend on the state of the write buffer. An important detail about instance guard conditions are that they, too, can be split between two decision points: at Decision 1, only the local state and the step input are checked, while at Decision 2, one can also check the values from the memory. The idea is that at Decision 1, one checks whether an access to those addresses is at all feasible, while at Decision 2, one can check the overall validity of the step. This distinction will play an important role in the definition of the software conditions: for detecting races we include all feasible accesses, even if they are not valid. For a concrete example, in the formalization of MIPS86 given by Schmaltz [Sch13], a page fault step is feasible if the given incomplete walk is in the TLB and matches the address to be accessed, but it is only valid if the extension of the walk would indeed be faulty. Note that if the step is invalid there would be a valid step that accesses the same address, namely the walk extension, so no additional memory accesses (and thus races) are created. Our theorem write buffer reduction theorem obviously works best if no races are created by feasible but invalid steps.

We specify the instance guard conditions for feasibility by

$$\Phi_P : \bigcup_i Val(A_{PR,i}) \times \Sigma_{P,i} \to \mathbb{B},$$

and for validity by by

$$\Upsilon_P : \bigcup_i \Pi core \in Val(A_{PR,i}), x \in \Sigma_{P,i}.Val(F(core,x)) \to \mathbb{B}.$$

We consider three sources of write buffer drains:

1. Explicit drain requests issued by the program. In MIPS-86 these are fence and compare-and-swap instructions. We specify these by a predicate *fence*

   $$fence : \bigcup_i \Pi core \in Val(A_{PR,i}), x \in \Sigma_{P,i}.Val(F(core,x)) \to \mathbb{B},$$

   which again is defined in the instantiation.

2. Partial hits, which are situations in which a processor is issuing a read through the write buffer, for which there is no consistent most-up-to-date value in the write buffer. We define the most-up-to-date value for a given set of addresses A as the *hit of A in wb* recursively by

   $$hit(A, \varepsilon) = \bot,$$
   $$hit(A, wb \circ \nu) = \begin{cases} \nu & Dom(\nu) \dot{\cap} A \\ hit(A, wb) & \text{o.w.} \end{cases}$$

   Note that $hit(A, wb) = \bot$ if and only if there is no hit for A. We overload *hit* as a predicate which is true iff there is a hit

   $$hit(A, wb) \equiv hit(A, wb) \neq \bot.$$

   We say *A has a partial hit in wb* when there is a hit which can not serve A because it does not have values for all addresses in a:

   $$phit(A, wb) \equiv hit(A, wb) \neq \bot \wedge A \not\subseteq Dom(hit(A, wb)).$$

3. Writes into the local registers of other processors, which drain the buffers of those processors. This situation occurs, for example, in case one places the interrupt request registers of an APIC into the local registers of a processor, so that each step can snoop the value of the processor before deciding what and if to fetch, and without creating a race. This is true in particular in the semantics of MIPS86, and we will explain why the hardware-enforced flush is necessary in that case. The interrupt command register of the sending processor changes in the same cycle as the interrupt request registers of the victim processors. These registers are also immediately used to interrupt the processor and thus prevent the current threads on the victim processors from making any further steps. In a weak memory machine, the victim processors might still have buffered writes. If the writes would be allowed to discharge after the interrupt is sent, these steps would look to the sending processor like additional steps of the interrupted thread, which contradicts the strong view of the sending processor. Consider the following example, where the ICR has two fields *target* and *status*, which define which thread to interrupt and whether an interrupt has been sent, respectively

```
fence; x = 1; || fence; ICR.target = Thread1;
fence;        || fence; ICR.status = PENDING;
              || fence; while (ICR.status != IDLE);
              || fence; y = x;
              || fence; z = x;
```

In a strong memory model, one easily verifies that `y = z` after every execution of the program. The fences in this program do nothing, however, to prevent the following execution:

```
                        || fence; ICR.target = Thread1;
                        || fence; ICR.status = PENDING;
fence; x = 1; -> BUF    ||
                        || IPI: Thread1.IRR = 1;
                        || fence; ICR.status != IDLE;
                        || fence; y = x;
BUF -> x = 1;           ||
                        || fence; z = x;
```

The Intel manual [Int, p. 2397] specifies that the buffer is drained when an "interrupt is generated". This likely means the point where the processor actually reacts to the IPI, which can be considerably later than the step in which the IPI is considered "delivered" by the sending processor. This does not prevent the weak memory behavior of the program above. There are at least two other ways to regain sequential consistency in this setting. The first is to define weaker constraints for the ordering between the change of the ICR and the actual interrupt, i.e., specify that processors may continue to make a finite but undetermined amount of steps after receiving an interrupt request. In our semantics, that would mean making the IRR an external register (in A_{MEM} or A_{DEV}), and non-deterministically fetching from the IRR to snoop for interrupts. In order to prevent spurious steps, one could make steps that snoop the IRR but do not find an interrupt invalid (this is permissible, since snooping the IRR is done with the

fetching accesses). Note that this works independently of whether the hardware actually reacts to the interrupt immediately or not, since one can simply add a trivial layer of simulation between a hardware machine that reacts immediately and an abstract write buffer machine that uses oracle inputs to snoop for the interrupt: whenever the hardware machine is interrupted, we simulate a snooping step in the abstract write buffer machine. Our machine model is compatible with this computational model.

The second is to enforce a software protocol where the interrupt has to be acknowledged by the victim processors by writing to shared variables, and a busy wait of all involved processors until all victims have sent their acknowledgment. How exactly such a protocol and a write buffer reduction proof based on such a protocol would look like is an open question.

We combine explicit flushes and partial hits in internal buffer conditions for processor steps:

$$\Delta_P(core, fetch, wb, x) = \begin{cases} wb = \varepsilon & fence(core, x, fetch) \\ \neg phit(Dom(R(core, x, fetch).wba), wb) & \text{o.w.} \end{cases}$$

Since APICs for us are just processors that access the processor registers of other processors, we formalize the drain on writing to interrupt registers of other units by an external buffer condition for processor steps. This condition takes a set of victims V and a mapping wbp of unit indexes to write buffer configurations of the processors (short for *write buffers of processors*)

$$wbp : U \to PVal(BA)^*$$

and holds if the write buffers of all victim processors are empty

$$\Delta_{IPI}(V, wbp) \equiv \bigwedge_{j \in V} wbp(j) = \varepsilon.$$

Finally we add for write buffer steps the condition that the buffer be non-empty:

$$\Delta_{WB}(wb) \equiv wb \neq \varepsilon.$$

For write buffers of processors, we use the notation

$$wbp =_i wbp' \equiv wbp(i) = wbp'(i).$$

We state a series of trivial lemmas without proof.
If there is a hit, the hit is an element of the buffer.

Lemma 47.

$$hit(A, wb) \to hit(A, wb) \in wb.$$

If A subsumes B, buffers that have a hit with B also have a hit with A (although not necessarily the same one).

Lemma 48.

$$B \subseteq A \wedge hit(B, wb) \rightarrow hit(A, wb).$$

If there is a hit with $A \cup B$ there is also a hit with A or a hit with B and vice versa.

Lemma 49.

$$hit(A \cup B, wb) \iff hit(A, wb) \vee hit(B, wb).$$

Equivalently, there is no hit with $A \cup B$ iff there is no hit with A nor a hit with B.

Lemma 50.

$$\neg hit(A \cup B, wb) \iff \neg hit(A, wb) \wedge \neg hit(B, wb).$$

A hit in a write buffer exists if there is a hit in the upper half or in the lower half. If there is a hit in the lower half, the hit is the hit of the lower half.

Lemma 51.

1. $hit(A, wb \circ wb') = \begin{cases} hit(A, wb') & hit(A, wb') \\ hit(A, wb) & o.w. \end{cases}$.
2. $hit(A, wb \circ wb') \iff hit(A, wb) \vee hit(A, wb')$.

Lemmas 33, 34, 28, 30, and 35 can be easily extended to sequences of writes by simply iterating the Lemma. We again do not present proofs.

Lemma 52.

$$\nu =_A \nu' \rightarrow \nu \circledcirc wb =_A \nu' \circledcirc wb.$$

Lemma 53.

$$\neg hit(A, wb) \rightarrow \nu \circledcirc wb =_A \nu.$$

Lemma 54.

$$closed(A) \wedge \neg hit(A, wb) \rightarrow \nu \circledast wb =_A \nu.$$

Lemma 55.

$$\neg hit(idc(Dom(w)), wb) \rightarrow \nu \circledast w \circledast wb = \nu \circledast wb \circledast w.$$

Lemma 56.

$$\neg hit(dc(A), wb) \rightarrow \nu \circledast wb =_A \nu \circledcirc wb.$$

Note that buffers are, in a sense, monotone. If a processor step can be done with one configuration of the buffers, dropping more elements from the buffer will not invalidate that step. This is true because fewer writes in the buffer cause fewer hits, and thus also fewer partial hits.

Lemma 57.

$$\Delta_P(core, fetch, wb \circ wb', x) \rightarrow \Delta_P(core, fetch, wb', x).$$

Proof. We distinguish between fences and other steps.

$fence(core, fetch, x)$**:** Then the large write buffer is empty, and so is the suffix

$$wb \circ wb' = \varepsilon = wb'.$$

The claim follows

$$\Delta_P(core, fetch, wb', x) \equiv wb' = \varepsilon \equiv 1.$$

Otherwise: There can not be a partial hit in the large buffer

$$\neg phit(Dom(R(core,x,fetch).wba), wb \circ wb').$$

Let the accessed memory region be A

$$A = R(core,x,fetch).wba.$$

By unfolding the definition of partial hit, we obtain that there either is no hit with A in the large buffer or A can be served by the hit

$$\neg hit(A, wb \circ wb') \vee A \subseteq Dom(hit(A, wb \circ wb')). \tag{2.2}$$

We distinguish whether there was a hit in the suffix or not.

$\neg hit(A, wb')$**:** Therefore there is also no partial hit in the suffix, and the drain condition is satisfied

$$\Delta_P(core, fetch, x, wb') = \neg phit(A, wb') = \neg hit(wb') \vee \ldots = 1.$$

$hit(A, wb')$**:** By Lemma 51, that hit is the hit of the large buffer

$$hit(A, wb \circ wb') = hit(A, wb').$$

We substitute hits in the suffix for hits in the large buffer in Equation (2.2)

$$\neg hit(A, wb') \vee A \subseteq Dom(hit(A, wb')).$$

By definition of *phit*, we obtain that there is no partial hit

$$\neg phit(A, wb'),$$

and the claim follows

$$\Delta_P(core, fetch, wb', x) \equiv \neg phit(A, wb') \equiv 1.$$

□

2.5.1 Functions and Steps

We have defined and specified so far a lot of functions in terms of valuations and oracle inputs. Our machine semantics, however, will be defined in terms of configurations and oracle inputs, so we will now overload and generalize those functions to work in a uniform way with steps.

We extend the definition of a unit to steps in the obvious way for the sake of uniform notation

$$u_M(c,x) = u(x),$$

and say that $u_M(c,x)$ is *the unit making step* c,x.

We obviously have that the unit is independent of the configuration.

Lemma 58.

$$u_M(c,x) = u_N(c',x).$$

Proof.

$$u_M(c,x) = u(x) = u_N(c',x).$$

□

We also extend the definition of objects to steps

$$o_M(c,x) = o(x).$$

A step is done in sequentially consistent mode if the unit making that step is in sequentially consistent mode at the beginning of the step

$$SC_M(c,x) = SC_{u_M(c,x)M}(c).$$

We define the local state used in the step as *core inputs* or *local inputs* by

$$C_M(c,x) = \begin{cases} A_{PR,i} & x \in \Sigma_{P,i} \\ A_{SC,i} & x \in \Sigma_{WB,i}, \end{cases}$$

Lemma 59. *The local inputs do not depend on the configuration or machine type*

$$C_M(c,x) = C_N(c',x).$$

Proof. Straightforward

$$\begin{aligned} C_M(c,x) &= \begin{cases} A_{PR,i} & x \in \Sigma_{P,i} \\ A_{SC,i} & x \notin \Sigma_{P,i} \end{cases} \\ &= C_N(c',x). \end{aligned}$$

□

Only the unit to which the processor registers belong uses those registers as its core registers

Lemma 60.

$$C_M(c,x) \dot{\cap} A_{PR,i} \to u_M(c,x) = i.$$

Proof. By definition of $C_M(c,x)$ we obtain that the step is made by processor i or its write buffer

$$x \in \Sigma_{P,i} \cup \Sigma_{WB,i}$$

and the claim follows

$$u_M(c,x) = u(x) = i.$$

□

The core configuration used during step c,x is defined by

$$core_M(c,x) = c.m\big|_{C_M(c,x)}.$$

Note that this is curried notation: the core configuration is itself a valuation, i.e., a map from addresses to values

$$core_M(c,x)(a) \in V_a.$$

The fetched addresses in step c,x are defined by

$$F_M(c,x) = \begin{cases} F(core_M(c,x),x) & x \in \Sigma_{P,i} \\ \emptyset & \text{o.w.}, \end{cases}$$

Lemma 61. *The fetched addresses only depend on the core configuration*

$$core_M(c,x) = core_N(c',x) \rightarrow F_M(c,x) = F_N(c',x).$$

Proof. We distinguish between processor and write buffer steps.

$x \in \Sigma_{P,i}$**:** Straightforward

$$\begin{aligned} F_M(c,x) &= F(core_M(c,x),x) \\ &= F(core_N(c',x),x) = F_N(c',x). \end{aligned}$$

$x \notin \Sigma_{WB,i}$**:** Nothing is fetched

$$F_M(c,x) = \emptyset = F_N(c',x).$$

□

The fetch result is defined by simply taking the value of the fetched addresses in the memory

$$fetch_M(c,x) = c.m\big|_{F_M(c,x)}.$$

The addresses read in step c,x are defined by

$$R_M(c,x) = \begin{cases} R(core_M(c,x),x,fetch_M(c,x)) & x \in \Sigma_{P,i} \\ (\emptyset,\emptyset) & \text{o.w.} \end{cases}$$

Note that again the read addresses contain two components for the buffered and bypassing access

$$R_M(c,x).wba,\ R_M(c,x).bpa,$$

and that for the sake of brevity we coerce R_M to a set of addresses in appropriate contexts by taking the union of these two components

$$R_M(c,x) = R_M(c,x).wba \cup R_M(c,x).bpa.$$

Lemma 62. *The read addresses only depend on the core configuration and the fetch results*

$$core_M(c,x) = core_N(c',x) \wedge fetch_M(c,x) = fetch_N(c',x) \rightarrow R_M(c,x) = R_N(c',x).$$

Proof. We distinguish between processor and write buffer steps.

$x \in \Sigma_{P,i}$**:** Straightforward

$$\begin{aligned} R_M(c,x) &= R(core_M(c,x),x,fetch_M(c,x)) \\ &= R(core_N(c',x),x,fetch_N(c',x)) \\ &= R_N(c',x). \end{aligned}$$

$x \notin \Sigma_{WB,i}$**:** Nothing is read

$$R_M(c,x) = (\emptyset,\emptyset) = R_N(c',x).$$

□

We also combine the guard conditions

$$\Phi_M(c,x) = \begin{cases} \Phi_P(core_M(c,x),x) & x \in \Sigma_{P,i} \\ 1 & \text{o.w.}, \end{cases}$$

$$\Upsilon_M(c,x) = \begin{cases} \Upsilon_P(core_M(c,x),x,fetch_M(c,x)) & x \in \Sigma_{P,i} \\ 1 & \text{o.w.}, \end{cases}$$

Note that for all functions defined so far, the machine-type index is irrelevant. We say that these functions are independent of the machine type.

2.5.2 Machine Modes

We introduce now the machine modes. We distinguish between two types of semantics. In low-level semantics (or weak memory semantics), a processor steps uses the buffer for forwarding during reads, and write buffer steps commit the oldest write in the buffer to memory. In high-level semantics (or strong memory semantics), processors ignore the write buffer during reads, and write buffer steps do not modify the memory.

Whether a unit is currently using low-level semantics depends at one level between the machine type

$$M \in \{\downarrow,\uparrow\},$$

and at another level on the memory mode of the unit making the step.

In the low-level machine, units are always using low-level semantics

$$LL_{i\downarrow}(c) = 1,$$

but in the high-level machine, a unit uses low-level semantics only when it is not in strong memory mode

$$LL_{i\uparrow}(c) = \neg SC_{iM}(c).$$

This crucial definition implies that we have three main cases for our semantics:

1. Low-level machine running any software. Write buffers are completely visible.
2. High-level machine running untrusted software on processor i. Write buffers of processor i are completely visible, processor uses semantics of low-level machine.
3. High-level machine running trusted software on processor i. Write buffers of processor i are invisible; all writes of this processor are executed immediately; write buffer steps of this processor have no effect on the memory (but drop an element from the ghost buffer).

A step is using low-level semantics if the unit making the step is in low-level semantics while making the step

$$LL_M(c,x) \equiv LL_{u_M(c,x)\ M}(c).$$

We obtain that a step in the high-level machine is done with low-level semantics iff the step is not done in strong memory mode.

Lemma 63.

$$LL_{\uparrow}(c,x) \iff \neg SC_{\uparrow}(c,x).$$

Proof.

$$LL_{\uparrow}(c,x) \equiv LL_{u_{\uparrow}(c,x)\ M}(c) \equiv \neg SC_{u_{\uparrow}(c,x)M}(c) \equiv \neg SC_{\uparrow}(c,x).$$

□

The forwarding memory system used in step c,x is that of the processor making the step (write buffers do not use forwarding)

$$fms_M(c,x) = \begin{cases} fms_{iM}(c) & x \in \Sigma_{P,i} \\ \emptyset & \text{o.w.} \end{cases}$$

In low-level semantics, we define the *memory view* in step c,x as the union of the results from memory for the bypassing portion and from the forwarding memory system for the buffered portion. In high-level semantics, we use the results from memory for both portions

$$v_M(c,x) = \begin{cases} c.m\big|_{R_M(c,x).bpa} \cup fms_M(c,x)\big|_{R_M(c,x).wba} & LL_M(c,x) \\ c.m\big|_{R_M(c,x)} & \neg LL_M(c,x). \end{cases}$$

Clearly the domain of this memory view is the read-set; we do not state a proof.

Lemma 64.

$$Dom(v_M(c,x)) = R_M(c,x).$$

We define in the straightforward way the *prepared writes in step* c,x

$$PW_M(c,x) = \begin{cases} PW(core_M(c,x),x,fetch_M(c,x),v_M(c,x)) & x \in \Sigma_{P,i} \\ (\emptyset,\emptyset) & \text{o.w.} \end{cases}$$

We use the projections $.wba$ and $.bpa$

$$PW_M(c,x) = (PW_M(c,x).wba, PW_M(c,x).bpa).$$

The write buffered in step c,x is the buffered portion of the prepared write

$$BW_M(c,x) = PW_M(c,x).wba.$$

We define a list operation used in step c,x for the write buffer

$$Op_{iM}(c,x) : X^* \times X \to X,$$

using the functions *push*,*pop*, and *noop*. Processor steps of the same unit that are preparing to buffer a write use the push operation; write buffer steps of the same unit use the pop operation. All other steps are no-ops.

$$Op_{iM}(c,x) = \begin{cases} push & x \in \Sigma_{P,i} \wedge BW_M(c,x) \neq \emptyset \\ pop & x \in \Sigma_{WB,i} \\ noop & \text{o.w.} \end{cases}$$

Clearly the new write given to the transition operator is relevant only for processor steps that are buffering a write.

Lemma 65.

$$\neg(x \in \Sigma_{P,i} \wedge BW_M(c,x) \neq \emptyset) \rightarrow Op_{iM}(c,x)(wb,w) = Op_{iM}(c,x)(wb,w')$$

Proof. We distinguish between write buffer and other steps.

$x \in \Sigma_{WB,i}$**:** By definition the operation is a pop, and the claim follows

$$Op_{iM}(c,x)(wb,w) = pop(wb,w) = tl(wb) = pop(wb,w') = Op_{iM}(c,x)(wb,w').$$

Otherwise: By definition the operation is a no-op, and the claim follows

$$Op_{iM}(c,x)(wb,w) = noop(wb,w) = wb = noop(wb,w') = Op_{iM}(c,x)(wb,w').$$

□

Whenever we know that no write is being buffered, we may omit the (sometimes long) term for the write. We use three dots … to denote that such a sub-expression is irrelevant.

If in case of a buffered write the new writes are the same, the new buffers are also the same. This allows us to reduce certain proofs to the case where something is being buffered.

Lemma 66.

$$((x \in \Sigma_{P,i} \wedge BW_M(c,x) \neq \emptyset) \rightarrow w = w') \rightarrow Op_{iM}(c,x)(wb,w) = Op_{iM}(c,x)(wb,w').$$

Proof. Either no write is being buffered or the buffered writes are the same.

$\neg(x \in \Sigma_{P,i} \wedge BW_M(c,x) \neq \emptyset)$**:** The claim is exactly Lemma 65.

$w = w'$**:** The claim trivially follows

$$Op_{iM}(c,x)(wb,w) = Op_{iM}(c,x)(wb,w').$$

□

We also define the effect of a step on memory. For this we use a function *writes*, which is a partial valuation of addresses and acts as an update for the memory

$$W_M(c,x) \in PVal(\mathcal{A}).$$

For processor steps in low-level semantics, the function simply executes the bypassing portion of the prepared writes. For processor steps in high-level semantics, it also executes the buffered portion of the prepared writes. For write-buffer steps in low-level semantics, the function commits the head of the write buffer in question; in high-level semantics, it does nothing.

$$W_M(c,x) = \begin{cases} PW_M(c,x).bpa & x \in \Sigma_{P,i} \wedge LL_M(c,x) \\ PW_M(c,x).bpa \cup PW_M(c,x).wba & x \in \Sigma_{P,i} \wedge \neg LL_M(c,x) \\ hd(c.wb(i)) & x \in \Sigma_{WB,i} \wedge LL_M(c,x) \\ \emptyset & x \in \Sigma_{WB,i} \wedge \neg LL_M(c,x). \end{cases}$$

Clearly the domain of the write is a subset of the accessible registers.

Lemma 67.

$$Dom(W_M(c,x)) \subseteq ACC_{u(x)}.$$

Proof. We distinguish between processor steps and write buffer steps.

$x \in \Sigma_{P,i}$**:** By definition, the executed write is subsumed by the union of bypassing and buffered writes

$$\begin{aligned} W_M(c,x) &= \begin{cases} PW_M(c,x).bpa & LL_M(c,x) \\ PW_M(c,x).bpa \cup PW_M(c,x).wba & \neg LL_M(c,x) \end{cases} \\ &\subseteq PW_M(c,x).bpa \cup PW_M(c,x).wba. \end{aligned}$$

Each of these are partial valuations of the accessible registers, i.e., for $X \in \{ bpa, wba \}$

$$\begin{aligned} PW_M(c,x).X &= PW(core_M(c,x),x,fetch_M(c,x),v_M(c,x)).X, \\ Dom(PW_M(c,x).X) &= Dom(PW(core_M(c,x),x,fetch_M(c,x),v_M(c,x)).X) \\ &\subseteq ACC_{u(x)}. \end{aligned}$$

and the claim follows

$$\begin{aligned} Dom(W_M(c,x)) &\subseteq Dom(PW_M(c,x).bpa \cup PW_M(c,x).wba) \\ &= Dom(PW_M(c,x).bpa) \cup Dom(PW_M(c,x).wba) \\ &\subseteq ACC_{u(x)} \cup ACC_{u(x)} \\ &= ACC_{u(x)}). \end{aligned}$$

$x \in \Sigma_{WB,i}$**:** The claim follows by definition

$$\begin{aligned} Dom(W_M(c,x)) &= Dom(hd(c.wb(i))) \\ &\subseteq BA \\ &\subseteq ACC_{u(x)}. \end{aligned}$$

□

2.5.3 Transition Operator

We now define the transition operators for the two machines. The memory in the next configuration is easily defined by applying the bypassing write as an update to the memory

$$c \blacksquare_M x.m = c.m \circledast W_M(c,x).$$

For the write buffer, we apply the operation of that step on the current buffer, using the buffered write as the update to the buffers

$$c \blacksquare_M x.wb(i) = Op_{iM}(c,x)(c.wb(i), BW_M(c,x)).$$

Write buffer steps do not change the perfect world forwarding memory system.

Lemma 68.

$$c.wb(i) \neq \varepsilon \wedge x \in \Sigma_{WB,i} \rightarrow pfms_{iM}(c) = pfms_{iM}(c \blacksquare_M x).$$

Proof. The buffer can be split into head and tail

$$c.wb(i) = hd(c.wb(i)) \circ tl(c.wb(i)).$$

The claim follows with the facts that the head of the write buffer is the write committed in the step, and the tail of the write buffer is the new write buffer

$$\begin{aligned} pfms_{iM}(c) &= c.m \circledast c.wb(i) \\ &= c.m \circledast hd(c.wb(i)) \circledast tl(c.wb(i)) \\ &= c.m \circledast W_M(c,x) \circledast c \centerdot_M x.wb(i) \\ &= c \centerdot_M x.m \circledast c \centerdot_M x.wb(i) \\ &= pfms_{iM}(c \centerdot_M x). \end{aligned}$$

□

We define the victims of a step as all processors who have their private regions touched by another processor

$$victims_M(c,x) = \{\, i \mid A_{PR,i} \hat{\cap} Dom(W_M(c,x)) \wedge x \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \,\}$$

Consider the case where the low-level machine simulates a high-level machine processor step in strong memory mode. In the low-level machine, the set of victims is determined by the bypassing writes, but in the high-level machine, also by the buffered writes. It would follow that the two machines can have different sets of victims. We will later require as a software condition that a processor in sequential mode will never buffer writes to processor registers, and therefore the set of victims will always be the same in safe computations.

We also require that the set of victims is actually defined by the contents of the memory region $A_{IPI}(x)$

$$\forall c, c'.c.m =_{A_{IPI}(x)} c'.m \rightarrow victims_M(c,x) = victims_M(c',x). \tag{2.3}$$

We now combine the drains for processor and write buffer steps in the *local drain condition*

$$\Delta_M(c,x) = \begin{cases} \Delta_P(core_M(c,x), fetch_M(c,x), c.wb(i), x) & x \in \Sigma_{P,i} \\ \Delta_{WB}(c.wb(i)) & x \in \Sigma_{WB,i}. \end{cases}$$

We also define the instance condition used in the step as the combination of feasibility and validity

$$I_M(c,x) = \Phi_M(c,x) \wedge \Upsilon_M(c,x).$$

The guard condition holds when the instance condition and the local drain condition holds

$$\Gamma_M(c,x) = I_M(c,x) \wedge \Delta_M(c,x).$$

In addition, we define a global drain condition for IPIs

$$\Delta_{IPIM}(c,x) = \Delta_{IPI}(victims_M(c,x), c.wb).$$

For now, the global drain condition is not that relevant, and we will require that schedules are safe even if they violate this guard condition. On the other hand, we will later require that all schedules provided by the hardware (for which we will prove that they simulate a high-level execution) satisfy the global drain condition.

This raises the question of how much efficiency in the software discipline is lost by not considering the drains that are caused by IPIs. We believe that there is no lost efficiency due the following reasons.

1. The time of IPI delivery is usually impossible to predict, and can thus not be used to justify assumptions about the write buffer being empty.

2. After an IPI is delivered, the victim processors will usually execute a jump to the interrupt service routine (JISR) step. In many architectures this step acts as a fence, at which point the state of the write buffer is known to the software discipline anyways.

3. In real-world architectures, IPIs do not drain the write buffer of the victims on delivery. In this case, the IPI can not be modeled[5] as a direct-to-core IPI, but has to use shared registers (e.g., in A_{DEV}) to signal the interrupt, and non-deterministic processor steps to sample the interrupt. In the terms of this thesis, such IPI delivery steps are not considered IPIs but rather normal shared writes, and the behavior of drains on IPIs is completely irrelevant.

On the other hand, the advantages for the proof effort one obtains by strengthening the assumptions in such a way are considerable, because one never has to worry about the effect of a reordering on the write buffers of other processors.

We are sometimes in a situation where we know that two configurations agree on the buffers of the unit making the step, but not much more. During steps of write buffers, this is enough to obtain that the steps agree on the local drain condition. We define a predicate that captures the local drain condition in write buffer steps and is always true in other steps

$$\Lambda_M(c,x) \equiv \forall i.x \notin \Sigma_{WB,i} \vee \Delta_M(c,x).$$

We call this predicate the write buffer drain condition.

We show that the condition indeed only depends on the write buffer.

Lemma 69.

$$c.wb(i) = c'.wb(i) \wedge i = u(x) \rightarrow \Lambda_M(c,x) = \Lambda_M(c',x).$$

Proof. We distinguish between processor and write buffer steps.

$x \in \Sigma_{P,i}$**:** The condition vacuously holds in both configurations, and the claim follows

$$\Lambda_M(c,x) = 1 = \Lambda_M(c',x).$$

$x \in \Sigma_{WB,i}$**:** The condition unfolds to be the drain condition of the write buffer, which indeed only depends on the write buffer

$$\Lambda_M(c,x) = \Delta_M(c,x)$$

[5]This is not a shortcoming of our model, but rather a consequence of the fact that direct-to-core IPIs cause sequentially inconsistent behaviors if they do not drain the buffer of victims on delivery. Our model is versatile enough to be able to handle both types of IPIs.

$$
\begin{aligned}
&= \Delta_{WB}(c.wb(i)) \\
&= \Delta_{WB}(c'.wb(i)) \\
&= \Delta_M(c', x) \\
&= \Lambda_M(c', x),
\end{aligned}
$$

which was the claim.

□

2.6 Computations

We define a computation in terms of a schedule, which is an infinite sequence of oracle inputs

$$s : \mathbb{N} \to \Sigma.$$

The computation induced by s is an infinite sequence of configurations, denoted by

$$c_M[s],$$

and we write the index in the superscript. Thus the configuration after t steps is denoted as follows

$$c_M[s]^t.$$

The sequence is defined recursively by applying the transition operator to each individual step input, starting (in both machines) with configuration c^0

$$
\begin{aligned}
c_M[s]^0 &= c^0, \\
c_M[s]^{t+1} &= c_M[s]^t \centerdot_M s(t).
\end{aligned}
$$

Note that this means that every schedule s induces two computations: the low-level computation

$$c_\downarrow[s],$$

in which all steps always use low-level semantics, and the high-level computation

$$c_\uparrow[s]$$

in which steps use low-level semantics if the unit making that step is currently in weak memory mode, and high-level semantics if the unit is strong memory mode.

In particular, we can run a schedule on both machines in parallel, and argue about what happens in the steps that are executed in parallel. We will later define a class of schedules for which the two computations execute the same steps. By that we mean that many functions, such as BW, F, etc., are the same in the low-level computation and high-level computation in each step. What we explicitly do not mean is that the two computations are in the same configuration after each step; this generally would only hold for the very first configuration c^0, which is the same in both schedules. The reason why this is not true in general is simple: when running a processor in strong memory mode on the two machines in parallel, the low-level machine will use low-level semantics and put writes into the buffer without applying these writes to the memory configuration. The high-level machine, on the other hand, will use high-level semantics

and execute the write immediately, thus changing the memory configuration. After one such step, the configurations are different. This problem does not occur for untrusted code, which runs in low-level semantics on both machines.

We define for functions $f_M(c)$ as *f at t in schedule s (in machine-type M)*

$$f_M[s](t) = f_M(c_M[s]^t),$$

and for functions of a step $f_M(c,x)$ we define *f during t in schedule s (in machine-type M)*[6] as f applied to the configuration after t steps and the t-th oracle step input

$$f_M[s](t) = f(c_M[s]^t, s(t)).$$

For components $c.x$ we similarly define *x at t in schedule s* as the component of the configuration after t steps

$$x_M[s]^t = c_M[s]^t.x.$$

We say a schedule s is *valid* if the guard condition is satisfied at all steps

$$\Gamma_M(s) \equiv \forall t.\Gamma_M[s](t).$$

We also introduce *validity until t*, which indicates that step t and earlier steps are valid:

$$\Gamma^t_M(s) \equiv \forall t' \leq t.\Gamma_M[s](t').$$

We will often consider schedules where steps up to some point are valid and the next step is at least feasible. In that case we say that the schedule is *semi-valid until t* and write

$$\Gamma\Phi^t_M(s) \equiv \Gamma^{t-1}_M(s) \wedge \Phi_M[s](t).$$

We similarly define that a schedule s is *IPI-valid* when it satisfies the global drain condition at all steps

$$\Delta^t_{IPIM}(s) \equiv \forall t' \leq t.\Delta_{IPIM}[s](t).$$

2.7 Reordering

We reorder with three schedule operators, shown in Fig. 2.2. We swap t and $t+1$ by

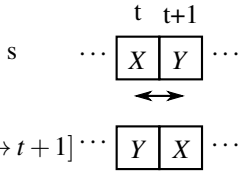
$$s[t \leftrightarrow t+1],$$

and move t to k by

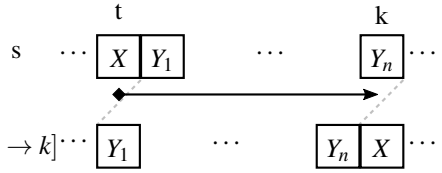
$$s[t \rightarrow k] \text{ or } s[k \leftarrow t],$$

if t is before or after k, resp. The latter are simply repeated application of the basic operator $[t \leftrightarrow t+1]$, which is defined as follows

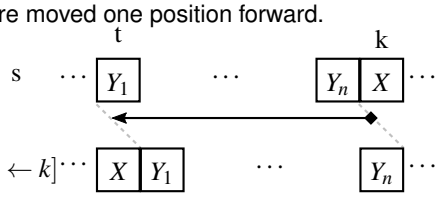
$$s[t \leftrightarrow t+1](t') = \begin{cases} s(t) & t' = t+1 \\ s(t+1) & t' = t \\ s(t') & \text{o.w.}, \end{cases}$$



(a) Swapping steps t and $t+1$. Other steps are unaffected.



(b) Delaying step t by moving it to position k. Intermediate steps are moved one position forward.



(c) Executing step k earlier by moving it to position t. Intermediate steps are delayed by one position.

Figure 2.2: The three reordering operators.

We use the variable Ω to range over instances of the basic operator

$$\Omega \in \{\, [t \leftrightarrow t+1] \mid t \in \mathbb{N} \,\}.$$

We can chain the basic operator several times

$$((s\Omega_1)\ldots)\Omega_{n-1}$$

and observe that operators are applied from left to right. We usually write these expressions without parentheses

$$s\Omega_1 \ldots \Omega_{n-1} = ((s\Omega_1)\ldots)\Omega_{n-1},$$

at which point it is tempting to treat the operators as a sequence O

$$O = \Omega_1 \circ \ldots \circ \Omega_{n-1}.$$

To make this formal, we define by recursion on O the reordered schedule sO, in which the basic operators are applied from left to right

$$s\varepsilon = s,\ s(\Omega \circ O) = (s\Omega)O.$$

[6]Strictly speaking, this creates a conflict due to our convention that the cartesian function $f(c,x)$ can always be used as a cascaded function $f(c)(x)$, and thus it is not clear which notation applies. We use the natural convention that the notation for steps has precedence over the notation for configurations in case both of them could be applied.

Note that the overloaded notation here is not problematic, since the singleton list containing Ω applies exactly Ω

$$s(\Omega \circ \varepsilon) = (s\Omega)\varepsilon = s\Omega.$$

We often call such sequences of basic operators O a *reordering*.

In what follows, we use two shorthands: 1) we do not write $\circ$ and 2) we flatten nested sequences of operators. Thus the following are all the same, where $O = \Omega_1 \circ \ldots \circ \Omega_{n-1}$ and $O' = \Omega'_1 \circ \ldots \circ \Omega'_{n'-1}$ are sequences of operators

$$\begin{aligned} OO' &= O \circ O' \\ &= (\Omega_1 \circ \ldots \circ \Omega_{n-1}) \circ (\Omega'_1 \circ \ldots \circ \Omega'_{n'-1}) \\ &= \Omega_1 \circ \ldots \circ \Omega_{n-1} \circ \Omega'_1 \circ \ldots \circ \Omega'_{n'-1} \\ &= \Omega_1 \ldots \Omega_{n-1} \Omega'_1 \ldots \Omega'_{n'-1}. \end{aligned}$$

This allows us to define the remaining two operators recursively on k simply as sequences of basic operators as follows.

To move step t to position t, no reordering has to take place.

$$[t \to t] = \varepsilon,$$

To move step t to position $k+1$, we move step t to position k, then swap k and $k+1$.

$$[t \to k+1] = [t \to k][k \leftrightarrow k+1],$$

To move to step t from position t, no reordering has to take place.

$$[t \leftarrow t] = \varepsilon,$$

To move to step t from position $k+1$, we swap k and $k+1$, then move to step t from position k.

$$[t \leftarrow k+1] = [k \leftrightarrow k+1][t \leftarrow k].$$

In contexts where there is exactly one original schedule s and reordered schedules $s_1, \ldots, s_n$, where s_i is obtained from s by applying a sequence of operators O_i, e.g., of the form

$$O_i = [t \to u][v \leftarrow k] \ldots,$$

so that we have

$$s_i = sO_i,$$

we may apply the operators directly to f and obtain f *at* t *after* O_i

$$f_M O_i(t) = f_M[sO_i](t).$$

Note that since there is only one original schedule, this may not cause ambiguity.

For example, when we move t to k in s, we obtain a new schedule $s[t \to k]$ ("s with t moved to k"). In that case we use

$$W_M[t \to k](k) = W_M[s[t \to k]](k),$$

$$R_M[t \to k](k) = R_M[s[t \to k]](k),$$

to denote the writes or read-set in step k after moving t to k. If we move t to $k+1$ we may use

$$\begin{aligned} W_M[t \to k+1](k+1) &= W_M[s[t \to k+1]](k+1) \\ &= W_M[s[t \to k][k \leftrightarrow k+1]](k+1) \\ &= W_M[t \to k][k \leftrightarrow k+1](k+1) \end{aligned}$$

to denote the writes in step $k+1$ after moving t to $k+1$, or equivalently, after first moving t to k and then k to $k+1$. We use a similar notation for components or the configuration $x \in \{ c, m, wb \}$

$$x_M O_i^t = x_M[sO_i]^t,$$

such as the memory

$$m_M[t \to k]^k = m_M[s[t \to k]]^k$$

or write buffers

$$wb_M[t \to k]^k = wb_M[s[t \to k]]^k.$$

In the special case where the sequence of operators is empty $O_i = \varepsilon$, the "reordered" schedule s_i is s. This allows us to make the original sequence implicit:

$$f_M(t) = f_M[s](t),\ x_M^t = x_M[s]^t.$$

Note that after reordering t to k, the step at position k is the step that was before at position t

$$s[t \to k](k) = s(t).$$

Similarly, steps t' between t and k are moved by one step

$$t' \in [t : k) \to s[t \to k](t') = s(t'+1),$$

while other steps are not moved at all

$$t' \notin [t : k] \to s[t \to k](t') = s(t').$$

Consequently we would expect that other functions, such as read-set and writes, or components such as the memory, also move similarly with the step:

$$\begin{aligned} W_M[t \to k](k) &\stackrel{?}{=} W_M(t), \\ t' \in [t : k) \to W_M[t \to k](t') &\stackrel{?}{=} W_M(t'+1), \\ t' \notin [t : k] \to W_M[t \to k](t') &\stackrel{?}{=} W_M(t'). \end{aligned}$$

However, this is usually not the case: when moving the oracle input, one changes the order of reads and writes, and therefore one changes the configuration in which the step is executed. If the function of the step depends on registers which are changed during the reordering, the value will obviously also change. Nevertheless, we will often only reorder in situations where we know at least for a portion of the functions f_M and components x that their values are unchanged, e.g.,

$$f_M[t \to k](k) = f_M(t) \wedge x_M[t \to k]^k = x^t.$$

Only during steps $t' < t$ that occurred before the reordered portion we can always show that nothing is changed.

Lemma 70. *Only portions of the schedule after the reordering are changed.*

1. $s[0:t-1] = s[t \leftrightarrow t+1][0:t-1]$.
2. $s[0:t-1] = s[t \to k][0:t-1]$.
3. $s[0:t-1] = s[t \leftarrow k][0:t-1]$.

Proof. The first one is by definition. The second and third are proven by induction on k, starting with t, and with s generalized (i.e., the induction hypothesis quantifies over all s). The base cases are trivial. In the inductive step from k to $k+1$, we have for the second claim with the first claim

$$\begin{aligned} s[t \to k+1][0:t-1] &= s[t \to k][k \leftrightarrow k+1][0:t-1] \\ &= s[t \to k][0:k-1][0:t-1] \\ &= s[t \to k][0:t-1]. \end{aligned}$$

The last equality holds because k is at least t.

For the third claim we have

$$\begin{aligned} s[t \leftarrow k+1][0:t-1] &= s[k \leftrightarrow k+1][t \leftarrow k][0:t-1] \\ &= s[k \leftrightarrow k+1][0:t-1] \\ &= s[0:k-1][0:t-1] \\ &= s[0:t-1]. \end{aligned}$$

The last equality holds because k is at least t. □

Lemma 71. *Steps which occur before the reordered step are unaffected by the reordering. Let $t' < t$ be a step occurring before the reordering.*

1. $f_M(t') = f_M[t \leftrightarrow t+1](t')$.
2. $f_M(t') = f_M[t \to k](t')$.
3. $f_M(t') = f_M[t \leftarrow k](t')$.

Proof. Let O be the singleton sequence of operators applied to the schedule

$$O \in \{\, [t \leftrightarrow t+1], [t \to k], [t \leftarrow k] \,\}.$$

We first show with Lemma 70 that $c^{t'}$ is unchanged

$$\begin{aligned} c_M[s]^{t'} &= c \centerdot_M s(0) \centerdot_M \ldots \centerdot_M s(t'-1) \\ &= c \centerdot_M sO(0) \centerdot_M \ldots \centerdot_M sO(t'-1) \\ &= c[sO]^{t'}. \end{aligned}$$

We obtain now for f with the equality of $c^{t'}$ and Lemma 70

$$\begin{aligned} f_M(t') &= f_M[s](t') \\ &= f(c_M[s]^{t'}, s(t')) \\ &= f(c_M[sO]^{t'}, sO(t')) && \text{L 70} \\ &= f_M O(t'). \end{aligned}$$

□

The notation also gives us recursive characterizations for components.

Lemma 72. *Let O be some sequence of operators of schedules. All of the following hold.*

1. $c_M O^{t+1} = c_M O^t \blacksquare_M sO(t)$.
2. $wb_M O^{t+1}(i) = Op_{iM} O(t)(wb_M O^t(i), BW_M O(t))$.
3. $m_M O^{t+1} = m_M O^t \circledast W_M O(t)$.

Proof.

$$\begin{aligned}
c_M O^{t+1} &= c_M[sO]^{t+1} \\
&= c_M[sO]^t \blacksquare_M sO(t) \\
&= c_M O^t \blacksquare_M sO(t), \\
wb_M O^{t+1}(i) &= c_M[sO]^{t+1}.wb(i) \\
&= c_M[sO]^t \blacksquare_M sO(t).wb(i) \\
&= Op_{iM}(c_M[sO]^t, sO(t))(c_M[sO]^t.wb(i), BW_M(c_M[sO]^t, sO(t))) \\
&= Op_{iM}[sO](t)(wb_M[sO]^t(i), BW_M[sO](t)) \\
&= Op_{iM} O(t)(wb_M O^t(i), BW_M O(t)), \\
m_M O^{t+1} &= c_M[sO]^{t+1}.m \\
&= c_M[sO]^t \blacksquare_M sO(t).m \\
&= c_M[sO]^t.m \circledast W_M(c_M[sO]^t, sO(t)) \\
&= m_M[sO]^t \circledast W_M[sO]^t \\
&= m_M O^t \circledast W_M O^t.
\end{aligned}$$

□

We wish to track steps as they get moved by a reordering. For each basic reordering operator Ω we define a function $mv\Omega(l)$ that tracks exactly how step l is moved

$$mv[t \leftrightarrow t+1](l) = \begin{cases} l & l \notin \{t, t+1\} \\ t+1 & l = t \\ t & l = t+1 \end{cases}$$

That this really tracks the movement of each step is shown in the following lemma.

Lemma 73.

$$s(l) = s\Omega(mv\Omega(l)).$$

Proof. We first split along the cases of $mv\Omega(l)$ and then of $s\Omega(\ldots)$.

$$\begin{aligned}
s[t \leftrightarrow t+1](mv[t \leftrightarrow t+1](l)) &= \begin{cases} s[t \leftrightarrow t+1](l) & l \notin \{t, t+1\} \\ s[t \leftrightarrow t+1](t+1) & l = t \\ s[t \leftrightarrow t+1](t) & l = t+1 \end{cases} \\
&= \begin{cases} s(l) & l \notin \{t, t+1\} \\ s(t) & l = t \\ s(t+1) & l = t+1 \end{cases}
\end{aligned}$$

$$= s(l).$$

□

This function is injective. We do not show the proof.

Lemma 74.

$$mv\Omega(l) = mv\Omega(l') \to l = l'.$$

We extend mv to sequences of operators O by applying the operators in the same order they are applied on the schedule, i.e., from left to right

$$mv\varepsilon(l) = l,\ mv\Omega O(l) = mvO(mv\Omega(l)).$$

Again the overloaded notation is not problematic since tracking the movement of the singleton list containing Ω exactly tracks the movement of Ω

$$mv\Omega\varepsilon(l) = mv\varepsilon(mv\Omega(l)) = mv\Omega(l).$$

For example, a single (long distance) operator is represented by a sequence of swaps

$$mv[t \to k](l) = mv[t \leftrightarrow t+1] \dots [k-1 \leftrightarrow k](l)$$
$$mv[t \leftarrow k](l) = mv[k-1 \leftrightarrow k] \dots [t \leftrightarrow t+1](l).$$

An easy lemma shows that for sequences of operators the reordering also works

Lemma 75.

$$s(l) = sO(mvO(l)).$$

Proof. By induction on O, with s and l generalized. The base case is trivial. The inductive step $O \to \Omega O$ is solved easily with Lemma 73 and the induction hypothesis for $s := s\Omega$ and $l := mv\Omega(l)$

$$\begin{aligned} s(l) &= s\Omega(mv\Omega(l)) && \text{L 73} \\ &= s\Omega O(mvO(mv\Omega(l))). && \text{IH} \\ &= s\Omega O(mv\Omega O(l)). \end{aligned}$$

□

We show that our shorthands for sequences of operators (dropping $\circ$ and flattening nested sequences) also makes sense for this function with the following lemma, which shows that it does not matter where we split between operators.

Lemma 76.

$$mvOO'(l) = mvO'(mvO(l)).$$

Proof. By induction on O with l generalized. The base case is trivial. The inductive step $O \to \Omega O$ is straightforward with the induction hypothesis applied for $l := mv\Omega(l)$

$$\begin{aligned} mv\Omega OO'(l) &= mvOO'(mv\Omega(l)) \\ &= mvO'(mvO(mv\Omega(l))) && \text{IH} \\ &= mvO'(mv\Omega O(l)). \end{aligned}$$

□

An explicit form for these reorderings is given by the following lemma.

Lemma 77.

$$mv[t \to k](l) = \begin{cases} l & l \notin [t:k] \\ l-1 & l \in (t:k] \\ k & l = t \end{cases}$$

$$mv[t \leftarrow k](l) = \begin{cases} l & l \notin [t:k] \\ l+1 & l \in [t:k) \\ t & l = k \end{cases}$$

Proof. We only show the proof for $[t \to k]$. The proof is by induction on k. The base case is trivial. In the inductive step $k \to k+1$, we obtain with Lemma 76 that we can first move t to position k

$$\begin{aligned} mv[t \to k+1](l) &= mv[t \to k][k \leftrightarrow k+1](l) \\ &= mv[k \leftrightarrow k+1](mv[t \to k](l)) \text{L 76} \end{aligned}$$

We distinguish between four cases (corresponding to 1) not moved at all, 2) not moved in the recursion but moved in the recursive step, 3) moved forward by the recursion, and 4) step t which is moved back).

$l \notin [t:k+1]$**:** In this case l is also neither in the interval $[t:k]$ nor in the set $\{k, k+1\}$ and the claim follows with the induction hypothesis

$$\begin{aligned} mv[t \to k+1](l) &= mv[k \leftrightarrow k+1](mv[t \to k](l)) \\ &= mv[k \leftrightarrow k+1](l) && \text{IH} \\ &= l. \end{aligned}$$

$l = k+1$**:** By the induction hypothesis, step l is not moved when t is moved to k, and it is moved to $k = l-1$ by the swap

$$\begin{aligned} mv[t \to k+1](l) &= mv[k \leftrightarrow k+1](mv[t \to k](l)) \\ &= mv[k \leftrightarrow k+1](l) && \text{IH} \\ &= mv[k \leftrightarrow k+1](k+1) \\ &= k \\ &= l-1, \end{aligned}$$

which is the claim.

$l \in (t:k]$**:** In this case $l-1$ is before k and thus not in the set $\{k, k+1\}$ and the claim follows with the induction hypothesis

$$\begin{aligned} mv[t \to k+1](l) &= mv[k \leftrightarrow k+1](mv[t \to k](l)) \\ &= mv[k \leftrightarrow k+1](l-1) && \text{IH} \\ &= l-1. \end{aligned}$$

$l = t$**:** By the induction hypothesis step l is first moved to position k, and then moved again by the swap

$$mv[t \to k+1](l) = mv[k \leftrightarrow k+1](mv[t \to k](l))$$

$$= mv[k \leftrightarrow k+1](k) \qquad \text{IH}$$
$$= k+1,$$

which is the claim.

□

Note that functions $mv[t \to k]$ and $m[t \leftarrow k]$ are injective. All reorderings are.

Lemma 78.

$$mvO(l) = mvO(l') \to l = l'.$$

Proof. Since mvO is just the chained application of injective functions $mv\Omega$, it is also injective. □

We now look at how to undo a reordering. We will define for each reordering O an inverse O^{-1} which undoes the reordering such that $sOO^{-1} = s$ (the inversion operator $\cdot^{-1}$ binds stronger than concatenation).

We begin by defining an inverse Ω^{-1} for the basic operators. Unsurprisingly, to undo a swap of t and $t+1$, one simply swaps t and $t+1$ again. Therefore each basic operator is its own inverse.

$$\Omega^{-1} = \Omega.$$

One easily sees that applying a reordering operator and then its inverse does not actually move any steps.

Lemma 79.

$$mv\Omega\Omega^{-1}(l) = l.$$

Inverting twice yields the same operator.

Lemma 80.

$$\Omega = (\Omega^{-1})^{-1}.$$

To undo a reordering O consisting of several chained basic operators, we undo each basic operator but in reverse order.

$$\varepsilon^{-1} = \varepsilon,\ (\Omega O)^{-1} = O^{-1}\Omega^{-1}.$$

Again the overloaded notation is not problematic since the inverse of the singleton list containing Ω is exactly the singleton list containing the inverse of Ω

$$(\Omega\varepsilon)^{-1} = \varepsilon^{-1}\Omega^{-1} = \varepsilon\Omega^{-1} = \Omega^{-1}.$$

Here too the nesting and $\circ$ do not matter and we can split between operators anywhere we want. We do not show the proof.

Lemma 81.

$$(OO')^{-1} = O'^{-1}O^{-1}.$$

One now easily obtains that the inverse of the inverse of a reordering is the reordering itself

Lemma 82.

$$(O^{-1})^{-1} = O.$$

Proof. By induction on O. The base case is trivial. The inductive step $O \to \Omega O$ is straightforward with Lemmas 80 and 81

$$\begin{aligned}((\Omega O)^{-1})^{-1} &= (O^{-1}\Omega^{-1})^{-1} \\ &= (\Omega^{-1})^{-1}O^{-1})^{-1} && \text{L 81} \\ &= \Omega O && \text{IH, L 80}\end{aligned}$$

□

Almost by definition one now obtains that the reorderings $[t \to k]$ and $[t \leftarrow k]$ undo each other.

Lemma 83.

$$\begin{aligned}[t \to k] &= [t \leftarrow k]^{-1}, \\ [t \leftarrow k] &= [t \to k]^{-1}.\end{aligned}$$

Proof. We show first the first claim by induction on k, the base case is trivial. The inductive step $k \to k+1$ is straightforward

$$\begin{aligned}[t \to k+1] &= [t \to k][k \leftrightarrow k+1] \\ &= [t \leftarrow k]^{-1}[k \leftrightarrow k+1] && \text{IH} \\ &= [t \leftarrow k]^{-1}[k \leftrightarrow k+1]^{-1} \\ &= ([k \leftrightarrow k+1][t \leftarrow k])^{-1} \\ &= [t \leftarrow k+1]^{-1}.\end{aligned}$$

which proves the first claim.

The second claim follows by inverting both sides of the first claim and applying Lemma 82 to cancel the double inversion

$$\begin{aligned}[t \to k]^{-1} &= ([t \leftarrow k]^{-1})^{-1} && \text{Claim 1} \\ &= [t \leftarrow k]. && \text{L 82}\end{aligned}$$

□

Applying a reordering and then its inverse moves no steps. For example, moving t to step k and then from k to t (or vice versa) does not move any steps at all (cf. Fig. 2.3).

Lemma 84.

$$mvOO^{-1}(l) = l.$$

Proof. By induction on O with l generalized. The base case is trivial. The inductive step $O \to \Omega O$ is straightforward with the induction hypothesis for $l := mv\Omega(l)$ and Lemmas 76 and 79

$$\begin{aligned}mv\Omega O(\Omega O)^{-1}(l) &= mv\Omega OO^{-1}\Omega^{-1}(l) \\ &= mv\Omega^{-1}(mvOO^{-1}(mv\Omega(l))) && \text{mv split} \\ &= mv\Omega^{-1}(mv\Omega(l)) && \text{IH} \\ &= mv\Omega\Omega^{-1}(l) \\ &= l. && \text{L 79}\end{aligned}$$

□

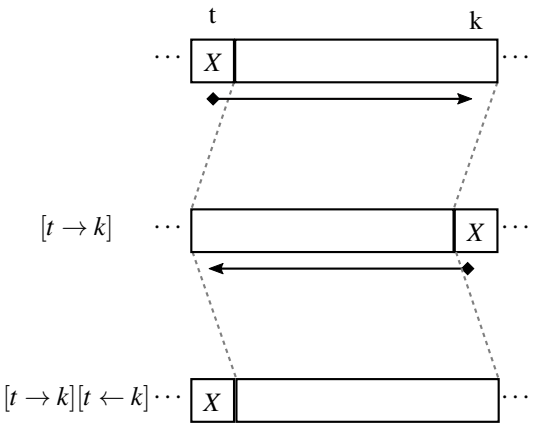


Figure 2.3: Moving step t to position k and then to position t from position k has no effect.

We can now give an alternative definition for $sO(l)$: rather than applying the reordering operators in O to s, we can find the oracle input used by the original position of step l in s by undoing the reordering.

Lemma 85.

$$sO(l) = s(mvO^{-1}(l)).$$

Proof. Straightforward with Lemmas 75, 76, 82 and 84

$$\begin{aligned} sO(l) &= sO(mvO^{-1}(O^{-1})^{-1}(l)) && \text{L 84} \\ &= sO(mvO^{-1}O(l)) && \text{L 82} \\ &= sO(mvO(mvO^{-1}(l)) && \text{L 76} \\ &= s(mvO^{-1}(l)). && \text{L 75} \end{aligned}$$

□

For example, to obtain the oracle input of step l after moving k to position t, we can simply backtrack and see where step l was before the reordering by moving t back to position k

$$s[t \leftarrow k](l) = s(mv[t \to k](l)).$$

Furthermore, applying a reordering O and its inverse has no effect

Lemma 86.

$$sOO^{-1}(l) = s(l).$$

Proof. Straightforward with Lemmas 75 and 84

$$\begin{aligned} sOO^{-1}(l) &= sOO^{-1}(mvOO^{-1}(l)) && \text{L 84} \\ &= s(l) && \text{L 75} \end{aligned}$$

□

Therefore, when comparing schedules s and sO, we can choose which schedule to consider the original schedule and which to consider the reordered schedule: either 1) s is the original schedule, and sO is the schedule with reordering O, or 2) schedule sO is the original schedule and $s = sOO^{-1}$ is the reordered schedule with reordering O^{-1}. We will use this observation several times.

2.8 Managing Layers of Notation

We have now introduced several layers of notation for lifting functions defined in terms of steps c,x to steps t in a computation. We wish to do the same for lemmas that range over steps c,x without having to tediously fold and unfold the notation for computations. Consider for example Lemma 67, which says that the domain of the writes only includes accessible registers

$$Dom(W_M(c,x)) \subseteq ACC_{u(x)}.$$

In order to obtain the analogous result for steps in a computation

$$Dom(W_M(t)) \subseteq ACC_{u_M(t)},$$

we have to unfold two layers of notation on the left, apply Lemma 67, and then fold the layers of notation again:

$$\begin{aligned} Dom(W_M(t)) &= Dom(W_M[s](t)) \\ &= Dom(W_M(c_M[s]^t, s(t))) \\ &\subseteq ACC_{u(s(t))} && \text{L 67} \\ &= ACC_{u_M[s](t)} \\ &= ACC_{u_M(t)}. \end{aligned}$$

As the following Lemma shows, we can always convert such statements.

Lemma 87. *Let f_{iM} be a finite family of functions with $i \in [1:n]$. Consider some predicate P over steps which is defined in terms of the functions f_{iM}, and which holds for all steps*

$$\forall c,x.P(c,x,f_{1M}(c,x),\cdots,f_{nM}(c,x)).$$

We immediately obtain that it also holds for all steps in computations

$$\forall s,t.P(c_M^t, s(t), f_{1M}(t),\cdots,f_{nM}(t)).$$

Proof. Unfolding the notation we reduce the claim to the following

$$\begin{aligned} & P(c_M^t, s(t), f_{1M}(t),\cdots,f_{nM}(t)) \\ \iff & P(c_M[s]^t, s(t), f_{1M}[s](t),\cdots,f_{nM}[s](t)) \\ \iff & P(c_M[s]^t, s(t), f_{1M}(c_M[s]^t, s(t)),\cdots,f_{nM}(c_M[s]^t, s(t))). \end{aligned}$$

The assumption solves this goal. □

We will thus implicitly convert statements about configurations c and steps c,x to statements about configurations at t and steps t in a schedule.

2.9 Inputs and Outputs

We now look at addresses read in a step and addresses modified in a step. We define first $in_M(c,x) \subseteq \mathcal{A}$ as the inputs of step c,x by

$$in_M(c,x) = C_M(c,x) \cup F_M(c,x) \cup R_M(c,x).$$

We define the *write set* as the domain of the write

$$WS_M(c,x) = Dom(W_M(c,x)).$$

We define the *device inputs* as the inputs taken during the device modification by

$$devin_M(c,x) = dc(WS_M(c,x)).$$

The state of device inputs *devin* only matters for the configuration of a device after a write, but not for the processor executing that write. Only the state of addresses in C,F,R matter, which is why we do not include *devin* in the definition of inputs *in*. One can split steps that modify devices into two substeps:

Processor step, depends on *in*, modifies A_{MEM}, $A_{PR,i}$.

Device step, triggered by processor step, depends on *devin*, modifies A_{DEV}.

This corresponds to the notion of passive transitions in [Sch13]. Note that the device step also depends on the value of inputs, because those determine the write that triggers the device step.

For write buffer steps, only the head element matters, and the rest can be removed or added freely. We define a relation between steps that use the same oracle input which holds during a write buffer step when the heads of the write buffers are the same, and for processor steps when the write buffer of the second step is a suffix of the write buffer of the first step and the elements in the prefix do not matter to the step.

We say that *wb subsumes wb′ when stepped with x on inputs A* and write

$$bufS(x,A,wb,wb') \equiv \begin{cases} hd(wb) = hd(wb') & x \in \Sigma_{WB,i} \\ \exists w.\ wb = w \circ wb' \wedge \neg hit(A,w) & x \in \Sigma_{P,i}. \end{cases}$$

This relation is transitive. We do not show the proof.

Lemma 88.

$$bufS(x,A,wb,wb') \wedge bufS(x,A,wb',wb'') \rightarrow bufS(x,A,wb,wb'').$$

This relation is also reflexive.

Lemma 89.

$$wb = wb' \rightarrow bufS(x,A,wb,wb').$$

Proof. By case distinction on x.

$x \in \Sigma_{P,i}$**:** With $w := \varepsilon$ the claim reduces to showing that the buffers are the same and that there is no hit in the empty buffer

$$\begin{aligned} & wb = \varepsilon \circ wb' \wedge \neg hit(A,\varepsilon) \\ \iff & wb = wb' \wedge \neg hit(A,\varepsilon). \end{aligned}$$

The former is an assumption and the latter trivially holds.

$x \in \Sigma_{WB,i}$**:** The buffers are the same and the claim follows

$$hd(wb) = hd(wb').$$

□

Note that forwarding is used only in low-level machine semantics. We define the *forwarding inputs* as either the inputs when the machine is using low-level semantics, or as the empty set

$$fin_M(c,x) = \begin{cases} in_M(c,x) & LL_M(c,x) \\ \emptyset & \text{o.w.} \end{cases}$$

We extend the definition to steps by using the forwarding inputs of the step and the buffers of the unit making the step

$$bufS_M(x,c,c') = bufS(x, fin_M(c,x), c.wb(u_M(c,x)), c'.wb(u_M(c,x))).$$

In this case we say that *the buffers of step c,x subsume those of step c',x.* In case of steps in a schedule, we allow ourselves the freedom to use step numbers, as such in the following example

$$bufS_M(s(t), c'_M, c_M[t \to k]^k)$$

we might say that the buffers of step t subsume those of step k after moving t to k.

We can extend Lemma 89 to steps. We do not show the proof.

Lemma 90.

$$c.wb =_{u_M(c,x)} c.wb' \to bufS_M(x,c,c').$$

We define a powerful simulation relation that couples the inputs of two configurations that are stepped with the same oracle input x, possibly in different machines M and N. For processor steps $x \in \Sigma_{P,i}$, we couple the value of core and fetched registers

$$c =^x_{M,N} c' \equiv c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge \Delta_M(c,x) \equiv \Delta_N(c',x).$$

For write buffer steps $x \in \Sigma_{WB,i}$, we do a little more. We require that the configurations agree on the core registers (i.e., the memory mode), as well as the head of the write buffers

$$c =^x_{M,N} c' \equiv c.m =_{C_M(c,x)} c'.m \wedge hd(c.wb(i)) = hd(c'.wb(i)).$$

If this relation holds between steps c and c', we say that the configurations *strongly agree when stepped with x.* If we are using reorderings and one of the configurations is at k in the reordered schedule, which is the new position of step t in the previous schedule which happens to be the other configuration, we say *step t can still be executed at k.*

Intuitively speaking, when c and c' strongly agree during step x, the steps c,x and c',x see the same values and therefore have the same effect.

We show that all steps that strongly agree also agree on the value of core and fetched values, on the memory view, and on the drain condition.

Lemma 91.

$$c =^x_{M,N} c' \to c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge \Delta_M(c,x) \equiv \Delta_N(c',x).$$

Proof. We distinguish between processor and write buffer steps.

$x \in \Sigma_{P,i}$**:** The claim holds by definition of strong agreement.

$x \in \Sigma_{WB,i}$**:** By assumption the configurations agree on the value of core registers

$$c.m =_{C_M(c,x)} c'.m,$$

Nothing is being fetched

$$F_M(c,x) = \emptyset = F_N(c',x).$$

and thus the configurations also vacuously agree on the value of fetched registers

$$c.m =_{F_M(c,x)} c'.m.$$

Similarly, nothing is being read

$$R_M(c,x) = \emptyset = R_N(c',x)$$

and thus the configurations vacuously agree on the memory view

$$\begin{aligned} v_M(c,x) &= \begin{cases} c.m\big|_\emptyset \cup fms_M(c,x)\big|_\emptyset & LL_M(c,x) \\ c.m\big|_\emptyset & \neg LL_M(c,x) \end{cases} \\ &= \emptyset \\ &= \begin{cases} c'.m\big|_\emptyset \cup fms_N(c',x)\big|_\emptyset & LL_N(c',x) \\ c'.m\big|_\emptyset & \neg LL_N(c',x) \end{cases} \\ &= v_N(c',x). \end{aligned}$$

It remains to be shown that the steps agree on the drain condition, which follows with the fact that they agree on the head element of the write buffer

$$\begin{aligned} & \Delta_M(c,x) \equiv \Delta_N(c',x) \\ \iff & \Delta_{WB}(c.wb(i)) \equiv \Delta_{WB}(c'.wb(i)) \\ \iff & c.wb(i) \neq \varepsilon \equiv c'.wb(i) \neq \varepsilon \\ \iff & hd(c.wb(i)) \neq \bot \equiv hd(c'.wb(i)) \neq \bot \\ \iff & hd(c.wb(i)) \neq \bot \equiv hd(c.wb(i)) \neq \bot. \end{aligned}$$

□

When we use the strong agreement for steps of the same machine-type, we omit the second index for the sake of brevity

$$c =^x_M c' \equiv c =^x_{M,M} c'.$$

We define $out_M(c,x)$ as the outputs of step c,x by the inclusive device closure of the write set

$$out_M(c,x) = idc(WS_M(c,x)).$$

In the high-level machine, a processor step in strong memory mode executes the buffered writes.

Lemma 92.

$$x \in \Sigma_{P,i} \wedge SC_{\uparrow}(c,x) \rightarrow Dom(BW_{\uparrow}(c,x)) \subseteq out_{\uparrow}(c,x).$$

Proof. The step is not using low-level machine semantics

$$\neg LL_{\uparrow}(c,x),$$

and therefore the processor is executing bypassing and buffered writes

$$W_{\uparrow}(c,x) = PW_{\uparrow}(c,x).bpa \cup PW_{\uparrow}(c,x).wba.$$

The claim follows

$$\begin{aligned} Dom(BW_{\uparrow}(c,x)) &= Dom(PW_{\uparrow}(c,x).wba) \\ &\subseteq Dom(W_{\uparrow}(c,x)) \\ &= Dom(PW_{\uparrow}(c,x).bpa \cup PW_{\uparrow}(c,x).wba) \\ &\subseteq idc(Dom(W_{\uparrow}(c,x))) \\ &= out_{\uparrow}(c,x). \end{aligned}$$

□

We also show that inputs and outputs are indeed confined to the set of accessible registers.

Lemma 93.

$$in_M(c,x), out_M(c,x) \subseteq ACC_{u_N(c,x)}.$$

Proof. The unit making step c,x in machine N is by definition the unit to which the oracle input x belongs

$$u_N(c,x) = u(x),$$

which reduces the claim to the following

$$in_M(c,x), out_M(c,x) \overset{!}{\subseteq} ACC_{u(x)}.$$

For the outputs, we know that the device closure are device registers which are accessible, and it suffices to show that the write-set is contained

$$\begin{aligned} & out_M(c,x) \subseteq ACC_{u(x)} \\ \iff & idc(WS_M(c,x)) \subseteq ACC_{u(x)} \\ \iff & WS_M(c,x) \subseteq ACC_{u(x)}. \qquad \text{L 16, 4} \end{aligned}$$

The write-set is just the domain of the writes, and the first claim is Lemma 67

$$WS_M(c,x) = Dom(W_M(c,x)) \subseteq ACC_{u(x)}.$$

For the inputs, we have that each of the individual parts C,F,R are restricted to accessible registers; for the core registers this is because they are core registers, which are accessible

$$C_M(c,x) \subseteq A_{PR,u(x)} \subseteq ACC_{u(x)},$$

for the fetched and read registers this is by definition

$$F_M(c,x), R_M(c,x) \subseteq ACC_{u(x)}.$$

Therefore their union is also contained in the set of accessible registers

$$in_M(c,x) = C_M(c,x) \cup F_M(c,x) \cup R_M(c,x) \subseteq ACC_{u(x)}.$$

□

Normal processor registers are not modified by other processors, which means that a modification of processor registers of another unit always means that the interrupt registers are being modified.

Lemma 94.

$$out_M(c,x) \dot{\cap} A_{PR,j} \wedge i \neq j \rightarrow out_M(c,x) \dot{\cap} A_{IPR}.$$

Proof. By Lemma 93, step t only modifies accessible registers

$$out_M(c,x) \subseteq ACC_i$$

which by definition do not include normal processor registers of unit j

$$A_{NPR,j} \not\dot{\cap} ACC_i.$$

Therefore step t does not modify normal processor registers of unit j

$$out_M(c,x) \not\dot{\cap} A_{NPR,j},$$

leaving only the interrupt registers

$$out_M(c,x) \dot{\cap} A_{IPR,j}.$$

The claim follows

$$out_M(c,x) \dot{\cap} A_{IPR}.$$

□

Different units do not modify each other's write buffers.

Lemma 95.

$$u_M(c,x) \neq i \rightarrow Op_{iM}(c,x)(wb,w) = wb.$$

Proof. Since the step is not made by unit i, it is neither a step of processor i nor its write buffer

$$\begin{aligned} & u_M(c,x) \neq i \\ \Longrightarrow\ & u(x) \neq i \\ \Longrightarrow\ & x \notin \Sigma_{P,i} \cup \Sigma_{WB,i}, \end{aligned}$$

and the claim follows

$$wb = noop(wb,w) = Op_{iM}(c,x)(wb,w).$$

□

Lemma 96.

$$u_M(c,x) \neq i \rightarrow c.wb =_i c \blacksquare_M x.wb.$$

Proof. By Lemma 95 a step by a different unit no effect on the write buffer

$$c \blacksquare_M x.wb(i) = Op_{iM}(c,x)(c.wb(i),\ldots) = c.wb(i).$$

□

Therefore the write never matters in write buffer transitions made by different units

Lemma 97.

$$u_M(c,x) \neq i \rightarrow Op_{iM}(c,x)(wb,w) = Op_{iM}(c',x)(wb,w').$$

Proof. Note that the unit making step c',x is the same as the unit making step c,x, and thus not unit i

$$u_M(c',x) = u(x) = u_M(c,x) \neq i.$$

The claim is now just twice Lemma 95

$$Op_{iM}(c,x)(wb,w) = wb = Op_{iM}(c',x)(wb,w').$$

□

The set of victims can now be restated as the set of those processors which have their local registers modified by other units

Lemma 98.

$$victims_M(c,x) = \{\, i \mid A_{PR,i} \dot{\cap} out_M(c,x) \wedge u_M(c,x) \neq i \,\}.$$

Proof. Note that x does not belong to processor i or its write buffer iff the step c,x is made by a different unit than unit i

$$\begin{aligned} x \notin \Sigma_{WB,i} \cup \Sigma_{P,i} &\iff \neg(x \in \Sigma_{WB,i} \cup \Sigma_{P,i}) \\ &\iff \neg(u(x) = i) \\ &\iff u(x) \neq 1 \\ &\iff u_M(c,x) \neq i. \end{aligned}$$

The claim follows with Lemmas 15 and 4

$$\begin{aligned} victims_M(c,x) &= \{\, i \mid A_{PR,i} \dot{\cap} Dom(W_M(c,x)) \wedge x \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \,\} \\ &= \{\, i \mid A_{PR,i} \dot{\cap} WS_M(c,x) \wedge x \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \,\} \\ &= \{\, i \mid A_{PR,i} \dot{\cap} WS_M(c,x) \wedge u_M(c,x) \neq i \,\} \\ &= \{\, i \mid A_{PR,i} \dot{\cap} idc(WS_M(c,x)) \wedge u_M(c,x) \neq i \,\} && \text{L 15, 4} \\ &= \{\, i \mid A_{PR,i} \dot{\cap} out_M(c,x) \wedge u_M(c,x) \neq i \,\} \end{aligned}$$

□

We identify steps made by different units by the following predicate

$$diffu(x,y) \equiv u(x) \neq u(y).$$

Lemma 99. *A step of another unit*

$$x \notin \Sigma_{P,i} \cup \Sigma_{WB,i}$$

commutes w.r.t. the write buffer

$$Op_{iM}(c,x)(Op_{iM}(c',y)(wb,w'),w) = Op_{iM}(c',y)(Op_{iM}(c,x)(wb,w),w').$$

Proof. The operation of x is a no-op. The claim follows

$$\begin{aligned} Op_{iM}(c,x)(Op_{iM}(c',y)(wb,w'),w) &= noop(Op_{iM}(c',y)(noop(wb,w),w'),w) \\ &= Op_{iM}(c',y)(Op_{iM}(c,x)(wb,w),w'). \end{aligned}$$

□

Lemma 100. *If two different units modify the write buffer*

$$diffu(x,y),$$

the order of the steps is irrelevant.

$$Op_{iM}(c,x)(Op_{iM}(c',y)(wb,w'),w) = Op_{iM}(c',y)(Op_{iM}(c,x)(wb,w),w').$$

Proof. At least one of the steps is not made by unit i. Let WLOG that step be x

$$x \notin \Sigma_{P,i} \cup \Sigma_{WB,i}.$$

The claim is now Lemma 99. □

We extend the definition of different units to timestamps in the obvious way

$$diffu[s](t,k) = diffu(s(t),s(k)).$$

It could be alternatively defined in terms of the units making those steps.

Lemma 101.

$$diffu(t,k) \equiv u_M(t) \neq u_N(k).$$

Proof. Straightforward

$$\begin{aligned} & u_M(t) \neq u_N(k) \\ \iff & u_M(c_M[s]^t, s(t)) \neq u_N(c_N[s]^k, s(k)) \\ \iff & u(s(t)) \neq u(s(k)) \\ \iff & diffu(t,k). \end{aligned}$$

□

We say step t interrupts step k if the unit making step k is a victim of step t

$$int_M(t,k) \equiv u_M(k) \in victims_M(t).$$

Note that this definition implies no particular order between t and k; a much later step may interrupt an earlier step. This makes sense in our context because we are not just looking at one particular ordering of steps, but also other potential orderings; the interrupt here simply means that steps t and k can not be reordered without seriously interfering with step k, by possibly changing all of the predicates and functions of step k (such as feasibility, fetched addresses, whether it is marked as shared or not, etc.).

A unit is interrupted iff its local registers are modified or read by a different unit

Lemma 102.

$$A_{PR,u_M(k)} \hat{\cap} out_M(t) \wedge diffu(t,k) \equiv int_M(t,k).$$

Proof. Straightforward with Lemma 98

$$\begin{aligned} int_M(t,k) &\equiv u_M(k) \in victims_M(t) \\ &\equiv u_M(k) \in \{\, i \mid A_{PR,i} \hat{\cap} out_M(t) \wedge u_M(t) \neq i \,\} && \text{L 98} \\ &\equiv A_{PR,u_M(k)} \hat{\cap} out_M(t) \wedge u_M(t) \neq u_M(k) \\ &\equiv A_{PR,u_M(k)} \hat{\cap} out_M(t) \wedge diffu(t,k). \end{aligned}$$

□

We say steps t and k are *unit-concurrent* if they are not steps of the same unit, and step t does not interrupt step k

$$ucon_M(t,k) \equiv \neg int_M(t,k) \wedge diffu(t,k).$$

For the intuition behind this definition, consider steps t and $t+1$ which are unit-concurrent. Because step t does not interfere with local inputs of step $t+1$, both $s(t)$ and $s(t+1)$ could be stepped next, i.e., are feasible in the configuration $c_M[s]^t$.

Note that this definition is not symmetric at all; when steps t and k are unit-concurrent, changing their order will simply not change the feasibility of step k (and thus races can not be hidden by changing the order and thus disabling step k), but step k might still interrupt step t, therefore changing its feasibility.

Note also that this definition does not imply an ordering between t and k; t may be well after k and still be unit-concurrent with it.

When a step accesses the registers of a unit making another step, the steps are not unit-concurrent.

Lemma 103.

$$out_M(t) \hat{\cap} A_{PR,u_M(k)} \rightarrow \neg ucon_M(t,k)$$

Proof. We apply Lemma 102 and obtain that the steps are made by different units iff step t interrupts k.

$$\begin{aligned} & A_{PR,u_M(k)} \hat{\cap} out_M(t) \wedge diffu(t,k) \equiv int_M(t,k) \\ \iff & 1 \wedge diffu(t,k) \equiv int_M(t,k) \\ \iff & diffu(t,k) \equiv int_M(t,k). \end{aligned}$$

The claim follows

$$\begin{aligned} ucon_M(t,k) &\equiv \neg int_M(t,k) \wedge diffu(t,k) \\ &\equiv \neg diffu(t,k) \wedge diffu(t,k) \\ &\equiv 0. \end{aligned}$$

□

When a step modifies or reads the local inputs of another step, they are not unit-concurrent.

Lemma 104.

$$out_M(t) \hat{\cap} C_M(k) \rightarrow \neg ucon_M(t,k)$$

Proof. By definition, the core registers of step k are processor registers of the unit making step k

$$C_M(k) \subseteq A_{PR,u_M(k)}$$

and we obtain an intersection between the outputs of step t and the core registers of step k

$$out_M(t) \dot{\cap} A_{PR,u_M(k)}.$$

The claim is now Lemma 103. □

We will often consider races where we do no not have unit-concurrency because the steps are made by the same unit, in particular, by the processor and the write buffer of the same unit. In those cases, races might occur for two reasons: 1) the processor step and write buffer step are in weak memory mode, and thus by definition annotated as a shared read and shared write respectively; or 2) the processor step modifies the mode registers, which are inputs (core registers) of the write buffer step.

In order to save ourselves pointless case distinctions, we define two steps as *object-concurrent* when the first does not modify the core registers of the second and they are made by different objects

$$ocon_M(t,k) \equiv out_M(t) \not\dot{\cap} C_M(k) \wedge o_M(t) \neq o_M(k).$$

Lemma 105.

$$ucon_M(t,k) \rightarrow ocon_M(t,k).$$

Proof. After unfolding definitions, we obtain the following claim

$$ucon_M(t,k) \rightarrow out_M(t) \not\dot{\cap} C_M(k) \wedge o_M(t) \neq o_M(k).$$

The first part is just the contraposition of Lemma 104. For the second part, assume for the sake of contradiction that the steps are made by different units but by the same object

$$u_M(t) \neq u_M(k) \wedge o_M(t) = o_M(k).$$

The proof continues by case distinction on the object making step t.

$o_M(t) = o_M(k) = P,i$**:** The oracle input of both steps belongs to processor i

$$s(t), s(k) \in \Sigma_{P,i}.$$

Thus both steps are made by unit i

$$u_M(t) = u_M(k) = i,$$

which is a contradiction.

$o_M(t) = o_M(k) = WB,i$**:** The oracle input of both steps belongs to write buffer i

$$s(t), s(k) \in \Sigma_{WB,i}.$$

Thus both steps are made by unit i

$$u_M(t) = u_M(k) = i,$$

which is a contradiction.

□

If steps are made by different objects, they are either made by different units, or the first step is made by a processor and the second by its write buffer, or vice versa. We do not show the proof.

Lemma 106.

$$\begin{aligned} o_M(t) \neq o_M(k) \equiv \quad & diffu(t,k) \\ & \vee s(t) \in \Sigma_{P,i} \wedge s(k) \in \Sigma_{WB,i} \\ & \vee s(t) \in \Sigma_{WB,i} \wedge s(k) \in \Sigma_{P,i}. \end{aligned}$$

Outputs of a step only remain visible to later steps if they have not been overwritten. We define the *set of addresses overwritten in the interval I* as the union of outputs of steps in I

$$out_M[s](I) = \bigcup_{t \in I} out_M[s](t).$$

If the interval is empty, the set of overwritten addresses is empty.

Lemma 107.

$$out_M(\emptyset) = \emptyset.$$

Proof. Straightforward

$$out_M[s](\emptyset) = \bigcup_{t \in \emptyset} out_M[s](t) = \emptyset.$$

□

The intervals can be concatenated.

Lemma 108.

$$out_M(I) \cup out_M(J) = out_M(I \cup J).$$

Proof. Straightforward

$$\begin{aligned} out_M(I) \cup out_M(J) &= (\bigcup_{t \in I} out_M[s](t)) \cup \bigcup_{t \in J} out_M[s](t) \\ &= \bigcup_{t \in (I \cup J)} out_M[s](t) \\ &= out_M(I \cup J). \end{aligned}$$

□

We define the *visible write-set of t at k* as the write-set minus all outputs from t to k

$$vws_M[s](t,k) = WS_M[s](t) \setminus out_M[s]((t:k)).$$

The visible write-set at step $t+1$ is just the write-set.

Lemma 109.

$$vws_M(t,t+1) = WS_M(t).$$

Proof. Straightforward with Lemma 107

$$\begin{aligned} vws_M(t,t+1) &= WS_M(t) \setminus out_M((t:t+1)) \\ &= WS_M(t) \setminus out_M(\emptyset) \\ &= WS_M(t) \setminus \emptyset && \text{L 107} \\ &= WS_M(t). \end{aligned}$$

□

The visible write-set can be extended to the right by removing more overwritten outputs.

Lemma 110.

$$k \leq l \rightarrow vws_M(t,k) \setminus out_M((k-1:l)) = vws_M(t,l).$$

Proof. Straightforward with Lemma 108. One easily sees that the intervals $(t:k)$ and $(k-1:l)$ can be joined together to interval $(t:l)$ (cf. Fig. 2.4)

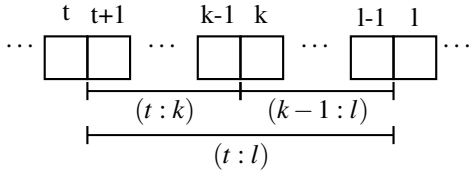


Figure 2.4: Interval $(t:k)$ ends with $k-1$, interval $(k-1:l)$ begins with k. They can thus be joined seamlessly.

$$\begin{aligned} vws_M(t,k) \setminus out_M((k-1:l)) &= WS_M(t) \setminus out_M((t:k)) \setminus out_M((k-1:l)) \\ &= WS_M(t) \setminus (out_M((t:k)) \cup out_M((k-1:l))) \\ &= WS_M(t) \setminus out_M((t:k) \cup (k-1:l)) && \text{L 108} \\ &= WS_M(t) \setminus out_M((t:l)) \\ &= vws_M(t,l). \end{aligned}$$

□

This also works for individual steps

Lemma 111.

$$vws_M(t,k) \setminus out_M(k) = vws_M(t,k+1).$$

Proof. Straightforward with Lemma 110

$$\begin{aligned} vws_M(t,k) \setminus out_M(k) &= vws_M(t,k) \setminus out_M((k-1:k+1)) \\ &= vws_M(t,k+1). && \text{L 110} \end{aligned}$$

□

For devices, that is not completely true: even when device registers are overwritten, the device might decide to "ignore" these changes or otherwise remember the original write. The device closure of the write-set of t thus always stays a visible output.

We define *visible outputs of t at k* by

$$vout_M[s](t,k) = dc(WS_M[s](t)) \cup vws_M[s](t,k).$$

The visible outputs at step $t+1$ are just the outputs.

Lemma 112.

$$vout_M(t,t+1) = out_M(t).$$

Proof. Straightforward with Lemma 109

$$\begin{aligned} vout_M(t,t+1) &= dc(WS_M(t)) \cup vws_M(t,t+1) \\ &= dc(WS_M(t)) \cup WS_M(t) && \text{L 109} \\ &= out_M(t). \end{aligned}$$

□

The visible outputs are a subset of the outputs

Lemma 113.

$$vout_M(t,k) \subseteq out_M(t).$$

Proof. Straightforward

$$\begin{aligned} vout_M(t,k) &= dc(WS_M(t)) \cup vws_M(t,k) \\ &= dc(WS_M(t)) \cup (WS_M(t) \setminus \bigcup_{t' \in (t:k)} out_M(t')) \\ &\subseteq dc(WS_M(t)) \cup WS_M(t) \\ &= out_M(t). \end{aligned}$$

□

2.10 Equivalence of Schedules

We wish to show that the reordering strategy which we will define does not affect the inputs of any of the steps in the low-level machine. For this we need a mechanism for identifying the same steps in different schedules, and for comparing how steps are executed in various schedules. We count steps of object X in a finite sequence s

$$\#X(s) = \#\{\, t \mid o(s(t)) = X \,\},$$

and in a schedule until t

$$\#X[s](t) = \#X(s[0:t-1]).$$

This allows us to split a schedule into parts, which are summed over individually.

Lemma 114.

$$\#X(s \circ r) = \#X(s) + \#X(r).$$

Proof.

$$\begin{aligned}
\#X(s \circ r) &= \#\{\, t \mid o((s \circ r)(t)) = X \,\} \\
&= \#\{\, t \mid t < |s| \wedge o(s(t)) = X \vee t \in [|s| : |s| + |r|) \wedge o(r(t - |s|)) = X \,\} \\
&= \#(\{\, t \mid o(s(t)) = X \,\} \cup \{\, t \mid t \in [|s| : |s| + |r|) \wedge o(r(t - |s|)) = X \,\}) \\
&= \#(\{\, t \mid o(s(t)) = X \,\} \cup \{\, t \mid t \in [0 : 0 + |r|) \wedge o(r(t - 0)) = X \,\}) \\
&= \#(\{\, t \mid o(s(t)) = X \,\} \cup \{\, t \mid o(r(t)) = X \,\}) \\
&= \#\{\, t \mid o(s(t)) = X \,\} + \#\{\, t \mid o(r(t)) = X \,\} \\
&= \#X(s) + \#X(r).
\end{aligned}$$

□

We also define the steps counts taken by an object in the whole schedule by

$$X(s) = \{\, n \mid \exists t . \#X[s](t) = n \,\}.$$

This is necessary to deal both with situations where object X makes a finite amount of steps, in which case $X(s)$ is the finite set of numbers less than or equal to the step count of object X, and where object X makes an infinite amount of steps, in which case $X(s)$ is the set of natural numbers. Finally we find the step where object X makes it's n-th step by

$$\#X{\approx}n(s) = \varepsilon\,\{\, t \mid o(s(t)) = X \wedge \#X[s](t) = n \,\}.$$

Since the step count is monotonic and increases at that point, there is always at most one such step.

Lemma 115.

$$\forall t, t' . o(s(t)) = o(s(t')) = X \wedge \#X[s](t) = n \wedge \#X[s](t') = n \rightarrow t' = t.$$

Proof. Assume for the sake of contradiction that they are unequal

$$t \neq t'.$$

Let without loss of generality t be the earlier step

$$t < t'.$$

Since t is a step of object X, the steps of object X before t are a proper subset of the steps before t'

$$\begin{aligned}
\{\, k < t \mid o(k) = X \,\} &\subsetneq \{\, k < t \mid o(s(k)) = X \,\} \cup \{\, t \,\} \\
&= \{\, k < t + 1 \mid o(s(k)) = X \,\} \\
&\subseteq \left\{\, k < t' \;\middle|\; o(s(k)) = X \,\right\}
\end{aligned}$$

and thus the step count at t is less than that at t'

$$\#X(t) = \#\{\, k < t \mid o(s(k)) = X \,\} < \#\left\{\, k < t' \;\middle|\; o(s(k)) = X \,\right\} = \#X(t').$$

But that contradicts our assumption that the step count at both is n

$$\#X(t) = n = \#X(t').$$

□

When a step number $n+1$ is reached by unit X, there is a point in time when unit X makes its n-th step.

Lemma 116.

$$\#X[s](t) = n+1 \to \exists t'.o(s(t')) = X \wedge \#X[s](t') = n.$$

Proof. By induction on t.

In the base case, the step count is zero

$$\#X[s](0) = \#X(s[0:-1]) = \#X(\varepsilon) = 0,$$

which contradicts our assumption that the step count equals $n+1$.

In the inductive step $t \to t+1$, the step count at $t+1$ can be split between the step count at t and step t

$$\begin{aligned}\#X[s](t+1) &= \#X(s[0:t]) \\ &= \#X(s[0:t-1]) + \#X(s(t)) \\ &= \#X[s](t) + \#X(s(t)).\end{aligned}$$

We distinguish whether step t is made by object X or not.

$o(s(t)) = X$**:** In this case the step number increases

$$\#X(s(t)) = 1 \wedge \#X[s](t+1) = \#X[s](t) + 1,$$

and thus the step count at t was n

$$X[s](t) = n$$

and the claim follows with $t' := t$.

$o(s(t)) \neq X$**:** In this case the step count is unchanged

$$\#X(s(t)) = 0 \wedge \#X[s](t+1) = \#X[s](t) = n+1,$$

and the claim follows directly by the induction hypothesis.

□

We say schedules s and r are *equivalent* when each object makes the same steps and both computations strongly agree on those steps

$$\begin{aligned}s \equiv_M r \iff \forall X.X(s) = X(r) \wedge \forall t,k.t = \#X{\approx}n(s) \wedge k = \#X{\approx}n(r) \\ \to s(t) = r(k) \wedge c_M[s]^t =_M^{s(t)} c_M[r]^k.\end{aligned}$$

This relation obviously is an equivalence relation.

If we only do a single reordering, it is easy to keep track of the position of each step: steps outside of the reordered portion are not moved, the moved step is now at its new position, and the other steps are off by one. The situation is depicted in Fig. 2.5.

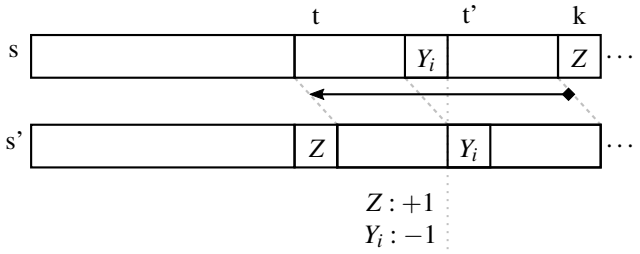


Figure 2.5: Step k of object Z is moved to position t, other steps in the reordered portion are made by objects Y_i other than Z. We pick some timestamp t' in the reordered interval (dotted vertical line) and compare the number of steps made by each object at that timestamp. Summing up the steps of each object from the left, we count one additional $Z = o(s(k))$ and one less $Y_i = o(s'(t'))$ in schedule s'. For all other objects the number of steps is unchanged.

Lemma 117. *Let s' be the schedule obtained from s by moving k to $t \leq k$*

$$s' = s[t \leftarrow k].$$

Assume that this did not reorder steps of the same object

$$\forall t' \in [t : k].o(s(k)) = o(s(t')) \rightarrow k = t'.$$

Then both of the following are true.

1. *The step counts are changed only in the reordered portion, in which the moved unit had one additional step before the reordering, and the unit that is now stepped at that position has one additional step after the reordering*

$$\#X[s'](t') = \begin{cases} \#X[s](t') + 1 & o(s(k)) = X \wedge t' \in (t : k] \\ \#X[s](t') - 1 & o(s'(t')) = X \wedge t' \in (t : k] \\ \#X[s](t') & o.w. \end{cases}$$

2. *Let n be one of the step counts reached by object X in s*

$$n \in X(s).$$

The position of the n-th step is changed by the reordering as follows

$$\#X{\approx}n(s') = \begin{cases} \#X{\approx}n(s) & \#X{\approx}n(s) \notin [t : k] \\ t & \#X{\approx}n(s) = k \\ \#X{\approx}n(s) + 1 & \#X{\approx}n(s) \in [t : k). \end{cases}$$

Proof. Note that the object of step $t' \in (t : k]$ in s' is the object of step $t' - 1$ in s

$$o(s'(t')) = o(s[t \leftarrow k](t')) = o(s(t' - 1)),$$

and by assumption it can never equal the object of step k in s

$$t' - 1 \neq k \wedge o(s(t' - 1)) \neq o(s(k)).$$

Note that s' can be characterized as a piece-wise reordering of s

$$s' = s[0:t-1] \circ s(k) \circ s[t:k-1] \circ s[k+1:\infty].$$

Thus the first t' steps in the sequence can also be characterized in piece-wise fashion

$$s'[0:t'-1] = \begin{cases} s[0:t'-1] & t' \leq t \\ s[0:t-1] \circ s(k) \circ s[t:t'-2] & t' \in (t:k] \\ s[0:t-1] \circ s(k) \circ s[t:k-1] \circ s[k+1:t'-1] & t' > k. \end{cases}$$

We can thus split the definition of the number of steps in an analogous fashion

$$\begin{aligned} \#X[s'](t') &= \#X(s'[0:t'-1]) \\ &= \begin{cases} \#X(s[0:t'-1]) & t' \leq t \\ \#X(s[0:t-1] \circ s(k) \circ s[t:t'-2]) & t' \in (t:k] \\ \#X(s[0:t-1] \circ s(k) \circ s[t:k-1] \circ s[k+1:t'-1]) & t' > k. \end{cases} \end{aligned}$$

We distinguish between those three cases in our proof of the first claim.

$t' \leq t$**:** Straightforward

$$\#X[s'](t') = \#X(s[0:t'-1]) = \#X[s](t').$$

$t' \in (t:k]$**:** We split the count and add a negative term to make up for the missing step $s(t'-1)$

$$\begin{aligned} \#X[s'](t') &= \#X(s[0:t-1] \circ s(k) \circ s[t:t'-2]) \\ &= \#X(s[0:t-1]) + \#X(s(k)) + \#X(s[t:t'-2]) \\ &= \#X(s[0:t-1]) + \#X(s(k)) + \#X(s[t:t'-2]) \\ &\quad + \#X(s(t'-1)) - \#X(s(t'-1)) \\ &= \#X(s[0:t-1] \circ s[t:t'-2] \circ s(t'-1)) + \#X(s(k)) - \#X(s(t'-1)) \\ &= \#X(s[0:t'-1]) + \#X(s(k)) - \#X(s(t'-1)) \\ &= \#X[s](t') + \#X(s(k)) - \#X(s(t'-1)) \\ &= \begin{cases} \#X[s](t') + 1 - 0 & o(s(k)) = X \\ \#X[s](t') + 0 - 1 & o(s(t'-1)) = X \\ \#X[s](t') + 0 - 0 & \text{o.w.} \end{cases} \\ &= \begin{cases} \#X[s](t') + 1 & o(s(k)) = X \\ \#X[s](t') - 1 & o(s'(t')) = X \\ \#X[s](t') & \text{o.w.,} \end{cases} \end{aligned}$$

which is the claim.

$t' > k$**:** We split the count and reorder it

$$\begin{aligned} \#X[s'](t') &= \#X(s[0:t-1] \circ s(k) \circ s[t:k-1] \circ s[k+1:t'-1]) \\ &= \#X(s[0:t-1]) + \#X(s(k)) + \#X(s[t:k-1]) + \#(s[k+1:t'-1]) \\ &= \#X(s[0:t-1] \circ s[t:k-1] \circ s(k) \circ s[k+1:t'-1]) \\ &= \#X(s[0:t'-1]) = \#X[s](t'), \end{aligned}$$

which is the claim.

Let now for the sake of brevity

$$t' = \#X{\approx}n(s)$$

be the original position of the n-th step, and

$$t'' = \begin{cases} t' & t' \notin [t:k] \\ t & t' = k \\ t'+1 & t' \in [t:k). \end{cases}$$

be its claimed new position

$$\#X{\approx}n(s') \stackrel{!}{=} t''.$$

By Lemma 115 it suffices to show that t'' is a step of object X and that object X had already reached n steps at t''

$$o(s'(t'')) \stackrel{!}{=} o(s(t')) \wedge \#X[s'](t'') \stackrel{!}{=} n.$$

We clearly obtain for each case in the definition of t'' that t'' is the new position of the step in the schedule

$$\begin{aligned} s(t') &= \begin{cases} s'(t') & t' \notin [t:k] \\ s'(t) & t' = k \\ s'(t'+1) & t' \in [t:k), \end{cases} \\ &= s'(t''), \end{aligned}$$

and therefore t'' is a step of object X

$$o(s'(t'')) = o(s(t')) = X.$$

Since none of the steps between k and t are made by the unit of which we move the step, we have

$$o(s(k)) = X \rightarrow \#X[s](k) = \#X[s](t).$$

With Claim 1 (shown above) we now obtain that the step count for each of the cases in the definition of t'' corresponds to the step count of t'

$$\begin{aligned} n = \#X[s](t') &= \begin{cases} \#X[s'](t') & t' \notin [t:k] \\ \#X[s](t) & t' = k \wedge o(s(k)) = X \\ \#X[s'](t')+1 & t' \in [t:k) \wedge o(s'(t')) = X \end{cases} \\ &= \begin{cases} \#X[s'](t') & t' \notin [t:k] \\ \#X[s'](t) & t' = k \wedge o(s'(t)) = X \\ \#X[s'](t'+1) & t' \in [t:k) \wedge o(s'(t')) = X \end{cases} \\ &= \#X[s'](t''), \end{aligned}$$

which was the claim. □

Furthermore, if we only do a single reordering, the steps before the reordered portion are the same, and the steps behind the reordered portion are executed in the same order. Therefore such schedules are equivalent if the inputs of the moved steps are the same and the schedules reach the same configuration at the end of the reordered portion.

Lemma 118. *If the step at t after moving k to t strongly agrees with step k before the reordering*

$$c_M^k =_M^{s(k)} c_M[t \leftarrow k]^t,$$

and steps $t' \in [t : k)$ after the reordering also strongly agree

$$c_M^{t'} =_M^{s(t')} c_M[t \leftarrow k]^{t'+1},$$

and the computations reach the same configuration after step k

$$c_M[t \leftarrow k]^{k+1} = c_M^{k+1},$$

then the schedules are equivalent

$$s[t \leftarrow k] \equiv_M s.$$

Proof. Obviously each unit is stepped equally often in the two schedules

$$X(s[t \leftarrow k]) = X(s).$$

Consider now steps t' and t'' which are the n-th step of unit X in s and $s[t \leftarrow k]$, respectively

$$t' = \#X{\approx}n(s),\ t'' = \#X{\approx}n(s[t \leftarrow k]).$$

We have to show that the steps have the same oracle input and that the configurations from which the steps are started strongly agree

$$s(t') \stackrel{!}{=} s[t \leftarrow k](t''),$$
$$c_M^{t'} \stackrel{!}{=}_M^{s(t')} c_M[t \leftarrow k]^{t''}.$$

By Lemma 117 we obtain that t'' is as follows

$$t'' = \begin{cases} t' & t' \notin [t : k] \\ t & t' = k \\ t'+1 & t' \in [t : k). \end{cases}$$

We distinguish between these three cases, but split the first case depending on whether t' is before the reordered portion or after it.

$t' < t$: In this case t' equals t''

$$t'' = t',$$

and t' is before the reordered portion. Thus the same step is made

$$s[t \leftarrow k](t'') = s(t'') = s(t'),$$

which is the first claim. Furthermore the steps are started from the same configuration

$$c_M[t \leftarrow k]^{t''} = c_M^{t''} = c_M^{t'},$$

and strong agreement holds by reflexivity

$$c_M^{t'} =_M^{s(t')} c_M[t \leftarrow k]^{t''},$$

which is the second claim.

$t' > k$**:** In this case t' equals t''

$$t'' = t',$$

It is easy to show that the same step is made

$$s[t \leftarrow k](t'') = s(t'') = s(t'),$$

For the memory view, however, we first prove by induction on t'', starting with $k+1$, that the configurations are the same

$$c_M[t \leftarrow k]^{t''} = c_M^{t''}.$$

- The base case is solved by the assumption

$$c_M[t \leftarrow k]^{k+1} = c_M^{k+1}.$$

- In the inductive step $t'' \to t''+1$, the inductive claim follows with the induction hypothesis and the fact that t'' is not in the reordered portion of s

$$\begin{aligned} c_M[t \leftarrow k]^{t''+1} &= c_M[t \leftarrow k]^{t''} \cdot_M s[t \leftarrow k](t'') \\ &= c_M^{t''} \cdot_M s(t'') \\ &= c_M^{t''+1}. \end{aligned}$$

We conclude that the configurations in which the step starts are the same

$$c_M^{t'} = c_M^{t''} = c_M[t \leftarrow k]^{t''},$$

and the second claim holds by reflexivity

$$c_M^{t'} =_M^{s(t')} c_M[t \leftarrow k]^{t''}.$$

$t' = k$**:** In this case t'' is t

$$t'' = t.$$

Clearly the same step is made

$$s[t \leftarrow k](t'') = s[t \leftarrow k](t) = s(k) = s(t').$$

The configurations at k and at t agree by assumption

$$c_M^k =_M^{s(k)} c_M[t \leftarrow k]^t,$$

and the claim follows with the equalities $k = t'$ and $t = t''$

$$c_M^{t'} =_M^{s(t')} c_M[t \leftarrow k]^{t''}.$$

$t' \in [t : k)$**:** In this case t'' is $t'+1$

$$t'' = t'+1.$$

Clearly the same step is made

$$s[t \leftarrow k](t'') = s[t \leftarrow k](t'+1) = s(t').$$

The configurations at t' and at $t'+1$ agree by assumption

$$c_M^{t'} =_M^{s(t')} c_M[t \leftarrow k]^{t'+1},$$

and the claim follows

$$c_M^{t'} =_M^{s(t')} c_M[t \leftarrow k]^{t''}.$$

□

2.11 Reads, Writes, Read-Modify-Writes

Normally it is useful to classify instructions into *reads*, i.e., instructions that read from memory, *writes*, i.e., instructions that modify memory, and *read-modify-writes*, i.e., instructions that atomically do both and where the written values depend on the read (e.g., test-and-set). In this thesis, this classification can not be done based on syntax of the instruction, e.g., bit patterns; but only on the semantics of the step.

In this sense, a memory write is a step that modifies not only the normal processor registers (or the write buffer):

$$mwrite_M(c,x) \equiv out_M(c,x) \not\subseteq A_{NPR,u_M(c,x)}.$$

Note that this ignores, e.g., store instructions in low-level machine mode, which go to the buffer and do not immediately modify the memory. Furthermore, there is no guarantee that an instruction which is fetched in one configuration and is a memory write is still a memory write in other configurations. For example, a test-and-set operation may modify a bit in one configuration, but not another.

Perhaps surprisingly, it is not necessary to formalize the notions of reads and read-modify-writes for the purpose of this work. The only reads that are relevant for us are *shared* reads, i.e., reads that access shared data. As mentioned before, shared reads have to be annotated by the programmer, and are not based on the semantics of the instruction; the formalization is found in Section 4.2.

A read-modify-write is simply a step which is both a read and a memory write. Again, since this classification is purely semantical, a compare-and-swap instruction with a failed test might is not considered an RMW in this text. We will need to deal with races between a write and a read-modify-write. In this case, whether the read-modify-write modifies the memory or not depends on the order between the write and the read-modify-write. Consider the following program:

```
x[0:7].store(1);     ‖ x[0:15].cas(0 → 2^8);
y = x[8:15].load();  ‖
```

In a high-level computation (without buffers), the compare-and-swap can never race with the load because it would have to be ordered after the store. This order can be reversed in a low-level computation if the write is buffered, thus causing the compare-and-swap to race with the load. Similarly, an ordinary load instruction can be disabled by a concurrent code modification, as the following program shows. We consider here `I` to be the address of the instruction of the second thread which loads the value of `x`. Thread 1 replaces that load by a constant assignment.

```
I.store('t = 0;'); ‖ I: t = x;
x := 1;            ‖
```

In a high-level computation, there is no race between the load and the store, and the value of t is always 0 (either because the instruction has been replaced by a constant assignment, or because the assignment to x has not taken place when Thread 1 is scheduled). But in the low-level machine, the assignment to x can be executed before the code modification takes place, resulting in a race between the assignment to x and the load from x, and a possible value of $t = 1$.

We will exclude programs like these with our software conditions.

We show that it suffices to look at the write-set to determine whether a step is memory write or not.

Lemma 119.

$$mwrite_M(c,x) \equiv WS_M(c,x) \not\subseteq A_{NPR,u_M(c,x)}.$$

Proof. By Lemma 4, the normal processor registers are closed

$$closed(A_{NPR,u_M(c,x)}).$$

The claim follows with Lemma 16

$$\begin{aligned} mwrite_M(c,x) &\equiv out_M(c,x) \not\subseteq A_{NPR,u_M(c,x)} \\ &\equiv idc(WS_M(c,x)) \not\subseteq A_{NPR,u_M(c,x)} \\ &\equiv WS_M(c,x) \not\subseteq A_{NPR,u_M(c,x)}. \qquad \text{L 16} \end{aligned}$$

□

For processor steps in the low-level machine, it suffices thus to look at the domain of the prepared bypassing writes.

Lemma 120.

$$x \in \Sigma_{P,i} \rightarrow mwrite_{\downarrow}(c,x) \equiv Dom(PW_{\downarrow}(c,x).bpa) \not\subseteq A_{NPR,u_{\downarrow}(c,x)}.$$

Proof. The low-level machine uses low-level machine semantics

$$LL_{\downarrow}(c,x)$$

and thus the executed write is the prepared bypassing write

$$W_{\downarrow}(c,x) = PW_{\downarrow}(c,x).bpa.$$

The claim follows with Lemma 119

$$\begin{aligned} mwrite_{\downarrow}(c,x) &\equiv WS_{\downarrow}(c,x) \not\subseteq A_{NPR,u_{\downarrow}(c,x)} \qquad \text{L 119} \\ &\equiv Dom(W_{\downarrow}(c,x)) \not\subseteq A_{NPR,u_{\downarrow}(c,x)} \\ &\equiv Dom(PW_{\downarrow}(c,x).bpa) \not\subseteq A_{NPR,u_{\downarrow}(c,x)}. \end{aligned}$$

□

We show that a step can only modify accessible registers of another unit with a memory write.

Lemma 121.

$$u_M(t) \neq i \wedge out_M(t) \cap ACC_i \rightarrow mwrite_M(t).$$

Proof. The processor registers of the unit making step t are not accessible to unit i, which is a different unit

$$A_{NPR,u_M(t)} \not\cap ACC_i,$$

Since the outputs of t intersect with the accessible registers, we conclude that the outputs of t are not contained in the normal processor registers

$$out_M(t) \not\subseteq A_{NPR,u_M(t)}$$

and by definition step t is a memory write

$$mwrite_M(t).$$

□

Therefore interaction between units — even across different machines M and N — is only possible using memory writes.

Lemma 122.

$$diffu(t,k) \wedge out_M(t) \dot{\cap} in_N(k) \cup out_N(k) \rightarrow mwrite_M(t).$$

Proof. By assumption the steps are made by different units

$$u_M(t) \neq u_N(k).$$

By Lemma 93 the inputs and outputs of step k are a subset of the addresses accessible to the unit making step k

$$in_N(k) \cup out_N(k) \subseteq ACC_{u_N(k)}.$$

We conclude that the outputs intersect with the accessible registers of step k

$$out_M(t) \dot{\cap} ACC_{u_N(k)}.$$

The claim is now Lemma 121. □

Chapter 3

MIPS ISA

In this part we instantiate the models of the previous chapters with a variant of the MIPS instruction set architecture, which has been enriched with the advanced hardware features of x86 discussed in the previous chapters (APICs, store buffers, devices). A formalization of that instruction set architecture is in the appendix, taken verbatim from the technical report of Schmaltz [Sch13]. Our instantiation differs in a few key points from the ISA in the technical report of Schmaltz.

- In order to correctly describe the behavior of the machine in case of misbehaving users, we will need to separate instruction fetch and instruction execute for users. We add a non-deterministically filled *instruction buffer*, which is a set of key-value pairs that maps physical instruction addresses to instruction words. Interrupts flush this buffer. For the operating system, one can easily enforce software conditions — such as waiting a few cycles between modifying an instruction and fetching from it on the same core, and never concurrently modifying code — that simulate single-cycle fetch and execute (even though in the pipelined hardware, an instruction is fetched long before it is executed). We therefore use the instruction buffer only in user mode.

- We allow the MMU to set the access bit in operating system mode, but only directly after jumping to the interrupt service routine. Allowing the step in operating system mode is necessary in case the hardware jumps to the interrupt service routine while the MMU is setting the accessed bit, in which case this write operation can not be aborted and completes after the interrupt service routine has been entered. However, in hardware we can stall the execution of the interrupt service routine until the MMU has completed the operation (by preventing the first instruction of the interrupt service routine from leaving the translation stages of the pipeline while a translation is still taking place). In the ISA this is reflected by a delayed user mode bit, which is active until the first instruction of the interrupt service routine is executed. The MMU is allowed to perform translations in operating system mode until that flag is lowered, i.e., after jumping to the interrupt service routine but before executing the first instruction of the interrupt service routine.

- We do not always lower the in-service register bits on an exception return, and we do not lower them when writing to the end-of-interrupt port of the APIC. This resolves the redundancy (which falsely causes two serviced interrupts to be

cleared by the interrupt service routine) and prevents the interrupt service routine from falsely clearing serviced interrupts when the interrupt service routine is not handling an external interrupt. One of the conditions of the software discipline (Condition IRRForwarding) will forbid us to have at the same time a) the in-service register as interrupt registers and b) allow changes to the in-service register to be performed by (buffered) store instructions. For the sake of implementing the direct-to-core IPI functionality of the Schmaltz ISA, we have to put the in-service registers into the processor registers, and can thus not allow writes to the end-of-interrupt port of the local APIC to have any effect on the in-service register[1]. Instead, we use the exception cause register to determine whether the in-service register bits should be flushed. Note that this means that the exception cause register can not be restored by a preempting interrupt service routine, and has to be stored on an interrupt stack at the beginning of the interrupt service routine before external interrupts are enabled.

- We add read- and write fences which on their own do nothing but together are an efficient tool to ensure that shared writes are never buffered by a thread that executes a shared read.
- We flush the write buffer when an interrupt is taken (i.e., during the processor step that jumps to the interrupt service routine). This is closer to x86 and allows us to use for the memory mode registers $A_{SC,p}$ essentially the operating system mode $c.p(p).core.spr(mode)$.
- The passive transition function of the end-of-interrupt register uses the stored value to determine the interrupt that has ended, rather than snooping the in-service register of its processor. The main reason for this is that the model does not allow us to snoop processor registers when the store buffer makes a step, only the other device registers; in short, this issue could be resolved as well by switching to voluntary IPIs.
- We do not consider a generic device model, but only a simple disk with an interrupt request bit.
- We add an emode register, which stores the value of the mode register on JISR and restores it on ERET.
- We add two delayed program counters, which correspond to the delay slots incurred by pipelined implementations of MIPS with address translation that do not have branch speculation.
- We allow faulty walks to enter the translation look-aside buffer,
- We do not consider the multiplication unit and its special registers HI and LO.

Most of these changes can be found in the literature [PBLS16, KMP14].

We also have modified several minor things, such as the order of some fields in the APIC, in order to streamline the definition somewhat.

[1] A better semantics of APICs, which would also be closer to the x86 APIC model, would use non-deterministic, voluntary IPIs rather than direct-to-core IPIs. In that model, the in-service and interrupt request registers would be in the address space of the local APIC. This alternative semantics of APICs can also be modeled by our machine model, by non-deterministically fetching interrupts. Steps that fetch interrupts drain the write buffer and are only valid if there is an interrupt to be fetched, and perform the jump to the interrupt service routine.

3.1 Notation

We use the notation for binary representation x_n, binary interpretation $\langle a \rangle$, and binary addition $a +_n b$ from [PBLS16]. For the convenience of the reader we have copied the definition of this notation in Appendix A.

We consider $P \in \mathbb{N}$ MIPS processors, denoted by natural numbers $p \in [1 : P]$. We use p, q for variables ranging over MIPS processors.

We will often come across situations in which a single component of the Schmaltz ISA, such as the program counter, is split into multiple addresses in our model, such as four individual bytes. In general, if component a is split into R parts, each a bit string of length d, we denote the addresses with

$$a_r \in \mathcal{A} \wedge V(a_r) = \mathbb{B}^d,\ r \in [0 : R-1].$$

We try to minimize the pain of this by treating a as the sum of the parts, with the meaning depending on the context:

- given a valuation ν that covers all bytes of such a component, we use the notation

  $$\nu(a) = \nu(a_{R-1}) \circ \ldots \circ \nu(a_0)$$

  to denote the concatenation of the individual bytes,

- when defining a valuation

  $$\nu(a) = v,$$

  by splitting the value into R equal parts

  $$\nu(a_r) = v[(r+1) \cdot d - 1 : r \cdot d],$$

- when an enumeration of addresses is required, we treat a as the enumeration of its parts

  $$a = a_0, \ldots, a_{R-1};$$

  for example,

  $$\nu =_{\{a,b\}} \nu' \iff \nu =_{\{a_0,\ldots,a_{R-1},b\}} \nu',$$

Furthermore, we define $d \in \mathbb{N}$ consecutive 32-bit addresses starting at $a \in \mathbb{B}^{32}$

$$(a \cdot d)_r = a +_{32} r_{32},\ r < d,$$

using the same notation as above. For example, when we wish to fetch four bytes starting from the delayed delayed program counter (ddpc), we might say

$$F(c,x) = \{\, c.\mathrm{p}(p).\mathrm{core.ddpc} \cdot 4 \,\}.$$

All components of the MIPS ISA are collapsed into one memory component in our model. In order to make the notation more uniform, we define for some addresses a specific notation that allows us to closely mirror the MIPS ISA notation of Sabine Schmaltz, such as

$$c.\mathrm{p}(p).\mathrm{core.pc} = c.m((\mathrm{p}, p, \mathrm{pc})).$$

Since nearly all addresses have a range of one byte, we define (unless stated otherwise) that the range of each address is one byte

$$V(a) = \mathbb{B}^8.$$

In order to avoid confusion between constant addresses and variables ranging over addresses or values of addresses, we use a non-italic font for fixed addresses, and italic names for variables, as in the following example

$$\mathrm{pc} \in A, pc \in V(\mathrm{pc}).$$

When we define F and R etc., we will often have multiple parameters that are valuations, namely the core configuration, the set of fetched addresses, and the read results

$$\begin{aligned} core &\in Val(A_{PR,i}), \\ fetch &\in Val(F(core,x)), \\ v &\in Val(R(core,x,fetch)), \end{aligned}$$

which are passed as parameters to functions such as F and R. We will define those functions $f \in \{F,R,\dots\}$ using the form

$$f(c,x) = \dots,$$

and simply be vigilant to only use components of c for which the parameters given to f in step c,x really provide values. In the case of F, for example, we use core registers such as the program counters, which would allow us to define something like the following

$$F(c,x) = \{\, c.\mathrm{p}(p).\mathrm{core.ddpc} \cdot 4 \,\},$$

and in the case of $R(c,x)$ we use core registers, but also registers fetched in step c,x as defined by $F(c,x)$.

This may cause conflicts when the parameters provide different value for the same address, for example when an instruction loads from its own address, and thus W obtains a value for that address from $fetch$ (taken directly from $c.m$) and a different value from v (taken from a more up-to-date write in the write buffer). We resolve such conflicts by using the value from the most recent memory access: results from v overwrite those from $fetch$, which overwrite those from $core$.

When defining a partial valuation, e.g., W, we sometimes use a tabular notation which lists in one column the address (and possible conditions) and in another column the value for that address.

3.2 Configurations

In this section we introduce all the processors and devices as well as all addresses and their ranges.

For the sake of brevity, we define for all units except for MIPS processors that there are no memory mode registers

$$i \neq p \to A_{SC,i} = \emptyset$$

and that the unit is always in strong memory mode

$$i \neq p \to SC_i(\dots) = 0.$$

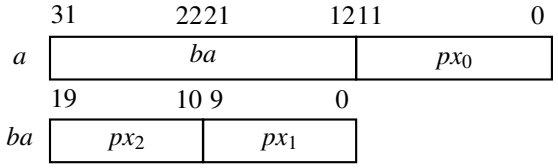


Figure 3.1: A bit string $a \in \mathbb{B}^{32}$ is split into base address $ba \in \mathbb{B}^{20}$ and a byte offset px_0. The base address is split further into two page indices px_2, px_1.

3.2.1 MMU Translations

As shown in Fig. 3.1, for each 32-bit string $a \in \mathbb{B}^{32}$ we define bitfields ba for base address

$$a.ba = a[31:12],$$

and a byte offset px_0

$$a.px_0 = a[11:0].$$

For each base address $ba \in \mathbb{B}^{20}$ we define two levels of page indices px_1, px_2

$$ba.px_2 = a[19:10],\ ba.px_1 = a[9:0].$$

When an address is translated, its base address is treated as a virtual address and translated to a physical base address, whereas the byte offset is used verbatim.

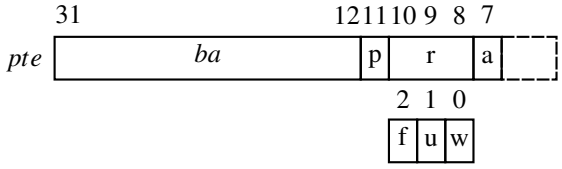


Figure 3.2: A page table entry is split into a base address, a present bit, three access-rights bits, and an accessed bit. The lowest six bits are unused.

As show in Fig. 3.2, we define for each page table entry $pte \in \mathbb{B}^{32}$ (in addition to the base address etc.) the following bit fields. A present bit p

$$pte.p = pte[11],$$

access rights bits r

$$pte.r = pte[10:8],$$

and an accessed bit a

$$pte.a = pte[7].$$

Access rights $r \in \mathbb{B}^3$ have fields for permission to fetch

$$r.f = r[2],$$

permission to access in user mode (redundant, since all translations are used in user mode)

$$r.u = r[1],$$

and permission to write

$$r.w = r[0].$$

A walk $w \in K_{walk}$ consists of the following bit fields

$w.va \in \mathbb{B}^{20}$**:** virtual page address to be translated.

$w.ba \in \mathbb{B}^{20}$**:** base address of the next memory location to be accessed, i.e., either the next page table entry, or the physical page address.

$w.asid \in \mathbb{B}^{6}$**:** address space identifier (ASID) of the translation.

$w.level \in \{0,1,2\}$**:** the number of remaining walk extensions needed to reach the physical page address.

$w.r \in \mathbb{B}^{3}$**:** rights remaining for memory accesses using this translation.

$w.fault \in \mathbb{B}$**:** indicates a faulting walk.

A walk is *complete* when the level is zero

$$complete(w) \equiv w.level = 0.$$

For incomplete walks, the address of the next page table entry to be accessed is computed by appending the page index corresponding to the level behind the base address of the walk.

$$\neg complete(w) \rightarrow ptea(w) = w.ba \circ w.va.px_{w.level} \circ 00$$

The physical address given by a translation and a complete virtual address $va \in \mathbb{B}^{32}$ is obtained by concatenating the base address of the walk with the byte offset of the virtual address

$$pma(w,va) = w.ba \circ va.px_0.$$

A new walk is created for a virtual address and ASID and base address of the page table by setting the level to two, the rights to full permission, and the fault bit to zero

$$\begin{aligned}
winit(va,ba,asid).va &= va,\\
winit(va,ba,asid).ba &= ba,\\
winit(va,ba,asid).asid &= asid,\\
winit(va,ba,asid).level &= 2,\\
winit(va,ba,asid).r &= 111,\\
winit(va,ba,asid).fault &= 0.
\end{aligned}$$

An incomplete walk is extended using a given page table entry by using the base address indicated by the page table entry, decreasing the level, taking the lower permissions of the walk and the page table entry, and indicating a fault if the page table entry was not present

$$\begin{aligned}
wext(w,pte).va &= w.va,\\
wext(w,pte).ba &= pte.ba,\\
wext(w,pte).asid &= w.asid,\\
wext(w,pte).level &= w.level - 1,\\
wext(w,pte).r_i &= w.r_i \wedge pte.r_i,\\
wext(w,pte).fault &= \neg pte.p.
\end{aligned}$$

3.2.2 MIPS Processor Configurations

We consider a machine with $P \in \mathbb{N}$ MIPS processors. Each *MIPS processor* $p \in [1:P]$ is a unit

$$p \in U.$$

Each MIPS processor p has the following registers.

- 32 word-sized general purpose registers (GPR) and special purpose registers (SPR). Each register $x \in [0:31]$ in each register file $pr \in \{\,\mathrm{gpr}, \mathrm{spr}\,\}$ is split into four bytes

  $$(\mathrm{p}, p, pr, x)_r \in A_{NPR,p}.$$

  For these we use the notation

  $$c.\mathrm{p}(p).\mathrm{core}.pr(x) = c.m((\mathrm{p}, p, pr, x)).$$

- A program counter and two delayed program counters. Each of these $pc \in \{\,\mathrm{pc}, \mathrm{dpc}, \mathrm{ddpc}\,\}$ is split into four bytes

  $$(\mathrm{p}, p, pc)_r \in A_{NPR,p}.$$

  For these we use the notation

  $$c.\mathrm{p}(p).\mathrm{core}.pc = c.m((\mathrm{p}, p, pc)).$$

- A translation look-aside buffer (TLB), which is a set of walks

  $$(\mathrm{p}, p, \mathrm{TLB}) \in A_{NPR,p},$$
  $$V((\mathrm{p}, p, \mathrm{TLB})) = 2^{K_{walk}}.$$

  For the TLB we use the notation

  $$c.\mathrm{p}(p).\mathrm{TLB} = c.m((\mathrm{p}, p, \mathrm{TLB})).$$

- An instruction buffer (IB), which is a set of pairs of aligned addresses and words

  $$(\mathrm{p}, p, \mathrm{IB}) \in A_{NPR,p},$$
  $$V((\mathrm{p}, p, \mathrm{IB})) = 2^{\mathbb{B}^{30} \times \mathbb{B}^{32}}.$$

  For the IB we use the notation

  $$c.\mathrm{p}(p).\mathrm{IB} = c.m((\mathrm{p}, p, \mathrm{IB})).$$

- Two flags that manage start and init interrupt requests. Each of the flags $IR \in \{\,\mathrm{sipirr}, \mathrm{initrr}\,\}$ is an interrupt register and has a single bit

  $$(\mathrm{p}, p, IR) \in A_{IPR,p},$$
  $$V((\mathrm{p}, p, IR)) = \mathbb{B}.$$

  For these we use the notation

  $$c.\mathrm{p}(p).\mathrm{apic}.IR = c.m((\mathrm{p}, p, IR)).$$

- The eight most significant bits of the program counter to which the processor jumps after a SIPI interrupt (called *SIPI-vector*, since it is sent as the interrupt vector in a SIPI)
$$(\mathrm{p}, p, \mathrm{sipivect}) \in A_{IPR,p}.$$
For the SIPI-vector we use the notation
$$c.\mathrm{p}(p).\mathrm{apic.sipivect} = c.m((\mathrm{p}, p, \mathrm{sipivect})).$$

- An interrupt request register (IRR), which is an interrupt register. We split the register into pieces of 32 bytes
$$(\mathrm{p}, p, \mathrm{IRR})_r \in A_{IPR,p}.$$
For the IRR we use the notation
$$c.\mathrm{p}(p).\mathrm{apic.IRR} = c.m((\mathrm{p}, p, \mathrm{IRR})).$$

- An in-service register (ISR), which is a normal register. We split the register into pieces of 32 bytes
$$(\mathrm{p}, p, \mathrm{ISR})_r \in A_{NPR,p}.$$
For the ISR we use the notation
$$c.\mathrm{p}(p).\mathrm{apic.ISR} = c.m((\mathrm{p}, p, \mathrm{ISR})).$$

- A binary running bit
$$(\mathrm{p}, p, \mathrm{running}) \in A_{NPR,p},$$
$$V((\mathrm{p}, p, \mathrm{running})) = \mathbb{B}.$$
For this register we use the notation
$$c.\mathrm{running}(p) = c.m((\mathrm{p}, p, \mathrm{running})).$$

- A binary delayed mode bit which records the mode of the previous configuration
$$(\mathrm{p}, p, \mathrm{dm}) \in A_{NPR,p},$$
$$V((\mathrm{p}, p, \mathrm{dm})) = \mathbb{B}.$$
For this register we use the notation
$$c.\mathrm{p}(p).\mathrm{dm} = c.m((\mathrm{p}, p, \mathrm{dm})).$$

Some special purpose registers have names, in which case we define the name to be an alias for the index of the register in the SPR, such as
$$sr = 0,\ esr = 1, \ldots.$$

Our *emode* register is SPR register 8

$$emode = 8.$$

A processor is in *delayed user mode* when it is in user mode or the delayed mode is user mode

$$dum_p(c) = c.\mathrm{p}(p).\mathrm{core.spr}(mode)[0] \vee c.\mathrm{p}(p).\mathrm{dm}.$$

The memory mode registers of a processor are simply the four bytes of the operating system mode register in the SPR

$$A_{SC,p} = \{ (\mathrm{p}, p, \mathrm{spr}, mode) \},$$

and a processor is in strong memory mode when it is not in user mode

$$SC_p(c) = \neg c.\mathrm{p}(p).\mathrm{core.spr}(mode)[0].$$

3.2.3 Memory Configuration

The main memory is a byte-addressable memory. All addresses have a length of 32 bit. However, we add two special addresses, the purpose of which will be explained shortly.

$$A_{MEM} = \mathbb{B}^{32} \cup \{ wf, rf \}.$$

The ranges of those special addresses do not really matter (as long as they are non-empty), but for the sake of completeness we define them as binary registers

$$V(wf) = V(rf) = \mathbb{B}.$$

For memory addresses $a \in \mathbb{B}^{32}$ we use for the sake of uniform notation the following notation

$$c.\mathrm{m}(a) = c.m(a).$$

3.2.4 Interrupt Command Registers

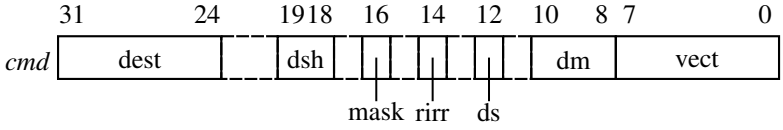


Figure 3.3: An interrupt command register is split into several fields. The dest and dsh fields are used to choose the targets of the IPI. The mask and rirr bits are only used by the IO APIC for masking and controlling device events. The ds bit indicates is raised to send an IPI and lowered by the APIC when the interrupt has been sent. The dm and vect fields are only used by the local APICs and determine the type of IPI to be sent.

Interrupt command registers are used by the IO APIC and the local APICs to determine which interrupts to deliver to which processors. We use the following fields (shown in Fig. 3.3). The *destination ID* contains the APIC ID of the target (if the APIC is in single target mode)

$$cmd.dest = cmd[31:24].$$

The *destination shorthand* distinguishes between four targeting modes: all APICs, all other APICs, self, and single target

$$cmd.dsh = cmd[19:18].$$

The *interrupt mask* (used only by the IO APIC) may prevent the interrupt from being sent

$$cmd.mask = cmd[16].$$

The *remote interrupt request* (used only by the IO APIC) is used as a synchronization mechanism between the IO APIC and the target APICs to prevent redundant sampling of an interrupt that is active in the device but is already being serviced by the target processor

$$cmd.rirr = cmd[14].$$

The *delivery status* indicates whether an interrupt is ready to be sent or not (raised when an interrupt should be sent, lowered when the APIC is idle)

$$cmd.ds = cmd[12].$$

The *delivery mode* (only used by the local APIC) distinguishes between three types of interrupts to be sent: a fixed interrupt vector, an INIT interrupt (for halting and resetting a core to an initial state), and a SIPI interrupt (for starting a halted processor)

$$cmd.dm = cmd[10:8].$$

The IO APIC ignores this field and always sends a fixed interrupt vector.

The *interrupt vector* to be sent in case of a fixed interrupt

$$cmd.vect = cmd[7:0].$$

We define the *targets* of an interrupt sent by the APIC of processor p based on the destination shorthand dsh and a destination APIC ID id as follows

$$q \in targets_p(c, dsh, id) \equiv \begin{cases} q \neq p & dsh = 11 \\ 1 & dsh = 10 \\ q = p & dsh = 01 \\ c.\mathrm{p}(q).\mathrm{apic.APICID} = id & dsh = 00. \end{cases}$$

3.2.5 APIC Configurations

We model APICs as units. For each MIPS processor $p \in [1:P]$, there is an APIC

$$(\mathrm{apic}, p) \in U.$$

APICs do not have processor registers. Instead, each APIC is also a device

$$(\mathrm{apic}, p) \in D$$

and uses the device registers for its components. We make two changes over the Schmaltz ISA. Firstly, we reorder the entries of the interrupt command register to correspond to the entries of the IO APIC one-to-one; secondly, we increase the length of the APIC ID from three to eight bits, therefore matching the length of the APIC ID in the destination register of the APIC and enabling the usage of a load byte to obtain the full APIC ID.

Each APIC has the following registers.

Subregister	Bits	Meaning
EOI	607-576	Signal end of interrupt
APICID	575-544	Unique number of IPI reception
ISR	543-288	Interrupts in service
IRR	287-32	Interrupt requests
ICR	31-0	Interrupt command register

Table 3.1: APIC Memory Map.

- An interrupt command register (ICR) of 32 bits, split into four bytes

$$(\text{apic}, p, \text{ICR})_r \in A_{DEV,(\text{apic},p)}.$$

We use the following notation

$$c.\text{p}(p).\text{apic.ICR} = c.m((\text{apic}, p, \text{ICR})).$$

- A pending end-of-interrupt (EOI) register of 256 bits, split into 32 bytes

$$(\text{apic}, p, \text{EOIpending})_r \in A_{DEV,(\text{apic},p)}.$$

We use the following notation

$$c.\text{p}(p).\text{apic.EOIpending} = c.m((\text{apic}, p, \text{EOIpending})).$$

- An end-of-interrupt register (EOI) of 32 bits, split into four bytes

$$(\text{apic}, p, \text{EOI})_r \in A_{DEV,(\text{apic},p)}.$$

We use the following notation

$$c.\text{p}(p).\text{apic.EOI} = c.m((\text{apic}, p, \text{EOI})).$$

- An APIC ID *register* of 32 bits, split into four bytes (of which only the least significant byte contains the APIC ID)

$$(\text{apic}, p, \text{APICID})_r \in A_{DEV,(\text{apic},p)}.$$

We use the following notation

$$c.\text{p}(p).\text{apic.APICID} = c.m((\text{apic}, p, \text{APICID})).$$

We generate a memory-mapping over the local APIC. This means that all memory accesses to a certain range of addresses will be redirected to registers of the APIC (i.e., of the APIC device and the processor registers IRR and ISR of the corresponding processor). A layout of this memory-mapping is given in Table 3.1.

The memory map of the local APIC starts at address $1^{20}0^{12}$. We define a function mm_p that reroutes accesses to an address $a \in A_{MEM}$ to the memory mapped register corresponding to that address. For the local APIC, the mapping is as follows

$$mm_p(1^{20}0^5 11001 r_2) = (\text{apic}, p, \text{EOI})_r,$$

$$mm_p(1^{20}0^5 11000 r_2) = (\text{apic}, p, \text{APICID})_r,$$
$$mm_p(1^{20}0^5 10 r_5) = (\text{p}, p, \text{ISR})_r,$$
$$mm_p(1^{20}0^5 01 r_5) = (\text{p}, p, \text{IRR})_r,$$
$$mm_p(1^{20}0^5 00000 r_2) = (\text{p}, p, \text{ICR})_r.$$

Note that this mapping is slightly different for each MIPS processor p because each processor only has access to its own local APIC, i.e., the same memory region is used for all local APICs, but memory accesses of processor p to that memory region are redirected to the local APIC of processor p.

We will extend this function when we memory-map the IO APIC.

We give a special name to the base address of the EOI register

$$a_{EOI} = 1^{20}0^5 1100100.$$

3.2.6 Disk Configuration and Controller

We model devices as regular memory ports plus processors (for the device controllers), which will allow us to model hidden device storage (such as swap memory) and non-deterministic device steps.

In this document, we only model a disk similar to that in the System Architecture textbook [PBLS16], but enhanced with an interrupt request bit. The disk model consists of a set of memory-mapped ports and a big swap memory. Data can be exchanged between a page-sized memory-mapped buffer $buff$ and the swap memory by giving read or write commands to the disk through a command-and-status register *csmr*; a write command is issued by raising the second least significant bit, whereas a read command is issued by raising the least significant bit. The page to be loaded or modified is given as a 28-bit address in the swap memory address register *sma*. Completion of the disk operation takes a non-deterministic amount of time, and will raise the interrupt request bit.

The disk is therefore not write-triggered, as the disk is updated non-deterministically when one of the command bits in the command-and-status register is raised, not as an (immediate) side-effect of writing to the command-and-status register. This means we can simply use normal memory addresses for the IO ports (and thus avoid defining a device transition function).

We add a unit

$$\text{disk} \in U$$

with one processor register: a big swap memory

$$\text{sm} \in A_{NPR,\text{disk}},$$

which consists of one terabyte, managed as a four kilobyte-addressable memory

$$V(\text{sm}) = \mathbb{B}^{28} \to \mathbb{B}^{8 \cdot 4K}.$$

We use the following notation

$$c.\text{sm} = c.m(\text{sm}).$$

We also give names to a few memory registers. The buffer occupies the four kilobyte starting at address $1^{18}0^{14}$

$$a_{\text{buff}} = 1^{18}000^80000,$$

the swap memory address occupies the next four byte

$$a_{\text{sma}} = 1^{18}010^{8}0000,$$

and the command and status register the four bytes after that

$$a_{\text{cmsr}} = 1^{18}010^{8}0100.$$

The three first bits of the command and status register correspond to a read command, a write command, and a status bit to signal that a disk operation has completed, and which causes the IO APIC to deliver an interrupt.

The memory region occupied by the disk is then defined as follows

$$A_{\text{disk}} = \{\, a_{\text{buff}} \cdot (4K) \,\} \cup \{\, a_{\text{sma}} \cdot 4 \,\} \cup \{\, a_{\text{cmsr}} \cdot 4 \,\}.$$

3.2.7 IO APIC Configuration

The IO APIC is similar to the normal APICs, except for a few central points.

- The IO APIC is shared between all processors,
- Instead of a single interrupt command register, the IO APIC is programmed through an array of interrupt command registers called the *redirection table*. Each row of the redirection table is an interrupt command register that is connected to a device and lists the destination of interrupts from that device,
- The IO APIC obtains the delivery status from the interrupt request bits of the devices (here: only the disk),
- The IO APIC does not interrupt a processor again until the APIC of that processor has sent an end-of-interrupt signal.

We allow our redirection table to have some arbitrary number of entries, as long as that number is a power of two and all entries of the redirection table can be represented by a word. In other words, there is some number of interrupt devices $ID \leq 2^{32}$ and the redirection table has 2^{ID} entries.

We add the IO APIC as a unit

$$\text{ioapic} \in U$$

and as a device

$$\text{ioapic} \in D.$$

The IO APIC has no processor registers, but has the following device registers.

- A redirection table "redirect" of 2^{ID} entries $d < 2^{ID}$, each of 32 bits, split into four bytes

  $$(\text{ioapic}, \text{redirect}, d)_r \in A_{DEV,\text{ioapic}} \text{ where } d < 2^{ID}.$$

  We use the following notation

  $$c.\text{ioapic.redirect}[d] = c.m((\text{ioapic}, \text{redirect}, d)).$$

  Each redirection table entry works analogous to the command register of the APIC, and could be considered the command register of the APIC of the device. In our model, we have only one device (the disk), which uses the first entry of the redirection table (i.e., entry 0).

- A register selection port (regsel) to select a entry in the redirection table, split into four bytes
$$(\text{ioapic}, \text{regsel})_r \in A_{DEV,(\text{ioapic})}.$$
We use the following notation
$$c.\text{p}(p).\text{ioapic.regsel} = c.m((\text{ioapic}, \text{regsel})).$$
- A so called "win" port to access the selected entry. Split into four bytes
$$(\text{ioapic}, \text{win})_r \in A_{DEV,(\text{ioapic})}.$$
We use the following notation
$$c.\text{p}(p).\text{ioapic.win} = c.m((\text{ioapic}, \text{win})).$$

Only the win and regsel registers are memory mapped

$$mm_p(1^{19}0^{10}0r_2) = (\text{ioapic}, p, \text{regsel})_r,$$
$$mm_p(1^{19}0^{10}1r_2) = (\text{ioapic}, p, \text{win})_r.$$

Intuitively, the IO APIC redirection table works like a web browser with no tabs. The programmer can point his web browser (the WIN register) to one server (a redirection table entry) on the Internet (the redirection table) by entering the web server's address (table index d) into the web browser's URL bar (the REGSEL register). Then the programmer can interact with that server by copying or changing its content (through the WIN register), or change the server he is connected to. Other clients (the IO APIC and the devices) can also connect to the servers independently of the programmer. However, in the case of the redirection table, the hardware grants atomicity: the content of the WIN register and the content of the server (the redirection table entry) are always the same. In Fig. 3.4 we show four example operations of the redirection table.

3.2.8 Memory Mapping

We have already defined how certain registers are memory mapped. We denote by MM the set of memory addresses which have been explicitly mapped, which is the same for all MIPS processors. For all remaining memory addresses $a \in A_{MEM} \setminus MM$ we use the identity mapping

$$mm_p(a) = a.$$

Clearly this function is injective and thus has an inverse mm_p^{-1}. We will later use this to define the prepared writes.

We also lift the function to sets of addresses

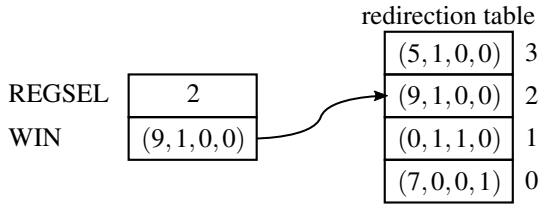
$$mm_p(A) = \left\{ mm_p(a) \;\middle|\; a \in A \right\}.$$

3.2.9 Instructions

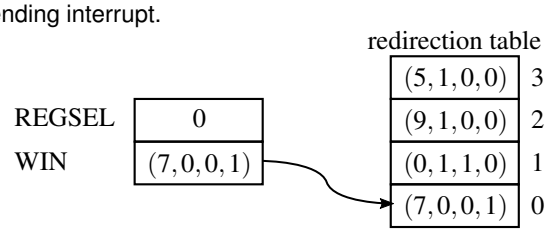
The definition of most functions over instruction words can be found in the Appendix (Section B.6.1). The following functions are added to simplify some definitions.

An instruction is a memory operation when it is a compare-and-swap, store, or load instruction

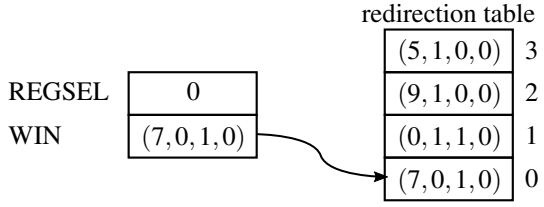
$$mop(I) = cas(I) \vee store(I) \vee load(I).$$



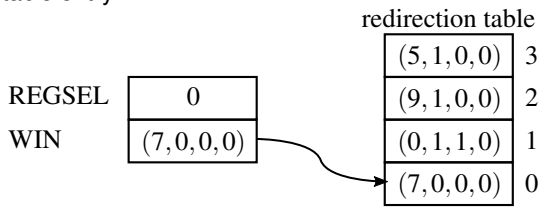
(a) Original configuration of the redirection table. Register REGSEL contains the value 2, and therefore WIN is pointed at the redirection table entry 2. All redirection table entries except for entry 0 are masked. Redirection table entry 0 has a pending interrupt.



(b) A MIPS processor writes a zero into the REGSEL register. As an atomic side-effect of this write, the WIN register is now pointed at the redirection table entry 0.



(c) The IO APIC delivers the pending interrupt of redirection table entry 0, thus clearing the delivery status bit and setting the remote interrupt request bit. Because the WIN register is currently pointed at that redirection table entry, the WIN register is updated as an atomic side-effect to reflect the new value of the redirection table entry.



(d) The interrupted MIPS processor modifies the WIN register to clear the remote irr bit, so that new interrupts can be sampled from the device. Because the WIN register is currently pointed at redirection table entry 0, that entry is updated as an atomic side-effect to reflect the new value of the WIN register.

Figure 3.4: We show four configurations of the redirection table and REGSEL and WIN registers. Registers REGSEL and WIN are on the left, the redirection table with four entries ($ID = 2$) is on the right. The values of redirection table entries and the WIN register are given as four-tuples $(dest, mask, rirr, ds)$, the value of the REGSEL register is given as an index $i \in [0:3]$ in the redirection table.

Each memory access has to be aligned. An address is d-misaligned for $d = 2^l$ if the last l bits are not equal to zero

$$mal_{2^l}(a) \equiv a[l:0] \neq 0^l.$$

Move instructions, exception return, and TLB invalidation instructions are restricted to operating system mode

$$restr(I) = movs2g(I) \vee movg2s(I) \vee eret(I) \vee invalpg(I) \vee flush(I).$$

Several functions were defined in terms of a *MIPS core configuration* (cf. Definition 21 in the Appendix). We show how to extract such a MIPS core configuration from our configuration by defining all components x of the *MIPS core configuration of processor p* as follows

$$core_p(c).x = c.\mathrm{p}(p).\mathrm{core}.x.$$

One easily notices that all such registers are processor registers of processor p, and thus the MIPS core configuration can always be used in steps of processor p to define F, Φ, etc.

We add *write fence* and *read fence* instructions as r-type instructions. The function codes below are somewhat arbitrary, but not occupied

$$wfence(I) \equiv rtype(I) \wedge fun(I) = 111010,$$
$$rfence(I) \equiv rtype(I) \wedge fun(I) = 111001.$$

On a high level, read fence is only executed if all writes that were issued before the last write fence have left the buffer. The implementation in our model is extremely technical, and somewhat abuses partial hits; the gist is that a write fence is buffering a write that has no effect except for causing partial hits during a read fence, consequently draining the buffer until all write fences and all writes before them have been drained. We use the registers $\{wf, rf\}$ to make this work.

3.3 Semantics

In this section we define the semantics of the different types of steps. Each type of steps uses an oracle input of a certain form. We define then the feasibility, the fetched addresses, the guard condition, the read addresses, and finally the prepared writes of the step.

We distinguish between *normal processor steps*, for which oracle step inputs x have the following form

$$x = (\mathrm{p}, \ldots),$$

steps of components such as the MMU and the instruction buffer (also belong to a MIPS processor, but do not have the form above), and steps of other units, such as the APICs and the devices.

Only normal processor steps use the write buffer. For all other steps, we keep the buffered accesses empty

$$x \neq (\mathrm{p}, \ldots) \implies R(c,x).wba = PW(c,x).wba = \emptyset.$$

Since only normal processor steps use the write buffer, other steps normally do not need to be fences and we define for those other steps unless specified otherwise

$$x \neq (\mathrm{p}, \ldots) \implies fence(c,x) = 0.$$

Only two types of steps will be allowed to interrupt other processors: IPI delivery steps of the APIC and of the IO APIC. Unless stated otherwise, we define that set of IPI-relevant registers is empty

$$A_{IPI(x)} = \emptyset.$$

3.3.1 MMU Walk Creation

The oracle input for MMU walk creation has the form

$$x = (\text{tlb}-\text{create}, va, p) \in \Sigma_{P,p}.$$

The step is feasible if the processor is in user mode and running

$$\Phi_P(c,x) \equiv c.\text{p}(p).\text{core.spr}(mode)[0] \wedge c.\text{running}(p).$$

It does not fetch anything

$$F(c,x) = \emptyset.$$

The step is always valid

$$\Upsilon_P(c,x) = 1.$$

The step does not read anything

$$R(c,x).bpa = \emptyset.$$

The step creates a new walk using the page table origin and the address space identifier

$$w = winit(va, c.\text{p}(p).\text{core.spr}(pto).ba, c.\text{p}(p).\text{core.spr}(asid)[5:0]),$$

and prepares to modify the TLB by inserting that new walk (recall that the value of the TLB is a set of walks)

$$PW(c,x).bpa((\text{p}, p, \text{TLB})) = c.\text{p}(p).\text{TLB} \cup w.$$

3.3.2 MMU Accessed Bits

The oracle input for the MMU step that sets accessed bits has the form

$$x = (\text{tlb}-\text{set}-\text{accessed}, w, p) \in \Sigma_{P,p}.$$

As described before, an MMU access in hardware that sets accessed bits and has reached the cache system can be neither canceled nor rolled back. In the case of an interrupt that is triggered after such an access has begun, this means that the processor is switched to operating system mode before the MMU completes the memory access. As a result, the hardware will simulate the MMU step only once the processor is in operating system mode. To model this behavior without giving the MMU the power to fully run in operating system mode, we have introduced the delayed user mode which is stays active after the interrupt but only until the first instruction of the interrupt service routine is executed.

The step is thus feasible if the processor is in delayed user mode and running, the walk w is from the TLB, incomplete, and matches the current address space identifier

$$\Phi_P(c,x) \equiv dum_p(c) \wedge c.\text{running}(p)$$

$$\wedge\, w \in c.\mathrm{p}(p).\mathrm{TLB} \wedge \neg \mathit{complete}(w) \wedge w.\mathit{asid} = c.\mathrm{p}(p).\mathrm{core.spr}(\mathit{asid})[5:0].$$

It does not fetch anything

$$F(c,x) = \emptyset$$

and is always valid

$$\Upsilon_P(c,x) = 1.$$

Note that this means that the step is valid even when the page table entry is not present, unlike in the ISA given by Schmaltz.

The step reads the page table entry

$$R(c,x) = \{\, \mathit{ptea}(w) \cdot 4 \,\},$$

and thus obtains the memory content of the page table entry

$$\mathit{pte} = c.\mathrm{m}(\mathit{ptea}(w) \cdot 4).$$

The step then raises the accessed bit (bit 7)

$$\mathit{PW}(c,x).\mathit{bpa}(\mathit{ptea}(w) \cdot 4) = \mathit{pte}[31:8]1\mathit{pte}[6:0].$$

Unlike in the ISA given by Schmaltz, we set the accessed bit even for page table entries that are not present. This is a side-effect of the fact that we wish to put faulty walks into the TLB, which clearly depend on the page table entry.

3.3.3 MMU Walk Extension

The oracle input for MMU walk creation has the form

$$x = (\mathrm{tlb{-}extend}, w, p) \in \Sigma_{P,p}.$$

The step is feasible if the processor is in user mode and running, the walk w is from the tlb and incomplete, and matches the current address space identifier

$$\Phi_P(c,x) \equiv c.\mathrm{p}(p).\mathrm{core.spr}(\mathit{mode})[0] \wedge c.\mathrm{running}(p)$$
$$\wedge\, w \in c.\mathrm{p}(p).\mathrm{TLB} \wedge \neg \mathit{complete}(w) \wedge w.\mathit{asid} = c.\mathrm{p}(p).\mathrm{core.spr}(\mathit{asid})[5:0].$$

It fetches the page table entry[2]

$$F(c,x) = \{\, \mathit{ptea}(w) \cdot 4 \,\},$$

and thus obtains the memory content of the page table entry

$$\mathit{pte} = c.\mathrm{m}(\mathit{ptea}(w) \cdot 4).$$

The step is valid if the page table entry has been accessed

$$\Upsilon_P(c,x) = \mathit{pte}.a.$$

[2]It is necessary to fetch the page table entry, rather than simply reading from it, since we want to make validity dependent on the entry, and validity only depends on what we have fetched, not on what we have read.

Note that it is important that we fetch the page table entry, rather than simply reading it: if we would not fetch the page table entry, we would not be able to make the step invalid in case it has not been accessed.

The step does not read anything

$$R(c,x).bpa = \emptyset.$$

The step creates a new walk using the page table origin and the address space identifier

$$w = wext(va, pte),$$

and prepares to modify the TLB by inserting that new walk

$$PW(c,x).bpa((\mathrm{p},p,\mathrm{TLB})) = c.\mathrm{p}(p).\mathrm{TLB} \cup w.$$

3.3.4 Instruction Buffering

While the processor is in user mode, the instruction buffer may speculatively fetch instructions and put them into the instruction buffer; but only if there is a valid translation for the instruction address.

The oracle input for instruction buffering has the form

$$x = (\mathrm{ib}, w, ia, p) \in \Sigma_{P,p}.$$

The step is feasible if the processor is in user mode and running, the walk w is from the tlb and complete, has fetching rights, and matches the current address space identifier and instruction address

$$\begin{aligned}\Phi_P(c,x) \equiv\ & c.\mathrm{p}(p).\mathrm{core.spr}(mode)[0] \wedge c.\mathrm{running}(p) \\ & \wedge w \in c.\mathrm{p}(p).\mathrm{TLB} \wedge complete(w) \\ & \wedge w.r.f \wedge w.r.u \wedge w.asid = c.\mathrm{p}(p).\mathrm{core.spr}(asid)[5:0] \wedge w.r.va = ia.ba.\end{aligned}$$

It fetches the given instruction

$$F(c,x) = \{\, pma(w,ia) \cdot 4 \,\},$$

and thus obtains an instruction word

$$I = c.\mathrm{m}(pma(w,ia) \cdot 4).$$

The step is valid

$$\Upsilon_P(c,x) = 1.$$

The step does not read anything

$$R(c,x).bpa = \emptyset.$$

The step prepares to put the pair of address and instruction into the instruction buffer

$$PW(c,x).bpa((\mathrm{p},p,\mathrm{IB})) = c.\mathrm{p}(p).\mathrm{IB} \cup (ia, I).$$

3.3.5 Normal Processor Steps

Oracle inputs for normal processor steps have the form

$$x = (\mathrm{p}, p, I' \in \mathbb{B}^{32}, w_I \in K_{walk}, w_E \in K_{walk}) \in \Sigma_{P,p}.$$

The idea is that for user steps, the oracle input must either provide a faulting instruction walk w_I or an instruction I' from the instruction buffer. If an instruction in user mode is a memory operation, then the execute walk w_E must be from the TLB and either faulting or complete. For operating system steps and steps that do not make use of the instruction, the provided parameters are irrelevant.

This gives nine possible feasible steps (shown also in the flowchart of Fig. 3.5):

1. An unmasked external interrupt. The step goes directly to the interrupt service routine. Oracle inputs I', w_I, w_E do not matter.

2. A misaligned instruction address is used. The step goes directly to the interrupt service routine. Oracle inputs I', w_I, w_E do not matter.

3. An operating system step. The instruction is fetched directly from memory and is executed normally. Oracle inputs I', w_I, w_E do not matter.

4. A user mode step where w_I is a faulting walk from the TLB that matches the fetch access (i.e., matches the instruction address and the address space identifier). Such steps are page faults and thus jump to the interrupt service routine. Oracle inputs I', w_E do not matter.

5. A user mode step where I' is an instruction from the instruction buffer and matches the instruction address, and some other interrupt (e.g., a misaligned effective address interrupt in case of a memory operation) occurs. In case of a continue interrupt (which are never memory operations), the instruction I' is executed normally. In any case, the step jumps to the interrupt service routine. Oracle inputs w_I, w_E do not matter.

6. A user mode step where I' is an instruction from the instruction buffer and matches the instruction address, and is not a memory operation. The instruction I' is executed normally. Oracle inputs w_I, w_E do not matter.

7. A user mode step where I' is an instruction from the instruction buffer and matches the instruction address, and is a memory operation, and where w_E is a complete walk from the TLB that matches the memory access performed by instruction I' (e.g., if it is a write, the walk has user and write permissions and matches the effective address and the address space identifier). The instruction I' is executed normally. Oracle input w_I does not matter.

8. A user mode step where I' is an instruction from the instruction buffer and matches the instruction address, and is a memory operation, and where w_E is a faulting walk from the TLB that matches the memory access (i.e., matches the effective address and the address space identifier). Such steps are page faults and thus jump to the interrupt service routine. Oracle input w_I does not matter.

The instruction address of the step is the ddpc

$$ia = c.\mathrm{p}(p).\mathrm{core.ddpc}.$$

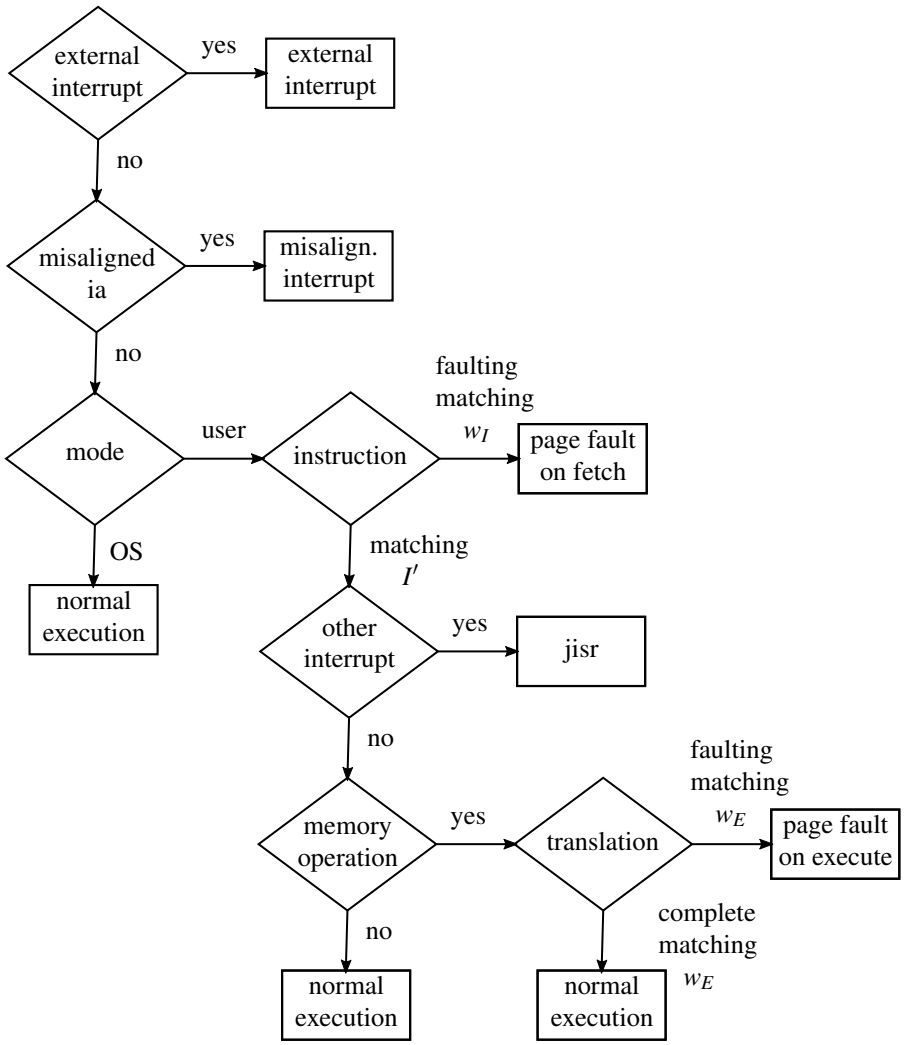


Figure 3.5: A flowchart detailing the eight possible feasible steps. A step is feasible if it can take a branch at each decision node (diamond) of the flow chart and ends up in one of the eight leaf nodes (rectangle). Steps that "get stuck" at a decision node are infeasible, e.g., a user step where neither w_I is a faulting matching walk nor I' is a matching instruction. Such steps do not occur in sane schedules and are never simulated by the hardware. Nodes on the bottom correspond to normal instruction execution, nodes on the right correspond to interrupted (but feasible) steps that do not execute an instruction (except for the jisr node, which may execute an instruction in a continue interrupt). In one case two branches are valid, i.e., when I' matches the instruction address and w_I is a faulting walk that matches the instruction address. In this case we follow the downwards-pointing branch.

We abbreviate the user mode

$$user = c.\mathrm{p}(p).\mathrm{core.spr}(mode)[0]$$

Interrupt	Index	Shorthand
reset	0	*reset*
external	1	e
misalignment on fetch	2	$malf$
page fault on fetch	3	pff
illegal instruction	4	ill
misalignment on execute	5	$male$
page fault on execute	6	pfe
system call	7	$sysc$
overflow	8	ovf

Table 3.2: List of interrupts.

and the address space identifier

$$asid = c.\mathrm{p}(p).\mathrm{core.spr}(asid)[5:0].$$

The given instruction I' is matching when the buffer maps the instruction address to that instruction

$$Imatch = (ia, I') \in c.\mathrm{p}(p).\mathrm{IB}.$$

As can be seen from the flow chart (Fig. 3.5), when we are in user mode and we are not given a matching I' from the instruction buffer, the step is only valid if we have a page fault on fetch. In this case we say that we *expect a page fault on fetch* and write $epff$

$$epff = user \wedge \neg Imatch.$$

The external event vector is given by the interrupt request vector. However, interrupts that are currently being serviced (i.e., have a bit raised in the in-service register) mask all interrupt requests of interrupts with the same or lower priority

$$eev_l = c.\mathrm{p}(p).\mathrm{apic.IRR}_l \wedge \nexists l' \leq l.c.\mathrm{p}(p).\mathrm{apic.ISR}_{l'}.$$

We partially define the masked cause vector of interrupts, introducing misalignment on fetch, and increasing the priority of page faults on fetch. An updated list of interrupts is found in Table 3.2

We assume that configuration zero is the configuration after reset, and during the rest of the execution reset is off

$$mca_{reset} = 0$$

There is an external interrupt when the status register is not masking external interrupts and the external event vector (taken from the IRR and the ISR) indicates an external event

$$mca_e = c.\mathrm{p}(p).\mathrm{core.spr}(sr)[1] \wedge \exists l.eev_l.$$

There is a misalignment on fetch when the instruction address is not word-aligned

$$mca_{malf} = mal_2(ia).$$

There is a page fault on fetch when we expect a page fault on fetch (thus prioritizing I' over w_I in case both could be used for the step, cf. Fig. 3.5) and the instruction walk is faulty or does not have sufficient permissions

$$mca_{pff} = epff \wedge (w_I.fault \vee \neg w_I.r.f \vee \neg w_I.r.u).$$

The given instruction walk is valid when there is a page fault on fetch and the walk matches the instruction address

$$wIvalid = mca_{pff} \wedge ia.ba = w_E.va \wedge w_I.asid = asid.$$

For instructions in user mode, we can already define some of the other interrupts

$$\begin{aligned} mca'_{ill} &= ill(I') \vee restr(I'), \\ mca'_{male} &= mop(I') \wedge mal_{d(I')}(ea(core_p(c), I')), \\ mca'_{pfe} &= mop(I') \wedge (w_E.fault \vee \neg w_E.r.w \vee \neg w_E.r.u). \end{aligned}$$

The given walk for execution is valid when the instruction is a memory operation, the walk matches the effective address, and produces a page fault or is complete

$$\begin{aligned} wEvalid \equiv mop(I') \wedge ea(core_p(c), I').ba = w_E.va \wedge w_E.asid = asid \\ \wedge (mca'_{pfe} \vee complete(w_E)). \end{aligned}$$

We encode the flowchart with predicate FC as follows

$$\begin{aligned} FC(c,x) \equiv mca_e \vee mca_{malf} \vee \neg user \vee wIvalid \\ \vee Imatch \wedge (mca'_{ill} \vee mca'_{male} \vee \neg mop(I') \\ \vee wEvalid) \end{aligned}$$

and say that the step is feasible if the processor is running and is feasible according to the flowchart

$$\Phi_P(c,x) \equiv c.\text{running}(p) \wedge \Phi'_P(c,x).$$

A normal processor step only has to fetch in operating system mode (otherwise, we use the instruction I' or have an interrupt), and when there are no interrupts preventing us from fetching. When we fetch, we fetch from the instruction address. We need to be a little careful with the memory mapping. We wrap the memory mapping around all addresses that go to the main memory

$$F(c,x) = \begin{cases} \emptyset & user \vee mca_e \vee mca_{malf} \\ mm_p(\{ia \cdot 4\}) & \text{o.w.} \end{cases}$$

After fetching, we can define the instruction to be executed as either the instruction given by the oracle input (for user mode steps), or the fetched instruction. In case no instruction was fetched due to an interrupt, the instruction does not matter and we default to I'

$$I = \begin{cases} I' & user \vee mca_e \vee mca_{malf} \\ c.\text{m}(ia \cdot 4) & \text{o.w.} \end{cases}$$

We can now define the remainder of the cause vector, filling all bits behind the overflow bit with zeros

$$\begin{aligned} mca_{ill} &= ill(I) \vee user \wedge restr(I), \\ mca_{male} &= mop(I) \wedge mal_{d(I)}(ea(core_p(c), I)), \\ mca_{pfe} &= user \wedge mop(I) \wedge (w_E.fault \vee \neg w_E.r.w \vee \neg w_E.r.u), \\ mca_{sysc} &= sysc(I), \end{aligned}$$

$$mca_{ovf} = ovf_p(c,I),$$
$$mca_l = 0. \qquad l \in (ovf : 32)$$

Note that the instruction may be meaningless in case there is a page fault or in case the instruction address is misaligned. In that case, all cause bits involving the instruction are meaningless, too. We blend out all interrupt bits except for the one with the highest priority. We call these the *normalized cause vector*

$$nca_l = mca_l \wedge \nexists l' < l.mca_{l'}.$$

The step jumps to the interrupt service routine if any bit in the vector is raised

$$jisr = \exists l.nca_l.$$

Compare-and-swap, memory fences, and the jump to the interrupt service routine are fences

$$fence(c,x) = cas(I) \vee mfence(I) \vee jisr.$$

The physical effective address is obtained in user mode by using the translation w_E, and in operating system mode by simply taking the effective address directly

$$pea = \begin{cases} pma(w_E, ea(core_p(c),I)) & user \\ ea(core_p(c),I) & \text{o.w.} \end{cases}$$

When the instruction is a compare-and-swap or a load instruction, and there is no interrupt, the instruction reads $d(I)$ bytes starting at the physical effective address through the buffer. In case of a read fence, we read the two special registers rf and wf

$$R(c,x).wba = \begin{cases} mm_p(\{\, pea \cdot d(I) \,\}) & \neg jisr \wedge (cas(I) \vee load(I)) \\ \{\, rf, wf \,\} & \neg jisr \wedge rfence(I) \\ \emptyset & \text{o.w.} \end{cases}$$

The step does not bypass the buffer while reading

$$R(c,x).bpa = \emptyset.$$

We define as the *load result* the memory at the read address

$$lres = (c.m \circ mm_p)(pea \cdot d(I)).$$

To deal with the memory mapping for the prepared writes, we define later an auxiliary function PW' that ignores the memory mapping, and then put the memory mapping in front of that function

$$PW(c,x).X = PW'(c,x).X \circ mm_p^{-1}.$$

Only uninterrupted store instructions buffer writes. Normally, we buffer the store value; but if the physical effective address is the memory mapped address of the EOI register (and we are storing four words), we use the binary encoding of the highest-priority interrupt currently being serviced.

In this case we are signaling an *end-of-interrupt*

$$eoi \equiv pea = a_{EOI} \wedge d(p) = 4.$$

The *ending interrupt* is the interrupt with the highest priority currently being serviced

$$ei = \min\{\, l \mid c.\mathrm{p}(p).\mathrm{apic}.\mathrm{ISR}_l \,\}.$$

The buffered prepared writes are then easily defined as follows

$$\neg jisr \wedge store(I) \rightarrow PW'(c,x).wba(pea \cdot d(I)) = \begin{cases} ei_{32} & eoi \\ sv(core_p(c),I) & \text{o.w.} \end{cases}$$

A write fence buffers a write to the special register wf

$$\neg jisr \wedge wfence(I) \rightarrow PW'(c,x).wba(wf) = 1.$$

One can now easily see the interplay between write fences, other writes, and read fences. A write fence enters the buffer and modifies wf; a read fence reads from the buffer and attempts to read both wf and rf, causing a partial hit until the write fence and all writes before it leave the buffer.

If there is an interrupt, we distinguish between *continue interrupts* and other interrupts. System call and overflow are the continue interrupts

$$continue = nca_{sysc} \vee nca_{ovf}.$$

We define the *intermediate core configuration* (called c^* in [Zah16]) either as the core configuration reached by executing the instruction (using the transition function δ_{instr} of Definition 49 in the appendix) if there was no interrupt or in case of a continue interrupt, or as the core configuration before the step

$$c_I = \begin{cases} \delta_{instr}(core_p(c),I,lres) & \neg jisr \vee continue \\ core_p(c) \end{cases}$$

The processor returns from exception if there is no interrupt and the instruction is eret

$$rfe = \neg jisr \wedge eret(I).$$

The delayed mode after the step is the mode before the step

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{dm})) = c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(mode)[0].$$

We define the remainder of the prepared bypassing writes by case distinction on whether there was a jump to the interrupt service routine, a return from exception, or neither.

$\neg jisr \wedge \neg rfe$: If there is no interrupt, we copy the pc into the dpc and the dpc into the ddpc

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{dpc})) = c.\mathrm{p}(p).\mathrm{code}.\mathrm{pc},$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{ddpc})) = c.\mathrm{p}(p).\mathrm{code}.\mathrm{dpc}.$$

All other core components are taken verbatim from the intermediate configuration

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{gpr},x)) = c_I.gpr(x),$$

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},x)) = c_I.spr(x),$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{pc})) = c_I.pc.$$

In case of an invalidating instruction, the TLB is updated accordingly. For flush, this means the TLB becomes empty.

$$flush(I) \rightarrow PW'(c,x).bpa((\mathrm{p},p,\mathrm{TLB})) = \emptyset$$

For invalpg, we obtain the invalidated base address from the *rd* register

$$invalpga = c.\mathrm{p}(p).\mathrm{core}.\mathrm{gpr}(rd(I)).ba$$

and the invalidated address space from the *rs* register

$$invalpgasid = c.\mathrm{p}(p).\mathrm{core}.\mathrm{gpr}(rs(I))[5:0].$$

A walk is then deleted if it matches the address and ASID

$$del(w) \equiv w.va = invalpga \wedge w.asid = invalpgasid.$$

The remaining TLB are all walks that are complete and not deleted

$$tlb' = \{\, w \in c.\mathrm{p}(p).\mathrm{TLB} \mid complete(w) \wedge \neg del(w) \,\}.$$

Only walks that are complete and not deleted remain

$$invalpg(I) \rightarrow PW'(c,x).bpa((\mathrm{p},p,\mathrm{TLB})) = tlb'.$$

A compare-and-swap is successful if the load result (which by definition has four bytes in the case of a cas instruction) equals the value of the GPR register indicated by the *rd* field

$$scas \equiv cas(I) \wedge lres = c.\mathrm{p}(p).\mathrm{core}.\mathrm{gpr}(rd(I)).$$

In case of a successful compare-and-swap, we modify the memory by writing the store value to the physical effective address

$$scas \rightarrow PW'(c,x).bpa(pea \cdot d(I)) = sv(core_p(c),I).$$

One complication is the fact that writes to the EOI register of the APIC should change the in-service register.

jisr: In the case of a jump to the interrupt service routine, we reset program counters to the beginning of the interrupt service routine

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{ddpc})) = 0_{32},$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{dpc})) = 4_{32},$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{pc})) = 8_{32},$$

storing the program counters of the intermediate core configuration in the exception registers

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},eddpc)) = c_I.ddpc,$$

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},edpc)) = c_I.dpc,$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},epc)) = c_I.pc.$$

We switch to operating system mode, storing the mode of the intermediate core configuration in the emode register

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},mode)) = 0_{32},$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},emode)) = c_I.spr(mode).$$

We mask all interrupts, and store the previous mask in the exception register.

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{sr})) = 0_{32},$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{esr})) = c_I.spr(sr).$$

The exception cause is taken from the normalized cause

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{eca})) = nca.$$

In the case of a page fault on execute, the exception data is the (virtual) effective address

$$nca_{pfe} \rightarrow PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},edata)) = ea(core_p(c),I).$$

In the case of an external interrupt, the exception data data is the binary encoding of the lowest active index in the external event vector

$$nca_e \rightarrow PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},edata)) = (\min\{\, l \mid eev_l \,\})_{32}.$$

The GPR is taken from the intermediate configuration

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{gpr},x)) = c_I.gpr(x).$$

The instruction buffer is cleared

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{IB})) = \emptyset.$$

To be consistent with the ISA of Lutsyk [Lut] we do not invalidate any walks in the TLB on jisr. This is different from the ISA of Schmaltz, which invalidates walks corresponding to a faulty address in case of a page fault. This design decision makes the implementation of the processor slightly simpler at the cost of having to do one additional (very fast) invalpg instruction in case of a page fault, but does not have any deeper reason.

Finally, in case of an external interrupt, the interrupt request and in-service registers are changed. The lowest active bit of the interrupt request register

$$k = \min\{\, l \mid c.\mathrm{p}(p).\mathrm{apic}.\mathrm{IRR}_l \,\}$$

is moved from the interrupt request register to the in-service register

$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{IRR}))_l = c.\mathrm{p}(p).\mathrm{apic}.\mathrm{IRR}_l \wedge (l \neq k),$$
$$PW'(c,x).bpa((\mathrm{p},p,\mathrm{ISR}))_l = c.\mathrm{p}(p).\mathrm{apic}.\mathrm{ISR}_l \vee (l = k).$$

rfe: The exception registers are restored

$$\begin{aligned} PW'(c,x).bpa((\mathrm{p},p,\mathrm{ddpc})) &= c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(eddpc),\\ PW'(c,x).bpa((\mathrm{p},p,\mathrm{dpc})) &= c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(edpc),\\ PW'(c,x).bpa((\mathrm{p},p,\mathrm{pc})) &= c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(epc),\\ PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},mode)) &= c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(mode),\\ PW'(c,x).bpa((\mathrm{p},p,\mathrm{spr},sr)) &= c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(esr). \end{aligned}$$

We are returning from an external interrupt if the corresponding bit of the exception cause is raised

$$ree = c.\mathrm{p}(p).\mathrm{core}.\mathrm{spr}(eca)_e,$$

in which case the lowest active bit of the in-service register

$$k = \min\{\, l \mid c.\mathrm{p}(p).\mathrm{apic}.\mathrm{ISR}_l \,\}$$

is quenched

$$ree \rightarrow PW'(c,x).bpa((\mathrm{p},p,\mathrm{ISR}))_l = c.\mathrm{p}(p).\mathrm{apic}.\mathrm{ISR}_l \wedge (l \neq k).$$

3.3.6 IPI Delivery

The oracle input for IPI delivery has the form

$$x = (\mathrm{apic}{-}\mathrm{sendIPI}, p) \in \Sigma_{P,(\mathrm{apic},p)}.$$

The IPI-relevant registers for this step are the interrupt command register, and the APIC IDs of all other processors

$$A_{IPI(x)} = \{\, (\mathrm{apic},p,\mathrm{ICR}) \,\} \cup \bigcup_q \{\, (\mathrm{apic},q,\mathrm{APICID}) \,\}.$$

The step is always feasible

$$\Phi_P(c,x) \equiv 1.$$

It fetches the interrupt command register

$$F(c,x) = \{\, (\mathrm{apic},p,\mathrm{ICR}) \,\},$$

yielding the *command*

$$cmd = c.\mathrm{p}(p).\mathrm{apic}.\mathrm{ICR}.$$

The step is valid if the command has an active delivery status

$$\Upsilon_P(c,x) = cmd.ds.$$

Note that we ignore the destination mode since only physical destination mode is supported anyways.

The step reads the interrupt request registers and APIC ids of all other processors

$$R(c,x).bpa = \bigcup_q \{\, (\mathrm{p},q,\mathrm{IRR}), (\mathrm{apic},q,\mathrm{APICID}) \,\}.$$

We can now compute the *targets* of the interrupt

$$t = targets_p(c, cmd.dsh, cmd.dest).$$

The step lowers its own delivery status (bit 12)

$$PW(c,x).bpa((\text{apic}, p, \text{ICR})) = cmd[31:13]0cmd[11:0].$$

We distinguish between the three categories of interrupts. A *fixed* interrupt vector is sent when the delivery mode is 000

$$fixed \equiv cmd.dm = 000.$$

An INIT interrupt is sent when the delivery mode is 101

$$init \equiv cmd.dm = 101.$$

A SIPI interrupt is sent when the delivery mode is 110

$$sipi \equiv cmd.dm = 110.$$

In case of a fixed interrupt, the step raises the interrupt bit in the interrupt request registers corresponding to the sent vector

$$fixed \wedge q \in t \rightarrow PW(c,x).bpa((\text{p}, q, \text{IRR}))_v \equiv c.\text{p}(q).\text{apic.ICR} \vee v = \langle cmd.vect \rangle.$$

In case of an INIT interrupt, the step raises the init request bit of the target processor

$$init \wedge q \in t \rightarrow PW(c,x).bpa((\text{p}, q, \text{initrr})) = 1,$$

and analogously for a SIPI interrupt

$$sipi \wedge q \in t \rightarrow PW(c,x).bpa((\text{p}, q, \text{sipirr})) = 1.$$

In case of a SIPI interrupt, however, the SIPI-vector is also updated to the vector given by the command register.

$$sipi \wedge q \in t \rightarrow PW(c,x).bpa((\text{p}, q, \text{sipivect})) = cmd.vect.$$

One easily checks that those predicates – and thus whether processor q is interrupted or not – indeed only depend on the IPI-relevant registers.

3.3.7 EOI Delivery

The oracle input for EOI delivery has the form

$$x = (\text{apic}-\text{sendEOI}, p) \in \Sigma_{P,(\text{apic},p)}.$$

The step is always feasible

$$\Phi_P(c,x) \equiv 1.$$

It fetches the interrupt command register

$$F(c,x) = \{\, (\text{apic}, p, \text{EOIpending}) \,\},$$

and is valid if there are pending EOI signals

$$\Upsilon_P(c,x) = \exists l.c.\mathrm{p}(p).\mathrm{apic.EOIpending}_l.$$

In this case we define the *ending interrupt* as the interrupt with highest priority among them

$$ei = \min\{\, l \mid c.\mathrm{p}(p).\mathrm{apic.EOIpending}_l \,\}.$$

The step reads the complete redirection table

$$R(c,x).bpa = \left\{\, (\mathrm{ioapic}, \mathrm{redirect}, d) \;\middle|\; d < 2^{ID} \,\right\},$$

yielding redirection entries

$$red_d = c.\mathrm{ioapic.redirect}[d],$$

and lowers the remote interrupt request signal (bit 14) if the ending interrupt matches the vector in the redirection table

$$red_d.vect = ei_{32} \rightarrow PW(c,x).bpa((\mathrm{ioapic}, \mathrm{redirect})) = red_d[31:15]0red_d[13:0].$$

In any case, it lowers the pending signal of the ending interrupt

$$PW(c,x).bpa((\mathrm{apic}, p, \mathrm{EOIpending}))_v = c.\mathrm{p}(p).\mathrm{apic.EOIpending}_v \wedge v \neq ei.$$

3.3.8 INIT Interrupt

The oracle input for INIT interrupts has the form

$$x = (\mathrm{apic{-}initCore}, p) \in \Sigma_{P,p}.$$

The step is feasible if the processor has a pending INIT request

$$\Phi_P(c,x) \equiv c.\mathrm{p}(p).\mathrm{apic.initrr}.$$

It fetches nothing

$$F(c,x) = \emptyset$$

and is always valid

$$\Upsilon_P(c,x) = 1.$$

Unlike in the Schmaltz ISA, we define that INIT Interrupts act as a fence

$$fence(c,x) = 1.$$

Otherwise, one can not send an INIT interrupt to a core in strong memory mode without violating the order between buffered writes and shared writes.

The step does not read anything

$$R(c,x).bpa = \emptyset.$$

The step prepares to clear the INIT request and lowers the running bit

$$PW(c,x).bpa((\mathrm{p}, p, \mathrm{initrr})) = 0,$$
$$PW(c,x).bpa((\mathrm{p}, p, \mathrm{running})) = 0.$$

Unlike the Schmaltz ISA, we do not modify the remaining core registers until the SIPI.

3.3.9 SIPI Interrupt

The oracle input for SIPI interrupts has the form

$$x = (\text{apic}-\text{startCore}, p) \in \Sigma_{P,p}.$$

The step is feasible if the processor has a pending SIPI request and is not running

$$\Phi_P(c,x) \equiv c.\text{p}(p).\text{apic.sipirr} \wedge \neg c.\text{running}(p).$$

It fetches nothing

$$F(c,x) = \emptyset$$

and is always valid

$$\Upsilon_P(c,x) = 1.$$

The step does not read anything

$$R(c,x).bpa = \emptyset.$$

The step prepares to clear the SIPI request and raises the running bit, resets the program counters according to what is given in the SIPI vector, switches to operating system mode, and sets the cause register to a state corresponding to a reset (least significant bit raised, other bits low)

$$\begin{aligned}
PW(c,x).bpa((\text{p},p,\text{sipirr})) &= 0,\\
PW(c,x).bpa((\text{p},p,\text{running})) &= 1,\\
PW(c,x).bpa((\text{p},p,\text{ddpc})) &= c.\text{p}(p).\text{apic.sipivect} \circ 0_{24},\\
PW(c,x).bpa((\text{p},p,\text{dpc})) &= c.\text{p}(p).\text{apic.sipivect} \circ 4_{24},\\
PW(c,x).bpa((\text{p},p,\text{pc})) &= c.\text{p}(p).\text{apic.sipivect} \circ 8_{24},\\
PW(c,x).bpa((\text{p},p,\text{spr},mode)) &= 0_{32},\\
PW(c,x).bpa((\text{p},p,\text{spr},eca)) &= 1_{32}.
\end{aligned}$$

Because a SIPI interrupt changes the mode registers, our software discipline will require that the write buffers are empty at the beginning of a SIPI step. Note that a SIPI interrupt can only be taken while the running bit is low. The running bit is only low after reset and after taking an INIT interrupt. In each case the write buffers are empty.

3.3.10 Disk Steps

Steps that indicate steps of the disk have the form

$$x = \text{d} \in \Sigma_{P,\text{disk}}.$$

Disk steps are always feasible

$$\Phi(c,x) = 1.$$

They fetch the command and status register and the swap memory address

$$F(c,x) = \{\, a_{\text{cmsr}} \cdot 4 \,\} \cup \{\, a_{\text{sma}} \cdot 4 \,\}.$$

The step is a read if the first bit of the command and status register is raised

$$hdr \equiv c.\mathrm{m}(a_{\mathrm{cmsr}} \cdot 4)[0]$$

and a write if the second bit is raised and the step is not a read (thus giving precedence to reads in case both bits are raised)

$$hdw \equiv c.\mathrm{m}(a_{\mathrm{cmsr}} \cdot 4)[1] \wedge \neg hdr.$$

The *swap page address* are the lower 28 bits of the swap memory address

$$spa = c.\mathrm{m}(a_{\mathrm{sma}} \cdot 4)[27:0].$$

The step is valid if it is a read or a write

$$\Upsilon(c,x) \equiv hdr \vee hdw.$$

In case of a write, the step reads the buffer (so it can write it into swap memory); a read step reads nothing

$$R(c,x).bpa = \begin{cases} \{\, a_{\mathrm{buff}} \cdot (4K) \,\} & hdw \\ \emptyset & \text{o.w.} \end{cases}$$

In case of a write, the step copies the memory into the page corresponding to the swap memory address. For this we define a new swap memory

$$sm(sma) = \begin{cases} c.\mathrm{m}(a_{\mathrm{buff}} \cdot (4K)) & sma = spa \\ c.\mathrm{sm}(sma) & \text{o.w.}, \end{cases}$$

which we prepare to set as the new swap memory

$$hdw \rightarrow PW(c,x).bpa(\mathrm{sm}) = sm$$

In case of a read, we instead copy the page from the swap memory to the buffer

$$hdr \rightarrow PW(c,x).bpa(a_{\mathrm{buff}} \cdot (4K)) = c.\mathrm{sm}(spa).$$

In each case, the step changes the state of the disk to idle by lowering the least two bits and raising the status bit

$$PW(c,x).bpa(a_{\mathrm{cmsr}} \cdot 4) = 0^{29}100.$$

3.3.11 Sampling Device Interrupts

The oracle input for sampling the interrupt request of the disk has the form

$$x = (\mathrm{ioapic{-}sample}, d) \in \Sigma_{P,\mathrm{ioapic}}.$$

The step is feasible if redirection table entry d is connected to a device, which in this case is true only when $d = 0$

$$\Phi_P(c,x) \equiv d = 0.$$

It fetches the redirection table entry for that device

$$F(c,x) = \{\, a_{\text{cmsr}} \cdot 4, (\text{ioapic}, \text{redirect}, d) \,\}$$

and thus obtains the redirection entry, which as mentioned before acts as an interrupt command register

$$cmd = c.\text{ioapic.redirect}[d].$$

The step is valid if the disk has completed an operation (and thus bit 2 of the disk's command and status register — the status bit — is set), the interrupt is not masked in the redirection table, and there is no remote interrupt request[3]

$$\Upsilon_P(c,x) = c.\text{m}(a_{\text{cmsr}} \cdot 4)[2] \wedge \neg cmd.mask \wedge \neg cmd.rirr.$$

The step does not read anything

$$R(c,x).bpa = \emptyset.$$

The step sets the delivery status (bit 12) to pending

$$PW(c,x).bpa((\text{ioapic}, \text{redirect}, 0)) = cmd[31:13]1cmd[11:0].$$

3.3.12 Delivering Device Interrupts

The delivery of device interrupts is essentially identical to the delivery of normal interrupts by local APICs, except that a) the command register is taken from the redirection table and b) the delivery mode of the command register is ignored, and the remote interrupt request is raised.

The oracle input for device interrupt delivery has the form

$$x = (\text{ioapic}-\text{deliver}) \in \Sigma_{P,\text{ioapic}}.$$

The IPI-relevant registers for this step are the disk's redirection table entry, and the APIC IDs of all other processors

$$A_{IPI(x)} = \{\, (\text{ioapic}, \text{redirect}, 0) \,\} \cup \bigcup_q \{\, (\text{apic}, q, \text{APICID}) \,\}.$$

The step is always feasible

$$\Phi_P(c,x) \equiv 1.$$

It fetches the redirection table entry

$$F(c,x) = \{\, (\text{ioapic}, \text{redirect}, 0) \,\},$$

yielding the *command*

$$cmd = c.\text{ioapic.redirect}[0].$$

The step is valid if the command has an active delivery status

$$\Upsilon_P(c,x) = cmd.ds.$$

[3]To keep things simple for now, we use the command-and-status register of the disk directly. If there were more devices, one would have to use a more flexible scheme for sampling device interrupts, e.g., by using a fixed map of addresses $a_{\text{cmsr}}(d)$.

Note that we ignore the destination mode since only physical destination mode is supported anyways.

The step reads the interrupt request registers and APIC ids of all other processors

$$R(c,x).bpa = \bigcup_q \{\,(\mathrm{p},q,\mathrm{IRR}),(\mathrm{apic},q,\mathrm{APICID})\,\}.$$

We can now compute the *targets* of the interrupt

$$t = targets_p(c, cmd.dsh, cmd.dest).$$

The step lowers its own delivery status (bit 12) and raises the remote interrupt request (bit 14)

$$PW(c,x).bpa((\mathrm{ioapic},\mathrm{redirect},0)) = cmd[31:15]1cmd[13]0cmd[11:0].$$

The step raises the interrupt bit in the interrupt request registers corresponding to the sent vector

$$q \in t \rightarrow PW(c,x).bpa((\mathrm{p},q,\mathrm{IRR}))_v \equiv c.\mathrm{p}(q).\mathrm{apic.ICR} \vee v = \langle cmd.vect \rangle.$$

One easily checks that whether processor q is a target – and thus whether it is interrupted or not – indeed only depends on the IPI-relevant registers.

3.3.13 Passive Steps of APIC

We now define the write-triggered semantics of the APIC device, i.e., we define the function

$$\delta_{DEV,(\mathrm{apic},p)} : K_{DEV,(\mathrm{apic},p)} \times \Sigma_{DEV,(\mathrm{apic},p)} \rightarrow K_{DEV,(\mathrm{apic},p)}.$$

Let $c \in K_{DEV,(\mathrm{apic},p)}$ be a configuration of the device, and $w \in \Sigma_{DEV,(\mathrm{apic},p)}$ be an input to the device. By definition, c is a valuation of the device addresses

$$c : Val(A_{DEV,(\mathrm{apic},p)}),$$

and w is a more up-to-date partial valuation of the device addresses

$$w : PVal(A_{DEV,(\mathrm{apic},p)}).$$

We will define a new device configuration c'

$$c' = \delta_{DEV,(\mathrm{apic},p)}(c,w).$$

We look at the shape of w and treat one interesting special case. In all other cases, we simply apply the update as if the device was a normal memory region. Because of the way our machine is built (aligned memory accesses, etc.), nothing can go wrong; it is, for example, impossible to write half a word into the end-of-interrupt register and another half of the word into the APICID, or to write to the EOIpending register.

The special case is when w is modifying the first byte of the EOI register. In this case we interpret the value of the update w as an interrupt vector for which a pending end-of-interrupt must be signaled.

$a_{EOI} \in Dom(w)$**:** We raise the EOIpending signal corresponding to the interrupt signaled by the value of the write

$$c'((\text{apic}, p, \text{EOIpending}))_v = c'((\text{apic}, p, \text{EOIpending}))_v \vee v = \langle w(a_{EOI}) \rangle.$$

All other values are unchanged

$$a \notin \{ (\text{apic}, p, \text{EOIpending}) \} \rightarrow c'(a) = c(a).$$

Otherwise: We simply apply the update directly

$$c'(a) = \begin{cases} w(a) & a \in Dom(w) \\ c(a) & \text{o.w.} \end{cases}$$

3.3.14 Passive Steps of IO APIC

We now define the write-triggered semantics of the IO APIC device, i.e., we define the function

$$\delta_{DEV,\text{ioapic}} : K_{DEV,\text{ioapic}} \times \Sigma_{DEV,\text{ioapic}} \rightarrow K_{DEV,\text{ioapic}}.$$

Let $c \in K_{DEV,\text{ioapic}}$ be a configuration of the device, and $w \in \Sigma_{DEV,\text{ioapic}}$ be an input to the device. By definition, c is a valuation of the device addresses

$$c : Val(A_{DEV,\text{ioapic}}),$$

and w is a more up-to-date partial valuation of the device addresses

$$w : PVal(A_{DEV,\text{ioapic}}).$$

We will define a new device configuration c'

$$c' = \delta_{DEV,\text{ioapic}}(c, w).$$

We look at the shape of w and treat two interesting special cases. In all other cases, we simply apply the update as if the device was a normal memory region.

The special cases are when w is modifying the WIN or the regsel registers.

$\{ (\text{ioapic}, \text{regsel}) \} \subseteq Dom(w)$**:** We select the redirection table entry corresponding to the value given by the write, using only the least significant ID bits of the value written to the regsel register

$$d = \langle w(\text{ioapic}, \text{regsel})[ID - 1 : 0] \rangle,$$

and copy its value into the WIN register

$$c'((\text{ioapic}, \text{win})) = c((\text{ioapic}, \text{redirect}, d)).$$

The register selection register is taken from the update w

$$c'((\text{ioapic}, \text{regsel})) = w((\text{ioapic}, \text{regsel})).$$

All other values are unchanged

$$a \notin \{ (\text{ioapic}, \text{win}), (\text{ioapic}, \text{regsel}) \} \rightarrow c'(a) = c(a).$$

$\{ (\text{ioapic}, \text{win}) \} \subseteq Dom(w) \wedge \{ (\text{ioapic}, \text{regsel}) \} \not\subseteq Dom(w)$**:** We select the redirection table entry given by the current value of the regsel register

$$d = \langle c(\text{ioapic}, \text{regsel})[ID - 1 : 0] \rangle,$$

and copy the value written to the WIN register into that redirection table entry

$$c'((\text{ioapic}, \text{redirect}, d)) = w((\text{ioapic}, \text{win})),$$

and to the WIN register

$$c'((\text{ioapic}, \text{win})) = w((\text{ioapic}, \text{win})).$$

All other values are unchanged

$$a \notin \{ (\text{ioapic}, \text{win}), (\text{ioapic}, \text{redirect}, d) \} \rightarrow c'(a) = c(a).$$

Otherwise: We first apply the update directly. However, if the port that is currently viewed in the WIN register is changed, we need to reflect that change in the WIN register. We select the redirection table entry given by the current value of the regsel register

$$d = \langle c(\text{ioapic}, \text{regsel})[ID - 1 : 0] \rangle,$$

and change the value of the overwritten address, and in case a byte of the redirection table entry d is overwritten, also of the WIN register

$$c'(a) = \begin{cases} w(a) & a \in Dom(w) \\ w(b) & \exists r.a = (\text{ioapic}, \text{win})_r \wedge b = (\text{ioapic}, \text{redirect}, d)_r \in Dom(w) \\ c(a) & \text{o.w.} \end{cases}$$

Note that this fixes a bug in the MIPS 86 ISA of Sabine Schmaltz, where changes made to the redirection table by IO APIC steps were not reflected in the WIN register.

Chapter 4

Write Buffer Reduction

In this chapter, we define the software discipline for write buffer reduction. To do so, we need to introduce some auxiliary functions and information. We then prove that any program in the high-level semantics that obeys the software discipline can be executed on the low-level machine without introducing any new computations. The proof has three parts:

1. We show that a certain technical definition of races really catches all races,
2. We show that we can keep the two machines — high-level and low-level — in sync as long as no shared reads are executed while a shared write is being buffered by another processor,
3. We show that we can reorder all schedules so that no shared reads are executed while a shared write is being buffered by another processor.

4.1 Enhancing the Buffer

We now enhance the buffer with considerable ghost information. In particular, we keep track for each unit i and each entry in the buffer of unit i the point at which it was added to the buffer. We use a sequence of functions $issue_M$, defined as follows

$$issue_M[s]^0(i) = \varepsilon$$
$$issue_M[s]^{t+1}(i) = Op_{iM}(t)(issue_M[s]^t(i), t).$$

That this function indeed keeps track of the right timestamps is given by the following lemma.

Lemma 123. *The write buffer can be expressed in terms of the issued timestamps.*

$$\left[BW_M(t') \;\middle|\; t' \in issue_M^t(i) \right] = wb_M^t(i).$$

Proof. By straightforward induction on t. The base case is trivial. In the inductive step, we commute the list comprehension using Lemma 36 with $\delta := Op_{iM}(t)$, $f := BW_M$, $l := issue_M^t(i)$, and $x := t$, and solve the claim using the induction hypothesis

$$\left[BW_M(t') \;\middle|\; t' \in issue_M^{t+1}(i) \right]$$

$$
\begin{aligned}
&= \left[BW_M(t') \;\middle|\; t' \in Op_{iM}(t)(issue^t_M(i),t) \right] \\
&= Op_{iM}(t)(\left[BW_M(t') \;\middle|\; t' \in issue^t_M(i) \right], BW_M(t)) && \text{L 36} \\
&= Op_{iM}(t)(wb^t_M(i), BW_M(t)) && \text{IH} \\
&= wb^{t+1}_M(i).
\end{aligned}
$$

□

Each sequence of issued writes is monotone in two ways: 1) it is sorted (the head is the minimum element), and 2) once a timestamp leaves the sequence, it never enters again. Furthermore, the elements in the buffer are all less than the current timestamp. We formalize these properties but we do not show the proofs.

Lemma 124.

$$e+1 < |issue^t_M(i)| \to issue^t_M(i)_e < issue^t_M(i)_{e+1}.$$

Lemma 125.

$$\forall t \in (t' : k].t' \in issue^k_M(i) \to t' \in issue^t_M(i).$$

Lemma 126.

$$e < |issue^t_M(i)| \to issue^t_M(i)_e < t.$$

We refer to these properties informally as "monotonicity of the sequence of issued writes".

The write buffer has a hit iff there is an issued write with that domain.

Lemma 127.

$$hit(A, wb^t_M(i)) \iff \exists t' \in issue^t_M(i).Dom(BW_M(t')) \dot{\cap} A$$

Proof. Obviously a hit for A exists in a buffer iff there is a write in the buffer the domain of which is intersecting with A. The claim follows with Lemma 123

$$
\begin{aligned}
hit(A, wb^t_M(i)) &\iff \exists w \in wb^t_M(i).Dom(w) \dot{\cap} A \\
&\iff \exists t' \in issue^t_M(i).Dom(BW_M(t')) \dot{\cap} A. && \text{L 123}
\end{aligned}
$$

□

Lemma 128. *If the steps from t to k are not write buffer steps of unit i*

$$\forall t' \in [t : k).s(t') \notin \Sigma_{WB,i}$$

then the sequence of issued writes of unit i at $l \in [t : k]$ is a prefix of the sequence at k

$$\exists q.issue^k_\downarrow(i) = issue^l_\downarrow(i) \circ q.$$

Proof. By downwards induction on l, starting at k. The base case is trivial with $q := \varepsilon$.

In the inductive step $l \to l-1$, we have a suffix q that has been added

$$issue^k_\downarrow(i) = issue^l_\downarrow(i) \circ q.$$

Since step $l-1 \in [t : k)$ is not a write buffer step of unit i

$$s(l-1) \notin \Sigma_{WB,i}$$

the operation performed by it is a push or noop

$$Op_{i\downarrow}(l-1) \in \{\, push, noop \,\}.$$

With Lemma 42 we obtain a suffix q' which is added in step $l-1$, and the claim follows with $q := q' \circ q$

$$\begin{aligned} issue^{k}_{\downarrow}(i) &= issue^{l}_{\downarrow}(i) \circ q \\ &= (issue^{l-1}_{\downarrow}(i) \circ q') \circ q && \text{L 42} \\ &= issue^{l-1}_{\downarrow}(i) \circ (q' \circ q). \end{aligned}$$

□

When reordering a schedule, the sequences of issued writes are also reordered, since the steps in which the writes were issued have been moved. We lift the function *mvO* to sequences of numbers

$$mvO(l) = [mvO(t') | t' \in l].$$

4.2 Shared Accesses and Global Steps

We consider a program annotation that marks some accesses as shared. The annotation comes with two proof obligations for the system programmer: potential races (i.e., memory accesses that in some schedule are part of a race) have to be marked as shared, and a processor must not issue a shared read while a shared write is in its buffer.

The decision to do a shared access is made before reading the memory, and we assume an annotation *intention to do a shared access*

$$iSh_i : Val(A_{PR,i}) \to \mathbb{B}.$$

This roughly corresponds to volatile bits in [CS10].

Our analysis of steps has to be more fine-grained. Our flushing discipline introduces flushes before certain shared reads. Consider a step that fetches a shared store instruction. This instruction is considered a read, because it is fetching from memory, and it is also considered shared, because the store portion of it is shared. A naive formalization would require us to insert a slow memory barrier before the shared store. However, the portion of the step that is potentially racing is the store instruction, not the fetch, and thus no memory barrier is needed. To distinguish between steps where a non-racing read is executed as well as a potentially racing (shared) store from those where the read is also potentially racing (and a memory barrier is needed), we introduce an annotation *intention to do a shared read*

$$iShR : Val(A_{PR,i}) \to \mathbb{B}$$

which is specifically used to mark steps that perform a potentially racing read.

We require that every step that is intending to do a shared read is also intending to do a shared access

$$iShR(core) \to iSh_i(core). \tag{4.1}$$

Steps that read from or modify devices or interrupt registers of other units, or modify their own interrupt registers, are always shared. We call these steps *volatile*. We distinguish between *volatile reads*

$$volR_M(c,x) \equiv in_M(c,x) \hat{\cap} A_{DEV} \cup \bigcup_{i \neq u_M(c,x)} A_{IPR,i}.$$

and *volatile writes*

$$volW_M(c,x) \equiv out_M(c,x) \hat{\cap} A_{DEV} \cup \bigcup_{i} A_{IPR,i}.$$

A step is *volatile* if it is a volatile read or a volatile write

$$vol_M(c,x) = volR_M(c,x) \vee volW_M(c,x).$$

Steps are *shared* if they have the intention to be shared or are volatile, but steps in weak-memory mode and non-processor steps are also shared:

$$Sh_M(c,x) = \begin{cases} iSh_i(core_M(c,x)) \vee vol_M(c,x) \vee \neg SC_M(c,x) & x \in \Sigma_{P,i} \\ 1 & \text{o.w.} \end{cases}$$

A step is a *shared read* if it has the intention to be a shared read or is a volatile read, but steps in weak memory mode are also shared reads:

$$ShR_M(c,x) = \begin{cases} iShR(core_M(c,x)) \vee volR_M(c,x) \vee \neg SC_M(c,x) & x \in \Sigma_{P,i} \\ 0 & \text{o.w.} \end{cases}$$

Intuitively, we consider all steps that the operating system programmer can not control, such as write buffer steps and user program steps, as shared. The reason for this is that they might be racing or not racing other steps without any control of the system programmer. We do know that write buffer steps are never reads, and thus do not consider them shared reads. For the operating system, we only consider those steps as shared which are racing or volatile. We will sometimes use the term *annotation of the step* c,x to refer to $Sh_M(c,x)$ and/or $ShR_M(c,x)$.

Lemma 129.

$$ShR_M(c,x) \rightarrow Sh_M(c,x).$$

Proof. The claim is solved easily with the fact that an intention to do a shared read always implies the intention to do a shared access. We do not show the proof. □

Volatile steps are always shared.

Lemma 130.

$$vol_M(c,x) \rightarrow Sh_M(c,x).$$

Proof. The proof distinguishes between processor and other steps.

$x \in \Sigma_{P,i}$**:** If a processor step is volatile it is shared

$$Sh_M(c,x) = \ldots \vee vol_M(c,x) \vee \ldots = 1,$$

which is the claim.

$x \notin \Sigma_{P,i}$**:** Non-processor steps are always shared.

□

Lemma 131.

$$volR_M(c,x) \to ShR_M(c,x).$$

Proof. The proof distinguishes between processor and other steps.

$x \in \Sigma_{P,i}$**:** If a processor step is a volatile read it is a shared read

$$ShR_M(c,x) = \ldots \vee volR_M(c,x) \vee \ldots = 1,$$

which is the claim.

$x \notin \Sigma_{P,i}$**:** By definition, non-processor only access the core registers

$$\begin{aligned} in_M(c,x) &= C_M(c,x) \cup F_M(c,x) \cup R_M(c,x) \\ &= C_M(c,x) \cup \emptyset \cup \emptyset \\ &= C_M(c,x) \\ &\subseteq A_{PR,i} \end{aligned}$$

which do not intersect with device registers or interrupt registers of other units

$$A_{PR,i} \mathbin{\dot{\cap}} A_{DEV} \cup \bigcup_{i \neq u_M(c,x)} A_{IPR,i}.$$

We conclude that the interrupts do not intersect with those registers

$$in_M(c,x) \mathbin{\dot{\cap}} A_{DEV} \cup \bigcup_{i \neq u_M(c,x)} A_{IPR,i},$$

and thus by definition of $volR$ there is no volatile read

$$\neg volR_M(c,x),$$

which is a contradiction.

□

We conclude that all accesses to devices are shared.

Lemma 132.

$$in_M(c,x) \mathbin{\dot{\cap}} A_{DEV} \cup \bigcup_{i \neq u_M(c,x)} A_{IPR,i} \to ShR_M(c,x).$$

Proof. The step is by definition of $volR$ a volatile read

$$volR_M(c,x),$$

and by Lemma 131 it is a shared read

$$ShR_M(c,x).$$

□

Lemma 133.

$$out_M(c,x) \cup devin_M(c,x) \mathbin{\dot\cap} A_{DEV} \cup \bigcup_i A_{IPR,i} \to Sh_M(c,x).$$

Proof. The device inputs are a subset of the outputs

$$\begin{aligned} devin_M(c,x) &= dc(WS_M(c,x)) \\ &\subseteq WS_M(c,x) \cup dc(WS_M(c,x)) \\ &= idc(WS_M(c,x)) = out_M(c,x), \end{aligned}$$

and thus the outputs or inputs of the step access the device or interrupt registers

$$out_M(c,x) \mathbin{\dot\cap} A_{DEV} \cup \bigcup_i A_{IPR,i}.$$

By definition of *volW* and *vol* such a step is a volatile write and thus volatile

$$vol_M(c,x) = volW_M(c,x) = 1.$$

The claim follows with Lemma 130

$$Sh_M(c,x).$$

□

We say that a step is *global* when it is a shared read or a memory write that is shared

$$G_M(c,x) = ShR_M(c,x) \vee mwrite_M(c,x) \wedge Sh_M(c,x).$$

This includes write buffer steps that commit non-shared writes. We will later reorder the schedule in a way that keeps the order of global steps intact. This corresponds to the intuition that on TSO, all processors agree on the order of stores (as they reach memory); therefore it is not surprising that in TSO an equivalent sequentially consistent execution can be found that does not reorder any stores. Since our machine model is slightly weaker than TSO, we will not get around changing the order of some write operations: if a processor issues a bypassing non-shared write while it has a buffered write, the bypassing non-shared write may be delayed until the buffered write can leave the write buffer.

We say that the step is *local* and write $L_M(c,x)$ if it is not global

$$L_M(c,x) = \neg G_M(c,x).$$

A local step has no victims.

Lemma 134.

$$L_M(c,x) \to victims_M(c,x) = \emptyset.$$

Proof. Assume for the sake of contradiction that there is a victim j

$$j \in victims_M(c,x).$$

By Lemma 98 we obtain that step t modifies the processor registers of unit j and is made by a different unit

$$out_M(c,x) \mathbin{\dot\cap} A_{PR,j} \wedge u_M(c,x) \neq j.$$

The processor registers are accessible for unit j

$$out_M(c,x) \dot{\cap} ACC_j.$$

With Lemma 121 we obtain that the step is a memory write

$$mwrite_M(c,x).$$

By Lemma 94 step t is modifying interrupt registers

$$out_M(c,x) \dot{\cap} A_{IPR}$$

and by Lemma 133, step t must be shared

$$Sh_M(c,x).$$

Since it is shared and a memory write, it is a global step

$$G_M(c,x)$$

which is a contradiction. □

A local step is also IPI-valid

Lemma 135.

$$L_M(c,x) \rightarrow \Delta_{IPIM}(c,x).$$

Proof. By Lemma 134 the step has no victims

$$victims_M(c,x) = \emptyset.$$

It is thus vacuously true that all write buffers of all victims are empty, and the claim follows

$$\begin{aligned}\Delta_{IPIM}(c,x) &= \bigwedge_{i \in victims_M(c,x)} \dots \\ &= \bigwedge_{i \in \emptyset} \dots \\ &= 1.\end{aligned}$$

□

Write buffer steps are global in the low-level machine.

Lemma 136.

$$s(c,x) \in \Sigma_{WB,i} \rightarrow G_{\downarrow}(c,x).$$

Proof. By Lemma 123, the head of the write buffer was issued at some time and is thus non-empty

$$hd(wb^t_{\downarrow}(i)) = BW_{\downarrow}(hd(issue^t_{\uparrow}(i))) \neq \emptyset,$$

and write buffer entries are always to the bufferable addresses

$$Dom(hd(wb^t_{\downarrow}(i))) \subseteq BA,$$

which do not contain normal processor registers

$$BA \not\cap A_{NPR,i}.$$

We conclude that the domain of the committed write is not contained in the normal processor registers

$$Dom(hd(wb^t_\downarrow(i))) \not\subseteq A_{NPR,i}.$$

The low-level machine uses low-level semantics

$$LL_\downarrow(c,x)$$

and therefore the executed write is exactly the committed write

$$W_\downarrow(c,x) = hd(wb^t_\downarrow(i)).$$

Therefore the outputs subsume the domain of the committed write

$$Dom(hd(wb^t_\downarrow(i))) = Dom(W_\downarrow(c,x)) \subseteq idc(Dom(W_\downarrow(c,x))) = out_\downarrow(c,x),$$

and the outputs are also not contained in the normal processor registers

$$out_\downarrow(c,x) \not\subseteq A_{NPR,i}.$$

Therefore the step is a memory write

$$mwrite_\downarrow(c,x).$$

Write buffer steps are by definition shared

$$Sh_\downarrow(c,x),$$

and shared memory writes are by definition global

$$G_\downarrow(c,x),$$

which was the claim. □

Local steps never modify IPI-relevant registers.

Lemma 137.

$$L_M(c,x) \rightarrow out_M(c,x) \not\cap A_{IPI}(x)$$

Proof. Assume for the sake of contradiction that the step modifies IPI-relevant registers

$$out_M(c,x) \cap A_{IPI}(x).$$

IPI-relevant registers are interrupt or device registers

$$A_{IPI}(x) \subseteq A_{IPR} \cup A_{DEV}$$

and we conclude that the step modifies those

$$out_M(c,x) \cap A_{IPR} \cup A_{DEV}.$$

Clearly the step does not only modify normal processor registers

$$out_M(c,x) \not\subseteq A_{NPR,i}$$

and is thus a memory write

$$mwrite_M(c,x).$$

Furthermore the step modifies interrupt or device registers and is by Lemma 133 shared

$$Sh_M(c,x),$$

and thus by definition global

$$G_M(c,x),$$

which is a contradiction. □

4.3 Input and Output Correctness

Under correctness of inputs and outputs we understand that the value of inputs determine the decisions made by a step, and only outputs are changed by a step. We show this by a series of trivial lemmas. We do not show proofs.

Lemma 138.

$$A \not\upharpoonright out_M(c,x) \rightarrow c \centerdot_M x.m =_A c.m.$$

Lemma 139.

$$A_{SC,i} \not\upharpoonright out_M(c,x) \rightarrow SC_{iM}(c \centerdot_M) = SC_{iM}(c).$$

Lemma 140.

$$v =_A v' \wedge v =_{A'} v' \rightarrow v =_{A \cup A'} v'.$$

Lemma 141.

$$v =_A v' \wedge A' \subseteq A \rightarrow v =_{A'} v'.$$

Lemma 142. *When two configurations agree on the value of local inputs during a step*

$$c.m =_{C_M(c,x)} c'.m,$$

then they agree on all functions X which are defined in terms of the core, i.e., for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_M(c,x) = X_N(c',x).$$

Lemma 143. *When two configurations agree on the local inputs and fetched inputs during a step*

$$c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m,$$

they agree on all functions X which are defined in terms of those values, i.e., for

$$X \in \{ fetch, R, \Upsilon, I, in, read \}$$

and for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_M(c,x) = X_N(c',x).$$

Lemma 144. *When two configurations strongly agree on the inputs during a step*

$$c =^{x}_{M,N} c',$$

then they agree on all of the functions X below

$$X_M(c,x) = X_N(c',x).$$

1. *All functions X which are defined in terms of core, fetched and read addresses, and the local buffer, i.e., for*

   $$X \in \{ PW, BW, Op_i, \Delta, \Gamma \}$$

   and for

   $$X \in \{ fetch, R, \Upsilon, I, in, read \}$$

   and for

   $$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

2. *When the machines also agree on whether they are using low-level semantics for the step*

   $$LL_M(c,x) = LL_N(c',x),$$

   also on all functions that additionally depend on the machine-type

   $$X \in \{ W, WS, out, mwrite, devin, victims, G \}.$$

Proof. We only show the second claim, and only in the case that the machine types are different. The other claims are all trivial. Assume thus that the machine types are different but the machine semantics are the same

$$N \neq M \wedge LL_M(c,x) = LL_N(c',x),$$

and that X is one of the functions that depend on the the machine type

$$X \in \{ W, WS, out, mwrite, devin, victims, G \}.$$

Note that since low-level machines always have low-level semantics, and at least one of the machines is a low-level machine, that machine is using low-level semantics

$$LL_M(c,x) \vee LL_N(c',x).$$

Since the machines agree on their machine semantics, the machines must both be in low-level semantics

$$LL_M(c,x) = LL_N(c',x) = 1.$$

We now show that the machines execute the same writes by case distinction on x.

$x \in \Sigma_{P,i}$**:** Both machines execute exactly the bypassing portion of the prepared writes. The claim follows with Claim 1 for $X := PW$

$$\begin{aligned} W_M(c,x) &= PW_M(c,x).bpa \\ &= PW_N(c',x).bpa && \text{Claim 1} \\ &= W_N(c',x). \end{aligned}$$

$x \in \Sigma_{WB,i}$**:** Both steps commit the head of the write buffer, which is the same

$$W_M(c,x) = hd(c.wb(i)) = hd(c'.wb(i)) = W_N(c',x).$$

Now that we have that the steps execute the same writes, the remaining claims trivially follow

$$\begin{aligned}
WS_M(c,x) &= Dom(W_M(c,x)) = Dom(W_N(c',x)) = WS_N(c',x),\\
out_M(c,x) &= idc(WS_M(c,x)) = idc(WS_N(c',x)) = out_N(c',x),\\
mwrite_M(c,x) &\equiv out_M(c,x) \not\subseteq A_{NPR,u_M(c,x)}\\
&\equiv out_N(c',x) \not\subseteq A_{NPR,u_N(c',x)}\\
&\equiv mwrite_N(c',x),\\
devin_M(c,x) &= dc(WS_M(c,x)) = dc(WS_N(c',x)) = devin_N(c',x),\\
victims_M(c,x) &= \{\, i \mid A_{PR,i} \hat{\cap} Dom(W_M(c,x)) \wedge x \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \,\}\\
&= \left\{\, i \;\middle|\; A_{PR,i} \hat{\cap} Dom(W_N(c',x)) \wedge x \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \,\right\}\\
&= victims_N(c',x),\\
G_M(c,x) &= Sh_M(c,x) \wedge mwrite_M(c,x) \vee ShR_M(c,x)\\
&= Sh_N(c',x) \wedge mwrite_N(c',x) \vee ShR_N(c',x)\\
&= G_N(c',x)
\end{aligned}$$

□

Strong agreement is in fact an equivalence relation.

Lemma 145.

$$\begin{aligned}
c &=^x_{M,M} c,\\
c =^x_{M,N} c' &\rightarrow c' =^x_{N,M} c,\\
c =^x_{M,N} c' \wedge c' =^x_{N,O} c'' &\rightarrow c =^x_{M,O} c''.
\end{aligned}$$

Proof. We only show symmetry, and only the claim for the agreement between memories on core and fetched registers when $x \in \Sigma_{P,i}$; the remaining goals are either trivial or analogous.

Assume for that case that the memories agree on the core and fetched registers in step c,x of machine M

$$c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m,$$

and we have to show that they also agree on the core and fetched registers in step c',x of machine N

$$c.m \stackrel{!}{=}_{C_N(c',x) \cup F_N(c',x)} c'.m.$$

We wish to rewrite C with Lemma 59 and F with Lemma 61, which would immediately solve the goal. The only thing we still need to show is that the steps have the same cores

$$core_M(c,x) \stackrel{!}{=} core_N(c',x),$$

which is Lemma 142. □

Lemma 146.

$$\neg SC_M(c,x) \wedge c.m =_{C_M(c,x)} c'.m \rightarrow LL_M(c,x) = LL_N(c',x).$$

Proof. By Lemma 142 the second machine also makes a weak memory mode step

$$\neg SC_N(c',x).$$

Independently of the machine type, weak memory mode steps always use low-level machine semantics. Thus both machines make a low-level machine step

$$LL_M(c,x) = LL_N(c',x) = 1.$$

□

Lemma 147.

$$c.m =_{C_M(c,x)} c'.m \rightarrow LL_M(c,x) = LL_M(c',x).$$

Proof. We only show the proof for the high-level machine

$$M = \uparrow,$$

since the claim trivially holds in the low-level machine. By Lemma 142, the steps use the same memory mode

$$SC_M(c,x) = SC_M(c',x)$$

and the claim follows

$$LL_M(c,x) = \neg SC_M(c,x) = \neg SC_M(c',x) = LL_M(c',x).$$

□

We can show that if two configurations strongly agree when stepped with some input x, and they agree on the state of the device inputs during the step, they have the same effect on the configuration and thus have the same outputs

$$c =^x_M c' \wedge c.m =_{devin_M(c,x)} c'.m \rightarrow c \blacksquare_M x.m =_{out_M(c,x)} c' \blacksquare_M x.m.$$

This turns out not to be strong enough. We need a generalization that talks about configurations where some of the devices modified by the step were not in the same state. In this case, we can not show that they are in the same state after the step.

Lemma 148.

$$c =^x_M c' \wedge c.m =_{devin_M(c,x) \setminus dc(B)} c'.m \rightarrow c \blacksquare_M x.m =_{out_M(c,x) \setminus dc(B)} c' \blacksquare_M x.m.$$

Proof. The device closure over the outputs minus the device closure of B is the same as the device inputs minus the device closure of B. With Lemmas 7, 10, 18 and 20 we obtain

$$\begin{aligned}
dc(out_M(c,x) \setminus dc(B)) &= dc(out_M(c,x)) \setminus dc(dc(B)) && \text{L 10, 20} \\
&= dc(out_M(c,x)) \setminus dc(B) && \text{L 18} \\
&= dc(idc(WS_M(c,x))) \setminus dc(B) \\
&= dc(WS_M(c,x)) \setminus dc(B) && \text{L 19} \\
&= devin_M(c,x) \setminus dc(B).
\end{aligned}$$

We conclude with the assumption that the memories agreed on the latter that they also agreed on the former

$$c.m =_{dc(out_M(c,x) \setminus dc(B))} c'.m.$$

With Lemma 23 we obtain further that the outputs without the device closure of B is closed

$$closed(out_M(c,x) \setminus dc(B)).$$

The claim follows with Lemmas 5 and 144

$$\begin{aligned} c \blacksquare_M x.m &= c.m \circledast W_M(c,x) \\ &=_{out_M(c,x)\setminus dc(B)} c'.m \circledast W_M(c,x) && \text{L 5} \\ &= c'.m \circledast W_M(c',x) && \text{L 144, 147} \\ &= c' \blacksquare_M x.m. \end{aligned}$$

□

Putting these together we obtain the following lemma.

Lemma 149. *When two configurations agree about a set of addresses A, except for the device closure of B, and they strongly agree on the inputs, and the complement of A is contained in the inclusive device closure of B*

$$c.m =_{A\setminus dc(B)} c'.m \wedge c =^x_M c' \wedge (\mathcal{A} \setminus A) \subseteq idc(B),$$

then the configurations after the step agree on A and the outputs of the step, except for the device closure of B

$$c \blacksquare_M x.m =_{(A\cup out_M(c,x))\setminus dc(B)} c' \blacksquare_M x.m.$$

Proof. Because set difference can be distributed over set union we obtain the following

$$\begin{aligned} (A \cup out_M(c,x)) \setminus dc(B) &= ((A \setminus out_M(c,x)) \cup out_M(c,x)) \setminus dc(B) \\ &= ((A \setminus out_M(c,x)) \setminus dc(B)) \cup (out_M(c,x) \setminus dc(B)) \\ &= ((A \setminus dc(B)) \setminus out_M(c,x)) \cup (out_M(c,x) \setminus dc(B)). \end{aligned}$$

Therefore the claim is equivalent to the following

$$c \blacksquare_M x.m \stackrel{!}{=}_{((A\setminus dc(B))\setminus out_M(c,x))\cup(out_M(c,x)\setminus dc(B))} c' \blacksquare_M x.m.$$

We also obtain that the device inputs minus the device closure of B are a subset of A minus the device closure of B

$$\begin{aligned} & (\mathcal{A} \setminus A) \subseteq idc(B) \\ \Longrightarrow\ & dc(\mathcal{A} \setminus A) \subseteq dc(idc(B)) && \text{L 9} \\ \Longrightarrow\ & A_{DEV} \cap (\mathcal{A} \setminus A) \subseteq dc(B) && \text{L 2} \\ \Longrightarrow\ & A_{DEV} \setminus A \subseteq dc(B) \\ \Longrightarrow\ & A_{DEV} \subseteq dc(B) \cup A \\ \Longrightarrow\ & devin_M(c,x) \subseteq A \cup dc(B) \\ \Longrightarrow\ & devin_M(c,x) \setminus dc(B) \subseteq A \setminus dc(B). \end{aligned}$$

By Lemma 148 we obtain the following

$$c \blacksquare_M x.m =_{out_M(c,x)\setminus dc(B)} c' \blacksquare_M x.m.$$

With Lemma 140 it now suffices to show that the addresses which are not outputs are correct

$$c \blacksquare_M x.m \overset{!}{=}_{(A\setminus dc(B))\setminus out_M(c,x)} c' \blacksquare_M x.m.$$

By Lemma 147 both machines have the same machine-mode

$$LL_M(c,x) = LL_M(c',x)$$

and by Lemma 144, the outputs of the two steps are the same

$$out_M(c,x) = out_M(c',x).$$

By applying Lemma 138 twice we can now reduce the claim to showing that the configurations agree on the non-outputs before the step

$$c \blacksquare_M x.m =_{(A\setminus dc(B))\setminus out_M(c,x)} c.m \overset{!}{=}_{(A\setminus dc(B))\setminus out_M(c,x)} c'.m =_{(A\setminus dc(B))\setminus out_M(c,x)} c' \blacksquare_M x.m.$$

With Lemma 141 we reduce the claim to the following

$$c.m \overset{!}{=}_{A\setminus dc(B)} c'.m,$$

which we have by assumption. □

We will often wish to apply Lemmas 142, 143, and 144 in situations where the functions X are applied after reordering, and usually on the same machine. The following theorems apply exactly in situations like that. We only give a proof for the first one, the others are analogous. For Lemma 144, we make use of the fact that the lemma proves more when the machine types are the same.

Lemma 150. *Let O_1, O_2 be sequences of operators and t_1, t_2 be the position of some step after the corresponding reorderings*

$$sO_1(t_1) = sO_2(t_2),$$

and the two configurations before the step strongly agree during that step

$$c_M O_1^{t_1} =_M^{sO_1(t_1)} c_M O_2^{t_2},$$

then the two computations agree on the following functions X which are used during the step, i.e., for

$$X \in \{ PW, Op_i, W, WS, out, victims, mwrite, read, \Delta, \Gamma, L \}$$

and for

$$X \in \{ R, \Upsilon, I, in \}$$

and for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_M O_1(t_1) = X_M O_2(t_2).$$

Proof. We simply unfold notations and rewrite with the equality of the steps

$$\begin{aligned} X_M O_1(t_1) &= X_M(c_M O_1^{t_1}, sO_1(t_1)), \\ X_M O_2(t_2) &= X_M(c_M O_2^{t_2}, sO_2(t_2)) \\ &= X_M(c_M O_2^{t_2}, sO_1(t_1)), \end{aligned}$$

and thus rewrite the claim to the following

$$X_M(c_M O_1^{t_1}, sO_1(t_1)) \stackrel{!}{=} X_M(c_M O_2^{t_2}, sO_1(t_1)).$$

We now obtain with Lemma 147 that the steps are done in the same machine-mode

$$LL_M(c,x) = LL_M(c',x).$$

The claim is now Lemma 144. □

Lemma 151. *Let O_1, O_2 be sequences of operators and t_1, t_2 be the position of some step after the corresponding reorderings*

$$sO_1(t_1) = sO_2(t_2),$$

and the two configurations before the step agree on the local inputs during that step

$$m_M O_1^{t_1} =_{C_M O_1(t_1)} m_N O_2^{t_2},$$

then the two computations agree on all functions X which are made before fetching, i.e., for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_M O_1(t_1) = X_N O_2(t_2).$$

Lemma 152. *Let O_1, O_2 be sequences of operators and t_1, t_2 be the position of some step after the corresponding reorderings*

$$sO_1(t_1) = sO_2(t_2),$$

and the two configurations before the step agree on the local inputs and fetch inputs during that step

$$m_M O_1^{t_1} =_{C_M O_1(t_1) \wedge F_M O_1(t_1)} m_N O_2^{t_2},$$

then the two computations agree on all functions X which are made before loading, i.e., for

$$X \in \{ R, \Upsilon, I, in \}$$

and for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_M O_1(t_1) = X_N O_2(t_2).$$

Forwarding is only dependent on the portion of write buffers that modifies the forwarded region, and the memory at the forwarded region.

Lemma 153. *When two valuations agree on some range of addresses A*

$$\nu =_A \nu'$$

and write buffers w_1, w_2 are equal except for a prefix w not used by A

$$w_1 = w \circ w_2 \wedge \neg hit(A, w),$$

then they have the same forwarding on A

$$\nu \circledcirc w_1 =_A \nu' \circledcirc w_2.$$

Proof. The claim follows with Lemmas 52 and 53

$$\begin{aligned}
& \nu \circledcirc w_1 =_A \nu' \circledcirc w_2 \\
\iff & \nu \circledcirc (w \circ w_2) =_A \nu' \circledcirc w_2 \\
\iff & (\nu \circledcirc w) \circledcirc w_2 =_A \nu' \circledcirc w_2 \\
\impliedby & (\nu \circledcirc w) =_A \nu' && \text{L 53} \\
\impliedby & \nu =_A \nu'. && \text{L 52}
\end{aligned}$$

□

Lemma 154. *When two configurations agree on some range of addresses A*

$$c.m =_A c'.m$$

and for processor steps write buffers are equal except for a portion not used by forwarding

$$x \in \Sigma_{P,i} \to c.wb(i) = w \circ c'.wb(i) \wedge \neg hit(A, w),$$

then they have the same forwarding system on A

$$fms_M(c, x) =_A fms_N(c', x).$$

Proof. By case distinction on x.

$x \in \Sigma_{P,i}$**:** The claim follows with Lemma 153

$$\begin{aligned}
fms_M(c, x) &= fms_{iM}(c) \\
&= c.m \circledcirc c.wb(i) \\
&=_A c'.m \circledcirc c'.wb(i) && \text{L 153} \\
&= fms_{iN}(c') \\
&= fms_N(c', x).
\end{aligned}$$

$x \in\in \Sigma_{WB,i}$**:** Both are empty by definition

$$fms_M(c, x) = \emptyset = fms_N(c', x).$$

□

4.4 Types of Races

Races are situations in which two concurrent steps access an address, and at least one of the accesses is modifying the value of the address. This gives rise to three types of races: write followed by read, write followed by write, and read followed by write. We can further distinguish between races in the main memory and races in devices (including the APIC).

Formally, steps t and t' are a write-read race in A if the outputs of t intersect with the inputs of t'

$$WR_M^A(t,t') \equiv out_M(t) \cap in_M(t') \dot{\cap} A.$$

They are a write-write race in A if the outputs intersect instead

$$WW_M^A(t,t') \equiv out_M(t) \cap out_M(t') \dot{\cap} A.$$

They are a read-write race in A if the inputs of t intersect with the outputs of t'

$$RW_M^A(t,t') \equiv in_M(t) \cap out_M(t') \dot{\cap} A.$$

When the race is in any register, i.e., $A = \mathcal{A}$, we drop the superscript. Formally, for $X \in \{ WR, RW, WW \}$ we have

$$X_M(t,t') = X_M^{\mathcal{A}}(t,t').$$

We add two more types of races, both of which are a subcases of the write-read race.

The first is code modification. Step t *modifies code* of step t' if it writes into the fetched addresses of that step

$$CM_M(t,t') \equiv out_M(t) \dot{\cap} F_M(t').$$

We also consider situations where outputs of a step have been overwritten by other steps, in which case we might no longer see that a write-read race originally took place. In such a case, one might be able to drop the writing step from the schedule without affecting the steps behind it, even though there is a write-read race. We call situations in which the visible outputs still overlap with the inputs of the later step *visible write-read races*

$$VR_M(t,t') \equiv vout_M(t,t') \dot{\cap} in_M(t').$$

Note that this is the only type of race which makes only sense if $t < t'$, due to the fact that the visible outputs do not make sense in the other case.

We state four simple lemmas about races. We do not show proofs. In the high-level machine, sequentially consistent buffered code modifications are write-read races in the fetch region.

Lemma 155.

$$SC_\uparrow(t) \wedge CM_\uparrow(t,t') \rightarrow WR_\uparrow^{idc(F_\uparrow(t'))}(t,t').$$

A race in some memory region is a race.

Lemma 156.

$$X \in \{ WR, RW, WW \} \wedge X^A_M(t,t') \to X_M(t,t').$$

Races can be defined in terms of output and input intersections.

Lemma 157.

$$\begin{aligned} WR_M(t,t') &\equiv out_M(t) \dot{\cap} in_M(t'), \\ WW_M(t,t') &\equiv out_M(t) \dot{\cap} out_M(t'), \\ RW_M(t,t') &\equiv in_M(t) \dot{\cap} out_M(t'). \end{aligned}$$

Code modifications and visible write-read races are write read races.

Lemma 158.

$$CM_M(t,t') \vee VR_M(t,t') \to WR_M(t,t').$$

In the software conditions, we will usually only consider races where steps t and t' are not just concurrent, but also adjacent, i.e., $t' = t+1$. That this suffices to also detect races where t' is much later than t is due to the fact that we will consider all schedules, and if there is no sequence of synchronizing memory operations (e.g., locking operations) that prevents us from reordering the schedule in such a way that the memory operations at t and t' occur directly one after another, we can always construct a new schedule where the race consists of adjacent memory operations. That this is really true is shown with Lemma 258 on page 241.

We show that only memory writes cause write races with steps made by a different unit, or object-concurrent steps.

Lemma 159.

$$diffu(t,k) \wedge (WR_M(t,k) \vee WW_M(t,k)) \to mwrite_M(t).$$

Proof. We have by assumption that there is a write-read or write-write race between those steps, and thus that the outputs of step t and inputs or outputs of step k intersect

$$out_M(t) \dot{\cap} in_M(k) \cup out_M(k).$$

The claim is now a special case of Lemma 122. □

We show that object concurrent write-read and write-write races are also only made by memory writes.

Lemma 160.

$$ocon_M(t,k) \wedge (WR_M(t,k) \vee WW_M(t,k)) \to mwrite_M(t).$$

Proof. By definition, the steps are made by different objects and step t does not modify processor registers of the unit making step k

$$o_M(t) \neq o_M(k) \wedge out_M(t) \not\dot{\cap} C_M(k).$$

With Lemma 106 we obtain that the steps are either made by different units, or by a processor and its write buffer in one order, or in the other order. We consider these three cases.

$diffu(t,k)$**:** The claim is just Lemma 159.

$s(t) \in \Sigma_{P,i} \wedge s(k) \in \Sigma_{WB,i}$**:** The only inputs of a write buffer step are its core registers

$$in_M(k) = C_M(k) \cup F_M(k) \cup R_M(k) = C_M(k).$$

Since step t by assumption is object concurrent and thus does not modify those

$$out_M(t) \not\cap C_M(k),$$

there is no intersection between the inputs of step k and the outputs of step t

$$out_M(t) \not\cap in_M(k)$$

and thus we can exclude a write-read race

$$\neg WR_M(t,k).$$

This leaves only the write-write race

$$WW_M(t,k),$$

and we have thus an intersection between the outputs

$$out_M(t) \dot{\cap} out_M(k).$$

For the write-write race we obtain with Lemmas 4 and 26 that the domain of the write of step k are bufferable addresses

$$\begin{aligned} out_M(k) &= idc(Dom(W_M(k))) \\ &\subseteq idc(BA) && \text{L 26} \\ &= BA. && \text{L 4} \end{aligned}$$

Consequently the outputs of t intersect with bufferable addresses

$$out_M(t) \dot{\cap} BA.$$

Since bufferable addresses do not include normal processor registers

$$BA \not\cap A_{NPR,u_M(t)},$$

there are some outputs that are not normal processor registers either

$$out_M(t) \not\subseteq A_{NPR,u_M(t)}.$$

Therefore step t is, as claimed, a memory write

$$mwrite_M(t).$$

$s(t) \in \Sigma_{WB,i} \wedge s(k) \in \Sigma_{P,i}$**:** The outputs of t are buffered addresses

$$\begin{aligned} out_M(t) &= idc(Dom(W_M(t))) \\ &\subseteq idc(BA) && \text{L 26} \\ &= BA. && \text{L 4} \end{aligned}$$

Since there is a write-write or write-read race, the outputs of step t intersect with some set X (either outputs or inputs of step k)

$$out_M(t) \barwedge X$$

and are thus non-empty

$$out_M(t) \neq \emptyset.$$

Since the outputs are buffered addresses and those do not include normal processor registers

$$BA \barwedge A_{NPR,u_M(t)},$$

there are some outputs that are not normal processor registers either

$$out_M(t) \not\subseteq A_{NPR,u_M(t)}.$$

Therefore step t is, as claimed, a memory write

$$mwrite_M(t).$$

□

When the outputs of a step made with input x do not affect the inputs of the next step made with input y, the step with input y could be executed first. By that we mean that the configuration after executing the step with input x strongly agrees with the configuration before that step, when stepped with y.

Lemma 161.

$$diffu(x,y) \wedge out_M(c,x) \barwedge in_M(c \blacksquare_M x,y) \rightarrow c \blacksquare_M x =^{y}_{M} c.$$

Proof. We apply Lemma 188, which gives us the following two subclaims.

Memory agrees: We have to show that the memory agrees on the inputs

$$c \blacksquare_M x.m \stackrel{!}{=}_{in_M(c \blacksquare_M x,y)} c.m.$$

This is exactly Lemma 138.

Same Buffers: By assumption the steps are made by different units

$$u_M(c,x) \neq u_M(c \blacksquare_M x,y).$$

We have to show that the write buffers are the same

$$c \blacksquare_M x.wb \stackrel{!}{=}_{u_M(c \blacksquare_M x,y)} c.wb.$$

This is exactly Lemma 96.

□

Similarly, the first step could be executed in the second position, but only if the second step also does not modify inputs of the first step.

Lemma 162.

$$diffu(x,y) \wedge out_M(c,x) \barwedge in_M(c \blacksquare_M x,y) \wedge out_M(c \blacksquare_M x,y) \barwedge in_M(c,x) \rightarrow c \blacksquare_M y =^{x}_{M} c.$$

Proof. With Lemma 161 we obtain that y could be stepped before x

$$c \blacksquare_M x =^y_M c,$$

and with Lemma 144 it has the same outputs when stepped before x

$$out_M(c,y) = out_M(c \blacksquare_M x, y).$$

Thus the step in its new position does not modify any inputs of x in its old position

$$out_M(c,y) \mathbin{\dot{\not\cap}} in_M(c,x),$$

and with Lemma 138 we obtain that the configurations before x agree on the inputs of step x

$$c.m =_{in_M(c,x)} c \blacksquare_M y.m.$$

With Lemma 142 we obtain that the step still has the same inputs

$$in_M(c,x) = in_M(c \blacksquare_M y, x),$$

and therefore there is no intersection between the outputs of the now first step and the inputs of the now second step

$$out_M(c,y) \mathbin{\dot{\not\cap}} in_M(c \blacksquare_M y, x).$$

Obviously $diffu$ is symmetric

$$diffu(y,x),$$

and the claim is Lemma 161. □

When there is also no intersection of the outputs of the two steps, then the resulting configuration is in fact equal.

Lemma 163. *Let x, y be steps of different units*

$$diffu(x,y).$$

Then if the steps can be swapped and still be executed at their new positions and there is also no write-write race between them

- $c \blacksquare_M x =^y_M c$,
- $c \blacksquare_M y =^x_M c$,
- $out_M(c,x) \mathbin{\dot{\not\cap}} out_M(c \blacksquare_M x, y)$,

then the order of the steps is irrelevant

$$c \blacksquare_M y \blacksquare_M x = c \blacksquare_M x \blacksquare_M y.$$

Proof. We show the equality of the final configurations component-wise.

$c \blacksquare_M y \blacksquare_M x.m = c \blacksquare_M x \blacksquare_M y.m$**:** By assumption the outputs do not intersection

$$out_M(c,x) \not\frown out_M(c \blacksquare_M x, y).$$

By definition, these are the inclusive device closures of the domains of the writes during those steps, which therefore also do not intersect

$$idc(Dom(W_M(c,x))) \not\frown idc(Dom(W_M(c \blacksquare_M x, y))).$$

The claim follows with Lemmas 30 and 144

$$\begin{aligned}
c \blacksquare_M y \blacksquare_M x.m &= c.m \circledast W_M(c,y) \circledast W_M(c \blacksquare_M y, x) \\
&= c.m \circledast W_M(c \blacksquare_M x, y) \circledast W_M(c,x) && \text{L } 144 \times 2 \\
&= c.m \circledast (W_M(c,x) \cup W_M(c \blacksquare_M x, y)) && \text{L } 30 \\
&= c.m \circledast W_M(c,x) \circledast W_M(c \blacksquare_M x, y) && \text{L } 30 \\
&= c \blacksquare_M x \blacksquare_M y.m.
\end{aligned}$$

$c \blacksquare_M y \blacksquare_M x.wb(i) = c \blacksquare_M x \blacksquare_M y.wb(i)$**:** The two steps are made by different units

$$diffu(x,y),$$

and the claim follows with Lemmas 100 and 144

$$\begin{aligned}
& c \blacksquare_M y \blacksquare_M x.wb(i) \\
&= Op_{iM}(c \blacksquare_M y, x)(Op_{iM}(c,y)(c.wb(i), BW_M(c,y)), BW_M(c \blacksquare_M y, x)) \\
&= Op_{iM}(c,y)(Op_{iM}(c \blacksquare_M y, x)(c.wb(i), BW_M(c \blacksquare_M y, x)), BW_M(c,y)) && \text{L } 100 \\
&= Op_{iM}(c \blacksquare_M x, y)(Op_{iM}(c,x)(c.wb(i), BW_M(c,x)), BW_M(c \blacksquare_M x, y)) && \text{L } 144 \\
&= c \blacksquare_M x \blacksquare_M y.wb(i).
\end{aligned}$$

□

We lift these Lemmas to schedules. We only show the proof for the first one, the others are analogous.

Lemma 164.

$$diffu(t,t+1) \wedge \neg WR_M(t,t+1) \rightarrow c_M^{t+1} =_M^{s(t+1)} c_M[t \leftrightarrow t+1]^t.$$

Proof. Because there is no race, the outputs and inputs do not intersect

$$out_M(t) \not\frown in_M(t+1).$$

The configuration at t is before the reordered portion and is thus the same in both schedules

$$c_M^t = c_M[t \leftarrow t+1]^t,$$

whereas the configuration at $t+1$ is the configuration at t stepped with $s(t)$

$$c_M^{t+1} = c_M^t \blacksquare_M s(t).$$

Rewriting this changes the claim to the following

$$c_M^t \blacksquare_M s(t) =_M^{s(t+1)} c_M^t.$$

The claim is now Lemma 161. □

Lemma 165.

$$diffu(t,t+1) \wedge \neg WR_M(t,t+1) \wedge \neg RW_M(t,t+1) \rightarrow c_M^t =_M^{s(t)} c_M[t \leftrightarrow t+1]^{t+1}.$$

Lemma 166. *If adjacent steps are made by different units and without races*

$$diffu(t,t+1) \wedge \neg WR_M(t,t+1) \wedge \neg RW_M(t,t+1) \wedge \neg WW_M(t,t+1)$$

their order can be swapped without affecting the final configuration

$$c_M^{t+2} = c_M[t \leftrightarrow t+1]^{t+2}.$$

Consequently, if two adjacent steps are made by different units and there is no race between them, they commute.

Lemma 167.

$$diffu(t,t+1) \wedge \neg(WR_M(t,t+1) \vee RW_M(t,t+1) \vee WW_M(t,t+1)) \rightarrow s \equiv_M s[t \leftrightarrow t+1].$$

Proof. Note now that changing the order of t and $t+1$ is the same as moving $t+1$ to t

$$s[t \leftrightarrow t+1] = s[t \leftarrow t+1],$$

which reduces our goal to showing that this move preserves equivalence

$$s \stackrel{!}{\equiv}_M s[t \leftarrow t+1].$$

We now apply Lemma 118 with $k := t+1$, and reduce the claim to the following three subgoals.

$c_M^{t+1} =_M^{s(t+1)} c_M[t \leftarrow t+1]^t$**:** This is Lemma 164.

$t' \in [t : t+1) \rightarrow c_M^{t'} =_M^{s(t')} c_M[t \leftarrow t+1]^{t'+1}$**:** Clearly the only such t' is t

$$t' = t.$$

Substituting this reduces the claim to the following

$$c_M^t =_M^{s(t)} c_M[t \leftarrow t+1]^{t+1},$$

which is Lemma 165.

$c_M^{t+2} = c_M[t \leftarrow t+1]^{t+2}$**:** This is Lemma 166.

□

4.5 Conditions

We now formalize the software conditions for write buffer reduction. We assume for all schedules s for the high-level machine and all t that all of the following conditions hold.

Races In a valid schedule, racing accesses are shared, i.e., when a unit-concurrent step satisfies the local guard condition and the steps before it are valid

$$ucon_\uparrow(t,t+1) \wedge \Gamma_\uparrow^{t+1}(s),$$

both of the following are true

If there is a write-write race, both steps have to be shared

$$WW_\uparrow^{A_{MEM}}(t,t+1) \rightarrow Sh_\uparrow(t) \wedge Sh_\uparrow(t+1).$$

If there is a write-read race, both steps have to be shared but the read has to be a shared read

$$WR_\uparrow^{A_{MEM}}(t,t+1) \rightarrow Sh_\uparrow(t) \wedge ShR_\uparrow(t+1).$$

We show with Lemma 210 that this definition is strong enough to account for read-write races as well as races in any register.

Flush In a semi-valid schedule, a processor in strong memory mode which might be buffering shared writes in the low-level machine:

$$\Gamma\Phi_\uparrow^t(s) \wedge s(t) \in \Sigma_{P,i} \wedge SC_\uparrow(t) \wedge \exists t' \in issue_\uparrow^t(i).Sh_\uparrow(t')$$

never issues a shared read

$$\neg ShR_\uparrow(t).$$

Note that the dirty bit of Cohen and Schirmer [CS10] is an elegant approximation of whether there is a shared write in the buffer. The dirty bit is set to one when a shared write enters the buffer, and is reset to zero when the buffer is flushed. The only downside to the dirty bit is that it ignores the effect of drains due to partial hits. Partial hits are extremely useful because they only partially drain the buffer and only as far as necessary, and can be used to simulate shared read and write fences (cf. Chapter 3).

Switch Modifying memory mode registers is only possible in a valid schedule when the write buffer is empty and stays empty:

$$\Gamma_\uparrow^t(s) \wedge A_{SC,i} \dot{\cap} out_\uparrow(t) \rightarrow wb_\uparrow^t(i) = \varepsilon = wb_\uparrow^{t+1}(i)).$$

TSO Processors in strong memory mode roughly obey TSO. This means that when

$$s(t) \in \Sigma_{P,i} \wedge SC_\uparrow(t),$$

we require all of the following:

IRRForwarding We never break local write/read ordering by buffering writes to local registers in a valid schedule

$$\Gamma_\uparrow^t(s) \wedge A_{PR,j} \not\dot{\cap} Dom(PW_\uparrow(t).wba).$$

Note that this includes modifications to other processors.

CodeOrder A semi-valid schedule never breaks local write/fetch order by fetching while a more up-to-date value is in the buffer

$$\Gamma\Phi^t_\uparrow(s) \rightarrow \neg hit(idc(F_\uparrow(t)), wb^t_\uparrow(i)).$$

WriteReadOrder A valid schedule never breaks local write/read order by reading directly from memory while a more up-to-date value is in the buffer

$$\Gamma^t_\uparrow(s) \rightarrow \neg hit(idc(R_\uparrow(t).bpa), wb^t_\uparrow(i)).$$

WriteWriteOrder A valid schedule never breaks local write/write ordering by writing directly to memory while an older value is still in the buffer

$$\Gamma^t_\uparrow(s) \rightarrow \neg hit(idc(Dom(PW_\uparrow(t).bpa)), wb^t_\uparrow(i)).$$

Note that there is no analogous condition for core registers, since writes to the core registers can never be buffered based on Condition IRRForwarding above.

AtomicWrite A shared write in a valid schedule is either buffered or bypassing, but never both

$$\Gamma^t_\uparrow(s) \wedge Sh_\uparrow(t) \rightarrow Dom(PW_\uparrow(t).wba) = \emptyset \vee Dom(PW_\uparrow(t).bpa) \subseteq A_{NPR,i}.$$

Otherwise, half of the write would become visible before the other half in the low-level machine, which is not sequentially consistent since the high-level machine executes the bypassing writes and buffered writes atomically. The normal processor registers can be ignored in this matter because they never become visible to other processors.

MessagePassing In a valid schedule, a shared write must not overtake other writes, i.e., when we might issue a shared bypassing write the buffer must be empty:

$$\Gamma^t_\uparrow(s) \wedge Sh_\uparrow(t) \wedge Dom(PW_\uparrow(t).bpa) \not\subseteq A_{NPR,i} \rightarrow wb^t_\uparrow(i) = \varepsilon.$$

Violating this condition destroys message-passing, e.g., when updating a concurrent list, unshared modifications to the list element might not be visible to other processors that see the shared write to the tail pointer which adds the element to the list.

AtomicRMW In a valid schedule there are no shared read-modify-writes which are buffered

$$\Gamma^t_\uparrow(s) \wedge ShR_\uparrow(t) \rightarrow Dom(BW_\uparrow(t)) = \emptyset.$$

These would appear to be atomic to observers in the high-level machine, while not being atomic in the low-level machine. That is not sequentially consistent.

Otherwise, it is possible to hide races by buffering code modifications.

CodeMod In a semi-valid schedule, a concurrent code modification is correctly annotated

$$\Gamma\Phi^{t+1}_\uparrow(s) \wedge ucon_\uparrow(t, t+1) \wedge CM_\uparrow(t, t+1) \rightarrow Sh_\uparrow(t) \wedge ShR_\uparrow(t+1).$$

RMWRace In a semi-valid schedule, a race with an RMW is never hidden by a buffered write. Formally, consider a feasible sequentially consistent step at $l+1$ with a buffered write from $k'+1$[1]

$$\Gamma\Phi^{l+1}_{\uparrow}(s) \wedge SC_{\uparrow}(l+1) \wedge s(l+1) \in \Sigma_{P,i} \wedge k'+1 \in issue^{l+1}_{\uparrow}(i),$$

and all steps between $k'+1$ and $l+1$ are made by the same object (i.e., processor i)

$$\forall t' \in [k'+1 : l+1].o_{\uparrow}(t') = o_{\uparrow}(k'+1),$$

then the buffered write may only read-write race a unit-concurrent instruction directly preceding it

$$ucon_{\uparrow}(k', k'+1) \wedge RW_{\uparrow}(k', k'+1)$$

if all of the following hold.

1. There is no visible write-read race:

$$\neg VR_{\uparrow}(k', l+1)$$

2. If there is a valid write-write race, step $l+1$ is shared

$$\Gamma_{\uparrow}(l+1) \wedge WW_{\uparrow}(k', l+1) \rightarrow Sh_{\uparrow}(l+1).$$

3. If there is a valid read-write race, step $l+1$ is shared

$$\Gamma_{\uparrow}(l+1) \wedge RW_{\uparrow}(k', l+1) \rightarrow Sh_{\uparrow}(l+1).$$

The last condition is annoying for two reasons. First, it is complicated and hard to decipher. We give in Figures 4.1a and 4.1b examples of situations in which the condition requires us to insert a fence. Consider the following variant of the Program from page 97, which corresponds roughly to Figure 4.1a.

```
x[0:7].store(1); || x[0:15].cas(0 → 2^8);
y = x[8:15];     ||
```

In all sequentially consistent configurations, the store in that program is executed before the load, and the compare-and-swap never directly races with the load. We show all sequentially consistent computations in which the compare-and-swap instruction and the load are executed in adjacent steps. In the first such schedule, the load is executed first:

```
x[0:7].store(1); ||
y = x[8:15];     ||
                 || x[0:15].cas(0 → 2^8);
```

In the second such schedule, the load is executed second:

```
x[0:7].store(1); ||
                 || x[0:15].cas(0 → 2^8);
y = x[8:15];     ||
```

[1] Note that the variables can be chosen more nicely, in particular $l+1$ could be aliased as t; but this form is closer to the description given in the introduction and to the version used in the proof.

In each case, the prior store operation makes the compare-and-swap fail, and therefore the compare-and-swap never is a write when executed next to the load. Condition Races never applies and never forces us to mark the load as shared. Furthermore, these are the only executions in which y loads a zero. In each case x ends up as one.

In the write buffer machine, however, the store can still be buffered when the load and the compare-and-swap are executed, causing not only a race, but also a computation where y loads a zero, but x ends up as 2^8+1:

```
                                                    x     y
x[0:7].store(1); -> BUF                             0     0
y = x[8:15];                                        0     0
                                 x[0:15].cas(0 → 2^8);  2^8   0
BUF -> x[0:7].store(1)                              2^8+1 0
```

Clearly this computation can not be simulated by the high-level machine. The program violates Condition RMWRace in the following execution of the high-level machine

```
  k                                 x[0:15].cas(0 → 2^8);
k+1   x[0:7].store(1);
l+1   y = x[8:15];
```

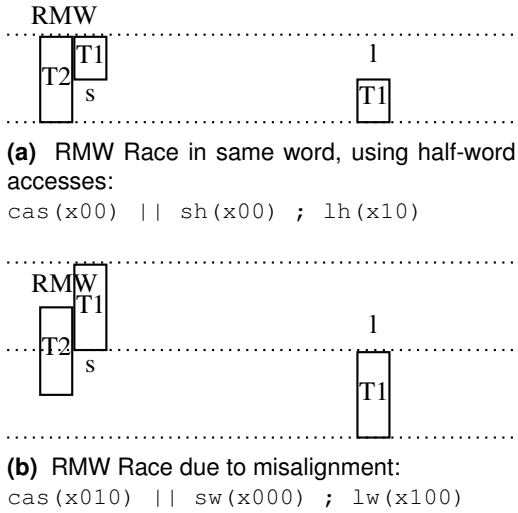
In this case, k is the step in which the compare-and-swap is (successfully) executed, $k+1$ is the step in which the store to x is buffered, and step $l+1$ is the step where the load to x is executed. The store is still buffered in step $l+1$, and the steps from $k+1$ until $l+1$ (which are just those two steps) are made by the same object, the processor executing Thread 1. Since the compare-and-swap is successfully executed and the store in step $k+1$ does not completely overwrite the modification of the compare-and-swap, there is a visible write-read race between k and $l+1$, which contradicts the condition.

The condition in fact detects all such programs by considering the execution in which the RMW (or instruction in general) is successfully executed, and looking for races that cover a long distance. That this only rarely creates false positives is simply explained by the fact that in the low-level machine, the store and all subsequent stores can be delayed until $l+1$ by simply being buffered. We can then delay the RMW until we find a race in the low-level machine that can not be explained by an execution in the high-level machine, such as the one above. That it finds all such inconsistencies is hard to prove but easily explained: if there was an inconsistency, it must be because both a) there is a buffered store that causes the memory configuration of the two machines to diverge and b) that store changes the effect of the RMW. Then by simply reversing the argument from before and moving the RMW to the front of the buffered store we obtain a sequentially consistent execution in which the RMW is executed directly before the store that changes its effect, and the race between the RMW and the step at $l+1$ persists.

The formal proof that this really works is one of the central results of this thesis, and the source of most of the effort in Section 4.12.

The second reason for why the condition is annoying is that it requires extensive backtracking in verification — every time we execute a write, we have to record whether it was racing with an RMW, the effect of the RMW, etc., until the write is finally drained. Both points can be resolved by using a less efficient simplification, and we give some examples for such simplifications.

- If a write only races with an RMW if the footprint of the write subsumes the footprint of the RMW, there is never an intersection of the visible outputs and



(a) RMW Race in same word, using half-word accesses:
`cas(x00) || sh(x00) ; lh(x10)`

(b) RMW Race due to misalignment:
`cas(x010) || sw(x000) ; lw(x100)`

Figure 4.1: An RMW Race: An RMW of Thread 2 races first with a store, then a load of Thread 1, on disjoint addresses. Accessed addresses are indicated by vertical size and position of the block, word alignment is indicated by dotted lines. Steps that are more to the right occur later.

the inputs because the visible outputs will be empty. This is the case in all architectures with aligned, same-sized memory accesses and no interrupts, such as the ones considered in the work of Cohen and Schirmer [CS10] or Owens [Owe10]. It is also easy to implement, but only works for high-level languages that make this restriction.

- One can strengthen the machine model (and thus obtain a stronger memory model than TSO) so that the write is drained by the hardware before executing the read. This is the case in Chen's PhD thesis [Che16], where the footprint of memory operations must always be within an aligned machine-word, and the definition of a partial hit is weakened to include situations in which the footprint of the write and the read do not need to overlap, i.e., all writes that modify the same machine-word as a later read trigger a partial hit and are drained from the buffer. This situation is shown in Figure 4.2. We are aware of no other work with mixed-size accesses.

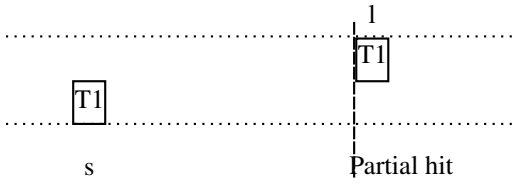


Figure 4.2: In Chen's PhD Thesis, a store operation and a later load operation cause a partial hit, even though there is no actual hit.

In general a stronger machine model with support for shared instructions or partial barriers would be more efficient than software-based solutions, which have to

over-approximate (e.g., draining the whole buffer to get rid of the shared writes rather than draining the buffer partially), and would allow additional hardware optimizations (e.g., write-merging on non-shared stores to the same cache line).

- Similarly one can place after every shared write a fence. This is the easiest solution and also the one used in state-of-the-art compilers, but it is also the least efficient solution.

- If one considers a concurrent access of an RMW and a read a race even if the RMW is not modifying memory (e.g., in case of a compare-and-swap because it is unsuccessful), then the normal flushing condition between the shared write and the shared read already prevents the situation altogether. We are not aware of any program where this simplification causes additional flushes, in particular because unsuccessful compare-and-swap operations usually only race with reads if there are also concurrent potentially successful compare-and-swap operations, as is the case in most lock-free algorithms. In this case there are schedules in which one of the compare-and-swap operations is successful directly before the read, which thus causes the read to be marked as shared. To make this work, races would have to be defined in terms of potential outputs *Pout* that 1) can be computed solely based off of the current core registers, similar to F

  $$Pout_M(c,x) = Pout(core_M(c,x),x)$$

  and 2) always subsume the outputs

  $$out_M(c,x) \subseteq Pout_M(c,x).$$

  Then one could define the write-read race in terms of *Pout* as follows

  $$WR'^A_\uparrow(t,t') = Pout_\uparrow(t) \cap in_\uparrow(t') \hat{\cap} A.$$

Note that Conditions Races and CodeMod require us to annotate certain steps under certain circumstances. Under those circumstances we informally say that the steps are "correctly annotated" when the steps have the annotation required by those conditions. For example, if there is a feasible code modification, the steps are correctly annotated if the writing step is shared and the fetching step is a shared read.

Note now that our assumptions are only on the OS code and the hardware[2]. Due to this, one can verify a program against this discipline without actually considering the exact user code or interleavings created by write buffers, and we quickly explain why that is the case. In order to detect races between user code and OS code, one can use the fact that users run in translated mode and thus only access page tables (during MMU steps) and pages for which they have sufficient rights (during user program steps). Thus one can abstract away from the actual user code and simply assume the worst case, namely that all addresses to which currently running users and MMUs have access have to be accessed shared by the OS. One small problem with this is the fact that changing the page tables immediately changes the access rights, but there may still be "legacy writes" in the buffer of user processors which were issued under the old rights; such situations have to either be modeled or, more likely, prevented, in order to have a sound model without users. Write buffers then become irrelevant in a sense because

[2]For example, we assume that the OS and hardware can restrict somehow the set of addresses that the user can access, e.g., by a memory management unit.

they are not considered in user mode and only create no-ops in OS mode, and are only used for the discipline in form of the write buffer. Again one can simply assume the worst case, namely that writes stay in the buffer as long as possible, i.e., until they are explicitly drained by a fence, partial hit, or IPI. The details are future work.

The main result of this thesis is now as follows. If every schedule satisfies the conditions, then for every schedule s that is valid and IPI-valid in the low-level machine

$$(\forall t.\Gamma_{\downarrow}[s](t) \wedge \Delta_{IPI\downarrow}[s](t))$$

there is an equivalent schedule s' which is also valid and IPI-valid, and where every step of the low-level machine strongly agrees with he step in the high-level machine

$$\exists s'.s' \equiv_{\downarrow} s \wedge \forall t.\Gamma_{\downarrow}[s'](t) \wedge \Delta_{IPI\downarrow}[s'](t) \wedge c_{\downarrow}[s']^t =_{\downarrow,\uparrow}^{s'(t)} c_{\uparrow}[s']^t.$$

4.6 Strengthening the Conditions

We will now prove that the conditions are strong enough to imply stronger or more useful variants, as well as several key facts.

We show that while a write is buffered, the memory mode is not changed. This means that a write that entered the buffer in strong memory mode can not leave the buffer in weak memory mode (and thus, e.g., be executed twice: once when it was issued in strong memory mode, and once when it leaves the buffer in weak memory mode) or vice versa.

Lemma 168.

$$\Gamma_{\uparrow}^{k-1}(s) \wedge t \in issue_{\uparrow}^{k}(i) \rightarrow SC_{i\uparrow}(t) = SC_{i\uparrow}(k).$$

Proof. By the monotonicity of the write buffer, step t is an issued time-stamp in all configurations $t' \in [t+1:k]$

$$t \in issue_{\uparrow}^{t'}(i).$$

By Lemma 123, the write buffered at t is thus always in the buffer

$$BW_{\uparrow}(t) \in wb_{\uparrow}^{t'}(i)$$

and thus the buffer is never empty

$$wb_{\uparrow}^{t'}(i) \neq \varepsilon.$$

By contraposition of Condition Switch we obtain that the mode registers were not an output of step $t'-1$

$$A_{SC,i} \not\parallel out_{\uparrow}(t'-1),$$

and thus by Lemma 139 the mode is not changed by any of the steps $t'-1 \in [t:k-1)$

$$SC_{i\uparrow}(t) = \ldots =_{A_{SC,i}} SC_{i\uparrow}(k).$$

□

When a write buffer makes a step that satisfies the write buffer drain condition, its write buffer is not empty.

Lemma 169.

$$s(t) \in \Sigma_{WB,i} \wedge \Lambda_{\uparrow}(t) \rightarrow wb_{\uparrow}^{t}(i) \neq \varepsilon.$$

Proof. Because the step is a write buffer step, the local drain condition is satisfied. By definition of the local drain condition, this means that the write buffer is non-empty

$$\begin{aligned} 1 &\equiv \Lambda_{\uparrow}(t) \\ &\equiv \Delta_{\uparrow}(t) \\ &\equiv \Delta_{WB}(wb^t_{\uparrow}(i)) \\ &\equiv wb^t_{\uparrow}(i) \neq \varepsilon, \end{aligned}$$

which is the claim. □

In a valid schedule, a step can not change the memory mode registers of a unit right before the write buffer of that unit makes a step that satisfies the write buffer drain condition.

Lemma 170.

$$s(t+1) \in \Sigma_{WB,i} \wedge \Gamma^t_{\uparrow}(s) \wedge \Lambda_{\uparrow}(t+1) \rightarrow A_{SC,i} \not\parallel out_{\uparrow}(t).$$

Proof. By Lemma 169, the write buffer of unit i is non-empty in step $t+1$

$$wb^{t+1}_{\uparrow}(i) \neq \varepsilon.$$

The claim is now the contraposition of Condition Switch. □

Since valid write buffer steps always satisfy the write buffer drain condition, we obtain immediately the following

Lemma 171.

$$s(t+1) \in \Sigma_{WB,i} \wedge \Gamma^t_{\uparrow}(s) \wedge \Gamma_{\uparrow}(t+1) \rightarrow A_{SC,i} \not\parallel out_{\uparrow}(t).$$

It can also not change the mode registers if it is a write buffer step of that unit itself.

Lemma 172.

$$s(t) \in \Sigma_{WB,i} \wedge \Gamma^t_{\uparrow}(s) \rightarrow A_{SC,i} \not\parallel out_{\uparrow}(t).$$

Proof. Because the step is valid, the local drain condition and thus the write buffer drain condition are satisfied

$$\Gamma_{\uparrow}(t) \implies \Delta_{\uparrow}(t) \implies \Lambda_{\uparrow}(t).$$

By Lemma 169, the write buffer of unit i is non-empty in step t

$$wb^t_{\uparrow}(i) \neq \varepsilon.$$

The claim is now the contraposition of Condition Switch. □

Furthermore, a processor in sequential mode never has a buffered write to a device that it is trying to load from.

Lemma 173.

$$\Gamma^t_{\uparrow}(s) \wedge s(t) \in \Sigma_{P,i} \wedge SC_{i\uparrow}(t) \rightarrow \neg hit(dc(in_{\uparrow}(t)), wb^t_{\uparrow}(i))$$

Proof. Assume for the sake of contradiction a hit

$$hit(dc(in_{\uparrow}(t)), wb^{t}_{\uparrow}(i)).$$

By Lemma 127, the hit was issued at some timestamp t'

$$\exists t' \in issue^{t}_{\uparrow}(i).Dom(BW_{\uparrow}(t')) \dot{\cap} dc(in_{\uparrow}(t)).$$

Because the device closure only contains device addresses, the domain of the buffered write must contain some device address. The device closure can also not be empty, and thus the inputs must also contain some device address.

$$A_{DEV} \dot{\cap} Dom(BW_{\uparrow}(t')) \wedge A_{DEV} \dot{\cap} in_{\uparrow}(t).$$

Clearly step t' which added a write to the buffer of processor i was made by processor i

$$s(t') \in \Sigma_{P,i}.$$

By Lemma 168, step t' was made in strong memory mode

$$SC_{\uparrow}(t') = SC_{\uparrow i}(t') = SC_{\uparrow i}(t) = 1.$$

By Lemma 92 the buffered write was therefore executed and part of the outputs

$$Dom(BW_{\uparrow}(t')) \subseteq out_{\uparrow}(t')$$

and the outputs of step t' intersected with the registers of some device

$$out_{\uparrow}(t') \dot{\cap} A_{DEV}$$

By definition of *volW* resp. *volR* step t' is a volatile write and step t is a volatile read

$$volW_{\uparrow}(t') \wedge volR_{\uparrow}(t)$$

and step t' by definition is volatile

$$vol_{\uparrow}(t').$$

Thus by Lemma 130 step t' is shared and by 131 step t is a shared read

$$Sh_{\uparrow}(t') \wedge ShR_{\uparrow}(t).$$

We conclude that t' is a shared buffered write still existing in the buffer of i when t is executing a shared read

$$(t' \in issue^{t}_{\uparrow}(i) \wedge Sh_{\uparrow}(t')) \wedge ShR_{\uparrow}(t),$$

and clearly the step is in strong memory mode

$$SC_{\uparrow}(t) = SC_{i\uparrow}(t) = 1,$$

which contradicts our Condition Flush. □

Sequentially consistent write buffer steps have no effect.

Lemma 174.

$$x \in \Sigma_{WB,i} \wedge SC_{\uparrow}(c,x) \rightarrow W_{\uparrow}(c,x) = \emptyset \wedge out_{\uparrow}(c,x) = \emptyset$$

Proof. The machine is using high-level machine semantics

$$\neg LL_{\uparrow}(c,x)$$

and the first claim follows by definition

$$W_{\uparrow}(c,x) = \emptyset.$$

The second claim immediately follows

$$\begin{aligned} out_{\uparrow}(c,x) &= idc(WS_{\uparrow}(c,x)) = idc(Dom(W_{\uparrow}(c,x))) \\ &= idc(Dom(\emptyset)) = idc(\emptyset) \\ &= \emptyset \cup dc(\emptyset) \\ &= \emptyset. \end{aligned}$$

□

Thus a write buffer step can never switch to weak memory mode.

Lemma 175.

$$x \in \Sigma_{WB,i} \wedge SC_{i\uparrow}(c) \rightarrow SC_{i\uparrow}(c \bullet_{\uparrow} x).$$

Proof. By definition the step is made in sequentially consistent semantics

$$SC_{\uparrow}(c,x) = SC_{i\uparrow}(c).$$

With Lemma 174 we obtain that there are no outputs

$$out_{\uparrow}(c,x) = \emptyset.$$

The claim follows with Lemma 138

$$c.m =_{\mathcal{A}} c \bullet_{\uparrow} x.m.$$

□

Writes to the IRR are never buffered by a processor in strong memory mode in the high-level machine in valid schedules.

Lemma 176.

$$\Gamma_{\uparrow}^{t-1}(s) \wedge SC_{i\uparrow}(t) \rightarrow \neg hit(A_{PR,j}, wb_{\uparrow}^{t}(i)).$$

Proof. By induction on t. The base case is trivial.

In the inductive step $t \rightarrow t+1$, we distinguish whether the processor was in strong memory mode at t or not.

$SC_{i\uparrow}(t)$**:** By the induction hypothesis, there is no such write in the buffer at t

$$\neg hit(A_{PR,j}, wb_{\uparrow}^{t}(i)).$$

The write buffer at $t+1$ is either the same, or a suffix, or increased by one element

$$wb_{\uparrow}^{t+1} = \begin{cases} push(wb_{\uparrow}^{t}, \ldots) & s(t) \in \Sigma_{P,i} \wedge BW_{\uparrow}(t) \neq \emptyset \\ pop(wb_{\uparrow}^{t}, \ldots) & s(t) \in \Sigma_{WB,i} \\ noop(wb_{\uparrow}^{t}, \ldots) & \text{o.w.} \end{cases}$$

We focus on the only difficult case where the write buffer is increased. Assume thus that the step is a processor step of unit i pushing a new write

$$s(t) \in \Sigma_{P,i} \wedge BW_{\uparrow}(t) \neq \emptyset.$$

By Condition IRRForwarding, the step is not buffering a write to processor registers

$$A_{PR,j} \mathbin{\dot{\not\cap}} Dom(BW_{\uparrow}(t))$$

and the claim follows with Lemma 51

$$\begin{aligned} hit(A_{PR,j}, wb_{\uparrow}^{t+1}(i)) &\equiv hit(A_{PR,j}, push(wb_{\uparrow}^{t}(i), BW_{\uparrow}(t))) \\ &\equiv hit(A_{PR,j}, wb_{\uparrow}^{t}(i) \circ BW_{\uparrow}(t)) \\ &\equiv hit(A_{PR,j}, wb_{\uparrow}^{t}(i)) \vee hit(A_{PR,j}, BW_{\uparrow}(t)) \\ &\equiv 0 \vee 0 \\ &\equiv 0. \end{aligned}$$

$\neg SC_{i\uparrow}(t)$**:** The processor is now in strong memory mode. Thus the memory mode has been changed

$$SC_{i\uparrow}(t+1) = 1 \neq 0 = SC_{i\uparrow}(t).$$

By contraposition of Lemma 139, the mode registers were an output of step t

$$A_{SC,i} \mathbin{\dot{\cap}} out_{\uparrow}(t).$$

By Condition Switch, the buffer is empty

$$wb_{\uparrow}^{t+1}(i) = \varepsilon$$

and the claim trivially follows

$$\neg hit(A_{PR,j}, wb_{\uparrow}^{t+1}(i)).$$

□

Lemma 177. *If step t is a processor step in strong memory mode executing a memory write*

$$s(t) \in \Sigma_{P,i} \wedge SC_{\uparrow}(t) \wedge mwrite_{\uparrow}(t),$$

and the step is not preparing bypassing writes except to the normal processor registers

$$Dom(PW_{\uparrow}(t).bpa) \subseteq A_{NPR,i},$$

then the step is buffering a write

$$BW_{\uparrow}(t) \neq \emptyset.$$

Proof. By Lemma 119, the write-set is not a subset of the normal processor registers

$$WS_{\uparrow}(t) \not\subseteq A_{NPR,u_{\uparrow}(t)},$$

and we obtain that the union of the domains of the prepared writes is not a subset of the processor registers

$$WS_{\uparrow}(t) \not\subseteq A_{NPR,u_{\uparrow}(t)}$$

$$\iff Dom(W_\uparrow(t)) \not\subseteq A_{NPR,u_\uparrow(t)}$$
$$\iff Dom(PW_\uparrow(t).bpa \cup PW_\uparrow(t).wba) \not\subseteq A_{NPR,u_\uparrow(t)}$$
$$\iff Dom(PW_\uparrow(t).bpa) \cup Dom(PW_\uparrow(t).wba) \not\subseteq A_{NPR,u_\uparrow(t)}.$$

By hypothesis the domain of the bypassing writes is a subset of the normal processor registers, but as we have shown the union of the prepared writes is not. Thus the prepared buffered writes are not a subset of the normal processor registers

$$Dom(PW_\uparrow(t).wba) \not\subseteq A_{NPR,u_\uparrow(t)}.$$

The buffered write can thus not be the empty write

$$Dom(PW_\uparrow(t).wba) \neq \emptyset,$$

and the claim follows

$$BW_\uparrow(t) = PW_\uparrow(t).wba \neq \emptyset.$$

□

We can also show that if the write buffer is non-empty, shared writes are always buffered writes.

Lemma 178. *Let the schedule be valid until step t, which is a processor step*

$$\Gamma^t_\uparrow(s) \wedge s(t) \in \Sigma_{P,i}.$$

If step t is a shared memory write in sequentially consistent mode and the buffer is non-empty

$$SC_\uparrow(t) \wedge Sh_\uparrow(t) \wedge mwrite_\uparrow(t) \wedge wb^t_\uparrow(i) \neq \varepsilon,$$

it must be a completely buffered write

$$BW_\uparrow(t) \neq \emptyset \wedge Dom(PW_\uparrow(t).bpa) \subseteq A_{NPR,i}.$$

Proof. The unit making the step is unit i

$$u_\uparrow(t) = i.$$

By Condition MessagePassing the step does not issue a bypassing write except to the normal processor registers

$$Dom(PW_\uparrow(t).bpa) \subseteq A_{NPR,u_\uparrow(t)},$$

and the claim follows with Lemma 177. □

We show that if during a processor step two configurations agree on the core and fetched registers, the drain condition is monotone under buffer subsumption.

Lemma 179.

$$c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge bufS_M(x,c,c') \wedge \Delta_M(c,x) \rightarrow \Delta_N(c',x).$$

Proof. We distinguish between the object making step c,x

$x \in \Sigma_{P,i}$**:** Since the buffers of step c,x subsume those of step c',x, we obtain that the write buffer of configuration c' is a suffix of that in configuration c

$$c.wb(i) = p \circ c'.wb(i).$$

The claim follows with Lemmas 57 and 143

$$\begin{aligned}
\Delta_M(c,x) &\iff \Delta_P(core_M(c,x), fetch_M(c,x), c.wb(i), x) \\
&\iff \Delta_P(core_N(c',x), fetch_N(c',x), c.wb(i), x) && \text{L } 143 \times 2 \\
&\iff \Delta_P(core_N(c',x), fetch_N(c',x), p \circ c'.wb(i), x) \\
&\implies \Delta_P(core_N(c',x), fetch_N(c',x), c'.wb(i), x) && \text{L } 57 \\
&\iff \Delta_N(c',x).
\end{aligned}$$

$x \in \Sigma_{WB,i}$**:** The heads of the write buffers are the same

$$hd(c.wb(i)) = hd(c'.wb(i)) \neq \bot$$

and thus the write buffer of configuration c' is not empty

$$c'.wb(i) \neq \varepsilon$$

and the claim follows

$$\begin{aligned}
\Delta_N(c',x) &\iff \Delta_{WB}(c'.wb(i)) \\
&\iff c'.wb(i) \neq \varepsilon.
\end{aligned}$$

□

When two configurations have the same buffers and agree on core and fetched registers, then validity of one implies validity of the other.

Lemma 180.

$$c.wb =_{u_M(c,x)} c'.wb \wedge c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge \Gamma_M(c,x) \to \Gamma_N(c',x).$$

Proof. By definition the local instance and write buffer guard conditions are satisfied in step c,x

$$I_M(c,x) \wedge \Delta_M(c,x).$$

By Lemma 143 with $X := I$ the instance guard condition can be moved to step c',x

$$I_N(c',x) = I_M(c,x) = 1.$$

By Lemma 90 the buffers of step c,x subsume those of step c',x

$$bufS_M(x,c,c'),$$

and with Lemma 179 we obtain that the drain condition in step c',x

$$\Delta_N(c',x).$$

The claim follows

$$\Gamma_N(c',x).$$

□

Lemma 181. *When two configurations agree on the inputs of a step*

$$c.m =_{in_M(c,x)} c'.m$$

and the buffers of step c,x subsume the buffers of step c',x

$$bufS_M(x,c,c'),$$

and step c,x is a processor step using low-level machine semantics

$$x \in \Sigma_{P,i} \wedge LL_M(c,x),$$

then the two steps agree on the forwarding memory system for the forwarded addresses

$$fms_M(c,x) =_{R_M(c,x).wba} fms_N(c',x)$$

Proof. Since the buffers of step c,x subsume those of step c',x and the step is made in low-level machine semantics, we obtain that the write buffer of configuration c' is a suffix of that in configuration c

$$c.wb(i) = w \circ c'.wb(i),$$

and that the missing elements are not relevant for forwarding

$$\neg hit(fin_M(c,x),w).$$

In low-level machine semantics, read addresses that use the buffer are by definition a subset of the forwarded inputs

$$R_M(c,x).wba \subseteq in_M(c,x) = fin_M(c,x).$$

We conclude by contraposition of Lemma 48 that there is also no hit with those addresses

$$\neg hit(R_M(c,x).wba,w).$$

The claim is now Lemma 154

$$fms_M(c,x)\big|_{R_M(c,x).wba} = fms_N(c',x)\big|_{R_M(c,x).wba}.$$

□

If two configurations agree on inputs and the buffers in one step subsume those in the other step, the steps agree on the memory view.

Lemma 182. *When two configurations agree on the inputs of a step*

$$c.m =_{in_M(c,x)} c'.m$$

and the buffers of step c,x subsume the buffers of step c',x

$$bufS_M(x,c,c'),$$

and at least one of the following is true

1. *the configurations agree on the machine semantics*

$$LL_M(c,x) = LL_N(c',x),$$

2. *or step c,x uses low-level semantics, step c',x high-level semantics, and the write buffer contains no information that should be forwarded*

$$LL_M(c,x) \wedge \neg LL_N(c',x) \wedge \neg hit(R_N(c',x).wba, c'.wb(u_N(c',x))),$$

then they agree on the memory view

$$v_M(c,x) = v_N(c',x).$$

Proof. We distinguish between the object making step c,x

$x \in \Sigma_{P,i}$**:** We obtain with Lemma 143 that the steps have the same core configuration and fetch results.

$$\begin{aligned} core_M(c,x) &= core_N(c',x), && \text{L 143} \\ fetch_M(c,x) &= fetch_N(c',x). && \text{L 143.} \end{aligned}$$

By assumption, the memories agree on the inputs, and thus in particular on all read addresses

$$c.m =_{R_M(c,x)} c'.m$$

and in particular on those addresses that are read using a bypassing read

$$c.m =_{R_M(c,x).bpa} c'.m.$$

By assumption, the steps use the same machine semantics unless step c,x uses low-level semantics and step c',x does not and the buffered writes are not relevant for forwarding. We distinguish between three cases: both steps use low-level semantics, both steps do not use low-level semantics, or step c,x does and step c',x does not.

$LL_M(c,x) = LL_N(c',x) = 1$**:** The claim follows with Lemmas 181 and 62

$$\begin{aligned} v_M(c,x) &= c.m\big|_{R_M(c,x).bpa} \cup fms_M(c,x)\big|_{R_M(c,x).wba} \\ &= c'.m\big|_{R_M(c,x).bpa} \cup fms_N(c',x)\big|_{R_M(c,x).wba} && \text{L 181} \\ &= c'.m\big|_{R_N(c',x).bpa} \cup fms_N(c',x)\big|_{R_N(c',x).wba} && \text{L 62} \\ &= v_N(c',x). \end{aligned}$$

$LL_M(c,x) = LL_N(c',x) = 0$**:** The claim follows with Lemma 62

$$\begin{aligned} v_M(c,x) &= c.m\big|_{R_M(c,x)} \\ &= c'.m\big|_{R_M(c,x)} \\ &= c'.m\big|_{R_N(c',x)} && \text{L 62} \\ &= v_N(c',x). \end{aligned}$$

$LL_M(c,x) \wedge \neg LL_N(c',x) \wedge \neg hit(R_N(c',x).wba, c'.wb(u_N(c',x)))$**:**
With Lemma 53 we obtain that forwarding and memory are the same in c'

$$fms_N(c',x) = fms_{u_N(c',x)}(c')$$

$$
\begin{aligned}
&= c'.m \circledcirc c'.wb(u_N(c',x)) \\
&=_{R_N(c',x).wba} c'.m. && \text{L 53}
\end{aligned}
$$

The claim follows with Lemmas 181 and 62

$$
\begin{aligned}
v_M(c,x) &= c.m\big|_{R_M(c,x).bpa} \cup fms_M(c,x)\big|_{R_M(c,x).wba} \\
&= c'.m\big|_{R_M(c,x).bpa} \cup fms_N(c',x)\big|_{R_M(c,x).wba} && \text{L 181} \\
&= c'.m\big|_{R_N(c',x).bpa} \cup fms_N(c',x)\big|_{R_N(c',x).wba} && \text{L 62} \\
&= c'.m\big|_{R_N(c',x).bpa} \cup c'.m\big|_{R_N(c',x).wba} \\
&= c'.m\big|_{R_N(c',x)} \\
&= v_N(c',x).
\end{aligned}
$$

$x \in \Sigma_{WB,i}$**:** The claim holds since write buffer steps do not read

$$R_M(c,x) = R_N(c',x) = \emptyset$$

and thus the steps do not have a memory view

$$v_M(c,x) = \emptyset = v_N(c',x).$$

□

Thus if two configurations agree on inputs and on the portion of the write buffers that is used by forwarding, the configurations agree on the memory view, and if the configuration with the larger buffer satisfies the drain condition, so does the other.

Lemma 183. *When two configurations agree on the inputs of a step*

$$c.m =_{in_M(c,x)} c'.m$$

and the buffers of step c,x subsume the buffers of step c',x

$$bufS_M(x,c,c'),$$

then they agree on the memory view, and if the configuration with the longer buffer satisfies the drain condition, so does the other

$$c.m =_{C_M(c,x)\cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge (\Delta_M(c,x) \rightarrow \Delta_N(c',x)).$$

Proof. Since local and fetched registers are a subset of the inputs

$$C_M(c,x) \cup F_M(c,x) \subseteq in_M(c,x),$$

the memories agree on the local and fetched registers

$$c.m =_{C_M(c,x)\cup F_M(c,x)} c'.m,$$

which is the first claim.

By Lemma 147, the machines have the same machine semantics

$$LL_M(c,x) = LL_M(c',x)$$

and the second claim is Lemma 182.

The third claim is Lemma 179. □

We consider the special case where the suffix of the buffer on which the configurations agree is actually the complete buffer. In this case we obtain strong agreement.

Lemma 184. *When two configurations agree on the inputs of a processor step*

$$x \in \Sigma_{P,i} \wedge c.m =_{in_M(c,x)} c'.m$$

and on the buffers

$$c.wb =_i c'.wb,$$

then they agree on the memory view, and if the configuration with the longer buffer satisfies the drain condition, so does the other

$$c =^x_{M,N} c'$$

Proof. By Lemma 90, the buffers of each configuration subsume the buffers of the other one

$$bufS_M(x,c,c') \wedge bufS_N(x,c',c).$$

We use Lemma 183 and obtain that the configurations agree on the core and fetched registers, on the memory view, and that the drain condition is satisfied by c' if it is satisfied by c

$$c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge (\Delta_M(c,x) \rightarrow \Delta_N(c',x)).$$

With Lemma 143 we obtain that step c',x has the same inputs as step c,x

$$in_N(c',x) = in_M(c,x)$$

and thus the configurations also agree on the inputs of step c',x

$$c.m =_{in_N(c',x)} c'.m.$$

We use again Lemma 183 (with reversed arguments) and obtain that the drain condition is satisfied by step c,x if it is satisfied by step c',x

$$\Delta_N(c',x) \rightarrow \Delta_M(c,x),$$

from which we immediately obtain that the configurations agree on the drain condition

$$\Delta_N(c',x) = \Delta_M(c,x).$$

The claim follows by definition

$$\begin{aligned} & c =^x_{M,N} c' \\ \equiv\ & c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge \Delta_M(c,x) = \Delta_N(c',x). \\ \equiv\ & 1. \end{aligned}$$

□

Lemma 185. *When two configurations agree on the inputs of a processor step*

$$x \in \Sigma_{P,i} \wedge c.m =_{in_M(c,x)} c'.m$$

and one of the configurations satisfies the local drain condition

$$\Delta_M(c,x)$$

and the buffers of that configuration subsume the buffers of the other configuration

$$bufS_M(x,c,c'),$$

then they agree on the memory view, and if the configuration with the longer buffer satisfies the drain condition, so does the other

$$c =^{x}_{M,N} c'$$

Proof. We apply Lemma 183 and obtain that the configurations agree on the core and fetched registers, on the memory view, and that the drain condition is satisfied by c' if it is satisfied by c

$$c.m =_{C_M(c,x)\cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge (\Delta_M(c,x) \rightarrow \Delta_N(c',x)).$$

Since the drain condition of step c,x is satisfied, so is the drain condition of step c',x

$$\Delta_N(c',x)$$

and the steps agree on the drain condition

$$\Delta_M(c,x) = \Delta_N(c',x).$$

The claim follows by definition of strong agreement. □

Lemma 186. *When two configurations agree on the inputs of a write buffer step*

$$x \in \Sigma_{WB,i} \wedge c.m =_{in_M(c,x)} c'.m$$

the buffers of that configuration subsume the buffers of the other configuration

$$bufS_M(x,c,c'),$$

then they strongly agree

$$c =^{x}_{M,N} c'$$

Proof. Since the core registers are inputs

$$C_M(c,x) \subseteq in_M(c,x),$$

the configurations agree on the core registers

$$c.m =_{C_M(c,x)} c'.m.$$

The claim follows with the assumption

$$\begin{aligned} c =^{x}_{M,N} c' &\equiv c.m =_{C_M(c,x)} c'.m \wedge hd(c.wb(i)) = hd(c'.wb(i)) \\ &\equiv 1. \end{aligned}$$

□

Lemma 187. *When step c,x is valid*

$$\Gamma_M(c,x)$$

and c agrees with a configuration c' on the inputs during step c,x

$$c.m =_{in_M(c,x)} c'.m$$

and write buffers of step c,x subsume those of step c',x

$$bufS_M(x,c,c'),$$

then the configurations strongly agree when stepped with x

$$c =^x_{M,N} c'.$$

Proof. By case distinction on x.

$x \in \Sigma_{P,i}$**:** By assumption the guard condition holds during the step

$$\Gamma_M(c,x),$$

and thus the local drain condition is satisfied

$$\Delta_M(c,x).$$

The claim is now Lemma 185.

$x \in \Sigma_{WB,i}$**:** The claim is just Lemma 186.

□

We also prove a simpler variant where the write buffers are equal also for write buffer steps, and the machines are the same.

Lemma 188. *When two configurations agree on the inputs*

$$c.m =_{in_M(c,x)} c'.m$$

and write buffers of the unit making the step are equal

$$c.wb =_{u_M(c,x)} c'.wb,$$

then the configurations strongly agree when stepped with x

$$c =^x_M c'.$$

Proof. By case distinction on x.

$x \in \Sigma_{P,i}$**:** The claim is just Lemma 184.

$x \in \Sigma_{WB,i}$**:** By Lemma 90 the buffers of step c,x subsume those of step c',x

$$bufS_M(x,c,c').$$

The claim is now Lemma 186.

□

We now consider a situation where we are running two schedules in parallel, where the second is obtained by a sequence of reorderings O from the first. We reach the configurations at k, k', respectively, which are stepped with the same oracle input. The reordering had the effect of moving step $t < k$, and thus the visible outputs of step t and the write buffers of the unit making step t have been changed, but the rest of the configuration happens to be the same. We now progressively show conditions under which steps k and k' can be executed in parallel, maintaining the invariant that visible outputs of step t and the write buffer of the unit making step t are the only things that are affected by the reordering O. When the invariant holds we say that *the configurations at k and k' are nearly the same* (after the reordering O) and write $\mathcal{I}^t_M[s](O,k,k')$

$$\mathcal{I}^t_M[s](O,k,k') \equiv m^k_M =_{\mathcal{A}\setminus vout_M(t,k)} m_M O^{k'} \wedge \forall i \neq u_M(t).wb^k_M =_i wb_M O^{k'}.$$

We first show that if the configurations strongly agree, the part of the invariant of the memory is maintained.

Lemma 189. *If the configurations at k and k' agree on everything except for the visible outputs of step t, and step k and configuration k' strongly agree when stepped with $s(k)$*

$$m^k_M =_{\mathcal{A}\setminus vout_M(t,k)} m_M O^k \wedge c^k_M =^{s(k)}_M c_M O^{k'},$$

and both configurations are actually stepped with $s(k)$

$$sO(k') = s(k),$$

the configurations after the step also agree on everything except for visible outputs

$$m^{k+1}_M =_{\mathcal{A}\setminus vout_M(t,k+1)} m_M O^{k'+1}$$

Proof. By Lemma 111 we can take the outputs of k out of the visible write set

$$vws_M(t,k+1) = vws_M(t,k) \setminus out_M(k).$$

We obtain that the complement of the visible outputs at $k+1$ is the complement of the visible writes at k, the outputs of k, but not the device closure of the write-set of step t

$$\begin{aligned}\mathcal{A} \setminus vout_M(t,k+1) &= \mathcal{A} \setminus (vws_M(t,k+1) \cup dc(WS_M(t))) \\ &= \mathcal{A} \setminus (vws_M(t,k) \setminus out_M(k+1)) \setminus dc(WS_M(t)) \\ &= ((\mathcal{A} \setminus vws_M(t,k)) \cup out_M(k+1)) \setminus dc(WS_M(t)).\end{aligned}$$

The last step is justified because $out_M(k+1)$ is a subset of $\mathcal{A}$.

Note that the configuration at $k'+1$ in the reordered schedule is obtained by stepping the configuration at k' with the oracle inputs of k in the original schedule

$$\begin{aligned}c_M O^{k'+1} &= c_M O^{k'} \bullet_M sO(k') \\ &= c_M O^{k'} \bullet_M s(k).\end{aligned}$$

We wish to apply Lemma 149 with the following parameters

$$\begin{aligned} A &:= \mathcal{A} \setminus vws_M(t,k), \\ B &:= WS_M(t), \\ c &:= c_M^k, \\ c' &:= c_M O^{k'}, \\ x &:= s(k). \end{aligned}$$

It suffices to show the following three subclaims.

$m_M^k =_{\mathcal{A} \setminus vws_M(t,k) \setminus dc(WS_M(t))} m_M O^{k'}$**:** The set of addresses is just the complement of the visible outputs

$$\begin{aligned} \mathcal{A} \setminus vws_M(t,k) \setminus dc(WS_M(t)) &= \mathcal{A} \setminus (vws_M(t,k) \cup dc(WS_M(t))) \\ &= \mathcal{A} \setminus vout_M(t,k). \end{aligned}$$

The claim is now an assumption

$$m_M^k =_{\mathcal{A} \setminus vout_M(t,k)} m_M O^{k'}.$$

$c_M^k =_M^{s(k)} c_M O^{k'}$**:** This is an assumption.

$(\mathcal{A} \setminus (\mathcal{A} \setminus vws_M(t,k))) \subseteq idc(WS_M(t))$**:** We cancel out the double-complement and the claim follows with Lemma 113

$$vws_M(t,k) \subseteq vout_M(t,k+1) \subseteq out_M(t) = idc(WS_M(t)).$$

□

The invariant can always be established when we are delaying a step, i.e., the configurations directly after step t and before step t are nearly the same.

Lemma 190. *The configurations at $t+1$ and t are nearly the same when step t is delayed to $k \geq t$*

$$\mathcal{I}_M^t([t \to k], t+1, t).$$

Proof. The configurations at t are the same

$$c_M^t = c_M[t \to k]^t$$

and by Lemma 138 the configurations at $t+1$ differ exactly by the outputs

$$m_M^{t+1} =_{\mathcal{A} \setminus out_M(t)} m_M^t = m_M[t \to k]^t.$$

By Lemma 112, those outputs are exactly the visible outputs

$$out_M(t) = vout_M(t, t+1),$$

and thus the first claim follows

$$m_M^t =_{\mathcal{A} \setminus vout_M(t,t+1)} m_M[t \to k]^t.$$

The second claim follows by Lemma 96

$$\forall i \neq u_M(t). wb_M^t =_i wb_M^{t+1}(i).$$

□

Lemma 191. *If step t does not visibly modify a memory region A*

$$vout_M(t) \not\downharpoonleft A,$$

the configurations agree on A if configurations are nearly the same

$$\mathcal{I}^t_M(O,k,k') \to m^k_M =_A m_M O^{k'}.$$

Proof. Since the configurations at k and k' are nearly the same, they agree on everything except for the visible outputs

$$m^k_M =_{\mathcal{A}\setminus vout_M(t,k)} m_M O^{k'}$$

and thus on A

$$m^k_M =_A m_M O^{k'},$$

which is the claim. □

Lemma 192. *If step t does not modify a memory region A*

$$out_M(t) \not\downharpoonleft A,$$

the configurations agree on A if configurations are nearly the same

$$\mathcal{I}^t_M(O,k,k') \to m^k_M =_A m_M O^{k'}.$$

Proof. Since by Lemma 113 the outputs subsume the visible outputs

$$vout_M(t,k) \subseteq out_M(k)$$

we conclude that there is no intersection between those and A either

$$vout_M(t,k) \not\downharpoonleft A.$$

The claim is Lemma 191. □

Lemma 193. *If step t is object-concurrent with step k*

$$ocon_M(t,k),$$

the configurations agree on the core registers if configurations are nearly the same

$$\mathcal{I}^t_M(O,k,k') \to m^k_M =_{C_M(k)} m_M O^{k'}.$$

Proof. By definition of *ocon* step t does not modify the core registers of step k

$$out_M(t) \not\downharpoonleft C_M(k).$$

The claim is now Lemma 192. □

Lemma 194. *If step t is unit concurrent with step k*

$$ucon_M(t,k),$$

the configurations agree on the core registers if configurations are nearly the same

$$\mathcal{I}^t_M(O,k,k') \to m^k_M =_{C_M(k)} m_M O^{k'}.$$

Proof. By Lemma 105 we obtain that the step is also object-concurrent

$$ocon_M(t,k)$$

and the claim is Lemma 193. □

Lemma 195. *If there is no visible write-read race*

$$\neg VR_M(t,k),$$

the memories agree on the inputs if the configurations are nearly the same

$$\mathcal{I}^t_M(O,k,k') \to m^k_M =_{in_M(k)} m_M O^{k'}.$$

Proof. There is no visible write-read race, and by definition there is no intersection between the visible outputs of step t and the inputs of step k

$$vout_M(t,k) \not\cap in_M(k).$$

The claim is Lemma 191. □

Lemma 196. *If step k is not made by the same unit as step t and there is no visible write-read race*

$$diffu(t,k) \wedge \neg VR_M(t,k),$$

the steps strongly agree if the configurations are nearly the same

$$\mathcal{I}^t_M(O,k,k') \to c^k_M =^{s(k)}_M c_M O^{k'}.$$

Proof. By Lemma 195 the configurations agree on the inputs

$$m^k_M =_{in_M(k)} m_M O^{k'}.$$

Since the unit making step k is not the unit making step t

$$u_M(t) \neq u_M(k),$$

buffers of the unit making the step are the same

$$wb^k_M =_{u_M(k)} wb_M O^{k'}.$$

The claim is now Lemma 188

$$c^k_M =^{s(k)}_M c_M O^k.$$

□

Lemma 197. *If step k can be executed at k' in the reordered schedule*

$$c^k_M =^{s(k)}_M c_M O^{k'},$$

and the configurations are stepped with the same input

$$sO(k') = s(k),$$

the invariant is maintained by steps k resp. k'

$$\mathcal{I}^t_M(O,k,k') \to \mathcal{I}^t_M(O,k+1,k'+1).$$

Proof. The first portion of the invariant is maintained by Lemma 189

$$m_M^{k+1} =_{\mathcal{A}\setminus vout_M(t,k+1)} m_M O^{k'+1}.$$

The second portion of the invariant is maintained by Lemma 150 and the invariant

$$\begin{aligned} wb_M^{k+1}(i) &= Op_{iM}(k)(wb_M^k(i), BW_M(k)) \\ &= Op_{iM}O(k')(wb_M^k(i), BW_M O(k')) && \text{L 150} \\ &= Op_{iM}O(k')(wb_M O^{k'}(i), BW_M O(k')) && \text{Inv} \\ &= wb_M O^{k'+1}(i). \end{aligned}$$

The claim follows. □

Lemma 198. *If step k is not made by the same unit as step t and there is no visible write-read race*

$$diffu(t,k) \wedge \neg VR_M(t,k),$$

and the configurations are stepped with the same input

$$sO(k') = s(k),$$

the invariant is maintained by steps k resp. k'

$$\mathcal{I}_M^t(O,k,k') \rightarrow \mathcal{I}_M^t(O,k+1,k'+1).$$

Proof. By Lemma 196 the steps agree

$$c_M^k =_M^{s(k)} c_M O^{k'},$$

and the claim is Lemma 197. □

We can also add the step back in if we swap the schedules and indices, as shown in Fig. 4.3.

Lemma 199. *If the configuration at k' is stepped with the same input as t and the steps strongly agree*

$$sO(k') = s(t) \wedge c_M^t =_M^{s(t)} c_M O^{k'},$$

then after step k' we can switch the direction of the invariant undoing the reordering and thus considering sO as the original and $s = sOO^{-1}$ as the reordered schedule

$$\mathcal{I}_M^t(O,k,k') \rightarrow \mathcal{I}_M^{k'} O(O^{-1},k'+1,k).$$

Proof. By assumption, the configurations agree on the content of everything except the visible outputs of step t

$$m_M^k =_{\mathcal{A}\setminus vout_M(t,k)} m_M O^{k'}.$$

By Lemma 138 the memories at $k'+1$ and k' agree on everything except for outputs of step k'

$$m_M O^{k'+1} =_{\mathcal{A}\setminus out_M O(k')} m_M O^{k'}.$$

By Lemma 150, step k' after the reordering has the same outputs as step t

$$out_M(t) = out_M O(k').$$

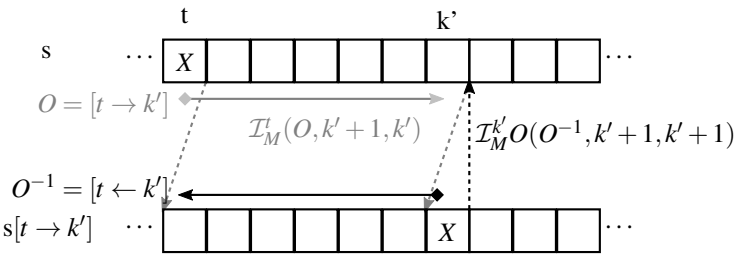


Figure 4.3: In dark gray: We move step t to position k', then run the schedules in parallel starting from $t+1$ in s and t in $s[t \to k']$ until we reach $k = k'+1$ and k', i.e., right before the new position of step t. In other words, step t has been executed in schedule s but not in schedule $s[t \to k']$, and thus the position in schedule s is always ahead by one step. Once we reach k', we execute the step but only in $s[t \to k']$. This changes only a) outputs of step k', which subsume visible outputs of step t in s and b) the write buffer of the unit making step k', which also made step t in s. As shown in black, we can now obtain again the invariant, but using $s[t \to k']$ as the original schedule and $s = s[t \to k'][t \leftarrow k']$ as the reordered schedule, and running the schedules in parallel starting from $k'+1$ *in both schedules*. This is due to the fact that now the same steps have been executed by both schedules, albeit in different order. We can now continue to run the schedules in parallel until we reach a step l' that has a visible write-read race or is made by the same unit as step k' in $s[t \to k']$.

By Lemma 113 the visible outputs of step t at k are a subset of the outputs of step t

$$vout_M(t,k) \subseteq out_M(t),$$

and by Lemma 112 the visible outputs of step k' at $k'+1$ are the outputs of step k'

$$out_M O(k') = vout_M O(k', k'+1)$$

and we conclude that the visible outputs of step t at k are a subset of the visible outputs of step k' at $k'+1$ in the reordered schedule

$$vout_M(t,k) \subseteq out_M(t) = out_M O(k') = vout_M O(k', k'+1).$$

By Lemma 141, the configurations at k' and $k'+1$ after the reordering therefore agree on everything except the visible outputs

$$m_M O^{k'+1} =_{\mathcal{A} \setminus vout_M O(k', k'+1)} m_M O^{k'}$$

and the configurations at k and k' agree on the content of everything except the visible outputs of k' at $k'+1$

$$m^k_M =_{\mathcal{A} \setminus vout_M O(k', k'+1)} m_M O^{k'},$$

and thus the configurations at k in the original schedule and $k'+1$ in the reordered schedule agree on everything except the visible outputs

$$m_M O^{k'+1} =_{\mathcal{A} \setminus vout_M O(k', k'+1)} m^k_M.$$

Since the original schedule is by Lemma 86 also obtained by applying first the operations O and then the operations O^{-1}

$$s = sOO^{-1},$$

the memory at k in those schedules is the same

$$m_M^k = m_M OO^{-1}{}^k$$

and the first part of the claim follows

$$m_M O^{k'+1} =_{\mathcal{A}\setminus vout_M O(k',k'+1)} m_M OO^{-1}{}^k.$$

The second part similarly follows: by the fact that step k' is made by the same unit as step t in the original schedule, we obtain with Lemma 96 for other units

$$i \neq u_M(t) = u_M O(k')$$

that step k' does not affect their write buffers

$$wb_M O^{k'+1} =_i wb_M O^{k'} =_i wb_M^k = wb_M OO^{-1}{}^k.$$

□

If the configurations when moving further than l are nearly the same at $l+1$ resp. l, so are the configurations when moving exactly to l.

Lemma 200. *The configurations at k and at l when moving t to $k' \geq l$ are nearly the same iff the ones when moving t to l are*

$$\mathcal{I}_M^t([t \to k'], k, l) \equiv \mathcal{I}_M^t([t \to l], k, l).$$

Proof. Moving t to k' is the same as moving t to l and then l to k'

$$s[t \to k'] = s[t \to l][l \to k']$$

and thus the steps before l are the same in the two reorderings

$$s[t \to k'][0 : l-1] = s[t \to l][l \to k'][0 : l-1] = s[t \to l][0 : l-1].$$

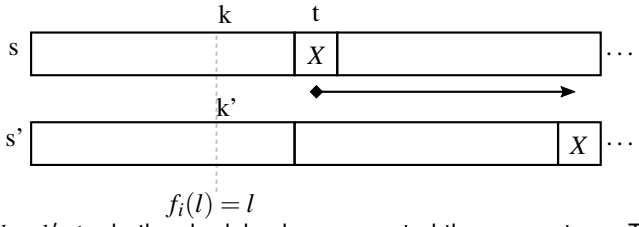
We conclude that the configurations at l are also the same

$$c_M[t \to k']^l = c_M[t \to l]^l.$$

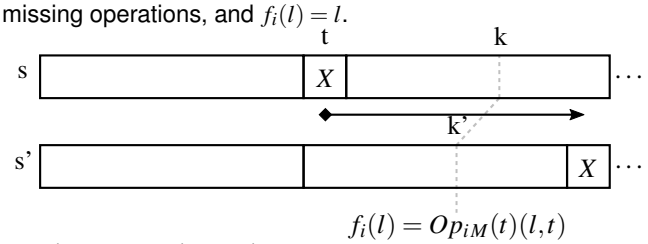
The claim follows

$$\begin{aligned}
\mathcal{I}_M^t([t \to k'], k, l) &\equiv m_M^k =_{\mathcal{A}\setminus vout_M(t,k)} m_M[t \to k']^l \wedge \forall i \neq u_M(t).wb_M^k =_i wb_M[t \to k']^l \\
&\equiv m_M^k =_{\mathcal{A}\setminus vout_M(t,k)} m_M[t \to l]^l \wedge \forall i \neq u_M(t).wb_M^k =_i wb_M[t \to l]^l \\
&\equiv \mathcal{I}_M^t([t \to l], k, l).
\end{aligned}$$

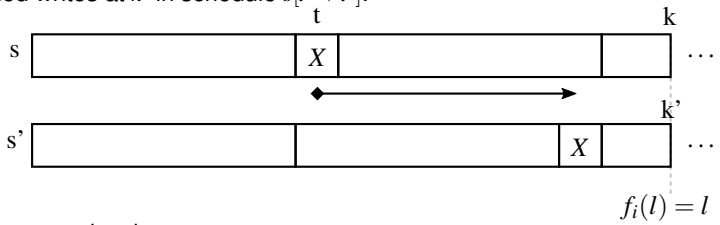
□



(a) For $k = k' \leq t$, both schedules have executed the same steps. Thus there are no missing operations, and $f_i(l) = l$.



(b) For $k = k' + 1$ and $k, k' \in (t : t']$, we have executed all steps except for step t of the original schedule in $s[t \to t']$. To synchronize the sequences of issued writes we have to apply the missing operation $f_i(l) = Op_{iM}(t)(l,t)$ to the sequence of issued writes at k' in schedule $s[t \to t']$.



(c) For $k = k' > t'$, both schedules have again been synchronized, as the missing step t has now been executed at t'. One can thus immediately synchronize the sequences of issued writes of each unit i, and can take f_i as the identity $f_i(l) = l$.

Figure 4.4: Examples of functions f_i for different k and k' such that we can obtain the sequence of issued writes at k by undoing the reordering $[t \to t']$, and then applying f_i, to the sequence of issued writes at k' in the reordered schedule

$$issue_M^k(i) = f_i(mv[t \to t']^{-1}(issue_M[t \to t']^{k'}(i))).$$

We sometimes need a similar invariant, namely to compare how sequences of issued writes are reordered. In this case the sequence of issued writes of the unit making step t is still missing the operation of step t, but unlike the write buffers, the other sequences are not simply the same. This is due to the fact that the write buffers do not care when the writes were buffered, only what writes were buffered; thus the order does not matter, only the result. In case of the sequence of issued writes, however, the order does matter. We use the function *mv* that tracks movement of steps from Section 2.7 to track how the order of steps changes.

We define a very general invariant $\mathcal{J}$, which uses a function f_i for each sequence of issued writes, an original schedule s and a reordering O. We will give the formal definition first, then explain the intuition behind that function in more detail. We say

that *the sequences of issued writes at k and k′ are nearly the same* when the sequences of issued writes in the original schedule can be obtained by undoing the reordering (using mvO^{-1}) and applying function f_i to the sequences of issued writes of unit i in the reordered schedule

$$\mathcal{J}_M^{f_i}[s](O,k,k') \equiv \forall i.issue_M^k(i) = f_i(mvO^{-1}(issue_M O^{k'}(i))).$$

We will use this invariant only in one particular setting, namely when O moves some step t to some position l. In this case, we use the function f_i to remember all of the operations that have been applied to the sequence of issued writes of unit i in schedule s until k but not in schedule $s[t \to t']$ until k'. There are three phases (shown in Fig. 4.4):

1. Steps k and k' are equal and before t. The sequences of issued writes are nearly the same because there is no difference between s and sO. Also, everything that has happened to sequences of issued writes in schedule s until k has happened in schedule sO until k', and therefore we do not have to remember any operations

$$f_i(l) = l.$$

2. Steps $k = k' + 1$ and k' are in the reordered portion $(t : t']$. Step t in the original schedule has been executed at k in the original schedule, but has not been executed at k' in the reordered schedule. All other steps have been executed in both schedules, but maybe at a different point in time. Therefore we have to remember the operation performed by step t

$$f_i(l) = Op_{iM}(t)(l,t).$$

3. Steps k and k' are equal and behind the reordered portion. All steps in the original schedule that have been executed until k have also been executed until k' in the reordered schedule: step t has been executed at position t' in $s[t \to t']$, and the reordered steps $(t : t']$ have been executed in the interval $[t : t')$. Thus once more we do not have to remember any operations

$$f_i(l) = l.$$

The first of these phases is easy, because nothing has happened. For the other two phases, we show three lemmas. The first lemma proves that the invariant can be established at the beginning of Phase 2, i.e., for $k = t+1$ and $k' = t$, and $f_i(l) = Op_{iM}(t)(l,t)$. The second lemma shows that the invariant is actually invariant under certain conditions, i.e., if those conditions hold and the invariant holds for some O and f_i at k and k', it also holds at $k+1$ and $k'+1$. Since we have established the invariant for $t+1$ and t using the first lemma, we can now move the invariant to any position in Phase 2, including the final point where k is $t'+1$ and k' is t'. At this point step t is added back into the schedule, and the third lemma shows that we can now enter Phase 3 by continuing the invariant with k (not increased), $k'+1$, and $f_i(l) = l$. After this we can again with the second lemma move the invariant to any position in Phase 3.

When f_i is the function applied in step t and $O = [t \to k]$ moves t to position k, the sequences of issued writes at $t+1$ and t are nearly the same

Lemma 201.

$$f_i(l) = Op_{iM}(t)(l,t) \to \mathcal{J}_M^{f_i}([t \to k],t+1,t).$$

Proof. Because all elements of the write buffers at t have entered before t, we obtain with Lemma 77 that they are not affected by the reordering

$$issue_M^t(i) = issue_M[t \to k]^t(i) = mv[t \leftarrow k](issue_M[t \to k]^t(i)).$$

The proof is now straightforward with Lemma 83

$$\begin{aligned} issue_M^{t+1}(i) &= Op_{iM}(t)(issue_M^t(i),t) \\ &= f_i(issue_M^t(i)) \\ &= f_i(mv[t \leftarrow k](issue_M[t \to k]^t(i))) \\ &= f_i(mv[t \to k]^{-1}(issue_M[t \to k]^t(i))). \qquad \text{L 83} \end{aligned}$$

□

Lemma 202. *When step k is moved to k'*

$$mvO(k) = k',$$

and f_i commutes with the operation in step k when applied to the sequence of issued writes at k' after the reordering is undone

$$f_i(Op_{iM}(k)(mvO^{-1}(issue_M O^{k'}(i)),k)) = Op_{iM}(k)(f_i(mvO^{-1}(issue_M O^{k'}(i))),k),$$

and steps k and k' strongly agree

$$c_M^k =_M^{s(k)} c_M O^{k'},$$

the invariant is maintained when the steps are executed in parallel.

$$\mathcal{J}_M^{f_i}(O,k,k') \to \mathcal{J}_M^{f_i}(O,k+1,k'+1).$$

Proof. With Lemmas 76, 82 and 84 we obtain that k is the original position of step k'

$$\begin{aligned} mvO^{-1}(k') &= mvO^{-1}(mvO(k)) \\ &= mvO^{-1}O(k) && \text{L 76} \\ &= mvO^{-1}(O^{-1})^{-1}(k) && \text{L 82} \\ &= k. && \text{L 84} \end{aligned}$$

By Lemma 75 the steps are stepped with the same oracle input

$$s(k) = sO(mvO(k)) = sO(k').$$

The proof is now straightforward with Lemmas 36 and 150

$$\begin{aligned} issue_M^{k+1}(i) &= Op_{iM}(k)(issue_M^k(i),k) \\ &= Op_{iM}(k)(f_i(mvO^{-1}(issue_M O^{k'}(i))),k) && \text{Inv} \\ &= f_i(Op_{iM}(k)(mvO^{-1}(issue_M O^{k'}(i)),k)) && \text{Comm} \\ &= f_i(Op_{iM}(k)(mvO^{-1}(issue_M O^{k'}(i)),mvO^{-1}(k'))) \\ &= f_i(mvO^{-1}(Op_{iM}(k)(issue_M O^{k'}(i),k'))) && \text{L 36} \\ &= f_i(mvO^{-1}(Op_{iM}O(k')(issue_M O^{k'}(i),k'))) && \text{L 150} \\ &= f_i(mvO^{-1}(issue_M O^{k'+1}(i))). \end{aligned}$$

□

By dropping f_i we can move step t back into the schedule. Unlike the memory region modified by step t in the corresponding lemma for invariant $\mathcal{I}$ (Lemma 199), the sequences of issued writes converge seamlessly after step t is added back into the schedule. This is why there is no need to play tricks with s and O/O^{-1}.

Lemma 203. *If after moving step t to position k the steps strongly agree*

$$c^t_M =^{s(t)}_M c_M[t \to k]^k,$$

and if f_i is the function applied in step t

$$f_i(l) = Op_{iM}(t)(l,t),$$

and the sequences of issued writes at $k+1$ and k are nearly the same

$$\mathcal{J}^{f_i}_M([t \to k], k+1, k),$$

then the sequences of issued writes are also nearly the same after k, and the functions f_i that were missing before is now already applied in the reordered schedule (and we use the identity function $f'_i(l) = l$)

$$\mathcal{J}^{f'_i}_M([t \to k], k+1, k+1).$$

Proof. Note that $[k \leftarrow t]$ is the inverse of $[t \to k]$ (Lemma 83) The proof is straightforward with Lemmas 36, 77 and 150

$$\begin{aligned}
issue^{k+1}_M(i) &= f_i(mv[k \leftarrow t](issue_M[t \to k]^k(i))) && \text{Inv}\\
&= Op_{iM}(t)(mv[k \leftarrow t](issue_M[t \to k]^k(i)), t)\\
&= Op_{iM}(t)(mv[k \leftarrow t](issue_M[t \to k]^k(i)), mv[k \leftarrow t](k)) && \text{L 77}\\
&= mv[k \leftarrow t](Op_{iM}(t)(issue_M[t \to k]^k(i), k)) && \text{L 36}\\
&= mv[k \leftarrow t](Op_{iM}[t \to k](k)(issue_M[t \to k]^k(i), k)) && \text{L 150}\\
&= mv[k \leftarrow t](issue_M[t \to k]^{k+1}(i))\\
&= f'_i(mv[k \leftarrow t](issue_M[t \to k]^{k+1}(i))).
\end{aligned}$$

□

Moving steps behind the reordered portion and behind k has no effect. This allows us to increase the reordered portion in specific instances.

Lemma 204.

$$\mathcal{J}^{f_i}_M([t \to k], k+1, k) \wedge k' > k \to \mathcal{J}^{f_i}_M([t \to k'], k+1, k).$$

Proof. Note that $[t \leftarrow k]$ is the inverse of $[t \to k]$ (Lemma 83). Furthermore, the steps that are in the sequence of issued writes at k in the reordered schedule are all before k

$$\forall t' \in issue_M[t \to k]^k(i).t' < k$$

and are thus by Lemma 77 not affected by the reordering

$$mv[k \leftarrow k'](issue_M[t \to k]^k(i)) = \left[mv[k \leftarrow k'](t') \;\middle|\; t' \in issue_M[t \to k]^k(i) \right]$$

$$
\begin{aligned}
&= \left[t' \;\middle|\; t' \in issue_M[t \to k]^k(i) \right] && \text{L 77}\\
&= issue_M[t \to k]^k(i)\\
&= issue_M[t \to k][k \to k']^k(i)\\
&= issue_M[t \to k']^k(i).
\end{aligned}
$$

The proof is now straightforward

$$
\begin{aligned}
issue_M^{k+1}(i) &= f_i(mv[t \leftarrow k](issue_M[t \to k]^k(i)))\\
&= f_i(mv[t \leftarrow k](mv[k \leftarrow k'](issue_M[t \to k]^k(i))))\\
&= f_i(mv[k \leftarrow k'][t \leftarrow k](issue_M[t \to k]^k(i)))\\
&= f_i(mv[t \leftarrow k'](issue_M[t \to k]^k(i)))\\
&= f_i(mv[t \leftarrow k'](issue_M[t \to k']^k(i))).
\end{aligned}
$$

□

Condition Races seems fishy, since it only considers situations where the modification occurs in the first step. We will show that it is admissible to consider modifications in the second step if the second step is valid, as one can then switch the order of the two steps. This is not as obvious as it sounds, as an RMW in the second step might no longer be modifying if the order is reversed. In that case, however, there must already be a write/read race in the original schedule, causing the steps to be marked as shared anyways.

We first prove a generalization of Condition Races that considers all addresses, rather than only the ones in memory or core. This works because steps that interact through devices must be shared, steps that interact through processor registers of other processors are shared, and steps that interact through the processor registers of the second processor are not unit-concurrent.

Lemma 205. *When s is valid until $t+1$*

$$\Gamma_\uparrow^{t+1}(s)$$

and there are unit-concurrent steps

$$ucon_\uparrow(t, t+1),$$

then both of the following must hold

Write-read races have to be annotated correctly

$$WR_\uparrow(t, t+1) \to Sh_\uparrow(t) \wedge ShR_\uparrow(t+1).$$

Write-write races have to be annotated correctly

$$WW_\uparrow(t, t+1) \to Sh_\uparrow(t) \wedge Sh_\uparrow(t+1).$$

Proof. We show the full proof for the first claim. The proof for the second claim is analogous and we only hint at the differences.

We have a write-read race, and thus an input output intersection somewhere in memory

$$out_\uparrow(t) \cap in_\uparrow(t+1)\,\hat{\cap}\,\mathcal{A}.$$

We distinguish now between the different regions of memory where the race could be: main memory, devices, processor registers of $u_\uparrow(t+1)$, or of any other processor.

$out_\uparrow(t) \cap in_\uparrow(t+1) \dot{\cap} A_{MEM}$**:** We obtain immediately that there is a race in the memory

$$WR_\uparrow^{A_{MEM}}(t, t+1),$$

and the claim is Condition Races

$$Sh_\uparrow(t) \wedge ShR_\uparrow(t+1).$$

In the second claim, one uses the part of Condition Races that talks about write-write races.

$out_\uparrow(t) \cap in_\uparrow(t+1) \dot{\cap} A_{DEV}$**:** Both processors access a device and by applying Lemma 133 we obtain that step t is shared, and by Lemma 132 step $t+1$ is a shared read

$$Sh_\uparrow(t) \wedge ShR_\uparrow(t+1),$$

which is the claim.

For the second claim, one simply uses Lemma 133 twice to prove that both steps are shared.

$out_\uparrow(t) \cap in_\uparrow(t+1) \dot{\cap} A_{PR,u_\uparrow(t+1)}$**:** Clearly step t is modifying processor registers of the unit making step t

$$out_\uparrow(t) \dot{\cap} A_{PR,u_\uparrow(t+1)}.$$

By Lemma 103 we obtain that if step t writes to the processor registers of $u_\uparrow(t+1)$, the steps are not concurrent

$$\neg ucon_\uparrow(t, t+1),$$

which is a contradiction.

The proof for the second claim is literally the same.

$out_\uparrow(t) \cap in_\uparrow(t+1) \dot{\cap} A_{PR,i} \wedge i \neq u_\uparrow(t+1)$**:** Step $t+1$ can not access processor registers of unit i except for the interrupt registers, and thus the intersection is there

$$out_\uparrow(t) \cap in_\uparrow(t+1) \dot{\cap} A_{IPR,i}.$$

Step t is shared because it modifies interrupt registers (Lemma 133), and step $t+1$ is a shared read because it reads from the interrupt registers of another processor (Lemma 132)

$$Sh_\uparrow(t) \wedge ShR_\uparrow(t+1).$$

For the second claim, one simply uses Lemma 133 twice to prove that both steps are shared.

□

We extend this as well as Condition CodeMod by showing that object-concurrent steps already have to be annotated. We unite the two in one lemma.

Lemma 206. *When s is valid until t*

$$\Gamma^{t}_{\uparrow}(s)$$

and there are object-concurrent steps

$$ocon_{\uparrow}(t,t+1),$$

then all of the following must hold

Feasible code-modifications have to be annotated correctly

$$\Phi_{\uparrow}(t+1) \wedge CM_{\uparrow}(t,t+1) \rightarrow Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(t+1).$$

Valid write-read races have to be annotated correctly

$$\Gamma_{\uparrow}(t+1) \wedge WR_{\uparrow}(t,t+1) \rightarrow Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(t+1).$$

Valid write-write races have to be annotated correctly

$$\Gamma_{\uparrow}(t+1) \wedge WW_{\uparrow}(t,t+1) \rightarrow Sh_{\uparrow}(t) \wedge Sh_{\uparrow}(t+1).$$

Proof. The steps are either made by the same unit or not

$$u_{\uparrow}(t) = u_{\uparrow}(t+1) \vee u_{\uparrow}(t) \neq u_{\uparrow}(t+1).$$

Since the steps are object-concurrent, they are made by different objects

$$o_{\uparrow}(t) \neq o_{\uparrow}(t+1)$$

and thus, if they are made by the same unit, one step must be made by the processor and the other by the write buffer. We distinguish between the resulting three cases: 1) steps by different units, 2) steps by same unit, where processor step is made first, and 3) steps by same unit, where write buffer step is made first.

$u_{\uparrow}(t) \neq u_{\uparrow}(t+1)$**:** We immediately obtain that step t is not interrupting step $t+1$

$$\neg int_{\uparrow}(t,t+1)$$

and thus the steps are unit-concurrent

$$ucon_{\uparrow}(t,t+1).$$

Claim 1 is now just Condition CodeMod. Claims 2 and 3 are just Lemma 205.

$u_{\uparrow}(t) = u_{\uparrow}(t+1) = i \wedge s(t) \in \Sigma_{P,i} \wedge s(t+1) \in \Sigma_{WB,i}$**:** Claim 1 trivially holds because step $t+1$ is not fetching

$$F_{\uparrow}(t+1) = \emptyset$$

and there is thus no code modification

$$\neg CM_{\uparrow}(t+1).$$

Claim 2 trivially holds because by assumption step t is not modifying core registers of step $t+1$

$$out_{\uparrow}(t) \not\parallel C_{\uparrow}(t+1),$$

and those are by definition the only inputs of a write buffer step

$$\begin{aligned} in_{\uparrow}(t+1) &= C_{\uparrow}(t+1) \cup F_{\uparrow}(t+1) \cup R_{\uparrow}(t+1) \\ &= C_{\uparrow}(t+1) \cup \emptyset \cup \emptyset \\ &= C_{\uparrow}(t+1). \end{aligned}$$

Thus there is no output-input intersection

$$out_{\uparrow}(t) \not\dot{\cap} in_{\uparrow}(t+1)$$

and thus also no write-read race between those two steps

$$\neg WR_{\uparrow}(t, t+1).$$

For Claim 3, assume that step $t+1$ is valid

$$\Gamma_{\uparrow}(t+1)$$

and that we have an output-output intersection

$$out_{\uparrow}(t) \dot{\cap} out_{\uparrow}(t+1).$$

Clearly the outputs of step $t+1$ are non-empty

$$out_{\uparrow}(t+1) \neq \emptyset,$$

and by contraposition of Lemma 174 we obtain that step $t+1$ is made in weak memory mode

$$\neg SC_{\uparrow}(t+1).$$

By Lemma 171 we obtain that the mode registers are not changed by step t

$$A_{SC,i} \not\dot{\cap} out_{\uparrow}(t)$$

and thus by Lemma 139 it is not modified by the step, and both steps are in weak memory mode

$$SC_{\uparrow}(t) = SC_{i\uparrow}(t) = SC_{i\uparrow}(t+1) = SC_{\uparrow}(t+1) = 0.$$

They are thus by definition shared, which was Claim 3

$$Sh_{\uparrow}(t) \wedge Sh_{\uparrow}(t+1).$$

$u_{\uparrow}(t) = u_{\uparrow}(t+1) = i \wedge s(t) \in \Sigma_{WB,i} \wedge s(t+1) \in \Sigma_{P,i}$**:** In each of the three cases, the outputs of step t intersect with some set X (either fetched registers, inputs, or outputs of step $t+1$)

$$out_{\uparrow}(t) \dot{\cap} X$$

and must thus be non-empty

$$out_{\uparrow}(t) \neq \emptyset.$$

By contraposition of Lemma 174 we obtain that step t is made in weak memory mode

$$\neg SC_{\uparrow}(t).$$

By Lemma 172 we obtain that the mode registers are not changed by step t

$$A_{SC,i} \not\parallel out_{\uparrow}(t),$$

and thus by Lemma 139 it is not modified by the step, and both steps are in weak memory mode

$$SC_{\uparrow}(t+1) = SC_{i\uparrow}(t+1) = SC_{i\uparrow}(t) = SC_{\uparrow}(t) = 0.$$

They are thus by definition shared, and step $t+1$ is a shared read, which solves all three claims

$$Sh_{\uparrow}(t) \wedge Sh_{\uparrow}(t+1) \wedge ShR_{\uparrow}(t+1).$$

□

We now wish to prove that read-write races are also annotated as shared with our discipline. The strategy will be to change the order of the read-write racing steps: in the reordered schedule we will have a write-read race, which is annotated correctly.

We show that unit-concurrency is preserved by reordering if the configurations of the interrupting step agree.

Lemma 207. *Let s be a schedule and O be a sequence of reorderings. Let t,k be two steps (in no particular order) in the original schedule which are unit-concurrent*

$$ucon_{\uparrow}(t,k),$$

which have been moved to t',k', respectively

$$s(t) = sO(t'),\ s(k) = sO(k').$$

If step t can be executed at its new position

$$c^t_{\uparrow} =^{s(t)}_{\uparrow} c_{\uparrow}O^{t'},$$

the steps are still concurrent in the new schedule

$$ucon_{\uparrow}O(t',k').$$

Proof. Note that the steps are still made by different units

$$\begin{aligned} & diffuO(t',k') \\ \equiv\ & u(sO(t')) \neq u(sO(k')) \\ \equiv\ & u(s(t)) \neq u(s(k)) \\ \equiv\ & diffu(t,k); \end{aligned}$$

and with Lemma 150 we obtain that the steps have the same victims, and thus that step t' after the reordering still does not interrupt step k'

$$\begin{aligned} int_{\uparrow}O(t',k') &\equiv u_{\uparrow}O(k') \in victims_{\uparrow}O(t') \\ &\equiv u_{\uparrow}(k) \in victims_{\uparrow}(t) && \text{L (150)} \\ &\equiv int_{\uparrow}(t,k). \end{aligned}$$

We conclude that the steps are still concurrent after the reordering

$$\begin{aligned} ucon_{\uparrow}O(t',k') &\equiv \neg int_{\uparrow}O(t',k') \wedge diffuO(t',k') \\ &\equiv \neg int_{\uparrow}(t,k) \wedge diffu(t,k) \\ &\equiv ucon_{\uparrow}(t,k) \\ &\equiv 1, \end{aligned}$$

which was the claim. □

The same holds for object-concurrency. The proof is completely analogous to that for weak-concurrency and omitted.

Lemma 208. *Let s be a schedule and O be a sequence of reorderings. Let t,k be two steps (in no particular order) in the original schedule which are object-concurrent*

$$ocon_{\uparrow}(t,k),$$

which have been moved to t',k', respectively

$$s(t) = sO(t'),\ s(k) = sO(k').$$

If step t can be executed at its new position

$$c^{t}_{\uparrow} =^{s(t)}_{\uparrow} c_{\uparrow}O^{t'},$$

the steps are still object-concurrent in the new schedule

$$ocon_{\uparrow}O(t',k').$$

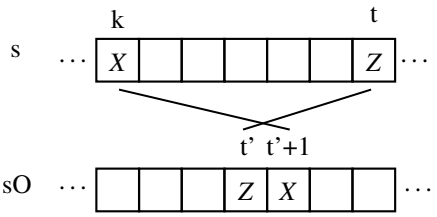


Figure 4.5: A race between t and k can be detected after moving t to t' and k to k', where they occur directly after one another.

Lemma 209. *Let s be a schedule and O be a sequence of reorderings. Let t,k be two steps (in no particular order) in the original schedule which are object-concurrent*

$$ocon_{\uparrow}(t,k),$$

which have been moved to $t',t'+1$, respectively (as in Fig. 4.5)

$$s(t) = sO(t'),\ s(k) = sO(t'+1).$$

Assume the new schedule is valid until right before t'

$$\Gamma^{t'-1}_{\uparrow}(sO),$$

and that steps t was valid

$$\Gamma_{\uparrow}(t).$$

Assume that step can be executed at its new position

$$c^{t}_{\uparrow} =^{s(t)}_{\uparrow} c_{\uparrow}O^{t'},$$

and that the configuration at t' only differs from the configuration at k by at most the outputs of step t

$$m_{\uparrow}O^{t'} =_{in_{\uparrow}(k)\setminus out_{\uparrow}(t)} m^{k}_{\uparrow},$$

and that the buffers of the original step k subsume those at $t'+1$ in the reordered schedule

$$bufS_{\uparrow}(sO(t'+1), c^{k}_{\uparrow}, c_{\uparrow}O^{t'+1}).$$

Then a race or code modification has to be correctly annotated:

1. $\Phi_{\uparrow}(k) \wedge CM_{\uparrow}(t,k) \rightarrow Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(k)$,
2. $\Gamma_{\uparrow}(k) \wedge WR_{\uparrow}(t,k) \rightarrow Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(k)$,
3. $\Gamma_{\uparrow}(k) \wedge WW_{\uparrow}(t,k) \rightarrow Sh_{\uparrow}(t) \wedge Sh_{\uparrow}(k)$.

Proof. By Lemma 150, the outputs of step t are the same when it is executed at t'

$$out_{\uparrow}O(t') = out_{\uparrow}(t).$$

We obtain that the configurations at t' and at k agree on inputs of step k that are not outputs of step t' after the reordering

$$m_{\uparrow}O^{t'} =_{in_{\uparrow}(k)\setminus out_{\uparrow}O(t')} m^{k}_{\uparrow},$$

and with Lemma 138 we obtain that the configuration at $t'+1$ agrees on the same addresses with the configuration at t'

$$m_{\uparrow}O^{t'+1} =_{in_{\uparrow}(k)\setminus out_{\uparrow}O(t')} m_{\uparrow}O^{t'} =_{in_{\uparrow}(k)\setminus out_{\uparrow}O(t')} m^{k}_{\uparrow}.$$

Note that partial agreement between functions is transitive, and thus the configuration at $t'+1$ agrees on the same addresses with the configuration at k

$$m_{\uparrow}O^{t'+1} =_{in_{\uparrow}(k)\setminus out_{\uparrow}O(t')} m^{k}_{\uparrow}.$$

By Lemma 208, step t' is object-concurrent with step $t'+1$ in the reordered schedule

$$ocon_{\uparrow}O(t', t'+1).$$

Thus step t' does not modify core registers of step $t'+1$

$$out_{\uparrow}O(t') \not\upharpoonright C_{\uparrow}(t'+1).$$

With Lemma 59 we can trace the core registers back to step k in the original schedule and obtain that step t' in the reordered schedule did not modify the core registers of step k in the original schedule

$$out_{\uparrow}O(t') \not\upharpoonright C_{\uparrow}(k).$$

Since the core registers are by definition inputs

$$C_\uparrow(k) \subseteq C_\uparrow(k) \cup R_\uparrow(k) \cup F_\uparrow(k) = in_\uparrow(k),$$

we obtain that the configurations at t' after the reordering and k before the reordering agree on the core registers

$$m_\uparrow O^{t'} =_{C_\uparrow(k)} m^k_\uparrow.$$

We now prove the three claims. The strategy is the same in each case: move the claim completely to the reordered schedule, where we can apply Lemma 206. Each claim uses the previous one in order to justify this move.

$\Phi_\uparrow(k) \wedge CM_\uparrow(t,k) \rightarrow Sh_\uparrow(t) \wedge ShR_\uparrow(k)$**:** We unfold the definition of concurrent code modification in the claim

$$\Phi_\uparrow(k) \wedge out_\uparrow(t) \barwedge F_\uparrow(k) \xrightarrow{!} Sh_\uparrow(t) \wedge ShR_\uparrow(k).$$

We rewrite all functions with parameter k using Lemma 151 and all functions with parameter t using Lemma 150

$$\Phi_\uparrow O(t'+1) \wedge out_\uparrow O(t') \barwedge F_\uparrow O(t'+1) \xrightarrow{!} Sh_\uparrow O(t') \wedge ShR_\uparrow O(t'+1).$$

We fold again the definition of a concurrent code modification in the claim

$$\Phi_\uparrow O(t'+1) \wedge CM_\uparrow O(t',t'+1) \xrightarrow{!} Sh_\uparrow O(t') \wedge ShR_\uparrow O(t'+1).$$

The claim is now Lemma 206.

$\Gamma_\uparrow(k) \wedge WR_\uparrow(t,k) \rightarrow Sh_\uparrow(t) \wedge ShR_\uparrow(k)$**:** We distinguish whether there is a concurrent code modification or not.

$CM_\uparrow(t,k)$**:** Because step k is valid, it is by definition also feasible

$$\Phi_\uparrow(k).$$

The claim is now solved by Claim 1 (shown above).

$\neg CM_\uparrow(t,k)$**:** By definition the outputs of step t do not modify fetched registers of step k

$$out_\uparrow(t) \barwedge F_\uparrow(k).$$

With Lemma 150 we obtain that the fetched registers are not modified by step t' after the reordering either

$$out_\uparrow O(t') \barwedge F_\uparrow(k)$$

and thus the configurations at $t'+1$ after and k before the reordering must also agree on the fetched registered registers

$$m_\uparrow O^{t'+1} =_{C_\uparrow(k) \cup F_\uparrow(k)} m^k_\uparrow.$$

Assume now that step k is valid in the original schedule

$$\Gamma_\uparrow(k).$$

By definition of validity, the step satisfies both the instance guard condition and local drain condition

$$I_{\uparrow}(k) \wedge \Delta_{\uparrow}(k).$$

With Lemma 152 we obtain that the instance guard condition is still satisfied

$$I_{\uparrow}O(t'+1) = I_{\uparrow}(k),$$

and with Lemma 179 and the assumption that the buffer is subsumed we obtain that the drain condition is still satisfied

$$\Delta_{\uparrow}O(t'+1).$$

Therefore the step is by definition still valid at its new position

$$\Gamma_{\uparrow}O(t'+1).$$

We unfold the write-read race in the claim with Lemma 157

$$out_{\uparrow}(t) \mathbin{\dot{\cap}} in_{\uparrow}(k) \xrightarrow{!} Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(k).$$

We rewrite all functions with parameter k using Lemma 152 and all functions with parameter t using Lemma 150

$$out_{\uparrow}O(t') \mathbin{\dot{\cap}} in_{\uparrow}O(t'+1) \xrightarrow{!} Sh_{\uparrow}O(t') \wedge ShR_{\uparrow}O(t'+1).$$

We fold again the definition of a write-read race

$$WR_{\uparrow}O(t',t'+1) \xrightarrow{!} Sh_{\uparrow}O(t') \wedge ShR_{\uparrow}O(t'+1).$$

The claim is now Lemma 206.

$\Gamma_{\uparrow}(k) \wedge WW_{\uparrow}(t,k) \rightarrow Sh_{\uparrow}(t) \wedge Sh_{\uparrow}(k)$**:** We distinguish whether there is a write-read race or not.

$WR_{\uparrow}(t,k)$**:** With Claim 2 (shown above) we obtain that the steps are shared and a shared read, respectively

$$Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(k).$$

The claim is now solved by Lemma 129

$$Sh_{\uparrow}(t) \wedge Sh_{\uparrow}(k).$$

$\neg WR_{\uparrow}(t,k)$**:** By definition the outputs of step t do not modify inputs of step k

$$out_{\uparrow}(t) \mathbin{\dot{\not\cap}} in_{\uparrow}(k).$$

With Lemma 150 we obtain that the inputs are not modified by step t' after the reordering either

$$out_{\uparrow}O(t') \mathbin{\dot{\not\cap}} in_{\uparrow}(k)$$

and thus the configurations at $t'+1$ after and k before the reordering must agree on all inputs

$$m_{\uparrow}O^{t'+1} =_{in_{\uparrow}(k)} m_{\uparrow}^{k}.$$

By Lemma 187 the step can still be executed at its new position

$$c^k_\uparrow =^{s(k)}_\uparrow c_\uparrow O^{t'+1}.$$

We unfold the write-write race in the claim with Lemma 157

$$\Gamma_\uparrow(k) \wedge out_\uparrow(t) \dot{\cap} out_\uparrow(k) \xrightarrow{!} Sh_\uparrow(t) \wedge Sh_\uparrow(k).$$

We move everything to the reordered schedule with Lemma 150

$$\Gamma_\uparrow O(t'+1) \wedge out_\uparrow O(t') \dot{\cap} out_\uparrow O(t'+1) \xrightarrow{!} Sh_\uparrow O(t') \wedge Sh_\uparrow O(t'+1).$$

We fold again the write-write race

$$\Gamma_\uparrow O(t'+1) \wedge WW_\uparrow O(t',t'+1) \xrightarrow{!} Sh_\uparrow O(t') \wedge Sh_\uparrow O(t'+1).$$

The claim is now Lemma 206.

□

We now prove the symmetry of our annotation.

Lemma 210. *Read-write races require shared accesses in valid schedules, i.e., when s is valid until $t+1$*

$$\Gamma^{t+1}_\uparrow(s)$$

and there are unit-concurrent (in both directions) steps

$$ucon_\uparrow(t,t+1) \wedge ucon_\uparrow(t+1,t),$$

that have a read-write race

$$RW_\uparrow(t,t+1),$$

then there is a write-read race and the steps are correctly annotated, or the read is a shared read and the write is shared

$$WR_\uparrow(t,t+1) \wedge Sh_\uparrow(t) \wedge ShR_\uparrow(t+1) \vee ShR_\uparrow(t) \wedge Sh_\uparrow(t+1).$$

Proof. We distinguish whether a write-read race exists or not.

$WR_\uparrow(t,t+1)$: By Lemma 205 we obtain that the steps are marked as shared and as a shared read, respectively

$$Sh_\uparrow(t) \wedge ShR_\uparrow(t+1),$$

and the claim follows.

$\neg WR_\uparrow(t,t+1)$: We will switch the order of the two accesses. This is possible because the second step does not read any registers that are changed by the first step. Since we are in the case without any write/read races, i.e.,

$$out_\uparrow(t) \not\dot{\cap} in_\uparrow(t+1),$$

we obtain with Lemma 138 that inputs are unchanged

$$m^{t+1}_\uparrow =_{in_\uparrow(t+1)} m^t_\uparrow = m_\uparrow[t \leftrightarrow t+1]^t.$$

By assumption the steps are unit-concurrent and thus made by different units

$$diffu(t,t+1).$$

By definition, the unit making step t is thus not the unit making step $t+1$

$$u_\uparrow(t) = u(s(t)) \neq u(s(t+1)) = u_\uparrow(t+1),$$

and by Lemma 95 we obtain that the write buffer of the unit making step $t+1$ is unchanged by the reordering

$$\begin{aligned} wb_\uparrow^{t+1}(u_\uparrow(t+1)) &= Op_{u_\uparrow(t)\uparrow}(wb_\uparrow^t(u_\uparrow(t+1)),\ldots) \\ &= wb_\uparrow^t(u_\uparrow(t+1)) && \text{L 95} \\ &= wb_\uparrow[t \leftrightarrow t+1]^t(u_\uparrow(t+1)). \end{aligned}$$

We conclude with Lemma 188 that the configurations strongly agree for the oracle input of step $t+1$

$$c_\uparrow^{t+1} =_\uparrow^{s(t+1)} c_\uparrow[t \leftrightarrow t+1]^t. \tag{4.2}$$

With Lemma 95 we also obtain that the buffers of the unit making step t are not changed

$$\begin{aligned} wb_\uparrow[t \leftrightarrow t+1]^{t+1}(u_\uparrow(t)) &= Op_{u_\uparrow[t\leftrightarrow t+1](t)\uparrow}(wb_\uparrow[t \leftrightarrow t+1]^t(u_\uparrow(t)),\ldots) \\ &= wb_\uparrow[t \leftrightarrow t+1]^t(u_\uparrow(t)) && \text{L 95} \\ &= wb_\uparrow^t(u_\uparrow(t)). \end{aligned}$$

and by Lemma 90 we obtain that the buffers of step t subsume those of step $t+1$ in the reordered schedule

$$bufS_\uparrow(s(t), c_\uparrow^t, c_\uparrow[t \leftrightarrow t+1]^{t+1}). \tag{4.3}$$

By definition, the new positions of the steps are now t and $t+1$

$$s(t+1) = s[t \leftrightarrow t+1](t),\ s(t) = s[t \leftrightarrow t+1](t+1). \tag{4.4}$$

Because the steps before t are not changed, the new schedule is valid until $t-1$

$$\Gamma_\uparrow^{t-1}(s[t \leftrightarrow t+1]). \tag{4.5}$$

Because the steps before t are not changed, the memory at t is the same

$$m_\uparrow[t \leftrightarrow t+1]^t = m_\uparrow^t.$$

Thus the memories at t agree in particular on those inputs which are not outputs of t

$$m_\uparrow[t \leftrightarrow t+1]^t =_{in_\uparrow(t)\setminus out_\uparrow(t+1)} m_\uparrow^t. \tag{4.6}$$

By assumption steps $t+1$ and t were valid

$$\Gamma_\uparrow(t+1) \wedge \Gamma_\uparrow(t). \tag{4.7}$$

Note now that a read-write race is a write-read race with swapped parameters, and thus we have a write-read race between $t+1$ and t

$$WR_{\uparrow}(t+1,t). \tag{4.8}$$

By Lemma 105 the steps are also object-concurrent

$$ocon_{\uparrow}(t,t+1). \tag{4.9}$$

From Lemma 209, with $O := [t \leftrightarrow t+1]$, $k := t$, $t := t+1$, and $t' := t$, and Eqs. (4.2) to (4.9) we obtain now that step $t+1$ is shared and step t is a shared read

$$Sh_{\uparrow}(t+1) \wedge ShR_{\uparrow}(t),$$

and the claim follows.

□

While a processor in sequential mode writes to interrupt registers, it can not buffer a write.

Lemma 211.

$$\Gamma^{t}_{\uparrow}(s) \wedge s(t) \in \Sigma_{P,i} \wedge SC_{\uparrow}(t) \wedge out_{\uparrow}(t) \dot{\cap} A_{IPR} \rightarrow BW_{\uparrow}(t) = \emptyset.$$

Proof. Let j be a unit of which the interrupt registers are modified by step t

$$out_{\uparrow}(t) \dot{\cap} A_{IPR,j}.$$

The step is modifying interrupt registers and by Lemma 133 must be shared

$$Sh_{\uparrow}(t).$$

By Condition IRRForwarding, the buffered writes may not include processor registers of unit j

$$Dom(PW_{\uparrow}(t).wba) \not\dot{\cap} A_{PR,j}$$

and thus also not its interrupt registers

$$Dom(PW_{\uparrow}(t).wba) \not\dot{\cap} A_{IPR,j}.$$

Since a sequentially consistent step in the high-level machine does not use low-level machine semantics

$$LL_{\uparrow}(t) = \neg SC_{\uparrow}(t) = 0,$$

the executed write is the union of the prepared writes

$$W_{\uparrow}(t) = PW_{\uparrow}(t).wba \cup PW_{\uparrow}(t).bpa.$$

With Lemmas 15 and 4 we obtain that the write is intersecting with the interrupt processor registers.

$$\begin{aligned}
& out_{\uparrow}(t) \dot{\cap} A_{IPR,j}. \\
\iff & idc(WS_{\uparrow}(t)) \dot{\cap} A_{IPR,j} \\
\iff & WS_{\uparrow}(t) \dot{\cap} A_{IPR,j} && \text{L 15, 4}
\end{aligned}$$

$$\iff Dom(W_\uparrow(t)) \dot\cap A_{IPR,j}.$$

Since the buffered portion of the prepared write does not intersect the interrupt registers, it must be the bypassing portion that causes the intersection

$$\begin{aligned}
&\iff Dom(PW_\uparrow(t).wba \cup PW_\uparrow(t).bpa) \dot\cap A_{IPR,j} \\
&\iff Dom(PW_\uparrow(t).wba) \cup Dom(PW_\uparrow(t).bpa) \dot\cap A_{IPR,j} \\
&\iff Dom(PW_\uparrow(t).wba) \dot\cap A_{IPR,j} \vee Dom(PW_\uparrow(t).bpa) \dot\cap A_{IPR,j} \\
&\iff 0 \vee Dom(PW_\uparrow(t).bpa) \dot\cap A_{IPR,j} \\
&\iff Dom(PW_\uparrow(t).bpa) \dot\cap A_{IPR,j}.
\end{aligned}$$

We conclude that the prepared bypassing writes are not just normal processor registers of unit i

$$Dom(PW_\uparrow(t).bpa) \not\subseteq A_{NPR,i},$$

and by Condition AtomicWrite the domain of the prepared buffered writes is empty

$$Dom(PW_\uparrow(t).wba) = \emptyset$$

and the claim follows

$$BW_\uparrow(t) = PW_\uparrow(t).wba = \emptyset.$$

□

4.7 Proof Strategy for Write Buffer Reduction

Our proof can be split into three phases. In the first phase we show that information flows between threads only if the threads are synchronized by a sequence of

1. shared write/shared read pairs
2. shared RMW/write pairs
3. IPIs

We then consider a schedule where processors are only allowed to perform a shared read if no other processor has a buffered shared write (to any address), and where the global order of shared writes is not changed by write buffers. By this we mean that the n-th shared write that is issued by a processor is also the n-th shared write to be committed to memory by the write buffers. In such a schedule, a processor in SC mode which has a buffered write can not be synchronized by either of the above three, because

1. the shared write is still buffered, preventing the shared read by assumption
2. a shared RMW drains the buffer (cf. Condition MessagePassing)
3. IPIs drain the buffer

By contraposition there is no information flow from a processor which has buffered writes, i.e., as long as a processor is buffering writes to an address, no other processor reads from that address. Consequently one can show that in such a schedule forwarding is correct for all addresses which are being accessed, and thus the low-level computation and the high-level computation see the same values.

Finally one shows that every schedule can be reordered to an equivalent schedule which has that property.

4.8 Communication and Synchronization

We now consider channels of communication between processors and APICs. We define a relation synchronized-with, which identifies situations in which an earlier step can not be moved across a later step because

- The later step depends on the earlier step,
- the later step would prevent the earlier step if the order would be reversed, in particular if this would hide a race, or
- the later step would change the program annotation of the first step. This is the case only if the first step is a processor step, and the second step is interrupting the step; in this case the IPRs may change, which however are used as an input to *iSh* and *iShR*.

For example, a read depends on the most recent writes to the same addresses, and a write can disable a compare-and-swap.

The results and definitions in this section can mostly be found in a less streamlined version in [Obe15]. The fact that they still hold in the new machine relies on a careful redefinition of the synchronized-with relation, which is now based on inputs and outputs rather than the actual steps. We first give a formal definition, then quickly describe the non-obvious rules.

We distinguish between forward-synchronization, which depends on step t and the oracle input of step k, and synchronization, which also depends on the step k.

We say *t is forward-synchronized-with k in s* and write $t \rhd [s]k$ when step t can not be moved/dropped due to the oracle input of step k.

$$\frac{t < k \qquad \neg ocon_{\uparrow}(t,k)}{t \rhd k}\ \textsc{NotConcurrent}$$

$$\frac{t = hd(issue^{k}_{\uparrow}(i)) \qquad s(k) \in \Sigma_{WB,i}}{t \rhd k}\ \textsc{IssueWrite}$$

$$\frac{t < k \qquad s(t) \in \Sigma_{WB,i} \qquad s(k) \in \Sigma_{P,i}}{t \rhd k}\ \textsc{ProcessorFence}$$

Rule NotConcurrent prevents steps that are not object-concurrent from overtaking each other.

Rule IssueWrite considers a buffered write, which can not overtake the write buffer step that commits it to memory.

Rule ProcessorFence prevents a fence (or partial hit) from overtaking write buffer steps on the same processor. It is slightly more general than that since it blocks all processor steps, including those that are not fences, from overtaking the write buffer step, but that will turn out to be fine.

We say *t is race-synchronized with k* and write $t \Rrightarrow [s]k$ if it is forward-synchronized or there is a visible write-read race that is annotated correctly

$$\frac{t \rhd k}{t \Rrightarrow k}$$

$$\frac{t < k \qquad VR_{\uparrow}(t,k) \qquad Sh_{\uparrow}(t) \qquad ShR_{\uparrow}(k)}{t \Rrightarrow k}\ \textsc{CommExt}$$

Rule CommExt is interesting because it seemingly ignores interaction which is not using shared accesses. This will turn out to be fine due to Condition Races, which will cause all interaction through the memory which is not marked as shared to be synchronized by another memory interaction which is marked as shared, e.g., the release and subsequent acquisition of a lock. This relation corresponds to the synchronization relation of [Obe15] and is insufficient for machines with IPIs and heterogeneous memory accesses. To tackle these, we define a new relation with two new rules.

We say *t is synchronized-with k* and write $t \blacktriangleright [s]k$ when step t can not be moved/-dropped because of what step k *does*.

$$\frac{t \Rrightarrow k}{t \blacktriangleright k}$$

$$\frac{t < k \qquad mwrite_{\uparrow}(t) \qquad int_{\uparrow}(k,t)}{t \blacktriangleright k}\ \textsc{Interrupted}$$

$$\frac{t < k \qquad ShR_{\uparrow}(t) \qquad mwrite_{\uparrow}(t) \qquad RW_{\uparrow}(t,k)}{t \blacktriangleright k}\ \textsc{RMWDisable}$$

Rule Interrupted prevents interrupts from overtaking writes: if the interrupt overtakes a write, the write is possibly not executed anymore. Rule RMWDisable prevents writes from overtaking read-modify-writes that read-write race them; such read-modify-writes might be disabled if the write is executed before the RMW.

We show that both race- and forward-synchronization synchronize steps. We do not show proofs.

Lemma 212.

$$t \rhd k \rightarrow t \blacktriangleright k.$$

Lemma 213.

$$t \Rrightarrow k \rightarrow t \blacktriangleright k.$$

4.9 Delaying Unsynchronized Steps

We prove that Condition Races implies that unshared accesses to the same memory region are also synchronized. The proof works by moving potentially unsynchronized

accesses next to each other, at which point they become races in the sense of Condition Races and have to be shared. This technique is familiar from Lemma 209.

If t is not synchronized-with any of the steps until k, then it can be reordered until k and the schedule changes significantly only after k, as in Figure 4.6.

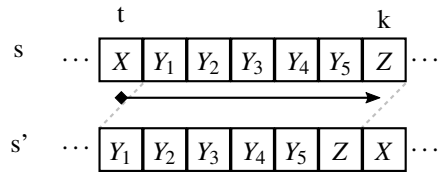


Figure 4.6: If step t (made by object X) is not synchronized with any of the steps until k (made by objects Y_i), and not forward-synchronized with step k (made by object Z), it can be delayed behind step k.

This will only work if the involved steps are valid, and the last step is at least feasible. For technical reasons we need to exclude write buffer steps that are made with an empty write buffer, and we define *WB-feasibility* as feasibility plus the write buffer drain condition

$$\Psi_M(c,x) = \Phi_M(c,x) \wedge \Lambda_M(c,x).$$

We define analogously to semi-validity a *WB-validity*

$$\Gamma\Psi^t_M(s) \equiv \Gamma^{t-1}_M(s) \wedge \Psi_M[s](t).$$

Interestingly, since the schedules will behave in the same way, it will not matter whether WB-feasibility and validity of the steps hold in the original schedule or the reordered one. In fact, we only need for each individual step that it is valid (resp. WB-feasible) in one of the schedules, so it is completely acceptable for some steps to be valid (resp. WB-feasible) in one and the other steps to be valid (resp. WB-feasible) in the other schedule. We express this with a predicate *pval* for pseudo-valid

$$\begin{aligned} pval[s](t,k) = \Gamma^t_\uparrow(s) \\ \wedge \forall t' \in (t:k).\Gamma_\uparrow(t') \vee \Gamma_\uparrow[t \to k](t'-1) \\ \wedge (\Psi_\uparrow(k) \vee \Psi_\uparrow[t \to k](k-1)). \end{aligned}$$

Note that the steps until step t are the same in both schedules, so we do not make this split for those steps. The steps t' after step t are executed at $t'-1$ after the reordering.

We prove with Invariant Valid below that this pseudo-validity indeed suffices to get validity in both of the two schedules.

We call step t *delayable until* k if the schedule is pseudo-valid and step t is not synchronized with any step until k

$$delay[s](t,k) \equiv pval[s](t,k) \wedge \forall t' \in (t:k].t \not\blacktriangleright [s]t'.$$

We define several invariants, all of which (as we will show) are preserved during reordering of delayable steps.

NearlySame The configurations at $k+1$ and k are nearly the same

$$\mathcal{I}^t_\uparrow([t \to k], k+1, k).$$

IssueSame Except for the operation made in step t

$$f_i(l) = Op_{i\uparrow}(t)(l,t),$$

the sequences of issued writes are nearly the same at $k+1$ and k

$$\mathcal{J}^{f_i}_\uparrow([t \to k], k+1, k).$$

NothingIssued No new write buffer entries are issued by the unit making step t

$$\forall t' \in issue_\uparrow[t \to k]^k(u_\uparrow(t)).t' < t.$$

Valid Steps before k are still valid resp. WB-feasible in both schedules

$$\Gamma\Psi^k_\uparrow(s) \wedge \Gamma\Psi^{k-1}_\uparrow(s[t \to k]).$$

The validity of step k is also the same

$$\Gamma_\uparrow(k) = \Gamma_\uparrow[t \to k](k-1).$$

SameState If step t is a write and step k is valid

$$mwrite_\uparrow(t) \wedge \Gamma_\uparrow(k),$$

step t can be moved to k and sees the same values from memory

$$m_\uparrow[t \to k]^k =_{in_\uparrow(t)} m^t_\uparrow,$$

and the buffers in step t subsume those in step k after the reordering

$$bufS_\uparrow(s(t), c^t_\uparrow, c_\uparrow[t \to k]^k).$$

We prove these invariants by induction on k. In each proof we can use the invariants for k but also invariants for $k+1$ which we have already shown. To refer to Invariant X when reordering t until k, we use the notation $X(t,k)$, e.g.,

$$NearlySame(t,k).$$

If all invariants hold at k, we write

$$Inv(t,k) \equiv NearlySame(t,k) \wedge \ldots \wedge SameState(t,k).$$

Clearly the invariants hold when moving t to t

Lemma 214.

$$Inv(t,t).$$

Proof. Note first that all predicates and components are unchanged by moving t to t

$$X_\uparrow[t \to t] = X_\uparrow.$$

We now prove the invariants.

NearlySame: The claim is just a special case of Lemma 190.

IssueSame: The claim is just a special case of Lemma 201.

NothingIssued Clearly all issued timestamps are before the current timestamp

$$\forall t' \in issue_{\uparrow}[t \to t]^t(u_{\uparrow}(t)).t' < t.$$

Valid: Pseudo-validity gives us validity until t

$$\Gamma^t_{\uparrow}(s).$$

The claims are all weaker than this and the invariant follows.

SameState: There is no reordering, and the claim about memory follows

$$m_{\uparrow}[t \to t]^t = m^t_{\uparrow},$$

and the write buffers are similarly unchanged

$$wb_{\uparrow}[t \to t]^t = wb^t_{\uparrow}.$$

The remaining claim follows with Lemma 90

$$bufS_{\uparrow}(s(t), c^t_{\uparrow}, c_{\uparrow}[t \to t]^t).$$

□

We now show that the invariants are actually maintained by the reordering.

The essence of the proof is quickly explained. We show that races between steps t and $k+1$ are maintained by moving step t until step k. Since steps k and $k+1$ are adjacent, this allows us to use Lemma 209 to show that races have to be correctly annotated. At this point we use the fact that there is no synchronization between steps t and k to deduce that there is no correctly annotated race, therefore no race, and thus also no race in the original schedule, which allows us to move step t further to position $k+1$. The situation is depicted in Figure 4.7.

In what follows, we assume that t is always less or equal to k

$$t \leq k.$$

Lemma 215.

$$t \not\rhd k+1 \to o_{\uparrow}(t) \neq o_{\uparrow}(k+1).$$

Proof. By contraposition of Rule NOTCONCURRENT, steps t and $k+1$ are object-concurrent

$$ocon_{\uparrow}(t, k+1).$$

The claim follows. □

Lemma 216. *If step t is not forward-synchronized with step $k+1$, it is not made by the write buffer of the unit that makes step $k+1$*

$$t \not\rhd k+1 \to s(t) \notin \Sigma_{WB, u_{\uparrow}(k+1)},$$

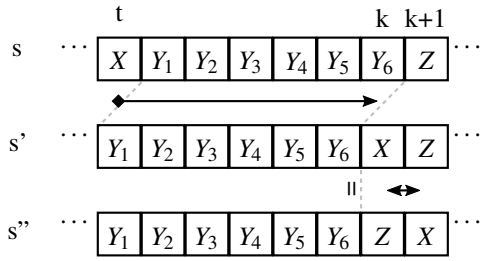


Figure 4.7: We obtain s' by moving step t to position k, and s'' by moving t one step further to the right. In s', we try to execute $s(k+1)$ after $s(t)$, but in s'' we execute $s(k+1)$ at position k.

Proof. Let the unit making step $k+1$ be i

$$i = u_\uparrow(k+1).$$

We distinguish between the object making step $k+1$

$s(k+1) \in \Sigma_{P,i}$**:** The claim is the contraposition of Rule PROCESSORFENCE.

$s(k+1) \in \Sigma_{WB,i}$**:** By Lemma 215, steps t and $k+1$ are not made by the same object

$$o_\uparrow(t) \neq o_\uparrow(k+1)$$

and the claim follows

$$s(t) \notin \Sigma_{WB,i}.$$

□

If step t is made by the same unit as step $k+1$ but is not forward-synchronized, step t is a processor step and step $k+1$ is a write buffer step.

Lemma 217.

$$u_\uparrow(t) = u_\uparrow(k+1) \wedge t \not\triangleright k+1 \rightarrow s(t) \in \Sigma_{P,u_\uparrow(t)} \wedge s(k+1) \in \Sigma_{WB,u_\uparrow(t)}.$$

Proof. By Lemma 216 step t is not made by the write buffer of step $k+1$

$$s(t) \notin \Sigma_{WB,u_\uparrow(k+1)},$$

and since it is made by unit $u_\uparrow(k+1)$

$$u_\uparrow(t) = u_\uparrow(k+1),$$

it must be made by the processor of that unit

$$s(t) \in \Sigma_{P,u_\uparrow(k+1)}.$$

By Lemma 215 we can now exclude the processor for step $k+1$ and obtain that it must have been made by the write buffer of unit $u_\uparrow(k+1)$

$$s(k+1) \in \Sigma_{WB,u_\uparrow(k+1)}.$$

By assumption, that is also the unit making step t and the claim follows

$$s(t) \in \Sigma_{P,u_\uparrow(t)} \wedge s(k+1) \in \Sigma_{WB,u_\uparrow(t)}.$$

□

Because no new writes have been added to the buffer of the unit making step t, the sequence of issued writes of that unit is unaffected by the reordering.

Lemma 218.

$$Inv(t,k) \wedge i = u_\uparrow(t) \rightarrow issue_\uparrow[t \rightarrow k]^k(i) = mv[t \leftarrow k](issue_\uparrow[t \rightarrow k]^k(i)).$$

Proof. By Invariant NothingIssued no new writes have entered the sequence of issued writes

$$\forall t' \in issue_\uparrow[t \rightarrow k]^k(i).t' < t.$$

The claim follows with Lemma 77

$$\begin{aligned} mv[t \leftarrow k](issue_\uparrow[t \rightarrow k]^k(i)) &= \left[mv[t \leftarrow k](t') \;\middle|\; t' \in issue_\uparrow[t \rightarrow k]^k(i) \right] \\ &= \left[t' \;\middle|\; t' \in issue_\uparrow[t \rightarrow k]^k(i) \right] && \text{L 77} \\ &= issue_\uparrow[t \rightarrow k]^k(i). \end{aligned}$$

□

This simplifies the invariant for the unit making step t.

Lemma 219.

$$Inv(t,k) \wedge i = u_\uparrow(t) \rightarrow issue_\uparrow^{k+1}(i) = Op_{i\uparrow}(t)(issue_\uparrow[t \rightarrow k]^k(i),t)$$

Proof. By Invariant IssueSame, the sequences of write buffers are nearly the same

$$issue_\uparrow^{k+1}(i) = Op_{i\uparrow}(t)(mv[t \leftarrow k](issue_\uparrow[t \rightarrow k]^k(i)),t),$$

and the claim follows immediately with Lemma 218

$$issue_\uparrow^{k+1}(i) = Op_{i\uparrow}(t)(issue_\uparrow[t \rightarrow k]^k(i),t).$$

□

Lemma 220.

$$Inv(t,k) \rightarrow wb_\uparrow^{k+1}(i) = Op_{i\uparrow}(t)(wb_\uparrow[t \rightarrow k]^k(i),BW_\uparrow(t)).$$

Proof. We distinguish whether unit i made step t or not

$u_\uparrow(t) = i$**:** By Lemma 219 step t is exactly the step missing in the sequence of issued writes

$$issue_\uparrow^{k+1}(i) = Op_{i\uparrow}(t)(issue_\uparrow[t \rightarrow k]^k(i),t).$$

Using Lemmas 123 and 36 we obtain the same for the write buffers

$$\begin{aligned} wb_\uparrow^{k+1}(i) &= \left[BW_\uparrow(t') \;\middle|\; t' \in issue_\uparrow^{k+1}(i) \right] && \text{L 123} \\ &= \left[BW_\uparrow(t') \;\middle|\; t' \in Op_{i\uparrow}(t)(issue_\uparrow[t \rightarrow k]^k(i),t) \right] \\ &= Op_{i\uparrow}(t)(\left[BW_\uparrow(t') \;\middle|\; t' \in issue_\uparrow[t \rightarrow k]^k(i) \right], BW_\uparrow(t)) && \text{L 36} \\ &= Op_{i\uparrow}(t)(wb_\uparrow[t \rightarrow k]^k(i),BW_\uparrow(t)), && \text{L 123} \end{aligned}$$

which was the claim.

$u_\uparrow(t) \neq i$**:** By Invariant NearlySame the buffers are the same, and the claim follows with Lemma 95

$$wb_\uparrow^{k+1}(i) = wb_\uparrow[t \to k]^k(i) = Op_{i\uparrow}(t)(wb_\uparrow[t \to k]^k(i), BW_\uparrow(t)).$$

□

Lemma 221. *Assume that the schedule is pseudo-valid until $k+1$, the invariants hold at k, t is delayable until $k+1$, and step t is not forward-synchronized with step $k+1$*

$$pval(t, k+1) \wedge Inv(t,k) \wedge t \not\triangleright k+1,$$

and that step $k+1$ is a step of the write buffer of processor i

$$s(k+1) \in \Sigma_{WB,i}.$$

Then the operation at t was not a push, or the sequence of issued writes before the reordering must be non-empty

$$Op_{i\uparrow}(t) \neq push \vee issue_\uparrow^{k+1}(i) \neq \varepsilon \wedge issue_\uparrow[t \to k]^k(i) \neq \varepsilon.$$

Proof. Assume that the operation was a push

$$Op_{i\uparrow}(t) = push.$$

Thus step t was made by the processor i and thus by unit i

$$u_\uparrow(t) = i.$$

By Lemma 219 step t is exactly the step missing in the sequence of issued writes

$$issue_\uparrow^{k+1}(i) = Op_{i\uparrow}(t)(issue_\uparrow[t \to k]^k(i), t).$$

We conclude that the sequence of issued writes at $k+1$ contains the issued writes at k after reordering and t

$$issue_\uparrow^{k+1}(i) = push(issue_\uparrow[t \to k]^k(i), t) = issue_\uparrow[t \to k]^k(i) \circ t,$$

and is thus clearly non-empty

$$issue_\uparrow^{k+1}(i) \neq \varepsilon,$$

which is the first half of the claim.

Step t is in the buffer at $k+1$

$$t \in issue_\uparrow^{k+1}(i)$$

but we obtain by contraposition of Rule IssueWrite that t can not be the step that issued the committed write

$$t \neq hd(issue_\uparrow^{k+1}(i)),$$

and by Lemma 38, the tail of the issued writes is not empty

$$tl(issue_\uparrow^{k+1}(i)) \neq \varepsilon.$$

By Lemma 39 we obtain the claim

$$issue_\uparrow[t \to k]^k(i) \neq \varepsilon.$$

□

If step $k+1$ is a write buffer step, the write buffer after and before the reordering must be non-empty.

Lemma 222. *Assume that the schedule is pseudo-valid until $k+1$, the invariants hold at k, t is delayable until $k+1$, and step t is not forward-synchronized with step $k+1$*

$$pval(t,k+1) \wedge Inv(t,k) \wedge t \not\rhd k+1,$$

that step $k+1$ is a step of the write buffer of processor i

$$s(k+1) \in \Sigma_{WB,i}.$$

Then the write buffers after and before the reordering must be non-empty

$$wb_{\uparrow}^{k+1}(i) \neq \varepsilon \wedge wb_{\uparrow}[t \to k]^k(i) \neq \varepsilon.$$

Proof. We distinguish between the three cases for the operator applied in step t.

$s(t) \in \Sigma_{P,i} \wedge BW_{\uparrow}(t) \neq \emptyset$**:** Since step t buffered a write, the operation is a push

$$Op_{i\uparrow}(t) = push$$

and by Lemma 221 the sequence of issued writes at $k+1$ in the original schedule and at k in the reordered schedule are non-empty

$$issue_{\uparrow}^{k+1}(i) \neq \varepsilon \wedge issue_{\uparrow}[t \to k]^k(i) \neq \varepsilon.$$

The claims follow with Lemma 123

$$wb_{\uparrow}^{k+1}(i) \neq \varepsilon \wedge wb_{\uparrow}[t \to k]^k(i) \neq \varepsilon.$$

$s(t) \in \Sigma_{WB,i}$**:** By contraposition of Rule NotConcurrent, the steps are object-concurrent

$$ocon_{\uparrow}(t,k+1)$$

and thus by definition made by different objects, which is a contradiction

$$WB,i = o_{\uparrow}(t) \neq o_{\uparrow}(k+1) = WB,i.$$

Otherwise: By Lemma 220 the buffers are missing exactly step t

$$wb_{\uparrow}^{k+1}(i) = Op_{i\uparrow}(t)(wb_{\uparrow}[t \to k]^k(i), BW_{\uparrow}(t)).$$

Since the operator is a noop, the buffers are the same

$$wb_{\uparrow}^{k+1}(i) = wb_{\uparrow}[t \to k]^k(i),$$

and it suffices to show that either of them is non-empty. Since step t can be delayed until $k+1$, by definition of *delay* the schedule is pseudo-valid

$$pval(t,k+1)$$

and thus by definition of *pval* step $k+1$ is WB-feasible in at least one of the schedules

$$\Psi_{\uparrow}(k+1) \vee \Psi_{\uparrow}[t \to k](k).$$

Since step $k+1$ is a write buffer step, this implies that the drain condition holds in at least one of the steps

$$\Delta_{\uparrow}(k+1) \vee \Delta_{\uparrow}[t \to k](k).$$

We conclude that the write buffer is non-empty in one of the steps

$$wb_{\uparrow}^{k+1}(i) \neq \varepsilon \vee wb_{\uparrow}[t \to k]^{k}(i) \neq \varepsilon.$$

Since they are equal, they must both be non-empty

$$wb_{\uparrow}^{k+1}(i) = wb_{\uparrow}[t \to k]^{k}(i) \neq \varepsilon,$$

which is the claim.

□

We can now show the first part of Invariant Valid.

Lemma 223. *If in a pseudo-valid schedule the invariants hold at k and step t is not forward-synchronized with $k+1$, then the schedule and the reordered schedule are WB-valid until $k+1$ resp. k*

$$pval(t,k+1) \wedge Inv(t,k) \wedge t \not\rhd k+1 \to \Gamma\Psi_{\uparrow}^{k+1}(s) \wedge \Gamma\Psi_{\uparrow}^{k}(s[t \to k+1]).$$

Proof. By Invariant Valid the steps before k are all valid

$$\Gamma_{\uparrow}^{k-1}(s) \wedge \Gamma_{\uparrow}^{k-2}(s[t \to k]).$$

The schedule where t is moved to $k+1$ is obtained by moving t to k and then k to $k+1$

$$s[t \to k+1] = s[t \to k][k \leftrightarrow k+1],$$

and the steps before k are unaffected by the latter move. Thus we obtain that the steps are also valid until $k-2$ when t is moved to $k+1$

$$\Gamma_{\uparrow}^{k-2}(s[t \to k+1]) = \Gamma_{\uparrow}^{k-2}(s[t \to k]) = 1,$$

and that the validity of step $k-1$ before and after this additional reordering is the same

$$\Gamma_{\uparrow}[t \to k+1](k-1) = \Gamma_{\uparrow}[t \to k](k-1).$$

The schedule is pseudo-valid until $k+1$, and thus step k is valid in one of the schedules

$$\Gamma_{\uparrow}(k) \vee \Gamma_{\uparrow}[t \to k](k-1).$$

By Invariant Valid the validity of the step is the same in both schedules, i.e., the step is valid

$$\Gamma_{\uparrow}(k) = \Gamma_{\uparrow}[t \to k](k-1) = 1.$$

Step $k-1$ after the additional reordering is the same and thus also valid

$$\Gamma_{\uparrow}[t \to k+1](k-1).$$

We conclude that the schedules are valid until k reps. $k-1$

$$\Gamma_{\uparrow}^{k}(s) \wedge \Gamma_{\uparrow}^{k-1}(s[t \to k+1]).$$

It thus only remains to be shown that they are WB-feasible at $k+1$ resp. k

$$\Psi_\uparrow(k+1) \overset{!}{\wedge} \Psi_\uparrow[t \to k+1](k).$$

The proof now depends on whether step $k+1$ is made by a processor or a write buffer.

$s(k+1) \in \Sigma_{P,i}$**:** For processor steps, WB-feasibility is just feasibility

$$\Phi_\uparrow(k+1) \overset{!}{\wedge} \Phi_\uparrow[t \to k+1](k).$$

By assumption, at least one of the steps is feasible

$$\Phi_\uparrow(k+1) \vee \Phi_\uparrow[t \to k+1](k).$$

By Invariant NearlySame, the configurations at $k+1$ and k are nearly the same

$$\mathcal{I}^t_\uparrow([t \to k], k+1, k)$$

and by contraposition of Rule NotConcurrent steps t and $k+1$ are object-concurrent

$$ocon_\uparrow(t, k+1).$$

By Lemma 193 we obtain that the configurations at $k+1$ and k agree on the core registers

$$m^{k+1}_\uparrow =_{C_\uparrow(k+1)} m_\uparrow[t \to k]^k = m_\uparrow[t \to k+1]^k.$$

By Lemma 151, both steps are feasible

$$\Phi_\uparrow(k+1) = \Phi_\uparrow[t \to k+1](k) = 1,$$

which was the claim.

$s(k+1) \in \Sigma_{WB,i}$**:** Write buffer steps are always feasible

$$\Phi_\uparrow(k+1) \wedge \Phi_\uparrow[t \to k+1](k),$$

and it suffices to show that the steps satisfy the local drain condition

$$\begin{aligned}
&\Psi_\uparrow(k+1) \wedge \Psi_\uparrow[t \to k+1](k) \\
\iff\ &\Phi_\uparrow(k+1) \wedge \Lambda(k+1) \wedge \Phi_\uparrow[t \to k+1](k) \wedge \Lambda[t \to k+1](k) \\
\iff\ &1 \wedge \Delta_\uparrow(k+1) \wedge 1 \wedge \Delta_\uparrow[t \to k+1](k) \\
\iff\ &\Delta_{WB}(wb^{k+1}_\uparrow(i)) \wedge 1 \wedge \Delta_{WB}(wb_\uparrow[t \to k+1]^k(i)),
\end{aligned}$$

i.e., that the write buffers are not empty

$$\begin{aligned}
\iff\ &wb^{k+1}_\uparrow(i) \neq \varepsilon \wedge wb_\uparrow[t \to k+1]^k(i) \neq \varepsilon \\
\iff\ &wb^{k+1}_\uparrow(i) \neq \varepsilon \wedge wb_\uparrow[t \to k]^k(i) \neq \varepsilon.
\end{aligned}$$

The claim is now exactly Lemma 222.

□

With Invariant SameState we can easily show that if step t is a memory write, it can be moved to k and executed at its new position.

Lemma 224.

$$Inv(t,k) \wedge mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k) \rightarrow c^t_{\uparrow} =^{s(t)}_{\uparrow} c_{\uparrow}[t \rightarrow k]^k.$$

Proof. By Invariant SameState we obtain that that the configurations agree on the inputs

$$m^t_{\uparrow} =_{in_{\uparrow}(t)} m_{\uparrow}[t \rightarrow k]^k,$$

and that the buffers in step t subsume those in step k after the reordering

$$bufS_{\uparrow}(s(t), c^t_{\uparrow}, c_{\uparrow}[t \rightarrow k]^k).$$

The claim is now Lemma 187. □

If the schedule is pseudo-valid until $k+1$, we already know that step k is valid. We can thus immediately use Lemma 224.

Lemma 225.

$$pval(t,k+1) \wedge Inv(t,k) \wedge mwrite_{\uparrow}(t) \rightarrow c^t_{\uparrow} =^{s(t)}_{\uparrow} c_{\uparrow}[t \rightarrow k]^k.$$

Proof. Since the schedule is pseudo-valid until $k+1$ the schedule is valid at k in one of the schedules

$$\Gamma_{\uparrow}(k) \vee \Gamma_{\uparrow}[t \rightarrow k](k-1).$$

By Invariant Valid, the validity is the same, i.e., valid

$$\Gamma_{\uparrow}(k) = \Gamma_{\uparrow}[t \rightarrow k](k-1) = 1.$$

The claim is now Lemma 224. □

Only writes have write-read and write-write races, and thus we can obtain the same lemma for races.

Lemma 226. *When the schedule is pseudo-valid until $k+1$, and the invariants hold until k, and the steps t and $k+1$ have a write-read or write-write race*

$$pval(t,k+1) \wedge Inv(t,k) \wedge t \not\triangleright k+1 \wedge (WR_{\uparrow}(t,k+1) \vee WW_{\uparrow}(t,k+1))$$

then step t can be executed in its new position

$$c^t_{\uparrow} =^{s(t)}_{\uparrow} c_{\uparrow}[t \rightarrow k]^k.$$

Proof. By contraposition of Rule NotConcurrent we obtain that steps t and $k+1$ are object-concurrent

$$ocon_{\uparrow}(t,k+1).$$

By Lemma 160, step t must be a memory write

$$mwrite_{\downarrow}(t).$$

The claim is now Lemma 225. □

Lemma 227. *When the schedule is pseudo-valid, the invariants hold, and step t is not forward-synchronized with step $k+1$*

$$pval(t,k+1) \wedge Inv(t,k) \wedge t \not\rhd k+1,$$

the buffers of step $k+1$ at configuration $k+1$ before the reordering subsume those at configurations $k+1$ and k after the reordering

$$bufS(s(k+1), c_{\uparrow}^{k+1}, c_{\uparrow}[t \to k]^{k+1}).$$
$$\wedge\, bufS(s(k+1), c_{\uparrow}^{k+1}, c_{\uparrow}[t \to k]^{k}).$$

Proof. The proof distinguishes between steps by different units and steps by the same unit.

$u_{\uparrow}(t) \neq u_{\uparrow}(k+1) = i$**:** Note that the unit making step t also makes step k after moving t to k

$$u_{\uparrow}[t \to k](k) = u(s[t \to k](k)) = u(s(t)) = u_{\uparrow}(t) \neq i.$$

By Invariant NearlySame the buffers are the same

$$wb_{\uparrow}^{k+1}(i) = wb_{\uparrow}[t \to k]^{k}(i).$$

Since step k after moving t to k is also made by a different unit, by Lemma 95 its operation has no effect, and the buffers at k and $k+1$ stay the same

$$\begin{aligned} wb_{\uparrow}^{k+1}(i) &= wb_{\uparrow}[t \to k]^{k}(i) \\ &= Op_{i\uparrow}[t \to k](k)(wb_{\uparrow}[t \to k]^{k}(i), BW_{\uparrow}[t \to k]^{k}(i)) \qquad \text{L 95} \\ &= wb_{\uparrow}[t \to k]^{k+1}(i). \end{aligned}$$

The claims are now Lemma 90.

$u_{\uparrow}(t) = u_{\uparrow}(k+1) = i$**:** By Lemma 217 step t is made by the processor and step $k+1$ by the write buffer of the unit

$$s(t) \in \Sigma_{P,i} \wedge s(k+1) \in \Sigma_{WB,i}.$$

By Lemma 220 we obtain that the buffers are missing exactly step t

$$wb_{\uparrow}^{k+1}(i) = Op_{i\uparrow}(t)(wb_{\uparrow}[t \to k]^{k}(i), t).$$

By Lemma 222, the buffers at k are non-empty

$$wb_{\uparrow}[t \to k]^{k}(i) \neq \varepsilon.$$

Since step t is a processor step, the operation is either *push* or *noop*, depending on whether the step is buffering a write or not

$$Op_{i\uparrow}(t) \in \{\, push, noop \,\},$$

and by Lemma 43 the head of the non-empty write buffer is unchanged

$$hd(wb_{\uparrow}^{k+1}(i)) = hd(Op_{i\uparrow}(t)(wb_{\uparrow}[t \to k]^{k}(i), BW_{\uparrow}(t)))$$

$$= hd(wb_\uparrow[t \to k]^k(i)).$$

Step k after moving t to k is also a processor step of unit i

$$s[t \to k](k) = s(t) \in \Sigma_{P,i}$$

and thus the operation is still a push or a noop

$$Op_{i\uparrow}[t \to k](k) \in \{ push, noop \}.$$

We conclude again with Lemma 43 that the head of the non-empty write buffer is unchanged by this operation

$$\begin{aligned} hd(wb_\uparrow^{k+1}(i)) &= hd(Op_{i\uparrow}[t \to k](k)(wb_\uparrow[t \to k]^k(i), BW_\uparrow[t \to k](k))) \\ &= hd(wb_\uparrow[t \to k]^{k+1}(i)) \end{aligned}$$

We conclude that the head of the write buffer at $k+1$ before the reordering is the head of the write buffer at k and $k+1$ after the reordering

$$\begin{aligned} hd(wb_\uparrow^{k+1}(i)) &= hd(wb_\uparrow[t \to k]^k(i)). \\ hd(wb_\uparrow^{k+1}(i)) &= hd(wb_\uparrow[t \to k]^{k+1}(i)). \end{aligned}$$

By definition, the buffer at $k+1$ subsumes those at k and $k+1$ after the reordering

$$\begin{aligned} &bufS_\uparrow(s(k+1), c_\uparrow^{k+1}, c_\uparrow[t \to k]^k) \\ \wedge\, &bufS_\uparrow(s(k+1), c_\uparrow^{k+1}, c_\uparrow[t \to k]^{k+1}), \end{aligned}$$

which was the claim.

□

Lemma 228. *When the schedule is pseudo-valid until $k+1$, and the invariants hold until k, and the steps t and $k+1$ are not forward-synchronized*

$$pval(t, k+1) \wedge Inv(t,k) \wedge t \not\triangleright k+1,$$

then all of the following are true.

1. *If there is a code modification, the steps are annotated correctly*

$$CM_\uparrow(t, k+1) \to Sh_\uparrow(t) \wedge ShR_\uparrow(k+1).$$

2. *If there is a valid write-read race, the steps are annotated correctly*

$$WR_\uparrow(t, k+1) \wedge \Gamma_\uparrow(k+1) \to Sh_\uparrow(t) \wedge ShR_\uparrow(k+1).$$

3. *If there is a valid write-write race, the steps are annotated correctly*

$$WW_\uparrow(t, k+1) \wedge \Gamma_\uparrow(k+1) \to Sh_\uparrow(t) \wedge Sh_\uparrow(k+1).$$

Proof. We apply Lemma 209 with $O := [t \to k]$, $t := t$, $k := k+1$, and $t' := k$ which reduces the goal to the following subclaims.

$ocon_\uparrow(t,k+1)$**:** Is the contraposition of Rule NOTCONCURRENT.

$s(t) = s[t \to k](k)$**:** Obviously true.

$s(k+1) = s[t \to k](k+1)$**:** Obviously true.

$\Gamma_\uparrow^{k-1}(s[t \to k])$**:** Follows trivially with Lemma 223.

$\Gamma_\uparrow(t)$**:** Follows trivially from the definition of *pval*.

$c_\uparrow^t =_\uparrow^{s(t)} c_\uparrow[t \to k]^k$**:** The claim is Lemma 226.

$m_\uparrow[t \to k]^k =_{in_\uparrow(k+1) \setminus out_\uparrow(t)} m_\uparrow^{k+1}$**:** Clearly the outputs do not intersect with the inputs minus the outputs

$$out_\uparrow(t) \not\parallel in_\uparrow(k+1) \setminus out_\uparrow(t).$$

By Invariant NearlySame the configurations at k and $k+1$ are nearly the same

$$\mathcal{I}_\uparrow^t([t \to k], k+1, k)$$

and the claim is Lemma 192.

$bufS(s[t \to k](k+1), c_\uparrow^{k+1}, c_\uparrow[t \to k]^{k+1})$**:** Since step $k+1$ is after the reordering, its oracle input is unaffected

$$s(k+1) = s[t \to k](k+1).$$

This reduces the claim to showing the following

$$bufS(s(k+1), c_\uparrow^{k+1}, c_\uparrow[t \to k]^{k+1}),$$

which is just Lemma 227.

□

We also obtain that of step $k+1$ is unchanged when executed at k after moving t to $k+1$.

Lemma 229. *Assume that t is delayable until $k+1$ and the invariants hold at k*

$$pval(t,k+1) \wedge Inv(t,k) \wedge t \not> k+1,$$

then if there is no visible write-read race

$$\neg VR_\uparrow(t,k+1),$$

the configurations at $k+1$ and k, before and after moving t to $k+1$, respectively, strongly agree when stepped with $s(k+1)$

$$c_\uparrow^{k+1} =_\uparrow^{s(k+1)} c_\uparrow[t \to k+1]^k.$$

Proof. Note first that the configuration at k after moving t to $k+1$ is the same as that at k after moving t to k, since the order of steps after k does not affect the configuration at k

$$c_\uparrow[t \to k+1]^k = c_\uparrow[t \to k]^k$$

and it suffices to show the following

$$c_\uparrow^{k+1} =_\uparrow^{s(k+1)} c_\uparrow[t \to k]^k.$$

We distinguish now with Lemma 217 between two cases: either the steps are made by different units, or step $k+1$ is a write buffer step.

$u_\uparrow(t) \neq u_\uparrow(k+1)$**:** We obtain that the steps are made by different units

$$diffu(t,k+1)$$

and by Invariant NearlySame that the configurations are nearly the same

$$\mathcal{I}^t_\uparrow([t \to k],k+1,k).$$

The claim is now Lemma 196.

$s(k+1) \in \Sigma_{WB,u_\uparrow(t)}$**:** We apply Lemma 186 and reduce the goal to the following two subclaims.

$m^{k+1}_\uparrow =_{in_\uparrow(k+1)} m_\uparrow[t \to k]^k$**:** By Invariant NearlySame the configurations are nearly the same

$$\mathcal{I}^t_\uparrow([t \to k],k+1,k)$$

and the claim is just Lemma 195.

$bufS_\uparrow(s(k+1),c^{k+1}_\uparrow,c_\uparrow[t \to k]^k)$**:** The claim is just Lemma 227.

□

Lemma 230.

$$pval(t,k+1) \wedge t \not\trianglerighteq k+1 \wedge Inv(t,k) \to \neg VR_\uparrow(t,k+1).$$

Proof. Assume for the sake of contradiction that the visible outputs are used by step $k+1$

$$vout_\uparrow(t,k+1) \owedge in_\uparrow(k+1).$$

With Lemma 113 we obtain that the outputs subsume those addresses and thus must also intersect the inputs

$$out_\uparrow(t) \owedge in_\uparrow(k+1).$$

Therefore there is a write-read race

$$WR_\uparrow(t,k+1).$$

Step t is not race-synchronized and thus also not forward-synchronized with step $k+1$

$$t \not\triangleright k+1.$$

With Lemma 228 we obtain that both steps are shared

$$Sh_\uparrow(t) \wedge ShR_\uparrow(k+1),$$

and by Rule CommExt the steps are race-synchronized

$$t \trianglerighteq k+1,$$

which is is a contradiction. □

Lemma 231. *Assume that the schedule is pseudo-valid until $k+1$, t is not race-synchronized with $k+1$, and the invariants hold at k*

$$pval(t,k+1) \wedge t \not\approx k+1 \wedge Inv(t,k),$$

then the configurations at $k+1$ and k, before and after moving t to $k+1$, respectively, strongly agree when stepped with $s(k+1)$

$$c_{\uparrow}^{k+1} =_{\uparrow}^{s(k+1)} c_{\uparrow}[t \to k+1]^k.$$

Proof. By Lemma 230, there is no visible write-read race

$$\neg VR_{\uparrow}(t,k+1).$$

The claim is now Lemma 229. □

Lemma 232. *Assume that the schedule is pseudo-valid until $k+1$, t is not race-synchronized with $k+1$, and the invariants hold at k*

$$pval(t,k+1) \wedge t \not\approx k+1 \wedge Inv(t,k),$$

then all functions X that depend on core, fetched and read addresses, or the write buffer, agree between step $k+1$ before the reordering and k after t was moved to $k+1$, i.e., for

$$X \in \{ PW, W, WS, out, victims, \Delta, \Gamma \}$$

and for

$$X \in \{ R, \Upsilon, I, in \}$$

and for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_{\uparrow}(k+1) = X_{\uparrow}[t \to k+1](k).$$

Proof. With Lemma 231 we obtain that the configurations during the step agree

$$c_{\uparrow}^{k+1} =_{\uparrow}^{s(k+1)} c_{\uparrow}[t \to k+1]^k.$$

Furthermore, the steps made in the configuration are the same

$$s[t \to k+1](k) = s[t \to k][k \leftrightarrow k+1](k) = s[t \to k](k+1) = s(k+1).$$

The claims are now Lemma 150. □

Lemma 233.

$$pval(t,k+1) \wedge t \not\approx k+1 \wedge Inv(t,k) \to NearlySame(t,k+1).$$

Proof. By Invariant NearlySame, the configurations are nearly the same at $k+1$ and k

$$\mathcal{I}_{\uparrow}^{t}([t \to k], k+1, k).$$

We can move step t one further without affecting the configuration at k

$$c_{\uparrow}[t \to k]^k = c_{\uparrow}[t \to k+1]^k.$$

Thus the configurations are still nearly the same when moving t until $k+1$

$$\begin{aligned}
&\mathcal{I}^t_\uparrow([t \to k], k+1, k)\\
\iff & m^{k+1}_\uparrow =_{\mathcal{A}\setminus vout_\uparrow(t,k+1)} m_\uparrow[t \to k]^k \wedge \forall i \neq u_\uparrow(t).wb^{k+1}_\uparrow =_i wb_\uparrow[t \to k]^k\\
\iff & m^{k+1}_\uparrow =_{\mathcal{A}\setminus vout_\uparrow(t,k+1)} m_\uparrow[t \to k+1]^k \wedge \forall i \neq u_\uparrow(t).wb^{k+1}_\uparrow =_i wb_\uparrow[t \to k+1]^k\\
\iff & \mathcal{I}^t_\uparrow([t \to k+1], k+1, k).
\end{aligned}$$

We now distinguish whether the steps are made by the same unit or not.

$u_\uparrow(t) \neq u_\uparrow(k+1)$**:** In this case the steps are made by different units

$$diffu(t, k+1)$$

and by Lemma 230 there is no visible write-read race

$$\neg VR_\uparrow(t, k+1).$$

The claim is thus Lemma 198.

$u_\uparrow(t) = u_\uparrow(k+1) = i$**:** Clearly the steps at $k+1$ and k in the schedules use the same oracle input

$$s[t \to k+1](k) = s(k+1).$$

By Lemma 231 the steps strongly agree

$$c^{k+1}_\uparrow =^{s(k+1)}_\uparrow c_\uparrow[t \to k+1]^k$$

and the claim is Lemma 197.

□

Lemma 234.

$$pval(t, k+1) \wedge Inv(t,k) \wedge t \not> k+1 \to IssueSame(t, k+1).$$

Proof. With Invariant IssueSame and Lemma 204 with $k' := k+1$ we obtain that the sequences of issued writes are nearly the same at $k+1$ and k when reordering until $k+1$

$$\mathcal{J}^{f_i}_\uparrow([t \to k+1], k+1, k).$$

We apply Lemma 202, which reduces the claim to the following three.

$mv[t \to k+1](k+1) = k$**:** This is just a special case of Lemma 77.

Commutativity: The claim is that the operations at t and $k+1$ can be commuted

$$\begin{aligned}
& f_i(Op_{i\uparrow}(k+1)(issue_\uparrow[t \to k+1]^k, k+1))\\
=\ & Op_{i\uparrow}(t)(Op_{i\uparrow}(k+1)(issue_\uparrow[t \to k+1]^k, k+1), t)\\
\overset{!}{=}\ & Op_{i\uparrow}(k+1)(Op_{i\uparrow}(t)(issue_\uparrow[t \to k+1]^k, t), k+1)\\
=\ & Op_{i\uparrow}(k+1)(f_i(issue_\uparrow[t \to k+1]^k), k+1).
\end{aligned}$$

We distinguish whether the steps are made by the same unit or not.

$u_\uparrow(t) = u_\uparrow(k+1) = i$**:** By Lemma 217 step t is made by the processor and step $k+1$ by the write buffer

$$s(t) \in \Sigma_{P,i} \wedge s(k+1) \in \Sigma_{WB,i}.$$

By Lemma 222 the write buffer at k is non-empty or step t did not push something

$$Op_{i\uparrow}(t) \neq push \vee issue_\uparrow[t \to k+1]^k(i) = issue_\uparrow[t \to k]^k(i) \neq \varepsilon,$$

whereas the operation at $k+1$ is certainly a pop

$$Op_{i\uparrow}(k+1) = pop.$$

The claim is now Lemma 46 with $\delta := Op_{i\uparrow}(t)$ and $\delta' := Op_{i\uparrow}(k+1)$

$$\begin{aligned} &Op_{i\uparrow}(t)(Op_{i\uparrow}(k+1)(issue_\uparrow[t \to k+1]^k, k+1), t) \\ &= Op_{i\uparrow}(k+1)(Op_{i\uparrow}(t)(issue_\uparrow[t \to k+1]^k, t), k+1). \end{aligned}$$

$u_\uparrow(t) \neq u_\uparrow(k+1)$**:** The claim is just Lemma 100

$$\begin{aligned} &Op_{i\uparrow}(t)(Op_{i\uparrow}(k+1)(issue_\uparrow[t \to k+1]^k, k+1), t) \\ &= Op_{i\uparrow}(k+1)(Op_{i\uparrow}(t)(issue_\uparrow[t \to k+1]^k, t), k+1). \end{aligned}$$

$c_\uparrow^{k+1} =_\uparrow^{s(k+1)} c_\uparrow[t \to k+1]^k$**:** This is just Lemma 231.

□

Lemma 235.

$$pval(t, k+1) \wedge Inv(t,k) \wedge t \not\rhd k+1 \to NothingIssued(t, k+1).$$

Proof. By Invariant NothingIssued no new writes have been issued at k when moving until k

$$\forall t' \in issue_\uparrow[t \to k]^k(u_\uparrow(t)).t' < t.$$

The sequence at k is unaffected by moving one step further, and we obtain the same when moving until $k+1$

$$\forall t' \in issue_\uparrow[t \to k+1]^k(u_\uparrow(t)).t' < t.$$

We want to show that this still holds at $k+1$, which is obtained by applying the operator of step k in the reordered schedule

$$issue_\uparrow[t \to k+1]^{k+1}(u_\uparrow(t)) = Op_{u_\uparrow(t)\uparrow}[t \to k+1](k)(issue_\uparrow[t \to k+1]^k(u_\uparrow(t)), k).$$

The proof now distinguishes between the three possible operators at k

$Op_{u_\uparrow(t)\uparrow}[t \to k+1](k) = push$**:** By definition of Op, step k in the reordering and thus step $k+1$ in the original schedule is a processor step of the unit that makes step t

$$s(k+1) = s[t \to k+1](k) \in \Sigma_{P,u_\uparrow(t)}$$

consequently both steps are made by the same unit

$$u_\uparrow(k+1) = u_\uparrow(t).$$

But by Lemma 217, step $k+1$ is made by a write buffer and thus not by a processor

$$s(k+1) \in \Sigma_{WB,u_\uparrow(t)} \wedge s(k+1) \notin \Sigma_{P,u_\uparrow(t)}.$$

$Op_{u_\uparrow(t)\uparrow}[t \to k+1](k) = pop$**:** The step only drops elements

$$issue_\uparrow[t \to k+1]^{k+1}(u_\uparrow(t)) = tl(issue_\uparrow[t \to k+1]^{k}(u_\uparrow(t)))$$

and thus all elements of the sequence at $k+1$ were already in the sequence at k. The claim follows

$$\forall t' \in issue_\uparrow[t \to k+1]^{k+1}(u_\uparrow(t)).t' < t.$$

$Op_{u_\uparrow(t)\uparrow}[t \to k+1](k) = noop$**:** The step does not change the sequence of issued writes

$$issue_\uparrow[t \to k+1]^{k+1}(u_\uparrow(t)) = issue_\uparrow[t \to k+1]^{k}(u_\uparrow(t))$$

and thus all elements of the sequence at $k+1$ were already in the sequence at k. The claim follows

$$\forall t' \in issue_\uparrow[t \to k+1]^{k+1}(u_\uparrow(t)).t' < t.$$

□

Lemma 236.

$$pval(t,k+1) \wedge t \not\Rightarrow k+1 \wedge Inv(t,k) \to Valid(t,k+1).$$

Proof. The first portion of the invariant is just Lemma 223. The second portion of the invariant is just Lemma 232 with $X := \Gamma$. □

To prove Invariant SameState, we also need that step t is not interrupted or a disabled RMW (i.e., we need $t \not\blacktriangleright k+1$). This is conveniently expressed by $delay(t,k+1)$. To go back to the form used before, we prove a small lemma: when step t can be delayed until $k+1$, the schedule is pseudo-valid, and t is neither race- nor forward-synchronized with $k+1$.

Lemma 237.

$$delay(t,k+1) \to pval(t,k+1) \wedge t \not\Rightarrow k+1 \wedge t \not\triangleright k+1.$$

Proof. The first claim is by definition of *delay*. By definition of *delay*, we obtain further that t is not synchronized with $k+1$

$$t \not\blacktriangleright k+1.$$

By contraposition of Lemma 212 we immediately obtain that there is no forward-synchronization

$$t \not\triangleright k+1,$$

and by contraposition of Lemma 213 that there is no race-synchronization

$$t \not\Rightarrow k+1,$$

and the claim follows. □

Lemma 238. *Assume that t can be delayed until $k+1$ and the invariants hold at k*

$$delay(t,k+1) \wedge Inv(t,k).$$

If step t is a memory write and step $k+1$ is a valid step of the same unit

$$mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k+1) \wedge u_{\uparrow}(t) = u_{\uparrow}(k+1),$$

the steps and step k after moving t to $k+1$ are all using the same memory mode

$$SC_{\uparrow}(t) = SC_{\uparrow}[t \to k+1](k) = SC_{\uparrow}(k+1).$$

Proof. By Lemma 217, step $k+1$ must be a write buffer step and step t must be a processor step

$$s(t) \in \Sigma_{P,i} \wedge s(k+1) \in \Sigma_{WB,i}.$$

The mode registers are inputs of step t

$$A_{SC,i} \subseteq in_{\uparrow}(t).$$

Due to Invariant SameState, the mode registers are unchanged by the reordering

$$m^t_{\uparrow} =_{A_{SC,i}} m_{\uparrow}[t \to k]^k.$$

Reordering further does not change the configuration and thus the mode registers at k

$$m_{\uparrow}[t \to k]^k =_{A_{SC,i}} m_{\uparrow}[t \to k+1]^k,$$

and since the unit making step k after moving t to $k+1$ is again unit i

$$u_{\uparrow}[t \to k+1](k) = u(s[t \to k+1](k)) = u(s(k+1)) = u_{\uparrow}(k+1) = i,$$

step k after moving t to $k+1$ is also done in the same memory mode

$$\begin{aligned} SC_{\uparrow}[t \to k+1](k) &= SC_{i\uparrow}[t \to k+1](k) \\ &= SC_{i\uparrow}(t) \\ &= SC_{\uparrow}(t). \end{aligned}$$

With Lemma 232 with $X := SC$ and Lemma 237 we obtain that the step was in the same memory mode in the original schedule

$$SC_{\uparrow}(k+1) = SC_{\uparrow}[t \to k+1](k) = SC_{\uparrow}(t),$$

which is the claim. □

Lemma 239. *Assume that t can be delayed until $k+1$ and the invariants hold at k*

$$delay(t,k+1) \wedge Inv(t,k).$$

If step t is a memory write and step $k+1$ is a valid step of the same unit, there is no read-write race between the two steps

$$mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k+1) \wedge u_{\uparrow}(t) = u_{\uparrow}(k+1) \to \neg RW_{\uparrow}(t,k+1).$$

Proof. Assume for the sake of contradiction that there is such a race

$$RW_{\uparrow}(t,k+1).$$

By contraposition of Rule RMWDISABLE, we obtain that step t is not a shared read

$$\neg ShR_{\uparrow}(t).$$

By Lemmas 217 and 237, step $k+1$ must be a write buffer step and step t must be a processor step

$$s(t) \in \Sigma_{P,i} \wedge s(k+1) \in \Sigma_{WB,i}.$$

By definition of *ShR*, a processor step that is not a shared read is performed in sequentially consistent mode

$$SC_{\uparrow}(t).$$

By Lemma 238 step $k+1$ is made in the same memory mode

$$SC_{\uparrow}(k+1)$$

and thus by definition the write at $k+1$ before the reordering is empty

$$W_{\uparrow}(k+1) = \emptyset.$$

Thus step $k+1$ does not have outputs

$$out_{\uparrow}(k+1) = \emptyset,$$

which contradicts the assumption that its outputs somehow modify inputs of step t.

$$in_{\uparrow}(t) \not\parallel out_{\uparrow}(k+1).$$

□

Lemma 240. *Assume that t can be delayed until $k+1$ and the invariants hold at k*

$$delay(t,k+1) \wedge Inv(t,k).$$

If step t is a memory write and step $k+1$ is valid, the buffers of step t subsume those of step $k+1$ after moving t to $k+1$

$$mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k+1) \rightarrow bufS_{\uparrow}(s(t), c^t_{\uparrow}, c_{\uparrow}[t \rightarrow k+1]^{k+1}).$$

Proof. We have by Invariant SameState that the buffers of step t subsume those of step k after moving t to k

$$bufS_{\uparrow}(s(t), c^t_{\uparrow}, c_{\uparrow}[t \rightarrow k]^k).$$

We move t on to $k+1$, which does not affect the configuration at k

$$c_{\uparrow}[t \rightarrow k+1]^k = c_{\uparrow}[t \rightarrow k]^k,$$

and obtain that the buffers of step t subsume those of step k after moving t to $k+1$

$$bufS_{\uparrow}(s(t), c^t_{\uparrow}, c_{\uparrow}[t \rightarrow k+1]^k).$$

We unfold notation in the assumption

$$bufS(s(t), fin_\uparrow(t), c^t_\uparrow.wb(u_\uparrow(t)), c_\uparrow[t \to k+1]^k.wb(u_\uparrow(t)))$$

and in the claim

$$bufS(s(t), fin_\uparrow(t), c^t_\uparrow.wb(u_\uparrow(t)), c_\uparrow[t \to k+1]^{k+1}.wb(u_\uparrow(t)))$$

and apply transitivity (Lemma 88), which reduces the claim to showing that the buffers at k after the reordering subsume those at $k+1$ after the reordering when stepped with $s(t)$ and using the forwarding inputs of step t

$$bufS(s(t), fin_\uparrow(t), c_\uparrow[t \to k+1]^k.wb(u_\uparrow(t)), c_\uparrow[t \to k+1]^{k+1}.wb(u_\uparrow(t))).$$

We distinguish whether steps t and $k+1$ were made by the same unit, or by different units.

$u_\uparrow(t) = u_\uparrow(k+1) = i$**:** By Lemmas 217 and 237 we obtain that the step were made by a processor and its write buffer

$$s(t) \in \Sigma_{P,i} \wedge s(k+1) \in \Sigma_{WB,i}.$$

Therefore we have to show that there is a prefix w that is missing and not used by the forwarding inputs

$$wb_\uparrow[t \to k+1]^k(i) = w \circ wb_\uparrow[t \to k+1]^{k+1}(i) \overset{!}{\wedge} \neg hit(fin_\uparrow(t), w);$$

that prefix is the head of the write buffer, which is committed during the write buffer step

$$w := hd(wb_\uparrow[t \to k+1]^k(i)).$$

The first claim holds now by definition

$$\begin{aligned} wb_\uparrow[t \to k+1]^k(i) &= hd(wb_\uparrow[t \to k+1]^k(i)) \circ tl(wb_\uparrow[t \to k+1]^k(i)) \\ &= w \circ wb_\uparrow[t \to k+1]^{k+1}(i). \end{aligned}$$

For the second claim, we distinguish between weak and strong memory mode.

$SC_\uparrow(t)$**:** By definition the machine uses high-level semantics and the forwarding inputs are empty

$$LL_\uparrow(t) = 0 \wedge fin_\uparrow(t) = \emptyset,$$

and the claim trivially holds.

$\neg SC_\uparrow(t)$**:** By Lemma 238, step k after moving t to $k+1$ is also in weak memory mode

$$\neg SC_\uparrow[t \to k+1](k).$$

Since step $k+1$ is a write buffer step, so is step k after moving t to $k+1$

$$s[t \to k+1](k) = s(k+1) \in \Sigma_{WB,i}$$

and thus the write executed in step k after moving t to $k+1$ is the head of the write buffer

$$W_\uparrow[t \to k+1](k) = hd(wb_\uparrow[t \to k+1]^k(i)).$$

The second claim is thus equivalent to showing that the write-set of that step does not intersect with the inputs

$$\begin{aligned}\neg hit(fin_\uparrow(t),w) &\equiv \neg hit(in_\uparrow(t), hd(wb_\uparrow[t \to k+1]^k(i))) \\ &\equiv in_\uparrow(t) \not\parallel Dom(hd(wb_\uparrow[t \to k+1]^k(i))) \\ &\equiv in_\uparrow(t) \not\parallel Dom(W_\uparrow[t \to k+1](k)) \\ &\equiv in_\uparrow(t) \not\parallel WS_\uparrow[t \to k+1](k).\end{aligned}$$

By Lemmas 232 and 237, that is the write-set of step $k+1$ before the re-ordering

$$WS_\uparrow[t \to k+1](k) = WS_\uparrow(k+1).$$

By Lemma 239 there is no read-write race between these steps, and thus no intersection between inputs and outputs

$$in_\uparrow(t) \not\parallel out_\uparrow(k+1),$$

and the claim follows with the observation that the outputs subsume the write-set

$$\begin{aligned}WS_\uparrow(k+1) &\subseteq WS_\uparrow(k+1) \cup dc(WS_\uparrow(k+1)) \\ &= idc(WS_\uparrow(k+1)) \\ &= out_\uparrow(k+1).\end{aligned}$$

$u_\uparrow(t) \neq u_\uparrow(k+1)$**:** The unit making step k after moving t to $k+1$ is also not the same as the one making step t

$$u_\uparrow[t \to k+1](k) = u(s[t \to k+1](k)) = u(s(k+1)) = u_\uparrow(k+1) \neq u_\uparrow(t).$$

By Lemma 96, step k after moving t to k has no effect on the write buffer

$$c_\uparrow[t \to k+1]^k.wb(u_\uparrow(t)) = c_\uparrow[t \to k+1]^{k+1}.wb(u_\uparrow(t))$$

and the claim is Lemma 89.

□

Lemma 241. *Assume step t can be delayed until $k+1$ and the invariants hold at k*

$$delay(t,k+1) \wedge Inv(t,k).$$

If step t is a memory write and step $k+1$ is a valid step of another unit, there is no read-write race between the two steps

$$mwrite_\uparrow(t) \wedge \Gamma_\uparrow(k+1) \wedge u_\uparrow(t) \neq u_\uparrow(k+1) \to \neg RW_\uparrow(t,k+1).$$

Proof. Assume for the sake of contradiction that there is such a race

$$RW_\uparrow(t,k+1).$$

By contraposition of Rule INTERRUPTED we obtain that step t is not interrupted by step $k+1$

$$\neg int_\uparrow(k+1,t).$$

By assumption the steps are made by different units

$$diffu(t,k).$$

Therefore step $k+1$ is by definition of *ucon* unit-concurrent with t

$$ucon_{\uparrow}(k+1,t),$$

and by Lemma 105 also object-concurrent

$$ocon_{\uparrow}(k+1,t) \tag{4.10}$$

Clearly step $k+1$ and t have been moved to k and $k+1$, respectively

$$s(k+1) = s[t \to k+1](k),\ s(t) = s[t \to k+1](k+1). \tag{4.11}$$

With Lemmas 231 and 237 step $k+1$ can be executed at its new position

$$c_{\uparrow}^{k+1} =_{\uparrow}^{s(k+1)} c_{\uparrow}[t \to k+1]^k. \tag{4.12}$$

By Lemmas 223 and 237 the new schedule is valid until $k-1$

$$\Gamma_{\uparrow}^{k-1}(s[t \to k+1]), \tag{4.13}$$

and the old schedule is valid until k

$$\Gamma_{\uparrow}^{k}(s).$$

In particular step t is valid

$$\Gamma_{\uparrow}(t) \tag{4.14}$$

and step k is valid

$$\Gamma_{\uparrow}(k)$$

and by Invariant SameState we obtain that the value of inputs is unchanged

$$m_{\uparrow}[t \to k]^k =_{in_{\uparrow}(t)} m_{\uparrow}^t,$$

in particular for those inputs which are not outputs of $k+1$

$$m_{\uparrow}[t \to k]^k =_{in_{\uparrow}(t) \setminus out_{\uparrow}(k+1)} m_{\uparrow}^t.$$

Since that memory configuration at k is not affected by moving t one more step

$$m_{\uparrow}[t \to k+1]^k = m_{\uparrow}[t \to k][k \leftrightarrow k+1]^k = m_{\uparrow}[t \to k]^k,$$

we obtain that the memory configuration at k after the reordering equals that at t before the reordering

$$m_{\uparrow}[t \to k+1]^k =_{in_{\uparrow}(t) \setminus out_{\uparrow}(k+1)} m_{\uparrow}^t. \tag{4.15}$$

By Lemma 240, the buffers of step t subsume those of step $k+1$ after moving t to $k+1$

$$bufS_{\uparrow}(s(t), c_{\uparrow}^t, c_{\uparrow}[t \to k+1]^{k+1}).$$

Rewriting with the equality $s(t) = s[t \to k+1](k+1)$ we obtain the following

$$bufS_{\uparrow}(s[t \to k+1](k+1), c_{\uparrow}^t, c_{\uparrow}[t \to k+1]^{k+1}). \tag{4.16}$$

Note that a read-write race is a write-read race with the step indices swapped, and so there is a write-read race between $k+1$ and t

$$WR_{\uparrow}(k+1,t).$$

We apply Lemma 209 with $O := [t \to k+1]$, $t := k+1$, $k := t$, $t' := k$, the assumption that step $k+1$ is valid and Eqs. (4.10) to (4.16), and obtain that step t is a shared read

$$ShR_{\uparrow}(t).$$

By Rule RMWDISABLE, the steps are synchronized

$$t \blacktriangleright k+1,$$

which is a contradiction. □

Lemma 242.

$$delay(t,k+1) \wedge Inv(t,k) \wedge mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k+1) \to \neg RW_{\uparrow}(t,k+1).$$

Proof. By case distinction on whether the units are the same.

$u_{\uparrow}(t) = u_{\uparrow}(k+1)$**:** The claim is just Lemma 239.

$u_{\uparrow}(t) \neq u_{\uparrow}(k+1)$**:** The claim is just Lemma 241.

□

Lemma 243.

$$delay(t,k+1) \wedge Inv(t,k) \to SameState(t,k+1).$$

Proof. Let step t be a memory write and step $k+1$ be valid

$$mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k+1).$$

We show the two claims.

$m_{\uparrow}[t \to k+1]^{k+1} =_{in_{\uparrow}(t)} m^t_{\uparrow}$**:** By definition of *delay*, the schedule is pseudo-valid and by Lemma 223 valid until k

$$\Gamma^k_{\uparrow}(s),$$

in particular at k

$$\Gamma_{\uparrow}(k).$$

By Invariant SameState we have that the memory at k when moving until k is the same, which is obviously the same as that at k when moving until $k+1$

$$m_{\uparrow}[t \to k+1]^k = m_{\uparrow}[t \to k]^k =_{in_{\uparrow}(t)} m^t_{\uparrow}.$$

With Lemma 242 we obtain that there is no read-write race

$$\neg RW_{\uparrow}(t,k+1),$$

and thus the outputs of step $k+1$ are not intersecting with the inputs of t

$$out_{\uparrow}(k+1) \not\parallel in_{\uparrow}(t).$$

The outputs at the new position of k are the same by Lemmas 232 and 237

$$out_\uparrow(k+1) = out_\uparrow[t \to k+1](k),$$

and we obtain that these do not intersect the inputs either

$$out_\uparrow[t \to k+1](k) \not\parallel in_\uparrow(t).$$

The claim follows with Lemma 138

$$m_\uparrow[t \to k+1]^k =_{in_\uparrow(t)} m_\uparrow[t \to k+1]^k =_{in_\uparrow(t)} m_\uparrow^t.$$

$bufS_\uparrow(s(t), c_\uparrow^t, c_\uparrow[t \to k+1]^{k+1})$**:** The claim is just Lemma 240.

□

Furthermore, if a step can be delayed until $k+1$ it can also be delayed until k

Lemma 244.

$$pval(t, k+1) \to pval(t, k).$$

Proof. The proof is completely trivial except for showing that the last step is WB-feasible in one schedule

$$\Psi_\uparrow(k) \overset{!}{\vee} \Psi_\uparrow[t \to k](k-1).$$

However we have by assumption that that step $k \in (t : k+1)$ is valid in one schedule

$$\Gamma_\uparrow(k) \vee \Gamma_\uparrow[t \to k](k-1),$$

and by unfolding definitions we obtain that it is feasible and satisfies the drain condition

$$\Phi_\uparrow(k) \wedge \Delta_\uparrow(k) \vee \Phi_\uparrow[t \to k](k-1) \wedge \Delta_\uparrow[t \to k](k-1)$$

and thus by definition of Λ also the write buffer drain condition

$$\Phi_\uparrow(k) \wedge \Lambda_\uparrow(k) \vee \Phi_\uparrow[t \to k](k-1) \wedge \Lambda_\uparrow[t \to k](k-1)$$

which by definition of Ψ is the claim. □

Lemma 245.

$$delay(t, k+1) \to delay(t, k)$$

Proof. By assumption the schedule is pseudo-valid until $k+1$ and thus by Lemma 244 also until k

$$pval(t, k).$$

By assumption none of the steps t to $k+1$ are synchronized, so clearly also not the ones until k

$$\forall t' \in (t : k].t \not\blacktriangleright t'.$$

The claim follows. □

Combining all of these lemma gives us

Lemma 246.

$$delay(t,k) \rightarrow Inv(t,k).$$

Proof. Induction on k, starting with t. In the base case we apply Lemma 214. In the inductive step from k to $k+1$, we first apply Lemma 245 and obtain that t can be delayed until k

$$delay(t,k).$$

We use the induction hypothesis and obtain the invariants at k

$$Inv(t,k).$$

The inductive claim now follows by simply applying the right lemma from above for each invariant (Lemmas 233, 234, 236 and 243) □

Often we need to argue about steps $k' \leq k$ when t can be delayed until k. By repeatedly applying Lemmas 244 and 245 we can use lemmas from above, including Lemmas 232 and 246, to argue about those steps.

Lemma 247. *When step t is not race-synchronized with k and can be delayed until $k-1$*

$$t \not\Rightarrow k \wedge delay(t,k-1),$$

then step t is not race-synchronized with any $k' \in (t:k]$

$$t \not\Rightarrow k'.$$

Proof. By case distinction on $k \leq k'$.

$k = k'$**:** In this case we have the claim as an assumption.

$k < k'$**:** In this case we have by definition of $delay$ that step t is not synchronized with k'

$$t \not\blacktriangleright k'$$

and the claim is the contraposition of Lemma 213.

□

Lemma 248. *When a schedule is pseudo-valid until k, step t is not race-synchronized with k and can be delayed until $k-1$*

$$pval(t,k) \wedge t \not\Rightarrow k \wedge delay(t,k-1),$$

then for all $k' \in (t:k]$, the schedule is pseudo-valid until k', step t is not race-synchronized with k', and the invariants hold at $k'-1$

$$pval(t,k') \wedge t \not\Rightarrow k' \wedge Inv(t,k'-1).$$

Proof. By repeated application of Lemma 245 we obtain that t can be delayed until $k' - 1 \leq k - 1$

$$delay(t, k' - 1)$$

and thus by Lemma 246 the invariants hold at $k' - 1$

$$Inv(t, k' - 1).$$

By repeated application of Lemma 244 we obtain that the schedule is pseudo-valid until $k' \leq k$

$$pval(t, k').$$

By Lemma 247 we obtain that t is not race-synchronized with k'

$$t \not\Rrightarrow k'$$

which completes the proof. □

Lemma 249. *When a schedule is pseudo-valid until k, step t is not race-synchronized with k and can be delayed until* $k - 1$

$$pval(t, k) \wedge t \not\Rrightarrow k \wedge delay(t, k - 1),$$

then for all $k' \in (t : k]$*, the schedule is pseudo-valid until* k'*, step t is not forward-synchronized with* k'*, and the invariants hold at* $k' - 1$

$$pval(t, k') \wedge t \not\triangleright k' \wedge Inv(t, k' - 1).$$

Proof. Follows directly with Lemma 248 and the definition of $\Rrightarrow$. □

Lemma 250. *When a schedule is pseudo-valid until k, step t is not race-synchronized with k and can be delayed until* $k - 1$

$$pval(t, k) \wedge t \not\Rrightarrow k \wedge delay(t, k - 1),$$

then all functions X that depend on core, fetched and read addresses, or the write buffer, agree between step k' *before the reordering and* $k' - 1$ *after t was moved to k for all* $k' \in (t : k]$*, i.e., for*

$$X \in \{ PW, W, WS, out, victims, \Delta, \Gamma \}$$

and for

$$X \in \{ R, \Upsilon, I, in \}$$

and for

$$X \in \{ core, C, F, \Phi, Sh, ShR, SC \}$$

we have

$$X_{\uparrow}(k') = X_{\uparrow}[t \to k](k' - 1).$$

Proof. By Lemma 248, the schedule is pseudo-valid until k', step t is not race-synchronized with k', and the invariants hold at $k'-1$

$$pval(t,k') \wedge t \not\Rrightarrow k' \wedge Inv(t,k'-1).$$

With Lemma 232 we obtain that X is unchanged when moving t until k'

$$X_\uparrow(k') = X_\uparrow[t \to k'](k'-1)$$

and the claim follows as moving k' further does not affect step $k'-1$

$$X_\uparrow[t \to k'](k'-1) = X_\uparrow[t \to k'][k' \to k](k'-1) = X_\uparrow[t \to k](k'-1).$$

□

We can now show that delaying a step that can be delayed until k does not affect synchronization and races of any of the steps before k. Together with Lemma 246, this lemma is extremely powerful since it allows us to drop unsynchronized steps from a schedule without affecting the steps around it.

Lemma 251. *Assume that the schedule is pseudo-valid until k, t is not race-synchronized with k, and we can delay t until* $k-1$

$$pval(t,k) \wedge t \not\Rrightarrow k \wedge delay(t,k-1),$$

we can move t to k without changing synchronization between steps before t and steps after t: for all $t' < t$ *and all* $k' \in (t:k]$ *we have*

1. *The races are unchanged, i.e., for each type of race* $T \in \{WR, WW, RW, CM\}$ *we have*
$$T_\uparrow(t',k') \equiv T_\uparrow[t \to k](t',k'-1).$$
2. *The interrupts are unchanged, i.e., we have*
$$int_\uparrow(t',k') \equiv int_\uparrow[t \to k](t',k'-1) \wedge int_\uparrow(k',t') \equiv int_\uparrow[t \to k](k'-1,t').$$
3. *Visible write-read races are unchanged*
$$VR_\uparrow(t',k') \equiv VR_\uparrow[t \to k](t',k'-1).$$
4. *The synchronization is unchanged, i.e., for* $R \in \{\rhd, \Rrightarrow, \blacktriangleright\}$
$$t'\ R\ k' \equiv t'\ R[t \to k]k'-1.$$

Proof. The situation is depicted in Fig. 4.8.

We obtain directly with Lemma 250 that outputs, inputs, and fetched registers $X \in \{in, out, F\}$ of step k' are unchanged

$$X_\uparrow(k') = X_\uparrow[t \to k](k'-1),$$

and from the fact that t' is less than t and thus in the prefix of s which is unchanged, that the outputs and inputs of step t' are also unchanged

$$in_\uparrow(t') = in_\uparrow[t \to k](t'),$$

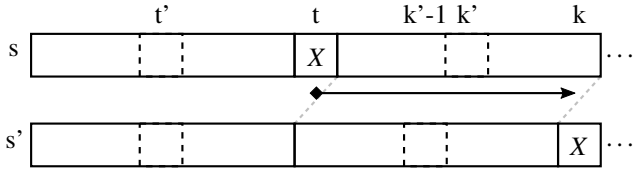


Figure 4.8: Steps t' and k' (represented by rectangles with dashed outlines) can be anywhere in the given ranges (represented by long rectangles). By moving step t to position k, step k' is moved to position $k'-1$ but step t' is not moved at all.

$$out_\uparrow(t') = out_\uparrow[t \to k](t').$$

The first claim follows. We only show the proof for $T = WR$, the others are analogous.

$$\begin{aligned} WR_\uparrow(t',k') &\equiv out_\uparrow(t') \dot\frown in_\uparrow(k') \\ &\equiv out_\uparrow[t \to k](t') \dot\frown in_\uparrow[t \to k](k'-1) \\ &\equiv WR_\uparrow[s[t \to k]](t',k'-1). \end{aligned}$$

With Lemma 250 we obtain analogously that the victims of the steps are the same

$$\begin{aligned} victims_\uparrow[t \to k](k'-1) &= victims_\uparrow(k'), && \text{L 250} \\ victims_\uparrow[t \to k](t') &= victims_\uparrow(t'). \end{aligned}$$

The second claim follows. We show the proof only for $int_\uparrow(t',k')$

$$\begin{aligned} int_\uparrow(t',k') &= u_\uparrow(k') \in victims_\uparrow(t') \\ &= u_\uparrow[t \to k](k'-1) \in victims_\uparrow[t \to k](t') \\ &= int_\uparrow[t \to k](t',k'-1). \end{aligned}$$

For the third claim, we obtain equalities for $WS_\uparrow$ and $in_\uparrow$

$$\begin{aligned} WS_\uparrow(t') &= WS_\uparrow[t \to k](t'), \\ in_\uparrow(k') &= in_\uparrow[t \to k](k'-1), && \text{L 250} \\ Sh_\uparrow(t') &= Sh_\uparrow[t \to k](t') = 1, \\ Sh_\uparrow(k') &= Sh_\uparrow[t \to k](k'-1) = 1, && \text{L 250} \end{aligned}$$

as well as that t' is also less than $k'-1$

$$t' < t \leq k'-1.$$

We also obtain that the addresses overwritten in the interval $(t' : t)$ are the same, and the addresses overwritten in the interval $(t : k')$ are now overwritten in the interval $(t-1 : k'-1)$

$$\begin{aligned} out_\uparrow((t' : t)) &= \bigcup_{t'' \in (t' : t)} out_\uparrow(t'') \\ &= \bigcup_{t'' \in (t' : t)} out_\uparrow[t \to k](t'') \end{aligned}$$

$$
\begin{aligned}
&= out_\uparrow[t \to k]((t' : t)),\\
out_\uparrow((t : k')) &= \bigcup_{t'' \in (t:k')} out_\uparrow(t'')\\
&= \bigcup_{t'' \in (t:k')} out_\uparrow[t \to k](t'' - 1) && \text{L 250}\\
&= \bigcup_{t'' \in (t-1:k'-1)} out_\uparrow[t \to k](t'')\\
&= out_\uparrow[t \to k]((t-1 : k'-1)).
\end{aligned}
$$

On the other hand, it is not necessarily true that the visible outputs are unchanged. Since the outputs of t are missing in schedule $s[t \to k]$, there might be additional visible outputs of t' in that schedule. In fact, the reordering increases the visible write-set exactly by the visible outputs of the moved step t

$$
\begin{aligned}
vws_\uparrow(t',k') &= WS_\uparrow(t') \setminus out_\uparrow((t' : k'))\\
&= WS_\uparrow(t') \setminus out_\uparrow((t' : t)) \setminus out_\uparrow(t) \setminus out_\uparrow((t : k'))\\
&= WS_\uparrow[t \to k](t') \setminus out_\uparrow[t \to k]((t' : t)) \setminus out_\uparrow(t) \setminus out_\uparrow((t : k'))\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow(t) \setminus out_\uparrow((t : k'))\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow((t : k')) \setminus out_\uparrow(t)\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow((t : k')) \setminus (dc(WS_\uparrow(t)) \cup WS_\uparrow(t))\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow((t : k')) \setminus (dc(WS_\uparrow(t)) \cup WS_\uparrow(t) \setminus out_\uparrow((t : k')))\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow((t : k')) \setminus (dc(WS_\uparrow(t)) \cup vws_\uparrow(t,k'))\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow((t : k')) \setminus vout_\uparrow(t,k')\\
&= vws_\uparrow[t \to k](t',t) \setminus out_\uparrow[t \to k]((t-1 : k'-1)) \setminus vout_\uparrow(t,k')
\end{aligned}
$$

We can pull these outputs into the visible write-set using Lemma 110

$$= vws_\uparrow[t \to k](t',k'-1) \setminus vout_\uparrow(t,k').$$

By Lemmas 230 and 248 there is no visible write-read race, and the visible outputs of t are not used by k'

$$vout_\uparrow(t,k') \not\frown in_\uparrow(k')$$

and thus they do not affect the intersection with inputs

$$
\begin{aligned}
& vws_\uparrow(t',k') \mathbin{\dot\cap} in_\uparrow(k')\\
\iff\ & vws_\uparrow[t \to k](t',k'-1) \setminus vout_\uparrow(t,k') \mathbin{\dot\cap} in_\uparrow(k')\\
\iff\ & vws_\uparrow[t \to k](t',k'-1) \mathbin{\dot\cap} in_\uparrow(k').
\end{aligned}
$$

We conclude that the visible outputs still intersect the inputs

$$
\begin{aligned}
VR_\uparrow(t',k') &\iff vout_\uparrow(t',k') \mathbin{\dot\cap} in_\uparrow(k')\\
&\iff vws_\uparrow(t',k') \cup dc(WS_\uparrow(t')) \mathbin{\dot\cap} in_\uparrow(k')\\
&\iff vws_\uparrow(t',k') \mathbin{\dot\cap} in_\uparrow(k') \vee dc(WS_\uparrow(t')) \mathbin{\dot\cap} in_\uparrow(k')\\
&\iff vws_\uparrow[t \to k](t',k'-1) \mathbin{\dot\cap} in_\uparrow(k') \vee dc(WS_\uparrow(t')) \mathbin{\dot\cap} in_\uparrow(k')\\
&\iff vws_\uparrow[t \to k](t',k'-1) \cup dc(WS_\uparrow(t')) \mathbin{\dot\cap} in_\uparrow(k')
\end{aligned}
$$

$$\begin{aligned}
&\iff vws_\uparrow[t \to k](t', k'-1) \cup dc(WS_\uparrow[t \to k](t')) \mathbin{\dot\cap} in_\uparrow[t \to k](k'-1),\\
&\iff vout_\uparrow[t \to k](t', k'-1) \mathbin{\dot\cap} in_\uparrow[t \to k](k'-1)\\
&\iff VR_\uparrow[t \to k](t', k'-1),
\end{aligned}$$

which is the third claim.

For the fourth claim, we look through all of the synchronization rules and observe that almost all remaining subterms X that depend on one step are also equal in s and $s[t \to k]$ due to Lemma 250

$$\begin{aligned}
X_\uparrow(k') &= X_\uparrow[t \to k](k'-1),\\
X_\uparrow(t') &= X_\uparrow[t \to k](t').
\end{aligned}$$

Together with the first three claims, this works for all rules except ISSUEWRITE, where we additionally have to show that step k' commits t' before the reordering iff step $k'-1$ does after the reordering

$$\begin{aligned}
&t' = hd(issue^{k'}_\uparrow(i)) \wedge s(k') \in \Sigma_{WB,i}\\
\overset{!}{\equiv}\ &t' = hd(issue_\uparrow[t \to k]^{k'-1}(i)) \wedge s[t \to k](k'-1) \in \Sigma_{WB,i}
\end{aligned}$$

Note first that step $k'-1$ is a write buffer step of some unit i in the reordered schedule iff step k' was in the original schedule because the oracle inputs are the same

$$s[t \to k](k'-1) = s(k').$$

Using simple boolean logic, this reduces the equivalence above to showing that when step k' is a write buffer step of unit i, then the head of the sequence of issued writes of that unit at k' in the original schedule was t' iff the one at $k'-1$ in the reordered schedule is. Assume thus that step k' is made by a write buffer step of unit i

$$s(k') \in \Sigma_{WB,i}$$

and we have to show that the head in the original schedule is t' iff it is in the reordered schedule

$$t' = hd(issue^{k'}_\uparrow(i)) \overset{!}{\equiv} t' = hd(issue_\uparrow[t \to k]^{k'-1}(i)).$$

Since $mv[t \leftarrow k]$ is injective we apply it to the right side and obtain the following claim

$$t' = hd(issue^{k'}_\uparrow(i)) \overset{!}{\equiv} mv[t \leftarrow k](t') = mv[t \leftarrow k](hd(issue_\uparrow[t \to k]^{k'-1}(i))).$$

Reducing the claim further with Lemma 77 we obtain

$$t' = hd(issue^{k'}_\uparrow(i)) \overset{!}{\equiv} t' = hd(mv[t \leftarrow k](issue_\uparrow[t \to k]^{k'-1}(i))).$$

We show the stronger claim that both heads are equal

$$hd(issue^{k'}_\uparrow(i)) \overset{!}{=} hd(mv[t \leftarrow k](issue_\uparrow[t \to k]^{k'-1}(i))).$$

By Lemma 249, the invariants hold at $k'-1$, the schedule is pseudo-valid until k', and step t is not forward-synchronized with step k'

$$pval(t, k') \wedge t \not\rhd k' \wedge Inv(t, k'-1).$$

By Invariant IssueSame and Lemma 204 with $k' := k$ we obtain that the write buffer at $k'-1$ is exactly missing step t and has been reordered

$$issue_{\uparrow}^{k'}(i) = Op_{i\uparrow}(t)(mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i)),t),$$

We distinguish between the possible operations.

$Op_{i\uparrow}(t) = push$**:** In this case by Lemma 221 the sequence of issued writes is non-empty

$$issue_{\uparrow}[t \to k]^{k'-1}(i) \neq \varepsilon$$

and neither is the sequence of writes after undoing the reordering, since that does not change the length

$$mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i)) \neq \varepsilon.$$

Step t adds additional elements

$$\begin{aligned} issue_{\uparrow}^{k'}(i) &= Op_{i\uparrow}(t)(mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i)),t) \\ &= mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i)) \circ t, \end{aligned}$$

and with Lemma 40 the claim follows

$$hd(issue_{\uparrow}^{k'}(i)) = hd(mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i))).$$

$Op_{i\uparrow}(t) = pop$**:** In this case steps t and k' are made by the write buffer of the same unit

$$s(t) \in \Sigma_{WB,i},$$

which contradicts Lemma 217.

$Op_{i\uparrow}(t) = noop$**:** The step did not change the sequence of issued writes

$$Op_{i\uparrow}(t)(mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i)),t) = mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i))$$

and the claim follows

$$hd(issue_{\uparrow}^{k'}(i)) = hd(mv[t \leftarrow k](issue_{\uparrow}[t \to k]^{k'-1}(i))).$$

□

Synchronization between steps in the reordered interval is also unchanged.

Lemma 252. *Assume that the schedule is pseudo-valid until k, t is not race-synchronized with k, and we can delay t until* $k-1$

$$pval(t,k) \wedge t \not\Rrightarrow k \wedge delay(t,k-1),$$

we can move t to k without changing synchronization between steps in the reordered interval: for all $t',k' \in (t:k]$ *we have that the synchronization is unchanged, i.e., for* $R \in \{\rhd, \Rrightarrow, \blacktriangleright\}$

$$t' \; R \; k' \equiv t'-1 \; R[t \to k]k'-1.$$

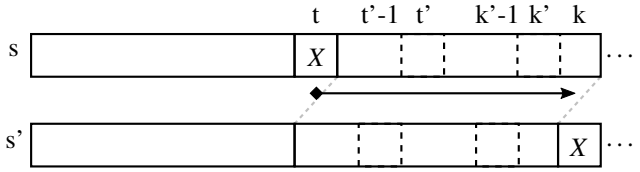


Figure 4.9: Intermediate steps t', k' are both moved one step forward, but synchronization is unchanged.

Proof. The situation is depicted in Fig. 4.9.

We observe first that the reordering moves t' correctly by Lemma 77

$$mv[t \to k](t') = t' - 1.$$

The proof is mostly analogous to that of Lemma 251 and we state the corresponding proof steps without a detailed proof. For all important functions of the step X we obtain that the value is unchanged

$$X(t') = X[t \to k](t'-1) \wedge X(k') = X[t \to k](k'-1).$$

Races $T \in \{ WR, RW, WW \}$ are unchanged

$$T(t', k') \equiv T[t \to k](t'-1, k'-1).$$

Interrupts between k' and t' are unchanged

$$int(k', t') \equiv int[t \to k](k'-1, t'-1).$$

The proof is much easier for the visible write-read races, since all steps are still there. The outputs in the interval $(t' : k')$ are completely transferred to the interval $(t'-1 : k'-1)$

$$\begin{aligned} out_\uparrow((t' : k')) &= \bigcup_{t'' \in (t' : k')} out_\uparrow(t'') \\ &= \bigcup_{t'' \in (t' : k')} out_\uparrow[t \to k](t''-1) && \text{L 250} \\ &= \bigcup_{t'' \in (t'-1 : k'-1)} out_\uparrow[t \to k](t'') \\ &= out_\uparrow[t \to k]((t'-1 : k'-1)). \end{aligned}$$

Consequently the visible write-set is unchanged

$$\begin{aligned} vws_\uparrow(t', k') &= WS_\uparrow(t') \setminus out_\uparrow((t' : k')) \\ &= WS_\uparrow[t \to k](t'-1) \setminus out_\uparrow[t \to k]((t'-1 : k'-1)) \\ &= vws_\uparrow[t \to k](t'-1, k'-1) \end{aligned}$$

and visible write-read races are stable

$$\begin{aligned} VR_\uparrow(t', k') &\iff vout_\uparrow(t', k') \barwedge in_\uparrow(k') \\ &\iff vws_\uparrow(t', k') \cup dc(WS_\uparrow(t')) \barwedge in_\uparrow(k') \end{aligned}$$

$$\begin{aligned}
&\iff vws_\uparrow[t \to k](t'-1,k'-1) \cup dc(WS_\uparrow[t \to k](t'-1)) \mathbin{\dot\cap} in_\uparrow[t \to k](k'-1),\\
&\iff vout_\uparrow[t \to k](t'-1,k'-1) \mathbin{\dot\cap} in_\uparrow[t \to k](k'-1)\\
&\iff VR_\uparrow[t \to k](t'-1,k'-1).
\end{aligned}$$

As before, this works for all rules except IssueWrite, where we additionally have to show that step k' commits t' iff step $k'-1$ commits $t'-1$

$$\begin{aligned}
&t' = hd(issue_\uparrow^{k'}(i)) \wedge s(k') \in \Sigma_{WB,i}\\
&\stackrel{!}{\equiv} t'-1 = hd(issue_\uparrow[t \to k]^{k'-1}(i)) \wedge s[t \to k](k'-1) \in \Sigma_{WB,i}
\end{aligned}$$

The proof slightly differs because we need to consider step $t'-1$ rather than step t'. Analogously to before we can assume that step k' is a write buffer step of unit i

$$s(k') \in \Sigma_{WB,i}$$

and have to show that the step being committed is t' in the original schedule iff the step being committed is $t'-1$ in the reordered schedule

$$t' = hd(issue_\uparrow^{k'}(i)) \stackrel{!}{\equiv} t'-1 = hd(issue_\uparrow[t \to k]^{k'-1}(i)).$$

Since $mv[t \leftarrow k]$ is injective we apply it to the right-hand side of the equivalence and obtain the following claim

$$t' = hd(issue_\uparrow^{k'}(i)) \stackrel{!}{\equiv} mv[t \leftarrow k](t'-1) = mv[t \leftarrow k](hd(issue_\uparrow[t \to k]^{k'-1}(i))).$$

Reducing the claim further with Lemma 77 we obtain the following claim

$$t' = hd(issue_\uparrow^{k'}(i)) \stackrel{!}{\equiv} t' = hd(mv[t \leftarrow k](issue_\uparrow[t \to k]^{k'-1}(i))).$$

We show the stronger claim that both heads are equal

$$hd(issue_\uparrow^{k'}(i)) \stackrel{!}{=} hd(mv[t \leftarrow k](issue_\uparrow[t \to k]^{k'-1}(i))).$$

The proof can now be concluded literally like the proof of Lemma 251. □

4.10 Synchronization of Unshared Accesses

Using these results, we prove that communication needs some sequence of synchronization. We define *far forward-synchronization* as the relation composed from some number of synchronized steps and one forward-synchronized step.

$$\blacktriangleright\!\!\triangleright = \blacktriangleright^* \circ \triangleright.$$

Analogously we define *far race-synchronization*

$$\blacktriangleright\!\!\blacktriangleright = \blacktriangleright^* \circ \blacktriangleright\!\!\triangleright.$$

We obtain that both relations are left-compositional with synchronization.

Lemma 253.

$$\blacktriangleright \circ \blacktriangleright\!\!\blacktriangleright \subseteq \blacktriangleright\!\!\blacktriangleright,$$
$$\blacktriangleright \circ \blacktriangleright\!\!\blacktriangleright \subseteq \blacktriangleright\!\!\blacktriangleright.$$

Proof. Follows with the associativity of composition and the fact that closure under composition is closed under composition. We only show the proof for far race-synchronization

$$\begin{aligned}\blacktriangleright \circ \blacktriangleright\!\!\blacktriangleright &= \blacktriangleright \circ (\blacktriangleright^* \circ \blacktriangleright\!\!\blacktriangleright) \\ &= (\blacktriangleright \circ \blacktriangleright^*) \circ \blacktriangleright\!\!\blacktriangleright \\ &\subseteq \blacktriangleright^* \circ \blacktriangleright\!\!\blacktriangleright \\ &= \blacktriangleright\!\!\blacktriangleright.\end{aligned}$$

□

Far race-synchronization subsumes race-synchronization.

Lemma 254.

$$\blacktriangleright\!\!\blacktriangleright \subseteq \blacktriangleright\!\!\blacktriangleright.$$

Proof. Straightforward

$$\begin{aligned}\blacktriangleright\!\!\blacktriangleright &\subseteq \blacktriangleright^* \circ \blacktriangleright\!\!\blacktriangleright \\ &= \blacktriangleright\!\!\blacktriangleright.\end{aligned}$$

□

Recall that in some lemmas above, e.g., Lemma 251, we require that a step can be delayed until some $k-1$ and that it is not race-synchronized with k. By unfolding the definition of *delay* we obtain that the step must not be synchronized with any step until $k-1$ and that it is not race-synchronized with k. This happens to be the case whenever step t is the last step that is not far race-synchronized with k, as the following lemma shows.

Lemma 255. *Assume that step t is not far race-synchronized with k but all steps in between are*

$$t \not\blacktriangleright\!\!\blacktriangleright\, k \wedge \forall t' \in (t:k).t' \blacktriangleright\!\!\blacktriangleright\, k.$$

Then step t is neither race-synchronized with k nor synchronized with any of the steps in between

$$t \not\blacktriangleright\!\!\blacktriangleright\, k \wedge \forall t' \in (t:k).t \not\blacktriangleright\, t'.$$

Proof. The first part of the claim is the contraposition of Lemma 254.

The second part of the claim is the contraposition of Lemma 253. □

Lemma 256. *A schedule which is semi-valid until k*

$$\Gamma\Phi^k_\uparrow(s)$$

and where some step t modifies code of step k or step k is valid

$$CM_\uparrow(t,k) \vee \Gamma_\uparrow(k)$$

is pseudo-valid for $l < k$

$$pval(l,k).$$

Proof. We immediately obtain that the schedule is valid until l

$$\Gamma^{l}_{\uparrow}(s)$$

and the steps $t' \in (l : k)$ are all valid

$$\Gamma_{\uparrow}(t'),$$

and we will show that step k is WB-feasible, which will by definition of *pval* complete the proof. Note first that step k is by assumption feasible

$$\Phi_{\uparrow}(k),$$

and by definition of Ψ we only have to show that it also satisfies Λ. By assumption there is either a code modification or step k is valid, and we distinguish between these two cases.

$CM_{\uparrow}(t,k)$**:** We obtain that the fetched registers are modified

$$out_{\uparrow}(t) \dot{\cap} F_{\uparrow}(k),$$

and thus the step fetches something

$$F_{\uparrow}(k) \neq \emptyset$$

and we conclude that it is not a write buffer step

$$s(k) \notin \Sigma_{WB,i}$$

and the claim follows by definition of Λ

$$\Lambda_{\uparrow}(k).$$

$\Gamma_{\uparrow}(k)$**:** The claim follows by definition of Γ

$$\Lambda_{\uparrow}(k).$$

□

We show that far-synchronization is not affected by delaying a step that is not synchronized.

Lemma 257. *Assume that the schedule is pseudo-valid until k, t is not race-synchronized with k, and t can be delayed until* $k-1$

$$pval(t,k) \wedge t \not\blacktriangleright\!\!\blacktriangleright k \wedge delay(t,k-1),$$

and is also not synchronized with any step before k

$$\forall l \in (t : k].t \not\blacktriangleright k,$$

we can move t to k without affecting far-synchronization between $t' < t$ *and* $k' \in (t : k]$

$$t' \blacktriangleright\!\!\blacktriangleright k' \equiv t' \blacktriangleright\!\!\blacktriangleright [t \to k]k' - 1.$$

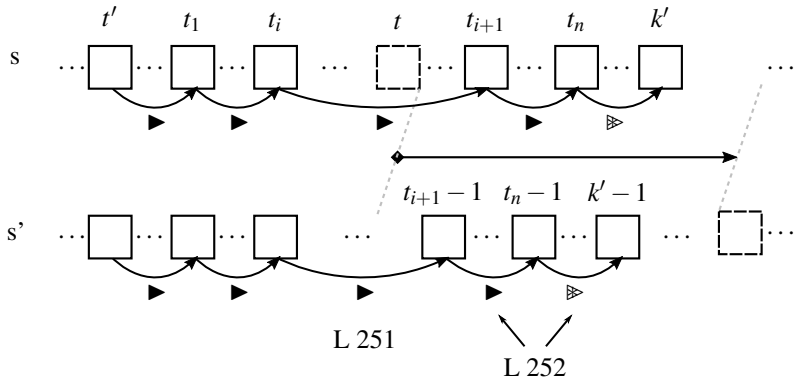


Figure 4.10: After step t is moved to position k, steps $t', \ldots, t_i < t$ are not moved and their synchronization is unaffected. Steps $t_{i+1}, \ldots, k' > t$ are moved forward by one step, and their synchronization can be inferred with Lemma 252. The synchronization of steps t_i and t_{i+1} on the other hand is maintained by Lemma 251.

Proof. We show only the direction from left to right, the other direction is analogous. The step is far-synchronized iff there is a sequence of steps $[t_0, \ldots, t_{n+1}]$ with $t_0 = t'$ and $t_{n+1} = k'$ that are each synchronized with one another except for last two steps, which are race synchronized (cf. Fig. 4.10)

$$t' \blacktriangleright\!\!\blacktriangleright k' \equiv t' \blacktriangleright t_1 \blacktriangleright \ldots \blacktriangleright t_n \blacktriangleright\!\!\blacktriangleright k'.$$

We show that by tracking the movement of all steps t_i, we again get a sequence $[mv[t \to k](t_0), \ldots, mv[t \to k](t_{n+1})]$ which has the same synchronizations

$$t_i \; R \; t_{i+1} \overset{!}{\equiv} mv[t \to k](t_i) \; R[t \to k] \; mv[t \to k](t_{i+1}).$$

Step t is not synchronized with any intermediate step l, and thus step t can not be in that sequence of steps

$$\forall i.t \neq t_i.$$

This allows us to distinguish between three cases, depending on whether the synchronization is before t, crosses over t, or is after t.

$t_i, t_{i+1} < t$**:** The schedule is the same until t_{i+1} and the synchronization rules can be transferred 1:1 for the same steps.

$$t_i \; R \; t_{i+1} \equiv t_i \; R[t \to k] \; t_{i+1}$$

The claim follows with Lemma 77 since the two steps are not moved

$$t_i \; R \; t_{i+1} \equiv mv[t \to k](t_i) \; R[t \to k] \; mv[t \to k](t_{i+1}).$$

$t_i < t$, $t_{i+1} > t$**:** By Lemma 251 we obtain that the synchronization is stable when step t_{i+1} is pushed forward

$$t_i \; R \; t_{i+1} \equiv t_i \; R[t \to k] \; t_{i+1} - 1,$$

and the claim follows with Lemma 77 since that is exactly how step t_{i+1} is moved by the reordering

$$t_i \; R \; t_{i+1} \equiv mv[t \to k](t_i) \; R[t \to k] \; mv[t \to k](t_{i+1}).$$

$t_i, t_{i+1} > t$**:** By Lemma 252 we obtain that the synchronization is stable when both steps are pushed forward

$$t_i \; R \; t_{i+1} \equiv t_i - 1 \; R[t \to k] \; t_{i+1} - 1,$$

and the claim follows with Lemma 77 since that is exactly how the steps are moved by the reordering

$$t_i \; R \; t_{i+1} \equiv mv[t \to k](t_i) \; R[t \to k] \; mv[t \to k](t_{i+1}).$$

And the claim follows with Lemma 77

$$\begin{aligned} t' \blacktriangleright\!\!\!\blacktriangleright k' &\implies t' \blacktriangleright \ldots \blacktriangleright k' \\ &\iff mv[t \to k](t') \blacktriangleright [t \to k] \ldots \blacktriangleright [t \to k] mv[t \to k](k') \\ &\implies t' \blacktriangleright\!\!\!\blacktriangleright [t \to k] k' - 1. \end{aligned}$$

□

We now prove a fundamental property of the synchronization relation and the simple annotation: when two threads access the same memory region without any synchronization, the accesses are always correctly annotated.

Lemma 258. *Assume that in a schedule which is semi-valid until k*

$$\Gamma \Phi^k_\uparrow(s)$$

and that some step t before k is not far race-synchronized with it

$$t < k \wedge t \not\blacktriangleright\!\!\!\blacktriangleright k.$$

Then all of the following are true

1. *If there is a code modification, the steps are annotated correctly*

$$CM_\uparrow(t,k) \to Sh_\uparrow(t) \wedge ShR_\uparrow(k).$$

2. *If there is a valid write-read race, the steps are annotated correctly*

$$WR_\uparrow(t,k) \wedge \Gamma_\uparrow(k) \to Sh_\uparrow(t) \wedge ShR_\uparrow(k).$$

3. *If there is a valid write-write race, the steps are annotated correctly*

$$WW_\uparrow(t,k) \wedge \Gamma_\uparrow(k) \to Sh_\uparrow(t) \wedge Sh_\uparrow(k).$$

Proof. With Lemma 251 we will now drop all non-far race-synchronized steps between t and k (cf. Figure 4.11). To see how this can be done, let $l \in (t : k)$ be the last such non-synchronized step

$$l \not\blacktriangleright\!\!\!\blacktriangleright k \wedge \forall t' \in (l : k).t' \blacktriangleright\!\!\!\blacktriangleright k.$$

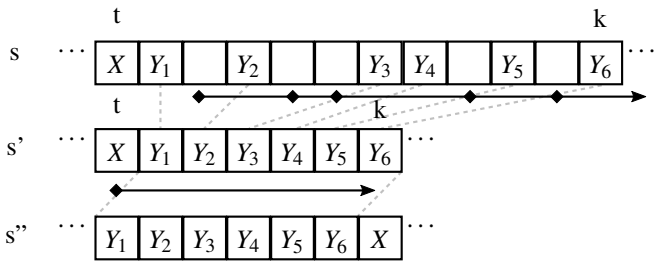


Figure 4.11: Steps which are not synchronized with step k (left blank) can be moved behind step k, and thus dropped from the schedule. The proof can focus on the case where step t is the only step not synchronized with step k.

By Lemma 255 step l is neither race-synchronized with k nor synchronized with any of the steps in between

$$l \not\Rrightarrow k \wedge \forall t' \in (l:k).l \not\blacktriangleright t'.$$

By Lemma 256 the schedule is pseudo-valid until k in all three cases (in Case 1 because of the code modification, in Cases 2 and 3 because step k is valid)

$$pval(l,k)$$

and by Lemma 244 also until $k-1$

$$pval(l,k-1).$$

By definition of *delay* it follows that t can be delayed until $k-1$

$$delay(l,k-1).$$

We move step l behind step k. By Lemmas 250 and 251 we immediately obtain that the schedule is still valid and all races and annotations are unchanged. With Lemma 257 with $t := l$, $k := k$, $k' := k$, and $t' := t$ we obtain further that there is still no far race-synchronization.

Thus we assume without loss of generality that there are no such steps l which are not far race-synchronized with k, and thus step t is the last step not far race-synchronized with step k

$$\forall t' \in (t:k).t' \blacktriangleright\!\!\blacktriangleright k.$$

By Lemma 255 step t is neither forward-synchronized with k nor synchronized with any of the steps in between

$$t \not\Rrightarrow k \wedge \forall t' \in (t:k).t \not\blacktriangleright t'.$$

By Lemmas 244 and 256 we obtain again that the schedule is pseudo-valid until k and $k-1$

$$pval(t,k) \wedge pval(t,k-1).$$

It immediately follows that t can be delayed until $k-1$

$$delay(t,k-1),$$

and with Lemma 246 we obtain that the invariants hold when moving until $k-1$

$$Inv(t, k-1).$$

Since step t is not race-synchronized with k, it is also not forward-synchronized with it

$$t \not\rhd k.$$

The claim is now Lemma 228. □

4.11 Simulation of the High-Level Machine

Note that we have set up the semantics of the high-level machine to be able to handle exactly the schedules from the low-level machine. This allows us to define a straightforward simulation relation. We simply use strong agreement between the low-level machine step and the high-level machine step. Thus when the low-level machine configuration $c_\downarrow$ strongly agrees with the high-level machine configuration c when stepped with x

$$c_\downarrow =^{x}_{\downarrow,\uparrow} c,$$

we say that $c_\downarrow$ *simulates* c *when stepped with* x.

Note that this is a simulation relation between steps, not configurations. In order to obtain a state-based simulation relation, one would need an operation buffer in the abstract machine, such as the one used by Cohen and Schirmer [CS10], where processor steps simulate a push of the operation into the operation buffer of that processor, and write buffer steps simulate popping a sequence of operations leading to the next write on that operation buffer; and where the simulation relation couples each local processor state in the low-level machine with the state one would obtain by popping the entire operation buffer of that processor in the abstract machine. This is obviously more complicated. Simpler purely state-based coupling relations do not work, because buffered steps will desynchronize the memory component of the abstract machine and of the low-level machine, and therefore one can not know that the next step will see the correct value from the memory.

We say that a schedule s is *reduced (until k)* or *k-abstract* if at all steps (until k) the low-level execution simulates the high-level execution:

$$s \in \mathrm{ABS}_k \equiv \forall t < k. c_\downarrow[s]^t =^{s(t)}_{\downarrow,\uparrow} c_\uparrow[s]^t.$$

Due to the transitivity of strong agreement, we can transfer strong agreement between the machines; i.e., strong agreement with a low-level machine configuration in an abstract schedule is equivalent to agreement with the high-level machine configuration at the same step.

Lemma 259. *When a schedule is* $t'+1$*-abstract*

$$s \in \mathrm{ABS}_{t'+1},$$

we have strong agreement at t' *with the low-level machine iff we have strong agreement with the high-level machine*

$$c^{t'}_\uparrow =^{s(t')}_{\uparrow,M} c \iff c^{t'}_\downarrow =^{s(t')}_{\downarrow,M} c.$$

Proof. Because the schedule is $t'+1$-abstract, at step t' the machines strongly agree

$$c_{\downarrow}^{t'} =_{\downarrow,\uparrow}^{s(t')} c_{\uparrow}^{t'}.$$

We show the equivalence by showing an implication in each direction.

$\Longrightarrow$**:** The claim follows by transitivity

$$c_{\downarrow}^{t'} =_{\downarrow,\uparrow}^{s(t')} c_{\uparrow}^{t'} =_{\uparrow,M}^{s(t')} c.$$

$\Longleftarrow$**:** The claim follows by transitivity and symmetry

$$c_{\uparrow}^{t'} =_{\uparrow,\downarrow}^{s(t')} c_{\downarrow}^{t'} =_{\downarrow,M}^{s(t')} c.$$

□

Lemma 260. *Let O be some reordering that moves $t'' < k''$ to $t' < k'$*

$$sO(t') = s(t'').$$

If the schedule and its reordering sO are abstract until k'' and k', respectively

$$s \in \mathrm{ABS}_{t''+1} \wedge sO \in \mathrm{ABS}_{t'+1},$$

we have strong agreement with the low-level machine iff we have strong agreement with the high-level machine

$$c_{\uparrow}O^{t'} =_{\uparrow}^{sO(t')} c_{\uparrow}^{t''} \iff c_{\downarrow}O^{t'} =_{\downarrow}^{sO(t')} c_{\downarrow}^{t''}.$$

Proof. The claim is twice Lemma 259

$$\begin{aligned}
& c_{\uparrow}O^{t'} =_{\uparrow}^{sO(t')} c_{\uparrow}^{t''} && \\
\iff\ & c_{\uparrow}O^{t'} =_{\uparrow,\uparrow}^{sO(t')} c_{\uparrow}^{t''} && \\
\iff\ & c_{\downarrow}O^{t'} =_{\downarrow,\uparrow}^{sO(t')} c_{\uparrow}^{t''} && \text{L 259} \\
\iff\ & c_{\downarrow}O^{t'} =_{\downarrow,\downarrow}^{sO(t')} c_{\downarrow}^{t''} && \text{L 259, Symmetry} \\
\iff\ & c_{\downarrow}O^{t'} =_{\downarrow}^{sO(t')} c_{\downarrow}^{t''} &&
\end{aligned}$$

□

We add a simple state invariant, which says that processor configurations and write buffers are the same for all processors. In that case we say $c_{\downarrow}$ is *locally simulating* c and write

$$c_{\downarrow} \doteq c \equiv \forall i. c_{\downarrow}.m =_{A_{PR,i}} c.m \wedge c_{\downarrow}.wb =_i c.wb.$$

We show that abstract schedules also satisfy the local simulation relation by a series of lemmas.

Lemma 261.

$$c_{\downarrow} \doteq c \to c_{\downarrow}.m =_{C_{\downarrow}(c_{\downarrow},x)} c.m.$$

Proof. Clearly the core registers are processor registers

$$C_{\downarrow}(c_{\downarrow},x) \subseteq A_{PR,u(x)},$$

and thus the configurations agree on it

$$c_{\downarrow}.m =_{C_{\downarrow}(c_{\downarrow},x)} c.m.$$

□

Lemma 262.

$$c_{\downarrow} \doteq c \rightarrow core_{\downarrow}(c_{\downarrow},x) = core_{\uparrow}(c,x).$$

Proof. With Lemma 261 we obtain that the configurations agree on the core registers

$$c_{\downarrow}.m =_{C_{\downarrow}(c_{\downarrow},x)} c.m.$$

The claim is now Lemma 142. □

Lemma 263.

$$c_{\downarrow} \doteq c \rightarrow c_{\downarrow}.m =_{A_{SC,i}} c.m.$$

Proof. Because $c_{\downarrow}$ locally simulates c, it has the same core configuration for each processor, including processor i

$$c_{\downarrow}.m =_{A_{PR,i}} c.m.$$

Since the mode registers are core registers

$$A_{SC,i} \subseteq A_{PR,i},$$

the claim follows

$$c_{\downarrow}.m =_{A_{SC,i}} c.m.$$

□

If a step is done in weak memory mode, the machines execute the same write.

Lemma 264. *If $c_{\downarrow}$ simulates c and $c_{\downarrow},x$ is a weak memory mode step*

$$c_{\downarrow} =^{x}_{\downarrow,\uparrow} c \wedge \neg SC_{\downarrow}(c_{\downarrow},x),$$

the low-level machine step $c_{\downarrow},x$ performs the same writes as the high-level machine step c,x that it simulates

$$W_{\downarrow}(c_{\downarrow},x) = W_{\uparrow}(c,x) \wedge out_{\downarrow}(c_{\downarrow},x) = out_{\uparrow}(c,x).$$

Proof. Steps that strongly agree agree on the core registers

$$c_{\downarrow}.m =_{C_{\downarrow}(c_{\downarrow},x)} c.m.$$

By Lemma 146 both machines use low-level machine semantics

$$LL_{\downarrow}(c_{\downarrow},x) = LL_{\uparrow}(c,x).$$

The claim is now Lemma 144. □

Only the outputs of sequentially consistent write buffer steps are not visible in the high-level machine.

Lemma 265. *If $c_\downarrow$ simulates c and $c_\downarrow, x$ is not a sequentially consistent write buffer step*

$$c_\downarrow =^{x}_{\downarrow,\uparrow} c \wedge \neg(SC_\downarrow(c_\downarrow, x) \wedge x \in \Sigma_{WB, u_\downarrow(c_\downarrow, x)}),$$

writes and outputs of the low-level machine are visible in the high-level machine

$$W_\downarrow(c_\downarrow, x) \subseteq W_\uparrow(c, x) \wedge out_\downarrow(c_\downarrow, x) \subseteq out_\uparrow(c, x).$$

Proof. We distinguish between the following two cases.

$\neg SC_\downarrow(c_\downarrow, x)$: Lemma 264 proves the claim with equality

$$W_\downarrow(c_\downarrow, x)) = W_\uparrow(c, x) \wedge out_\downarrow(c_\downarrow, x) = out_\uparrow(c, x).$$

$SC_\downarrow(c_\downarrow, x)$: By assumption we also have that x is also not an input for a write buffer step

$$x \notin \Sigma_{WB, u_\downarrow(c_\downarrow, x)},$$

leaving only processor steps

$$x \in \Sigma_{P,i}.$$

By Lemma 144 the steps prepare the same writes

$$PW_\downarrow(c_\downarrow, x) = PW_\uparrow(c, x),$$

and since a processor in the high-level machine in sequential mode executes both buffered and bypassing portions of this write whereas in the low-level machine it only executes the bypassing portions of this write, we obtain that write executed in the high-level machine subsumes the write executed in the low-level machine

$$\begin{aligned} W_\downarrow(c_\downarrow, x) &= PW_\downarrow(c_\downarrow, x).bpa \\ &= PW_\uparrow(c, x).bpa \\ &\subseteq PW_\uparrow(c, x).bpa \cup PW_\uparrow(c, x).wba \\ &= W_\uparrow(c, x), \end{aligned}$$

which is the first claim. The second claim follows with Lemma 26

$$out_\downarrow(c_\downarrow, x) = idc(Dom(W_\downarrow(c_\downarrow, x))) \subseteq idc(Dom(W_\uparrow(c, x))) = out_\uparrow(c, x).$$

□

In each case, however, the machines keep the core registers in sync.

Lemma 266. *Let a schedule be valid until $t+1$ in the low-level machine and step t simulate the high-level step*

$$\Gamma^{t+1}_\downarrow(s) \wedge c^t_\downarrow \doteq c^t_\uparrow \wedge c^t_\downarrow =^{s(t)}_{\downarrow,\uparrow} c^t_\uparrow,$$

then the two machines have the same effect on the core registers

$$W_\downarrow(t) =_{A_{PR,i}} W_\uparrow(t).$$

Proof. We distinguish between steps in weak memory mode, processor steps in strong memory mode, and write buffer steps in strong memory mode.

$\neg SC_\downarrow(t)$**:** In this case we obtain that the complete write is the same in both machines with Lemma 264

$$W_\downarrow(t) = W_\uparrow(t),$$

and the claim easily follows

$$W_\downarrow(t) =_{A_{PR,i}} W_\uparrow(t).$$

$SC_\downarrow(t) \wedge s(t) \in \Sigma_{P,j}$**:** By Lemma 144, the high level machine makes a step in strong memory mode

$$SC_\uparrow(t) = SC_\downarrow(t) = 1,$$

and the low-level machine executes exactly the bypassing portion of the prepared writes

$$\begin{aligned} W_\downarrow(t) &= PW_\downarrow(t).bpa \\ &= PW_\uparrow(t).wba, && \text{L 144} \end{aligned}$$

whereas the high-level machine also executes the buffered portion

$$W_\uparrow(t) = PW_\uparrow(t).bpa \cup PW_\uparrow(t).wba.$$

By Condition IRRForwarding no core registers are modified by buffered writes

$$A_{PR,i} \not\cap Dom(PW_\uparrow(t).wba).$$

The claim follows

$$\begin{aligned} W_\uparrow(t) &= PW_\uparrow(t).bpa \cup PW_\uparrow(t).wba \\ &=_{A_{PR,i}} PW_\uparrow(t).bpa \\ &= W_\downarrow(t). \end{aligned}$$

$s(t) \in \Sigma_{WB,j} \wedge SC_\downarrow(t)$**:** By Lemma 144, the high level machine makes a step in strong memory mode

$$SC_\uparrow(t) = SC_\downarrow(t) = 1.$$

By Lemma 174 the write buffer step has no effect in the high-level machine

$$W_\uparrow(t) = \emptyset \wedge A_{PR,i} \not\cap Dom(W_\uparrow(t)).$$

By Lemma 176 we obtain that the processor also does not have buffered such writes

$$\neg hit(A_{PR,i}, wb^t_\uparrow(j))$$

and by the local simulation we obtain that the low-level machine does not buffer any such writes either

$$hit(A_{PR,i}, wb^t_\downarrow(j)) = hit(A_{PR,i}, wb^t_\uparrow(j)) = 0.$$

Therefore the head of the write buffer in the low-level machine can not modify the processor registers either

$$A_{PR,i} \not\upharpoonright Dom(hd(wb^t_\downarrow(j))),$$

which is the write that is executed by the low-level machine

$$Dom(hd(wb^t_\downarrow(j))) = Dom(W_\downarrow(t)).$$

The claim follows

$$W_\downarrow(t) =_{A_{PR,i}} \emptyset = W_\uparrow(t).$$

□

Lemma 267. *In a schedule $s \in \mathrm{ABS}_k$ which is reduced until k the following are all true.*

1. *The low-level execution also locally simulates the high-level execution at k*

   $$c^k_\downarrow \doteq c^k_\uparrow,$$

   in particular, the machines have the same write buffers

   $$wb^k_\uparrow(i) = wb^k_\downarrow(i),$$

2. *The sequence of issues is the same*

   $$issue^k_\uparrow(i) = issue^k_\downarrow(i).$$

Proof. By induction on k. The base case $k = 0$ is trivial. In the inductive step from k to $k+1$, we obtain with the induction hypothesis that the low-level machine locally simulates the high-level machine; by assumption it also simulates its step

$$c^k_\downarrow \doteq c^k_\uparrow \wedge c^k_\downarrow =^{s(k)}_{\downarrow,\uparrow} c^k_\uparrow.$$

We now show the claims individually, splitting the claim about local simulation further between equality of processor configurations and equality of write buffers.

Same Issued Writes: With Lemma 144 we obtain that the processor performs the same operation on the buffers

$$Op_{i\downarrow}(k) = Op_{i\uparrow}(k).$$

The claim follows

$$\begin{aligned} issue^{k+1}_\downarrow(i) &= Op_{i\downarrow}(k)(issue^k_\downarrow(i), k) \\ &= Op_{i\uparrow}(k)(issue^k_\downarrow(i), k) \\ &= Op_{i\uparrow}(k)(issue^k_\uparrow(i), k) && \text{IH} \\ &= issue^{k+1}_\uparrow(i). \end{aligned}$$

Same buffers: With Lemma 144 we obtain that the processor is buffering the same writes and performs the same operation on the buffers

$$BW_{\downarrow}(k) = BW_{\uparrow}(k) \wedge Op_{i\downarrow}(k) = Op_{i\uparrow}(k).$$

The claim follows

$$\begin{aligned} wb_{\downarrow}^{k+1}(i) &= Op_{i\downarrow}(k)(wb_{\downarrow}^{k}(i), BW_{\downarrow}(k)) \\ &= Op_{i\uparrow}(k)(wb_{\downarrow}^{k}(i), BW_{\uparrow}(k)) \\ &= Op_{i\uparrow}(k)(wb_{\uparrow}^{k}(i), BW_{\uparrow}(k)) && \text{IH} \\ &= wb_{\uparrow}^{k+1}(i). \end{aligned}$$

Same Local Configurations: With Lemma 266 we obtain first that the updates are done the same way

$$W_{\downarrow}(k) =_{A_{PR,i}} W_{\uparrow}(k).$$

The claim follows with the induction hypothesis and Lemmas Lemma 29, 32, and 4

$$\begin{aligned} m_{\downarrow}^{k+1} &= m_{\downarrow}^{k} \circledast W_{\downarrow}(k) \\ &=_{A_{PR,i}} m_{\uparrow}^{k} \circledast W_{\downarrow}(k) && \text{IH, L 29, 4} \\ &=_{A_{PR,i}} m_{\uparrow}^{k} \circledast W_{\uparrow}(k) && \text{L 32, 4} \\ &= m_{\uparrow}^{k+1}. \end{aligned}$$

□

Lemma 268.

$$s \in ABS_k \rightarrow SC_{i\downarrow}(k) = SC_{i\uparrow}(k).$$

Proof. With Lemma 267 we obtain that there is a local simulation

$$c_{\downarrow}^{k} \doteq c_{\uparrow}^{k},$$

and the claim follows with Lemma 263. □

Lemma 269. *Let $s \in ABS_k$ be a k-abstract schedule. Then the following functions*

$$X \in \{\, core, C, F, \Phi, Sh, ShR, SC \,\}$$

are the same between the low-level and high-level machine at k

$$X_{\downarrow}(k) = X_{\uparrow}(k).$$

Proof. With Lemma 267 we obtain that there is a local simulation

$$c_{\downarrow}^{k} \doteq c_{\uparrow}^{k}.$$

With Lemma 261 we obtain that the configurations agree on the core registers

$$m_{\downarrow}^{k} =_{C_{\downarrow}(k)} m_{\uparrow}^{k}.$$

The claims follow with Lemma 142. □

If the simulation also holds during the step, we can go even further and simulate all functions which depend on fetch and read results.

Lemma 270. *Let $s \in ABS_{k+1}$ be an abstract schedule until $k+1$. Then functions that depend on the fetch and read results but not on the machine type*

$$X \in \{ PW, BW, Op_i, in, I, \Delta, \Gamma \},$$

but also the victims

$$X = victims$$

and in weak memory mode also the writes and outputs

$$\neg SC_{\uparrow}(k) \wedge X = \{ W, out \}$$

are the same between the low-level and high-level machine at k

$$X_{\downarrow}(k) = X_{\uparrow}(k).$$

Proof. The first claims are just Lemma 144. The claim about the victims is harder. With Lemma 266 we obtain first that the writes in both machines agree about the processor registers

$$W_{\downarrow}(k) =_{A_{PR,i}} W_{\uparrow}(k),$$

and thus the domains of the writes – and therefore the write-sets of the steps – also agree

$$\begin{aligned} WS_{\downarrow}(k) \hat{\cap} A_{PR,i} &\equiv Dom(W_{\downarrow}(k)) \hat{\cap} A_{PR,i} \\ &\equiv Dom(W_{\uparrow}(k)) \hat{\cap} A_{PR,i} \\ &\equiv WS_{\uparrow}(k) \hat{\cap} A_{PR,i}. \end{aligned}$$

The claim follows

$$\begin{aligned} victims_{\uparrow}(k) &= \{ i \mid A_{PR,i} \hat{\cap} WS_{\uparrow}(k) \wedge s(k) \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \} \\ &= \{ i \mid A_{PR,i} \hat{\cap} WS_{\downarrow}(k) \wedge s(k) \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \} \\ &= victims_{\downarrow}(k). \end{aligned}$$

The claim for the write and outputs is Lemma 264. □

Lemma 271. *In a schedule $s \in \mathrm{ABS}_k$ which is reduced until k the following are all true.*

1. *The schedules agree on validity until $k-1$*

$$\Gamma_{\downarrow}^{k-1}(s) = \Gamma_{\uparrow}^{k-1}(s),$$

2. *The schedules agree on semi-validity until k*

$$\Gamma\Phi_{\downarrow}^{k}(s) = \Gamma\Phi_{\uparrow}^{k}(s),$$

3. *The schedules agree on IPI validity until $k-1$*

$$\Delta_{IPI\downarrow}^{k-1}(s) = \Delta_{IPI\uparrow}^{k-1}(s).$$

Proof. Because the schedule is reduced, every step until $k-1$ the machines strongly agree

$$\forall t \leq k-1.c^t_\downarrow =^{s(t)}_{\downarrow,\uparrow} c^t_\uparrow.$$

The first claim follows directly with Lemma 144

$$\begin{aligned}\Gamma^{k-1}_\downarrow(s) &= \forall t \leq k-1.\Gamma_\downarrow(t) \\ &= \forall t \leq k-1.\Gamma_\uparrow(t) && \text{L 144} \\ &= \Gamma^{k-1}_\uparrow(s),\end{aligned}$$

and the second claim follows with Lemma 269

$$\begin{aligned}\Gamma\Phi^k_\downarrow(s) &= \Gamma^{k-1}_\downarrow(s) \wedge \Phi_\downarrow(k) \\ &= \Gamma^{k-1}_\uparrow(s) \wedge \Phi_\uparrow(k) && \text{Claim 1, L 269} \\ &= \Gamma\Phi^k_\uparrow(s).\end{aligned}$$

With Lemma 267 we obtain that processors have the same write buffers in both machines at all steps until k

$$\forall t \leq k.wb^t_\downarrow = wb^t_\uparrow,$$

and with Lemma 270 we obtain the steps until $k-1$ have the same victims

$$\forall t \leq k-1.victims_\downarrow(t) = victims_\uparrow(t).$$

The third claim follows

$$\begin{aligned}\Delta^{k-1}_{IPI\downarrow}(s) &= \forall t \leq k-1.\Delta_{IPI\downarrow}(t) \\ &= \forall t \leq k-1. \bigwedge_{j \in victims_\downarrow(t)} wb^t_\downarrow(j) = \varepsilon \\ &= \forall t \leq k-1. \bigwedge_{j \in victims_\uparrow(t)} wb^t_\uparrow(j) = \varepsilon \\ &= \forall t \leq k-1.\Delta_{IPI\uparrow}(t) \\ &= \Delta^{k-1}_{IPI\uparrow}(s).\end{aligned}$$

□

We therefore sometimes drop the machine-type index for these functions where appropriate, replacing it by an asterisk[3]. Let for example $s \in \mathrm{ABS}_k$ be a k-abstract schedule. We write the following

$$\Gamma^{k-1}_*(s) = \Gamma^{k-1}_\downarrow(s) = \Gamma^{k-1}_\uparrow(s).$$

[3]A person who likes to live dangerously can drop the asterisk; we use it to distinguish between typographic errors where an arrow was accidentally not written but should have been and cases where the arrow does not matter. Personal experience shows that this distinction is indeed useful, especially if one overlooks the error in the statement of a lemma and then proceeds to use the lemma as if it held for both machines.

4.11.1 Ordered Schedules

We will now define a set of schedules for which we will show that they are reduced. These schedules never execute global steps ($G_M(t)$, page 141) while a shared write is buffered by a processor in strong memory mode (except for global steps that empty that processor's store buffer).

We say the buffer is *dirty* if it has a buffered shared write issued in strong memory mode:

$$dirty_M[s](t,i) \equiv SC_{iM}[s](t) \wedge \exists t' \in issue_M[s]^t(i).Sh_M[s](t').$$

We say *the configuration at t is clean* if no write buffers are dirty at the beginning of step t

$$clean_M[s](t) \equiv \forall i.\neg dirty_M[s](t,i).$$

Lemma 272. *When the schedule is k reduced*

$$s \in \mathrm{ABS}_k,$$

we can ignore the machine-type index for dirtiness and cleanness.

$$\begin{aligned} dirty_\downarrow(k,i) &= dirty_\uparrow(k,i), \\ clean_\downarrow(k) &= clean_\uparrow(k). \end{aligned}$$

Proof. With Lemmas 267 to 269 the claims follow

$$\begin{aligned} dirty_\downarrow(k,i) &= SC_{i\downarrow}(t) \wedge \exists t' \in issue^t_\downarrow(i).Sh_\downarrow(t') \\ &= SC_{i\uparrow}(t) \wedge \exists t' \in issue^t_\uparrow(i).Sh_\uparrow(t') \\ &= dirty_\uparrow(k,i), \\ clean_\downarrow(k) &= \forall i.\neg dirty_\downarrow(k,i) \\ &= \forall i.\neg dirty_\uparrow(k,i) \\ &= clean_\uparrow(k). \end{aligned}$$

□

We may thus also drop the machine-type index in these definitions when the schedule is reduced.

A step is *ordered* if a dirty buffer prevents the step from being global unless the step is draining the dirty buffer

$$ord_M[s](t) \equiv \forall i.dirty_M[s](t,i) \wedge G_M[s](t) \rightarrow s(t) \in \Sigma_{WB,i}.$$

Therefore, the order of shared write buffer steps is exactly the order of the processor steps that issue them. Furthermore, while a shared write is buffered, different processors may have a different view of memory. However, since every shared read is a global step, other processors can not notice this difference. Note that the condition allows situations where multiple processors in strong memory mode have shared writes in their buffers, since adding a write to the buffer is not a global step. Note also that in

such a situation each dirty buffer prevents the other buffers from draining their writes, consequently locking the execution in a state where no more global steps are allowed.

Note that steps that are global in the low-level machine are not always global in the high-level machine, and so we may not drop the machine-type index for this definition.

In a clean configuration, all steps are ordered.

Lemma 273.

$$clean_{\downarrow}(t) \rightarrow ord_{\downarrow}(t).$$

Proof. Assume for the sake of contradiction that step t is not ordered. There is some processor i which is dirty

$$dirty_{\downarrow}(t,i),$$

which contradicts the assumption that the configuration at t is clean. □

A schedule is *k-ordered* if steps before k are ordered (in the low-level machine)

$$s \in \mathrm{ORD}_k \equiv \forall t < k.ord_{\downarrow}(t).$$

We consider also a slightly weaker definition that only excludes shared reads, but not shared writes. We call a step during which this is true *semi-ordered*

$$sord_M[s](t) \equiv \forall i.dirty_M[s](t,i) \rightarrow \neg ShR_M[s](t).$$

We define as *k-semi-ordered* a schedule which is $k-1$-ordered, and semi-ordered in step $k-1$

$$s \in \mathrm{ORD}_k^- \equiv s \in \mathrm{ORD}_{k-1} \wedge sord_{\downarrow}[s](k-1).$$

Lemma 274.

$$s \in \mathrm{ABS}_k \rightarrow sord_{\downarrow}(k) = sord_{\uparrow}(k)$$

Proof. By Lemmas 269 and 272

$$\begin{aligned} sord_{\downarrow}(k) &= \forall i.dirty_{\downarrow}(k,i) \rightarrow \neg ShR_{\downarrow}(k) \\ &= \forall i.dirty_{\uparrow}(k,i) \rightarrow \neg ShR_{\uparrow}(k) \\ &= sord_{\uparrow}(k). \end{aligned}$$

□

We may also drop the machine-type index in this definition.

While the last step of a semi-ordered schedule may break the ordering of writes and therefore cause an inconsistency in the memory configurations, the step will still at least not see any memory inconsistencies, and thus be executed correctly.

Ordered steps are semi-ordered

Lemma 275.

$$ord_{\downarrow}(t) \rightarrow sord_{\downarrow}(t)$$

Proof. Assume for the sake of contradiction that the step is ordered but dirty and a shared read

$$dirty_{\downarrow}(t,i) \wedge ShR_{\downarrow}(t).$$

By definition it is not a write buffer step, in particular not of i

$$s(t) \notin \Sigma_{WB,i}.$$

It is also global because it is a shared read

$$G_{\downarrow}(t),$$

which contradicts the definition of *ord*. □

We state a few trivial consequences of this Lemma but do not give proofs

Lemma 276.

$$s \in \mathrm{ORD}_{k+1} \rightarrow sord_{\downarrow}(k).$$

Lemma 277.

$$\mathrm{ORD}_k \subseteq \mathrm{ORD}_k^-.$$

Lemma 278. *Let in a $k+1$-semi-ordered and k-abstract schedule $s \in \mathrm{ORD}_{k+1}^- \cap \mathrm{ABS}_k$ processor i in sequentially consistent mode*

$$SC_{i\uparrow}(k)$$

have a buffered write from t at k

$$t \in issue_{\uparrow}^{k}(i).$$

It can not be the case that step t is shared and step k is a shared read

$$\neg(Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(k)).$$

Proof. Assume for the sake of contradiction that the steps are both marked as shared resp. a shared read

$$Sh_{\uparrow}(t) \wedge ShR_{\uparrow}(k).$$

We immediately obtain from the hypothesis that step t is buffered that the buffer is dirty

$$dirty_{*}(k,i)$$

and thus the step is not semi-ordered

$$\neg sord_{*}(k)$$

which is a contradiction. □

Lemma 279. *If in a schedule that is valid in the high-level machine until $k-1$*

$$\Gamma_{\uparrow}^{k-1}(s)$$

step t is forward-synchronized with k

$$t \rhd k$$

and is made by a processor in strong memory mode which keeps a buffered write at $k+1$[4]*,*

$$s(t) \in \Sigma_{P,i} \wedge SC_{\uparrow}(t) \wedge t' \leq t \wedge t' \in issue_{\uparrow}^{k+1}(i),$$

then step k must be made by the same processor.

$$s(k) \in \Sigma_{P,i}.$$

Proof. By case distinction on the rule used to prove $t \rhd k$.

NOTCONCURRENT:

$$\neg ocon_{\uparrow}(t,k)$$

We obtain that step t is either made by the same object as step k or the outputs of step t modify the core registers of step $k+1$

$$out_{\uparrow}(t) \dot{\cap} C_{\uparrow}(k) \vee o_{\uparrow}(t) = o_{\uparrow}(k).$$

We distinguish between those two cases.

$o_{\uparrow}(t) = o_{\uparrow}(k)$**:** The object making step t is processor i and the claim follows

$$s(k) \in \Sigma_{P,i}.$$

$out_{\uparrow}(t) \dot{\cap} C_{\uparrow}(k)$**:** Since we are not in the case above, assume that the steps are made by different objects

$$o_{\uparrow}(t) \neq o_{\uparrow}(k)$$

and therefore step k is not made by processor i

$$s(k) \notin \Sigma_{P,i}.$$

Since t' is still an issued time stamp at $k+1$ and buffers are monotone, t' must have been an issued time stamp at $t+1$

$$t' \in issue_{\uparrow}^{t+1}(i).$$

By Lemma 123 we obtain that the write of step t' was buffered in the write buffer at $t+1$

$$BW_{\uparrow}(t') \in wb_{\uparrow}^{t+1}(i),$$

which was therefore non-empty

$$wb_{\uparrow}^{t+1}(i) \neq \emptyset.$$

By contraposition of Condition Switch the mode registers of unit i are not outputs of step t

$$A_{SC,i} \notin out_{\uparrow}(t).$$

Since there is an intersection between the outputs of step t and the core registers of step k, the core registers of step k can not be only the memory mode registers of unit i

$$C_{\uparrow}(k) \neq A_{SC,i}$$

[4]If the write is only kept at k, step k might also be the write buffer step that commits the write (synchronized by Rule ISSUEWRITE).

and thus the step can not be made by the write buffer of unit i

$$s(k) \notin \Sigma_{WB,i}.$$

Since step t is made by processor i and step k is neither made by processor i nor its write buffer, step k is made by a different unit j

$$j = u_{\uparrow}(k) \neq u_{\uparrow}(t) = i.$$

The core registers of step k are processor registers of unit j

$$C_{\uparrow}(k) \subseteq A_{PR,j}$$

and thus step t modifies processor registers of unit j

$$out_{\uparrow}(k) \dot{\cap} A_{PR,j}.$$

Since the steps are made by different units, we obtain with Lemma 94 that step t is modifying interrupt registers

$$out_{\uparrow}(k) \dot{\cap} A_{IPR}.$$

By Lemma 211 the step is not buffering a write

$$BW_{\uparrow}(t) = \emptyset.$$

Since t' is a buffered write, we obtain that t' is not t and thus before it

$$t' < t.$$

By Lemma 123 we obtain that the write of step t' was buffered in the write buffer at t

$$BW_{\uparrow}(t') \in wb^{t}_{\uparrow}(i),$$

and thus the write buffer was non-empty at t

$$wb^{t}_{\uparrow}(i) \neq \varepsilon.$$

With Lemma 133 we obtain that a step that is modifying interrupt processor registers is shared

$$Sh_{\uparrow}(t),$$

and since it is modifying interrupt registers it clearly is not only modifying normal processor registers

$$out_{\uparrow}(t) \not\subseteq A_{NPR,i}$$

and is thus a memory write

$$mwrite_{\uparrow}(t).$$

With Lemma 178 we obtain that step t is buffering a write

$$BW_{\uparrow}(t) \neq \emptyset,$$

which is a contradiction.

PROCESSORFENCE:

$$s(t) \in \Sigma_{WB,i} \wedge s(t) \in \Sigma_{P,i}$$

Contradicts the assumption that step t is a processor step.

ISSUEWRITE:

$$t = hd(issue^k_\uparrow(i)) \wedge s(k) \in \Sigma_{WB,i}$$

Since buffers are monotone and $t' \leq t$, t' must have left the buffer by the time t left the buffer

$$t' \notin issue^{k+1}_\uparrow(i),$$

which contradicts the assumption.

□

In an ordered schedule, a processor with buffered writes is not synchronized-with other units.

Lemma 280. *In an ordered, reduced schedule*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1}$$

which is valid and IPI-valid until k

$$\Gamma^k_*(s) \wedge \Delta^k_{IPI*}(s),$$

a step of a strong memory mode processor with buffered writes is not far synchronized-with other units

$$s(t) \in \Sigma_{P,i} \wedge SC_\uparrow(t) \wedge t \in issue^{k+1}_\uparrow(i) \wedge t \blacktriangleright^* k \rightarrow s(k) \in \Sigma_{P,i}.$$

Proof. By induction on the number of $\blacktriangleright$ in $t \blacktriangleright^* k$. In the base case, $t = k$ and the claim follows.

In the inductive step we have

$$t \blacktriangleright^* k \blacktriangleright k'$$

where the induction hypothesis holds for $t \blacktriangleright^* k$, and we have to show the claim for $t \blacktriangleright^* k'$.

Since the buffer is monotonic, t must have been buffered after k

$$t \in issue^{k+1}_\uparrow(i)$$

and by the induction hypothesis we conclude k was a step of processor i

$$s(k) \in \Sigma_{P,i}.$$

By Lemma 168 we also have that the mode is unchanged

$$SC_{i\uparrow}(k) = SC_{i\uparrow}(k') = 1,$$

and can use the conditions about TSO.

We distinguish now the cases for $k \blacktriangleright k'$.

$k \triangleright k'$: The claim is simply Lemma 279 with $t' := t$, $t := k$, and $k := k'$.

COMMEXT:

$$VR_{\uparrow}(k,k') \wedge Sh_{\uparrow}(k) \wedge ShR_{\uparrow}(k')$$

By Lemma 158, there is also a write-read race between the two steps

$$WR_{\uparrow}(k,k').$$

Assume for the sake of contradiction that step k' is not a step of processor i

$$s(k') \notin \Sigma_{P,i}.$$

Because the step is a shared read, by definition it is not a write buffer step, in particular not of processor i

$$s(k') \notin \Sigma_{WB,i}.$$

Since step k' is neither a processor nor a write buffer step of unit i, it is not made by unit i

$$u_{\uparrow}(k') \neq i$$

and the steps are made by different units

$$diffu(k,k').$$

With Lemma 159 we obtain that step k is a memory write

$$mwrite_{\uparrow}(k).$$

By Lemma 278 it now suffices to show that k is an issued timestamp at k'

$$k \overset{!}{\in} issue_{\uparrow}^{k'}(i).$$

The proof distinguishes whether t is k or before it.

$t = k$**:** The claim is an assumption

$$t \in issue_{\uparrow}^{k'}(i).$$

$t < k$**:** By the monotonicity of the buffer, t is an issued timestamp at k

$$t \in issue_{\uparrow}^{k}(i).$$

We conclude with Lemma 123 that the write issued at t is still buffered at k

$$BW_{\uparrow}(t) \in wb_{\uparrow}^{k}(i)$$

and thus the buffer is not empty at k

$$wb_{\uparrow}^{k}(i) \neq \emptyset.$$

With Lemma 178 we obtain that step k buffered a write

$$BW_{\uparrow}(k) \neq \emptyset,$$

and by the monotonicity of the list of issued timestamps, that shared buffered write is still buffered at k'

$$k \in issue_{\uparrow}^{k'}(i).$$

INTERRUPTED:

$$mwrite_{\uparrow}(k) \wedge int_{\uparrow}(k', k)$$

Unit i is a victim of step k'

$$i \in victims_{\uparrow}(k').$$

Because the drain condition for IPI holds, the write buffer of i is the empty list

$$wb_{\uparrow}^{k'}(i) = \varepsilon,$$

and by Lemma 123 this holds also for the list of issued writes

$$issue_{\uparrow}^{k'}(i) = \varepsilon.$$

This clearly contradicts the fact that t is still issued after the step

$$t \notin issue_{\uparrow}^{k'+1}(i).$$

RMWDISABLED:

$$ShR_{\uparrow}(k) \wedge mwrite_{\uparrow}(k) \wedge RW_{\uparrow}(k, k')$$

By Lemma 129 step k is shared

$$Sh_{\uparrow}(k)$$

Since k is shared and the buffer of processor i is non-empty, we obtain by contraposition of Condition MessagePassing that there are no bypassing prepared writes except for the normal processor registers

$$Dom(PW_{\uparrow}(k).bpa) \subseteq A_{NPR,i}.$$

Since the step is a memory write, it has outputs other than the normal processor registers

$$out_{\uparrow}(k) \not\subseteq A_{NPR,i}$$

and thus the inclusive device closure of the domain of the executed writes is not contained in the set of normal processor registers

$$idc(Dom(W_{\uparrow}(k))) \not\subseteq A_{NPR,i}.$$

With Lemma 16 and the fact that the set of normal processor registers is closed (Lemma 4) we can drop the inclusive device closure and obtain that the domain of the executed writes is subsumed by the normal processor registers

$$Dom(W_{\uparrow}(k)) \not\subseteq A_{NPR,i}.$$

Since the step is a sequentially consistent processor step we also know that the buffered portion of the prepared writes are executed

$$W_{\uparrow}(k) = PW_{\uparrow}(k).bpa \cup PW_{\uparrow}(k).wba,$$

and since the domain of the bypassing writes is contained in the set of normal processor registers, the domain of the executed writes are contained in the set of normal processor registers and the domain of the buffered prepared writes

$$Dom(W_{\uparrow}(k)) \subseteq A_{NPR,i} \cup Dom(PW_{\uparrow}(k).wba).$$

Since the domain of the executed writes is subsumed by the set of normal processor registers and the domain of the buffered prepared writes, we obtain that these are also not contained in the set of of normal processor registers

$$A_{NPR,i} \cup Dom(PW_\uparrow(k).wba) \not\subseteq A_{NPR,i}.$$

Thus the domain of the prepared buffered writes and thus the buffered writes is non-empty

$$Dom(BW_\uparrow(k)) = Dom(PW_\uparrow(k).wba) \neq \emptyset.$$

By the contraposition of Condition AtomicRMW we obtain that step k is not a shared read

$$\neg ShR_\uparrow(k),$$

which is a contradiction.

□

If the steps are far forward-synchronized, we get the claim even if the last step is not abstract or valid or ordered.

Lemma 281. *In an ordered, reduced schedule*

$$s \in \mathrm{ABS}_k \cap \mathrm{ORD}_k$$

which is valid and IPI valid until $k-1$

$$\Gamma_*^{k-1}(s) \wedge \Delta_{IPI*}^{k-1}(s),$$

a step of a strong memory mode processor with buffered writes is not far forward-synchronized with other units

$$s(t) \in \Sigma_{P,i} \wedge SC_\uparrow(t) \wedge t \in issue_\uparrow^k(i) \wedge t \blacktriangleright\!\!\blacktriangleright k \rightarrow s(k) \in \Sigma_{P,i} \cup \Sigma_{WB,i}.$$

Proof. Assume for the sake of contradiction that the step is a step of another unit

$$s(k) \notin \Sigma_{P,i} \cup \Sigma_{WB,i}.$$

Thus the operation performed by step k is a noop

$$Op_{i\uparrow}(k) = noop$$

and the write is still buffered at $k+1$

$$t \in issue_\uparrow^k(i) = issue_\uparrow^{k+1}(i).$$

We get that there is some k' with which t is synchronized and which is forward-synchronized with k

$$t \blacktriangleright^* k' \rhd k.$$

Clearly k' is before k and t is less than or equal to k'

$$t \leq k' < k,$$

and by monotonicity of the buffers t is already buffered at $k'+1$

$$t \in issue_\uparrow^{k'+1}(i).$$

Furthermore, the steps until k' are valid, and the schedule is $k'+1$-abstract and -ordered

$$s \in \mathrm{ABS}_{k'+1} \cap \mathrm{ORD}_{k'+1} \wedge \Gamma_{\downarrow}^{k'}(s).$$

We apply Lemma 280 with $t := t$, $k := k'$ and obtain that step k' was made by the same processor

$$s(k') \in \Sigma_{P,i}.$$

With Lemma 279 with $t' := t$, $t := k'$, and $k := k$ we obtain that step k is also made by the same processor

$$s(k) \in \Sigma_{P,i},$$

which is a contradiction. □

To use this lemma, we need a lemma similar to Lemma 258 where t is only not far forward-synchronized with k, but may be far race-synchronized. In this case the original race between t and k may be facilitated by some step l which is race-synchronized with k, and it is no longer necessary for t to be shared. In case step t issued a buffered write that has not been drained, we still obtain that the buffer must be dirty at k from a write that was issued at t or later.

Lemma 282. *In a k-ordered, k-abstract schedule*

$$s \in \mathrm{ABS}_k \cap \mathrm{ORD}_k$$

which is valid and IPI valid until $k-1$

$$\Gamma_*^{k-1}(s) \wedge \Delta_{IPI*}^{k-1}(s)$$

and where a step of a strong memory mode processor with buffered writes is not far forward-synchronized with step k

$$s(t) \in \Sigma_{P,i} \wedge SC_{\uparrow}(t) \wedge t \in issue_{\uparrow}^{k}(i) \wedge t \not\blacktriangleright k,$$

all of the following hold

1. *If there is a code modification, the buffer is dirty and step k is a shared read*

$$CM_{\uparrow}(t,k) \rightarrow \exists l \in issue_{\uparrow}^{k}(i).l \geq t \wedge Sh_{\uparrow}(l) \wedge ShR_{\uparrow}(k).$$

2. *If there is a valid write-read race, the buffer is dirty and step k is a shared read*

$$WR_{\uparrow}(t,k) \wedge \Gamma_{\uparrow}(k) \rightarrow \exists l \in issue_{\uparrow}^{k}(i).l \geq t \wedge Sh_{\uparrow}(l) \wedge ShR_{\uparrow}(k).$$

3. *If there is a valid write-write race, the buffer is dirty and step k is shared*

$$WW_{\uparrow}(t,k) \wedge \Gamma_{\uparrow}(k) \rightarrow \exists l \in issue_{\uparrow}^{k}(i).l \geq t \wedge Sh_{\uparrow}(l) \wedge Sh_{\uparrow}(k).$$

Proof. By Lemma 168 we obtain that the processor is still in strong memory mode

$$SC_{i\uparrow}(k) = SC_{i\uparrow}(t) = SC_{\uparrow}(t) = 1.$$

We now distinguish whether there is also no far race-synchronization.

$t \not\blacktriangleright\!\!\blacktriangleright k$**:** To show that the buffer of unit i is dirty, we show that $l := t$ is the shared write, and the claims follow with Lemma 258.

$t \blacktriangleright\!\!\blacktriangleright k$**:** By definition of $\blacktriangleright\!\!\blacktriangleright$, step t is synchronized with some l, which is race-synchronized with k

$$t \blacktriangleright^* l \blacktriangleright\!\!\blacktriangleright k.$$

Note that l is before k

$$l < k$$

and thus the schedule is l-abstract, l-ordered, valid until l, satisfies the IPI drain condition until l

$$s \in \mathrm{ABS}_l \cap \mathrm{ORD}_l \wedge \Gamma^l_*(s) \wedge \Delta^l_{IPI*}(s)$$

and with Lemma 280 we obtain that l is made by processor i

$$s(l) \in \Sigma_{P,i}.$$

Since t is by assumption not far forward-synchronized with k, we immediately obtain that l can not be forward-synchronized with k

$$l \not\blacktriangleright k,$$

and thus the race-synchronization must have been established by Rule CommExt

$$VR_\uparrow(l,k) \wedge Sh_\uparrow(l) \wedge ShR_\uparrow(k).$$

By Lemma 129 step k is also shared

$$Sh_\uparrow(k),$$

and it only remains to be shown that the buffer of unit i is dirty at k.

We show that $l := l$ is the shared write and distinguish whether the sequence of synchronizations between t and l is empty or not, i.e., whether $t = l$ or $t < l$.

$t = l$**:** The claims immediately follow.

$t < l$**:** By the monotonicity of the buffers, the write issued in step t is in the buffers at l

$$t \in issue^l_\uparrow(i)$$

and by Lemma 168 we obtain that step l is made in strong memory mode

$$SC_\uparrow(l) = SC_{i\uparrow}(l) = SC_{i\uparrow}(t) = SC_\uparrow(t) = 1.$$

By contraposition of Rule NotConcurrent we obtain that the steps are object-concurrent

$$ocon_\uparrow(l,k)$$

and since there is a visible write-read race, there is a write-read race

$$WR_\uparrow(l,k).$$

By Lemma 160 we obtain that step l is a memory write

$$mwrite_\uparrow(l).$$

By Lemma 123 we obtain that the write issued at t is indeed in the buffer

$$BW_{\uparrow}(t) \in wb^{l}_{\uparrow}(i)$$

which is thus non-empty

$$wb^{l}_{\uparrow}(i) \neq \varepsilon.$$

Since the processor is buffering a write and is in strong memory mode, it can by Lemma 178 only execute a shared write if that write is buffered as well

$$BW_{\uparrow}(l) \neq \emptyset,$$

and the write issued by step l is buffered at $l+1$

$$l \in issue^{l+1}_{\uparrow}(i).$$

By the monotonicity of write buffers this shared write is not committed before the write issued at t, and since the write issued at t is still buffered at k, so is this write

$$l \in issue^{k}_{\uparrow}(i),$$

and the claim follows.

□

This lemma is extremely useful and versatile, since it eliminates many races outright, or at least shows that they need synchronization; and as we have shown in Lemma 281, synchronization is extremely limited in ordered schedules.

Note that the conclusion of Cases 1. and 2. of Lemma 282 are never satisfied when the processor is in strong memory mode and step k is semi-ordered, as the following lemma shows.

Lemma 283. *If a processor is in strong memory mode and the step is semi-ordered*

$$SC_{i\uparrow}(k) \wedge sord_{\uparrow}(k),$$

there is no shared write in the buffer of that processor issued after step t in case step k is a shared read

$$\nexists l \in issue^{k}_{\uparrow}(i).l \geq t \wedge Sh_{\uparrow}(l) \wedge ShR_{\uparrow}(k).$$

Proof. Straightforward

$$\begin{aligned}
sord_{\uparrow}(k) &\implies dirty_{\uparrow}(k,i) \rightarrow \neg ShR_{\uparrow}(k) \\
&\iff \neg(dirty_{\uparrow}(k,i) \wedge ShR_{\uparrow}(k)) \\
&\iff \neg(SC_{i\uparrow}(k) \wedge (\exists l \in issue^{k}_{\uparrow}(i).Sh_{\uparrow}(l)) \wedge ShR_{\uparrow}(k)) \\
&\iff \neg((\exists l \in issue^{k}_{\uparrow}(i).Sh_{\uparrow}(l)) \wedge ShR_{\uparrow}(k)) \\
&\iff \nexists l \in issue^{k}_{\uparrow}(i).Sh_{\uparrow}(l) \wedge ShR_{\uparrow}(k). \\
&\implies \nexists l \in issue^{k}_{\uparrow}(i).l \geq t \wedge Sh_{\uparrow}(l) \wedge ShR_{\uparrow}(k).
\end{aligned}$$

□

Lemma 284. *Let in a k+1-ordered, k+1-abstract schedule $s \in \mathrm{ORD}_{k+1} \cap \mathrm{ABS}_{k+1}$ which is valid until k and IPI-valid until k*

$$\Gamma_*^k(s) \wedge \Delta_{IPI*}^{k-1}(s)$$

distinct processors i and j

$$i \neq j$$

have a buffered sequentially consistent write from t resp. t' at k+1

$$SC_\uparrow(t) \wedge t \in issue_\uparrow^k(i),$$
$$SC_\uparrow(t') \wedge t' \in issue_\uparrow^k(j).$$

Then if the writes overlap both buffers are dirty

$$out_\uparrow(t) \owedge out_\uparrow(t') \rightarrow dirty_\uparrow(k,i) \wedge dirty_\uparrow(k,j).$$

Proof. Without loss of generality t was before t'

$$t < t'.$$

Clearly t' to be buffered at k must be less than k

$$t' < k$$

and the schedule is also valid and IPI-valid until t'

$$\Gamma_*^{t'}(s) \wedge \Delta_{IPI*}^{t'-1}(s)$$

Because t' issued a write for processor j

$$t' \in issue_\uparrow^k(j),$$

it must have been a step of processor j

$$s(t') \in \Sigma_{P,j}$$

and thus not a step of processor i

$$s(t') \notin \Sigma_{P,i}$$

or of the write buffer of i

$$s(t') \notin \Sigma_{WB,i}.$$

We obtain also by the monotonicity of write buffers that t is still buffered at t'

$$t \in issue_\uparrow^{t'}(i),$$

and by contraposition of Lemma 281 with $k := t'$ step t is not far forward-synchronized with step t'

$$t \not\blacktriangleright t'.$$

We obtain from the hypothesis that there is a write-write race between steps t and t'

$$WW_\uparrow(t,t').$$

With Lemma 282 for $k := t'$ we obtain that there is a shared write from $l \geq t$ which is still being buffered and step t' is shared

$$l \in issue^{t'}_{\uparrow}(i) \wedge Sh_{\uparrow}(l) \wedge Sh_{\uparrow}(t').$$

By the monotonicity of write buffers, the write issued at l is not committed before the write issued at t, which is still buffered at k

$$l \in issue^{k}_{\uparrow}(i).$$

With Lemma 168 we obtain that the mode is unchanged and thus both units are in strong memory mode at k

$$SC_{i\uparrow}(k) = SC_{i\uparrow}(t) = SC_{\uparrow}(t) = 1,$$
$$SC_{i\uparrow}(k) = SC_{i\uparrow}(t') = SC_{\uparrow}(t') = 1.$$

The claim follows

$$dirty_{\uparrow}(k,i) \wedge dirty_{\uparrow}(k,j).$$

□

Lemma 285. *In a $k+1$-abstract, $k+1$-ordered schedule that is valid until k*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma^{k}_{*}(s),$$

no sequentially consistent processor is buffering writes that it is also modifying in the low-level machine

$$\forall i. SC_{i\uparrow}(k) \wedge hit(out_{\downarrow}(k), wb^{k}_{\uparrow}(i)) \rightarrow s(k) \notin \Sigma_{P,i}.$$

Proof. Assume for the sake of contradiction that the processor is overwriting its own buffered writes

$$s(k) \in \Sigma_{P,i}.$$

By Condition WriteWriteOrder, the step k was not preparing to directly overwrite its own buffered writes in the high-level machine

$$\neg hit(idc(Dom(PW_{\uparrow}(k).bpa)), wb^{k}_{\uparrow}(i)).$$

By Lemma 270, the steps are preparing the same writes, which in the low-level machine are the executed writes

$$PW_{\uparrow}(k).bpa = PW_{\downarrow}(k).bpa = W_{\downarrow}(k).$$

We obtain that the low-level machine was not overwriting buffered writes

$$\neg hit(idc(Dom(W_{\downarrow}(k))), wb^{k}_{\uparrow}(i)),$$

which results in a contradiction

$$\neg hit(out_{\downarrow}(k), wb^{k}_{\uparrow}(i)).$$

□

It follows that in an ordered schedule, no sequentially consistent processor is buffering writes to an address which is modified during that step.

Lemma 286. *In a $k+1$-abstract, $k+1$-ordered schedule that is valid until k and IPI-valid until $k-1$*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma^k_*(s) \wedge \Delta^{k-1}_{IPI*}(s),$$

no sequentially consistent processor is buffering writes to what the low-level machine is modifying (except if the low-level machine is committing a write from that buffer)

$$\forall i. SC_{i\uparrow}(k) \wedge s(k) \notin \Sigma_{WB,i} \rightarrow \neg hit(out_\downarrow(k), wb^k_*(i)).$$

Proof. Assume for the sake of contradiction that there is a sequentially consistent processor which is buffering such writes

$$SC_{i\uparrow}(k) \wedge s(k) \notin \Sigma_{WB,i} \wedge hit(out_\downarrow(k), wb^k_*(i)).$$

By Lemma 127 that hit must have been issued at some point t

$$t \in issue^k_*(i) \wedge Dom(BW_*(t)) \hat{\cap} out_\downarrow(k).$$

With Lemma 168 we obtain that the processor has already been in sequentially consistent mode at that point

$$SC_{i\uparrow}(t).$$

By Lemma 92 the outputs of step t in the high-level machine include the domain of the buffered write

$$Dom(BW_*(t)) \subseteq out_\uparrow(t).$$

Therefore there is an intersection between the outputs of step t in the high-level machine and the outputs of step k in the low-level machine

$$out_\uparrow(t) \hat{\cap} out_\downarrow(k). \tag{4.17}$$

By Lemma 285, step k can also not be a step of processor i

$$s(k) \notin \Sigma_{P,i}$$

and is thus made by a different unit

$$u_\downarrow(k) \neq i.$$

On the other hand, step k is a memory write by Lemma 122

$$mwrite_\downarrow(k).$$

It is thus global if it is shared

$$Sh_\downarrow(k) \rightarrow G_\downarrow(k).$$

Since the schedule is ordered and step k is not made by the write buffer of unit i, it can not both be true that the buffer of i is dirty and that step k is global in the low-level machine

$$\neg(dirty_*(k,i) \wedge G_\downarrow(k))$$

and thus it can also not be true that the buffer is dirty and step k is shared in the low-level machine

$$\neg(dirty_*(k,i) \wedge Sh_\downarrow(k)).$$

By Lemma 269 for $X := Sh$ we obtain the same for the high-level machine

$$\neg(dirty_*(k,i) \wedge Sh_\uparrow(k)).$$

We now distinguish between two cases: either step k is a sequentially consistent write buffer step that is committing a buffered write, or something else.

$s(k) \in \Sigma_{WB,j} \wedge SC_{\downarrow}(k)$**:** In this case step k is just executing in the low-level machine the head of the write buffer

$$W_{\downarrow}(k) = hd(wb^k_{\downarrow}(j)),$$

which by Lemma 267 is the head of the write buffer in the high-level machine

$$W_{\downarrow}(k) = hd(wb^k_{\uparrow}(j)).$$

By Lemma 123 that write was issued in some step t'

$$t' = hd(issue^k_{\uparrow}(j)) \wedge hd(wb^k_{\uparrow}(j)) = BW_{\uparrow}(t').$$

By Lemma 168, step t' was made in sequentially consistent mode

$$SC_{\uparrow}(t') \equiv SC_{j\uparrow}(t') \equiv SC_{j\uparrow}(k) \equiv SC_{\uparrow}(k) \equiv 1.$$

By Lemma 92, the buffered writes of step t' are outputs at t' in the high-level machine

$$Dom(BW_{\uparrow}(t')) \subseteq out_{\uparrow}(t'),$$

and thus by Lemma 26 the outputs of the low-level machine are contained in the inclusive device closure of the outputs of the high-level machine

$$\begin{aligned} out_{\downarrow}(k) &= idc(Dom(W_{\downarrow}(k))) \\ &= idc(Dom(hd(wb^k_{\uparrow}(j)))) \\ &= idc(Dom(BW_{\uparrow}(t'))) \\ &\subseteq idc(out_{\uparrow}(t')) \qquad \text{L 26} \end{aligned}$$

and since the outputs are closed under devices, that is exactly the outputs

$$= out_{\uparrow}(t').$$

With Eq. (4.17) we obtain that the outputs of t and t' overlap

$$out_{\uparrow}(t) \dot{\cap} out_{\uparrow}(t')$$

and by Lemma 284 the buffer of processor i is dirty

$$dirty_{\uparrow}(k,i).$$

Note that step k is shared by definition of Sh since it is a write-buffer step

$$Sh_{\uparrow}(k),$$

which leads to a contradiction

$$dirty_{\uparrow}(k,i) \wedge Sh_{\uparrow}(k).$$

$\neg(s(k) \in \Sigma_{WB,j} \wedge SC_{\downarrow}(k))$**:** By Lemma 265, the outputs in the low-level machine are a subset of those in the high-level machine

$$out_{\downarrow}(k) \subseteq out_{\uparrow}(k)$$

and therefore with Statement (4.17) there is an intersection between the outputs of steps t and k in the high-level machine

$$out_{\uparrow}(t) \hat{\cap} out_{\uparrow}(k),$$

i.e., a write-write race

$$WW_{\uparrow}(t,k).$$

Recall that step k is made by a different unit than unit i, and thus by contraposition of Lemma 281 step t is not far forward-synchronized with step k

$$t \not\blacktriangleright k$$

With Lemma 282 we obtain that there is some step l which buffers a shared write for unit i and that step k is shared

$$l \in issue^{k}_{\uparrow}(i) \wedge Sh_{\uparrow}(l) \wedge Sh_{\uparrow}(k),$$

and we conclude that the buffer of unit i is dirty and step k is shared

$$dirty_{*}(k,i) \wedge Sh_{\uparrow}(k),$$

which is a contradiction.

□

Lemma 287. *If in a $k+1$-abstract schedule $s \in \mathrm{ABS}_{k+1}$ step k is only in strong memory mode if it is made by a processor that will not be buffering writes to a set of addresses A*

$$SC_{\uparrow}(k) \rightarrow \exists j.s(k) \in \Sigma_{P,j} \wedge \neg hit(A, wb^{k+1}_{\uparrow}(j))$$

then the writes of the two machines in that step agree on A

$$W_{\uparrow}(k) =_{A} W_{\downarrow}(k).$$

Proof. We distinguish between strong and weak memory mode

$SC_{\uparrow}(k)$**:** By assumption there is a processor j that is not buffering writes to A

$$s(k) \in \Sigma_{P,j} \wedge \neg hit(A, wb^{k+1}_{\uparrow}(j)).$$

By Lemma 270, the machines are preparing the same writes

$$PW_{\uparrow}(k) = PW_{\downarrow}(k).$$

However, the low-level machine only executes the bypassing portion of the write, whereas the high-level machine also executes the bypassing portion

$$\begin{aligned} W_{\uparrow}(k) &= PW_{\uparrow}(k).wba \cup PW_{\uparrow}(k).bpa, \\ &\stackrel{!}{=}_{A} PW_{\uparrow}(k).bpa \\ &= PW_{\downarrow}(k).bpa \\ &= W_{\downarrow}(k). \end{aligned}$$

It thus suffices to show that the buffered portion of the write has no effect on A

$$Dom(PW_\uparrow(k).wba) \not\overset{!}{\cap} A.$$

Assume for the sake of contradiction that it does

$$Dom(PW_\uparrow(k).wba) \overset{!}{\cap} A.$$

Thus clearly the prepared buffered write, which is also the buffered write, is non-empty

$$BW_\uparrow(k) = PW_\uparrow(k).wba \neq \emptyset$$

and the operation of the step is a push operation

$$Op_{j\uparrow}(k) = push.$$

Since the buffered write intersects A, there is a hit in the singleton list containing only the buffered write

$$hit(A, BW_\uparrow(k)),$$

and by Lemma 51 also in the write buffer at k concatenated with that write

$$hit(A, wb^k_\uparrow(j) \circ BW_\uparrow(k)).$$

Since the operation of the step is a push operation, that is the buffer at $k+1$

$$hit(A, wb^{k+1}_\uparrow(j)),$$

which is a contradiction.

$\neg SC_\uparrow(k)$**:** The claim follows immediately with Lemma 264.

□

We say that a set of addresses A is *up to date for processor i* at k when there is no other unit in strong memory mode that has outstanding writes to that address

$$utd_i(k,A) \equiv \forall j \neq i.SC_{j\uparrow}(k) \rightarrow \neg hit(A, wb^k_\uparrow(j)).$$

The only type of step that can make an address up to date is a write buffer step of another unit in strong memory mode.

Lemma 288. *Let in a $k+1$-abstract, $k+1$-ordered schedule that is valid until k*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma^k_*(s)$$

a set of addresses A be up to date for unit i at $k+1$

$$utd_i(k+1, A)$$

and step k not be made by a write buffer in strong memory mode

$$SC_\uparrow(k) \rightarrow \nexists j.s(k) \in \Sigma_{WB,j} \wedge j \neq i.$$

Then A was already up to date at k

$$utd_i(k,A).$$

Proof. Assume for the sake of contradiction that A is up to date at $k+1$

$$\forall j \neq i.SC_{j\uparrow}(k+1) \rightarrow \neg hit(A, wb_{\uparrow}^{k+1}(j)),$$

and that j is another processor in strong memory mode at k which had a hit on A

$$j \neq i \wedge SC_{j\uparrow}(k) \wedge hit(A, wb_{\uparrow}^{k}(j)).$$

Because there is a hit, the write buffer is non-empty

$$wb_{\uparrow}^{k}(j) \neq \varepsilon$$

and by contraposition of Condition Switch, the mode registers are not changed

$$A_{SC,j} \not\upharpoonright out_{\uparrow}(k).$$

We conclude with Lemma 139 that processor j is still in strong memory mode at $k+1$

$$SC_{j\uparrow}(k+1) = SC_{j\uparrow}(k) = 1,$$

and since the addresses are now up to date, no longer has a hit on A in its write buffer

$$\neg hit(A, wb_{\uparrow}^{k+1}(j)).$$

We immediately obtain that the buffers must have changed

$$wb_{\uparrow}^{k+1}(j) \neq wb_{\uparrow}^{k}(j),$$

and distinguish now between the two cases that might have changed the write buffer: a processor step of unit j buffering a write, and a write buffer step of unit j committing one.

$s(k) \in \Sigma_{P,j} \wedge BW_{\uparrow}(k) \neq \emptyset$**:** We immediately obtain with Lemma 51 that the hit has not disappeared

$$\begin{aligned} hit(A, wb_{\uparrow}^{k+1}(j)) &\equiv hit(A, wb_{\uparrow}^{k}(j) \circ BW_{\uparrow}(k)) \\ &\equiv hit(A, wb_{\uparrow}^{k}(j)) \vee \ldots \qquad \text{L 51} \\ &\equiv 1, \end{aligned}$$

which is a contradiction.

$s(k) \in \Sigma_{WB,j}$**:** Clearly the step is made in strong memory mode

$$SC_{\uparrow}(k) = SC_{j\uparrow}(k) = 1,$$

which contradicts our assumption.

□

Even such a step can only make those addresses up to date that it modifies.

Lemma 289. *Let in a $k+1$-abstract, $k+1$-ordered schedule that is valid until k*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma^k_*(s)$$

a set of addresses A be up to date for unit i at $k+1$

$$utd_i(k+1, A)$$

and step k be made by a write buffer of another unit in strong memory mode

$$SC_\uparrow(k) \wedge s(k) \in \Sigma_{WB,j} \wedge j \neq i.$$

Then the portion of A not modified by the step was already up to date at k

$$utd_i(k, A \setminus out_\downarrow(k)).$$

Proof. By assumption processor j was sequentially consistent at k, and by Lemma 174 a sequentially write buffer step has no outputs

$$out_\uparrow(k) = \emptyset$$

and thus by Lemma 138 does not change the memory mode, so processor j is also sequentially consistent at $k+1$

$$SC_{j\uparrow}(k+1) = SC_{j\uparrow}(k) = 1.$$

By assumption, processor j does not have any buffered writes to A at $k+1$

$$\neg hit(A, wb^{k+1}_\uparrow(j)),$$

which is the tail of its write buffer at k

$$wb^{k+1}_\uparrow(j) = tl(wb^k_\uparrow(j)),$$

and with Lemma 48 we obtain that there is no hit for $A \setminus out_\downarrow(k)$ in the tail of the write buffer at k

$$\begin{aligned} hit(A \setminus out_\downarrow(k), tl(wb^k_\uparrow(j))) &\implies hit(A, tl(wb^k_\uparrow(j))) && \text{L 48} \\ &\iff hit(A, wb^{k+1}_\uparrow(j)) \\ &\iff 0. \end{aligned}$$

With Lemma 267 we obtain that the outputs at k subsume the domain of the head of the write buffer at k

$$\begin{aligned} out_\downarrow(k) &= idc(Dom(W_\downarrow(k))) \\ &= idc(Dom(hd(wb^k_\downarrow(j)))) \\ &= idc(Dom(hd(wb^k_\uparrow(j)))) && \text{L 267} \\ &\supseteq Dom(hd(wb^k_\uparrow(j))), \end{aligned}$$

and thus we obtain that there is no hit for $A \setminus out_\downarrow(k)$ in the head of the write buffer at k, either

$$hit(A \setminus out_\downarrow(k), hd(wb^k_\uparrow(j)))$$

$$\iff A \setminus out_{\downarrow}(k) \cap Dom(hd(wb^k_{\uparrow}(j)))$$
$$\iff 0.$$

Since there is a hit neither in the head nor in the tail, with Lemma 51 we obtain that j was not buffering writes to $A \setminus out_{\downarrow}(k)$ at k, just as at $k+1$; therefore the two hit predicates are the same

$$\begin{aligned}
&hit(A \setminus out_{\downarrow}(k), wb^k_{\uparrow}(j)) \\
&= hit(A \setminus out_{\downarrow}(k), hd(wb^k_{\uparrow}(j))) \vee hit(A \setminus out_{\downarrow}(k), tl(wb^k_{\uparrow}(j))) && \text{L 51} \\
&= 0 \vee 0 = 0 \\
&= hit(A \setminus out_{\downarrow}(k), wb^{k+1}_{\uparrow}(j)). && (4.18)
\end{aligned}$$

Since by Lemma 174 a strong memory write buffer step has no outputs

$$out_{\uparrow}(k) = \emptyset,$$

with Lemma 139 we obtain that the step did not change any mode

$$\forall j'.SC_{j'\uparrow}(k+1) = SC_{j'\uparrow}(k). \tag{4.19}$$

Since step k is a step of unit j

$$u_{\uparrow}(k) = j,$$

by Lemma 96 the write buffers of other units are also unchanged

$$\forall j' \neq j.wb^{k+1}_{\uparrow}(j') = wb^k_{\uparrow}(j')) \tag{4.20}$$

and thus we obtain for those processors also that the hit predicate is unchanged

$$\forall j' \neq j.hit(A \setminus out_{\downarrow}(k), wb^{k+1}_{\uparrow}(j')) = hit(A \setminus out_{\downarrow}(k), wb^k_{\uparrow}(j')).$$

Combining this with Equation (4.18) we obtain that the hit predicates are unchanged for all processors

$$\forall j'.hit(A \setminus out_{\downarrow}(k), wb^{k+1}_{\uparrow}(j')) = hit(A \setminus out_{\downarrow}(k), wb^k_{\uparrow}(j')). \tag{4.21}$$

Since the buffers for A are up to date at $k+1$, no other processor in sequential mode is buffering a write to A at k

$$\forall j' \neq i.SC_{j'\uparrow}(k+1) \rightarrow \neg hit(A, wb^{k+1}_{\uparrow}(j'))$$

and by contraposition of Lemma 48 also not to the portion of A not modified by the low-level machine

$$\forall j' \neq i.SC_{j'\uparrow}(k+1) \rightarrow \neg hit(A \setminus out_{\downarrow}(k), wb^{k+1}_{\uparrow}(j')).$$

Since by Eqs. (4.19) and (4.21) the memory mode registers and hits have not changed, we obtain the same at k

$$\forall j' \neq i.SC_{j'\uparrow}(k) \rightarrow \neg hit(A \setminus out_{\downarrow}(k), wb^k_{\uparrow}(j')),$$

which is the claim

$$utd_i(k, A \setminus out_{\downarrow}(k)).$$

□

We say that *the memory views of processor i of the machines are in sync in cycle k on a set of addresses A* when the memory of the high-level machine agrees either with the memory of the low-level machine or the perfect world forwarding memory system of the low-level machine, depending on the memory mode of that processor

$$sync_i(k,A) \equiv m^k_{\uparrow} =_A \begin{cases} m^k_{\downarrow} & \neg SC_{i\uparrow}(k) \\ pfms_{i\downarrow}(k) & SC_{i\uparrow}(k). \end{cases}$$

For a processor that currently has a strong memory view, the writes in the buffer have already been executed in the high-level machine, but not yet in the low-level machine, which is why we use perfect world forwarding. For a processor in weak memory mode, the writes in the buffer are buffered in both machines, and thus the memories are in sync.

When the memory views of a processor are in sync in cycle k on two sets of addresses, they are also in sync on the union. We do not show the proof.

Lemma 290.

$$sync_i(k,A) \wedge sync_i(k,A') \rightarrow sync_i(k,A \cup A').$$

We wish to show that sets of addresses that are up to date are also in sync. We call this the *memory invariant*

$$minv_i(k,A) \equiv utd_i(k,A) \rightarrow sync_i(k,A).$$

We show that if the mode of a processor is changed, the memory invariant is maintained.

Lemma 291. *Let in a $k+1$-abstract, $k+1$-ordered schedule that is valid until k*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma^k_*(s)$$

step k change the memory mode of some processor i

$$SC_{i\uparrow}(k) \neq SC_{i\uparrow}(k+1).$$

Then the memory invariant is maintained on every set of addresses A that is closed under devices

$$closed(A) \wedge minv_i(k,A) \rightarrow minv_i(k+1,A).$$

Proof. Assume that A is closed, the memory invariant holds at k, and A is up to date for processor i at $k+1$

$$closed(A) \wedge minv_i(k,A) \wedge utd_i(k+1,A).$$

Since the addresses are up to date, no other processor in strong memory mode has buffered a write to A

$$\forall j \neq i. SC_{j\uparrow}(k+1) \rightarrow \neg hit(A, wb^{k+1}_{\uparrow}(j)).$$

By Condition Switch the buffers in the high-level machine are empty before and after step k

$$wb^k_{\uparrow}(i) = wb^{k+1}_{\uparrow}(i) = \varepsilon.$$

We conclude that the write buffer of i has not hit on A either

$$\neg hit(A, wb_{\uparrow}^{k+1}(j)),$$

and thus there is no unit in strong memory mode that has such a hit

$$\forall j.SC_{j\uparrow}(k+1) \rightarrow \neg hit(A, wb_{\uparrow}^{k+1}(j)).$$

Furthermore, step k modifies the memory mode and thus by Lemma 139 has outputs

$$A_{SC,i} \mathbin{\hat{\cap}} out_{\uparrow}(k),$$

from which we conclude by contraposition of Lemma 174 that the step is not a sequentially consistent write buffer step

$$SC_{\uparrow}(k) \rightarrow \forall j.s(k) \notin \Sigma_{WB,j}.$$

We conclude that if step k is a sequentially consistent step, it is made by a processor which has no hit for A

$$SC_{\uparrow}(k) \rightarrow \exists j.s(k) \in \Sigma_{P,j} \wedge \neg hit(A, wb_{\uparrow}^{k+1}(j)).$$

By Lemma 287, the executed writes of the machines agree on A

$$W_{\uparrow}(k) =_A W_{\downarrow}(k).$$

We conclude further that step k can not be made by a write buffer of another unit in strong memory mode

$$SC_{\uparrow}(k) \rightarrow \nexists j.s(k) \in \Sigma_{WB,j} \wedge j \neq i,$$

and thus by Lemma 288, the addresses were up to date at k

$$utd_i(k,A)$$

and with the memory invariant we obtain that the memory views were in sync

$$sync_i(k,A).$$

By Lemma 267, the write buffers in the low-level machine are the same as those in the high-level machine and thus also empty

$$wb_{\downarrow}^{k}(i) = wb_{\downarrow}^{k+1}(i) = \varepsilon.$$

Since the write buffers of unit i are empty in both steps $k' \in \{k, k+1\}$, the perfect world forwarding memory system simply equals the memory in both steps

$$pfms_{i\downarrow}(k') = m_{\downarrow}^{k'} \circ \varepsilon = m_{\downarrow}^{k'}.$$

Therefore the memory views are in sync iff the memories agree on A

$$\begin{aligned} sync_i(k',A) \equiv m_{\uparrow}^{k'} &=_A \begin{cases} m_{\downarrow}^{k'} & \neg SC_{i\uparrow}(k') \\ pfms_{i\downarrow}(k') & SC_{i\uparrow}(k') \end{cases} \\ &\equiv m_{\uparrow}^{k'} =_A m_{\downarrow}^{k'}. \end{aligned}$$

Since the memory views are in sync at k we immediately conclude that the memories agree on A

$$m_{\uparrow}^{k} =_{A} m_{\downarrow}^{k}.$$

By Lemma 32, we can update the memory of both configurations only using the portion of the writes on which the machines agree,

$$\begin{aligned} m_{\uparrow}^{k+1} &= m_{\uparrow}^{k} \circledast W_{\uparrow}(k) \\ &=_{A} m_{\uparrow}^{k} \circledast W_{\downarrow}(k) && \text{L 32} \end{aligned}$$

and by Lemma 29 only the portion of the memories on which the machines agree matters, and we conclude the memories at $k+1$ are the same

$$\begin{aligned} &=_{A} m_{\downarrow}^{k} \circledast W_{\downarrow}(k) && \text{L 29} \\ &= m_{\downarrow}^{k+1}, \end{aligned}$$

which is equivalent to the claim

$$sync_{i}(k+1, A).$$

□

Lemma 292. *Let in a $k+1$-abstract, $k+1$-ordered schedule that is valid until k*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma_{*}^{k}(s)$$

a set of addresses A be up to date for processor i at $k+1$

$$utd_{i}(k+1, A).$$

If step k is not sequentially consistent or made by another processor

$$\neg SC_{\uparrow}(k) \vee s(k) \in \Sigma_{P,j} \wedge i \neq j,$$

both of the following hold.

1. *The step is not made by another write buffer in strong memory mode*

$$SC_{\uparrow}(k) \rightarrow \nexists j.s(k) \in \Sigma_{WB,j} \wedge j \neq i,$$

2. *If the step is made in sequentially consistent mode, it is made by a processor with no hit for A*

$$SC_{\uparrow}(k) \rightarrow \exists j.s(k) \in \Sigma_{P,j} \wedge \neg hit(A, wb_{\uparrow}^{k+1}(j)).$$

Proof. Assume for both claims that the step is made in strong memory mode

$$SC_{\uparrow}(k).$$

The step must be made by another processor

$$s(k) \in \Sigma_{P,j} \wedge i \neq j.$$

It is thus not a write buffer step and the first claim follows

$$\nexists j'.s(k) \in \Sigma_{WB,j'} \wedge j' \neq i.$$

That processor is in strong memory mode at k

$$SC_{j\uparrow}(k),$$

and our proof distinguishes between its memory mode at $k+1$.

$SC_{j\uparrow}(k+1)$**:** By assumption, other processors in strong memory mode do not have a hit for A

$$\neg hit(A, wb_{\uparrow}^{k+1}(j))$$

and the claim follows.

$\neg SC_{j\uparrow}(k+1)$**:** Clearly the memory mode has changed, and by contraposition of Lemma 139 the mode registers are an outputs of step k

$$A_{SC,j} \hat{\cap} out_{\uparrow}(k).$$

By Condition Switch, the write buffer of unit j is empty at $k+1$

$$wb_{\uparrow}^{k+1}(j) = \varepsilon$$

and thus has no hit for A

$$\neg hit(A, wb_{\uparrow}^{k+1}(j)),$$

and the claim follows.

□

Sequentially consistent write buffer steps keep the memory views of their unit in sync.

Lemma 293. *Let in a $k+1$-abstract, $k+1$-ordered schedule that is valid until k*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma_*^k(s)$$

step k be a sequentially consistent write buffer step of unit i

$$SC_{\uparrow}(k) \wedge s(k) \in \Sigma_{WB,i}.$$

Then the memory views of that unit are kept in sync and the mode is not changed

$$sync_i(k,A) \rightarrow SC_{i\uparrow}(k+1) \wedge sync_i(k+1,A).$$

Proof. In the high-level machine, the step executes an empty write that does not change the memory

$$m_{\uparrow}^k = m_{\uparrow}^k \circledast \emptyset = m_{\uparrow}^k \circledast W_{\uparrow}(k) = m_{\uparrow}^{k+1},$$

and in the low-level machine the forwarding is unchanged by Lemma 68

$$pfms_{i\downarrow}(k+1) = pfms_{i\downarrow}(k).$$

We immediately conclude that the memory mode is unchanged in the high-level machine

$$SC_{i\uparrow}(k+1) = SC_{i\uparrow}(k) = SC_{\uparrow}(k) = 1.$$

Assume now that the memory views are in sync, i.e., that the forwarding of the low-level machine agrees with the memory of the high-level machine

$$pfms_{i\downarrow}(k) =_A m_{\uparrow}^k.$$

The claim follows

$$pfms_{i\downarrow}(k+1) = pfms_{i\downarrow}(k) =_A m_{\uparrow}^k = m_{\uparrow}^{k+1}.$$

□

The machine is also kept in sync if the mode is not switched, although this is more difficult to prove.

Lemma 294. *Let in a $k+1$-abstract, $k+1$-ordered schedule that is valid until k and IPI-valid until $k-1$*

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma^k_*(s) \wedge \Delta^{k-1}_{IPI*}(s)$$

step k not change the mode of some processor i

$$SC_{i\uparrow}(k) = SC_{i\uparrow}(k+1).$$

Then if the memory invariant holds at k for every processor and closed set of addresses

$$\forall A', i'.closed(A') \rightarrow minv_{i'}(k, A'),$$

it also holds at $k+1$

$$\forall A.closed(A) \rightarrow minv_i(k+1, A).$$

Proof. Clearly, the lemma models the inductive step of an inductive proof. Therefore, we refer to the assumption that the memory invariant holds at k as the induction hypothesis throughout the proof. Unless we specify something else, we always apply the induction hypothesis with $i' := i$ and $A' := A$.

Let A be closed and up-to-date

$$closed(A) \wedge utd_i(k+1, A),$$

and we have to show that the memory is in sync

$$sync_i(k+1, A) \stackrel{!}{=} 1.$$

By Lemmas 267 and 270 we obtain that the machines agree on the buffers, the prepared writes, and the buffered writes

$$\begin{aligned} wb^k_\uparrow &= wb^k_\downarrow, && \text{L 267} \\ wb^{k+1}_\uparrow &= wb^{k+1}_\downarrow, && \text{L 267} \\ PW_\uparrow(k) &= PW_\downarrow(k), && \text{L 270} \\ BW_\uparrow(k) &= BW_\downarrow(k), && \text{L 270} \end{aligned}$$

and for the sake of brevity we will often discard the machine-type index with these terms, simply writing an asterisk in its place, e.g.,

$$wb^k_* = wb^k_\uparrow = wb^k_\downarrow.$$

Clearly, one of the following holds.

1. The step is made in weak memory mode

$$\neg SC_\uparrow(k),$$

2. The step is made in strong memory mode by processor i

$$SC_\uparrow(k) \wedge s(k) \in \Sigma_{P,i},$$

3. The step is made in strong memory mode by another processor $j \neq i$

$$SC_{\uparrow}(k) \wedge s(k) \in \Sigma_{P,j} \wedge j \neq i,$$

4. The step is made in strong memory mode by the write buffer of processor i,

$$SC_{\uparrow}(k) \wedge s(k) \in \Sigma_{WB,i},$$

5. The step is made in strong memory mode by the write buffer of another processor $j \neq i$.

$$SC_{\uparrow}(k) \wedge s(k) \in \Sigma_{WB,j} \wedge j \neq i,$$

We distinguish between those cases, but treat Cases 1 and 3 as one.

Case 1 or Case 3

$$\neg SC_{\uparrow}(k) \vee s(k) \in \Sigma_{P,j} \wedge i \neq j$$

By Lemma 292, if the step is made in strong memory mode, it is not done by the write buffer of another unit

$$SC_{\uparrow}(k) \rightarrow \nexists j.s(k) \in \Sigma_{WB,j} \wedge j \neq i,$$

and by Lemma 288, the addresses were up to date at k

$$utd_i(k,A)$$

and with the induction hypothesis we obtain that the memory view is at sync at k

$$sync_i(k,A).$$

Also by Lemma 292, if the step is made in strong memory mode, it is made by another processor with no hit for A

$$SC_{\uparrow}(k) \rightarrow \exists j.s(k) \in \Sigma_{P,j} \wedge \neg hit(A, wb_{\uparrow}^{k+1}(j)).$$

With Lemma 287 we immediately obtain that the writes of both machines agree on A

$$W_{\uparrow}(k) =_A W_{\downarrow}(k).$$

The proof now distinguishes whether unit i was in strong memory mode or not.

$SC_{i\uparrow}(k)$**:** Because the step is either made in weak memory mode or by another processor, it is not made by the same unit

$$u_{\uparrow}(k) \neq i$$

and by Lemma 96 the write buffer of unit i is not changed by the step

$$wb_*^k(i) = wb_*^{k+1}(i).$$

Step k is in particular not a step of the write buffer of processor i

$$s(k) \notin \Sigma_{WB,i}$$

and by Lemma 286 there is no hit between the outputs of the low-level machine and the write buffer of unit i

$$\neg hit(out_{\downarrow}(k), wb_*^k(i)),$$

and by repeated application of Lemma 30 we can pull the update of the low-level machine behind the buffered writes

$$\begin{aligned} pfms_{i\downarrow}(k+1) &= m_{\downarrow}^{k+1} \circledast wb_*^{k+1}(i) \\ &= m_{\downarrow}^{k} \circledast W_{\downarrow}(k) \circledast wb_*^{k}(i) \\ &= m_{\downarrow}^{k} \circledast wb_*^{k}(i) \circledast W_{\downarrow}(k) && \text{L 30} \\ &= pfms_{i\downarrow}(k) \circledast W_{\downarrow}(k) \end{aligned}$$

and the claim follows with the induction hypothesis and Lemmas 29 and 32

$$\begin{aligned} &=_A m_{\uparrow}^{k} \circledast W_{\downarrow}(k) && \text{IH, L 29} \\ &=_A m_{\uparrow}^{k} \circledast W_{\uparrow}(k) && \text{L 32} \\ &= m_{\uparrow}^{k+1}. \end{aligned}$$

$\neg SC_{i\uparrow}(k)$**:** The claim follows with the induction hypothesis and Lemmas 29 and 32

$$\begin{aligned} m_{\uparrow}^{k+1} &= m_{\uparrow}^{k} \circledast W_{\uparrow}(k) \\ &=_A m_{\downarrow}^{k} \circledast W_{\uparrow}(k) && \text{IH, L 29} \\ &=_A m_{\downarrow}^{k} \circledast W_{\downarrow}(k) && \text{L 32} \\ &= m_{\downarrow}^{k+1}. \end{aligned}$$

Case 2

$$SC_{\uparrow}(k) \wedge s(k) \in \Sigma_{P,i},$$

By Lemma 288, the addresses were up to date at k

$$utd_i(k, A)$$

and with the induction hypothesis we obtain that the memory view is at sync at k

$$sync_i(k, A).$$

Since the step is in strong memory mode, unit i is in strong memory mode

$$SC_{i\uparrow}(k) = SC_{\uparrow}(k) = 1.$$

In the low-level machine, the bypassing portion of the write is executed before the write buffer and the buffered portion is added behind the write buffer

$$\begin{aligned} pfms_{i\downarrow}(k+1) &= m_{\downarrow}^{k+1} \circledast wb_*^{k+1}(i) \\ &= \begin{cases} m_{\downarrow}^{k} \circledast PW_*(k).bpa \circledast (wb_*^{k}(i) \circ BW_*(k)) & BW_*(k) \neq \emptyset \\ m_{\downarrow}^{k} \circledast PW_*(k).bpa \circledast wb_*^{k}(i) & BW_*(k) = \emptyset \end{cases} \end{aligned}$$

$$= \begin{cases} m^k_\downarrow \circledast PW_*(k).bpa \circledast wb^k_*(i) \circledast BW_*(k) & BW_*(k) \neq \emptyset \\ m^k_\downarrow \circledast PW_*(k).bpa \circledast wb^k_*(i) \circledast \emptyset & BW_*(k) = \emptyset \end{cases}$$
$$= m^k_\downarrow \circledast PW_*(k).bpa \circledast wb^k_*(i) \circledast BW_*(k)$$
$$= m^k_\downarrow \circledast PW_*(k).bpa \circledast wb^k_*(i) \circledast PW_*(k).wba.$$

However, with Condition WriteWriteOrder we obtain that the inclusive device closure of the bypassing writes does not conflict with buffered writes

$$\neg hit(idc(Dom(PW_*(k).bpa)), wb^k_*(i))$$

and we can thus move the bypassing portion of the write buffer behind the buffer and combine it with the buffered write by repeated application of Lemma 30

$$\begin{aligned} pfms_{i\downarrow}(k+1) &= m^k_\downarrow \circledast PW_*(k).bpa \circledast wb^k_*(i) \circledast PW_*(k).wba \\ &= m^k_\downarrow \circledast wb^k_*(i) \circledast (PW_*(k).bpa \cup PW_*(k).wba) && \text{L 30} \\ &= pfms_{i\downarrow}(k) \circledast W_\uparrow(k) \end{aligned}$$

Using now the induction hypothesis and Lemma 29 we can replace the perfect world forwarding with the memory in the high-level machine, and obtain the claim

$$\begin{aligned} &=_A m^k_\uparrow \circledast W_\uparrow(k) && \text{IH, L 29} \\ &= m^{k+1}_\uparrow. \end{aligned}$$

Case 4

$$SC_\uparrow(k) \wedge s(k) \in \Sigma_{WB,i},$$

By Lemma 288, the addresses were up to date at k

$$utd_i(k,A)$$

and with the induction hypothesis we obtain that the memory view is at sync at k

$$sync_i(k,A).$$

The claim is now Lemma 293.

Case 5

$$SC_\uparrow(k) \wedge s(k) \in \Sigma_{WB,j} \wedge j \neq i$$

This case is the most difficult one, since it includes the case where a write buffer step clears the last write to a portion of A and thus makes A up to date.

Using Lemma 290 we therefore split the proof between the portions of A that are modified resp. not modified by step k.

$sync_i(k+1, A \cap out_\downarrow(k))$**:** This portion of A *was* buffered at k and we can not apply the induction hypothesis for i and A. However, by Lemma 286 no

processor in strong memory mode except for j may be buffering a write that conflicts with the low-level write in k

$$\begin{aligned}
1 &\iff \forall j'.SC_{j'\uparrow}(k) \wedge s(k) \notin \Sigma_{WB,j'} \rightarrow \neg hit(out_{\downarrow}(k), wb^k_{\uparrow}(j')) \\
&\iff \forall j'.SC_{j'\uparrow}(k) \wedge j' \neq j \rightarrow \neg hit(out_{\downarrow}(k), wb^k_{\uparrow}(j')) \\
&\iff \forall j' \neq j.SC_{j'\uparrow}(k) \rightarrow \neg hit(out_{\downarrow}(k), wb^k_{\uparrow}(j')) \\
&\implies \forall j' \neq j.SC_{j'\uparrow}(k) \rightarrow \neg hit(A \cap out_{\downarrow}(k), wb^k_{\uparrow}(j')) && \text{L 48} \\
& && (4.22) \\
&\implies utd_j(k, A \cap out_{\downarrow}(k)).
\end{aligned}$$

By Lemma 12 the intersection is closed

$$closed(A \cap out_{\downarrow}(k)).$$

We can thus apply the induction hypothesis with $i' := j$ and $A' := A \cap out_{\downarrow}(k)$, and obtain that the memory views of that processor are in sync

$$sync_j(k, A \cap out_{\downarrow}(k))$$

and by Lemma 293 that is still the case at $k+1$ and the mode is unchanged

$$sync_j(k+1, A \cap out_{\downarrow}(k)) \wedge SC_{i\uparrow}(k+1),$$

and thus the memory and the forwarding agree

$$m^{k+1}_{\uparrow} =_{A \cap out_{\downarrow}(k)} pfms_{\downarrow j}(k+1).$$

Because the memory is now up-to-date for unit i, unit j no longer buffers writes to A at $k+1$

$$\neg hit(A, wb^{k+1}_*(j))$$

and by Lemma 54 the write buffer can be dropped from the forwarding system

$$\begin{aligned}
m^{k+1}_{\uparrow} &=_{A \cap out_{\downarrow}(k)} pfms_{\downarrow j}(k+1) \\
&= m^{k+1}_{\downarrow} \circledast wb^{k+1}_*(j) \\
&=_A m^{k+1}_{\downarrow} && \text{L 54}
\end{aligned}$$

The proof now distinguishes between the memory mode of unit i.

$SC_{i\uparrow}(k+1)$: The step of another write buffer does not modify the write buffer of unit i

$$wb^k_*(i) = wb^{k+1}_*(i).$$

Since the machine mode of unit i is by assumption not changed by the step, the machine was already in strong memory mode at k

$$SC_{i\uparrow}(k)$$

and with Statement (4.22) with $j' := i$ we obtain that there is neither a hit in the write buffer of unit i at k

$$\neg hit(A \cap out_{\downarrow}(k), wb^k_{\uparrow}(i))$$

nor at $k+1$

$$\neg hit(A \cap out_{\downarrow}(k), wb_{\uparrow}^{k+1}(i)).$$

The claim follows with Lemma 54

$$\begin{aligned} m_{\uparrow}^{k+1} &=_{A \cap out_{\downarrow}(k)} m_{\downarrow}^{k+1} \\ &=_{A \cap out_{\downarrow}(k)} m_{\downarrow}^{k+1} \circledast wb_{\uparrow}^{k+1}(i) && \text{L 54} \\ &=_{A \cap out_{\downarrow}(k)} pfms_{i\downarrow}(k+1). \end{aligned}$$

$\neg SC_{i\uparrow}(k+1)$**:** The memory views are in sync when the memories agree and we are done

$$sync_i(k+1, A \cap out_{\downarrow}(k)).$$

$sync_i(k+1, A \setminus out_{\downarrow}(k))$**:** By Lemma 289 the addresses were up to date

$$utd_i(k, A \setminus out_{\downarrow}(k))$$

and with the induction hypothesis for $A' := A \setminus out_{\downarrow}(k)$ we obtain that the memory view is at sync at k

$$sync_i(k, A \setminus out_{\downarrow}(k)).$$

By Lemma 174 step k has no outputs in the high-level machine

$$out_{\uparrow}(k) = \emptyset$$

and by Lemma 138 neither the high-level machine nor the low-level machine modify the addresses

$$\begin{aligned} m_{\uparrow}^{k+1} &=_{A \setminus out_{\downarrow}(k)} m_{\uparrow}^{k}, \\ m_{\downarrow}^{k+1} &=_{A \setminus out_{\downarrow}(k)} m_{\downarrow}^{k}. \end{aligned}$$

The proof distinguishes now between the mode of unit i.

$SC_{i\uparrow}(k)$**:** By Lemma 96 the write buffer of unit i is not changed

$$wb_{*}^{k}(i) = wb_{*}^{k+1}(i)$$

and the claim follows with the induction hypothesis and Lemma 29

$$\begin{aligned} m_{\uparrow}^{k+1} &=_{A \setminus out_{\downarrow}(k)} m_{\uparrow}^{k} \\ &=_{A \setminus out_{\downarrow}(k)} pfms_{i\downarrow}(k) && \text{IH} \\ &= m_{\downarrow}^{k} \circledast wb_{*}^{k}(i) \\ &=_{A \setminus out_{\downarrow}(k)} m_{\downarrow}^{k+1} \circledast wb_{*}^{k}(i) && \text{L 29} \\ &= m_{\downarrow}^{k+1} \circledast wb_{*}^{k+1}(i) \\ &= pfms_{i\downarrow}(k+1) \end{aligned}$$

$\neg SC_{i\uparrow}(k)$**:** The claim follows with the induction hypothesis

$$\begin{aligned} m_{\uparrow}^{k+1} &=_{A \setminus out_{\downarrow}(k)} m_{\uparrow}^{k} \\ &=_{A \setminus out_{\downarrow}(k)} m_{\downarrow}^{k} && \text{IH} \\ &=_{A \setminus out_{\downarrow}(k)} m_{\downarrow}^{k+1}. \end{aligned}$$

□

We can now show that an ordered schedule keeps the two machines in sync.

Lemma 295. *In a k-abstract, k-ordered schedule that is valid until $k-1$ and IPI-valid until $k-2$*

$$s \in \mathrm{ABS}_k \cap \mathrm{ORD}_k \wedge \Gamma_*^{k-1}(s) \wedge \Delta_{IPI*}^{k-2}(s)$$

the memory invariant holds at k for every processor and every closed set of addresses

$$closed(A) \rightarrow minv_i(k,A).$$

Proof. We prove the claims by induction on k, with i and j generalized. The base case is trivial. In the inductive step $k \rightarrow k+1$ we have that s is a $k+1$-abstract, $k+1$-ordered schedule that is valid until k and IPI-valid until $k-1$

$$s \in \mathrm{ABS}_{k+1} \cap \mathrm{ORD}_{k+1} \wedge \Gamma_*^{k}(s) \wedge \Delta_{IPI*}^{k-1}(s)$$

and immediately obtain that it is also a k-abstract, k-ordered schedule that is valid until $k-1$ and IPI-valid until $k-2$

$$s \in \mathrm{ABS}_k \cap \mathrm{ORD}_k \wedge \Gamma_\downarrow^{k-1}(s) \wedge \Delta_{IPI*}^{k-2}(s).$$

By the inductive hypothesis we obtain that the memory invariant holds at k on all closed sets of addresses for every processor

$$\forall i', A'.closed(A') \rightarrow minv_i(k,A),$$

and we have to show the same for $k+1$

$$closed(A) \rightarrow minv_i(k+1,A).$$

We distinguish whether the mode changed or not.

$SC_{i\uparrow}(k) \neq SC_{i\uparrow}(k+1)$: Since A is closed, the memory invariant holds at k for A and i

$$minv_i(k,A).$$

The claim is now Lemma 291.

$SC_{i\uparrow}(k) = SC_{i\uparrow}(k+1)$: The claim is Lemma 294.

□

We prove the following corollary.

Lemma 296. *In a k-abstract, k-ordered schedule that is valid until $k-1$ and IPI-valid until $k-2$*

$$s \in \mathrm{ABS}_k \cap \mathrm{ORD}_k \wedge \Gamma_*^{k-1}(s) \wedge \Delta_{IPI*}^{k-2}(s)$$

for all A which are closed under devices and to which no writes are buffered by any processor in SC mode

$$closed(A) \wedge \forall j.SC_{j\uparrow}(k) \rightarrow \neg hit(A, wb_\uparrow^k(j))$$

the memories agree on A

$$m_\downarrow^k =_A m_\uparrow^k.$$

Proof. Let i be some processor.

By assumption all other processors in strong memory mode are not buffering writes to A

$$\forall j \neq i.SC_{j\uparrow}(k) \rightarrow \neg hit(A, wb^k_\uparrow(j)),$$

and thus the memory view is up to date for processor i

$$utd_i(k,A).$$

By Lemma 295 we obtain that the memory invariant holds

$$minv_i(k,A)$$

and thus the memory views are in sync

$$sync_i(k,A).$$

We distinguish between the memory mode of processor i.

$\neg SC_{i\uparrow}(k)$: Since the memory views are in sync, the memories must agree

$$m^k_\downarrow =_A m^k_\uparrow,$$

which is the claim.

$SC_{i\uparrow}(k)$: Since no processor in strong memory mode is buffering writes to A, neither is processor i

$$\neg hit(A, wb^k_\uparrow(i)),$$

and by Lemma 267 the same holds for the low-level machine

$$\neg hit(A, wb^k_\downarrow(i)).$$

Since the memory views are in sync, the memory of the high-level machine agrees with perfect world forwarding in the low-level machine, and the claim follows with Lemma 54

$$\begin{aligned} m^k_\uparrow &=_A pfms_{i\downarrow}(k) \\ &= m^k_\downarrow \circledast wb^k_\downarrow(i) \\ &=_A m^k_\downarrow. \end{aligned}$$ L 54

□

We now show that if a step uses some address as an input, the other units in sequentially consistent mode are not buffering writes to that address. We begin with fetched addresses, which are never modified by a buffered write.

Lemma 297. *Let in a $k+1$-ordered and k-abstract schedule $s \in \mathrm{ORD}^-_{k+1} \cap \mathrm{ABS}_k$ which is semi-valid until k and IPI-valid until $k-1$*

$$\Gamma\Phi^k_*(s) \wedge \Delta^{k-1}_{IPI*},$$

processor i in sequentially consistent mode

$$SC_{i\uparrow}(k)$$

have a write from t at k

$$t \in issue^k_\uparrow(i).$$

Then the processor does not modify code of other processors[5]

$$CM_\uparrow(t,k) \rightarrow s(k) \in \Sigma_{P,i}.$$

Proof. Assume that there was a code modification

$$CM_\uparrow(t,k).$$

By Lemma 168 we obtain that the buffered write was issued in sequentially consistent mode

$$SC_\uparrow(t) = SC_{i\uparrow}(t) = SC_{i\uparrow}(k) = 1.$$

Step k is semi-ordered

$$sord_*(k)$$

and by Lemma 283 we obtain that there is no shared write buffered for processor i in case step k is a shared read

$$\nexists l \in issue^k_\uparrow(i).l \geq t \wedge Sh_\uparrow(l) \wedge ShR_\uparrow(k).$$

By contraposition of Lemma 282 steps t and k must be far forward-synchronized

$$t \blacktriangleright\!\!\blacktriangleright k.$$

By Lemma 281 step k is a step of processor i or its write buffer

$$s(k) \in \Sigma_{P,i} \cup \Sigma_{WB,i},$$

but since there is a code modification, step k is fetching something

$$F_\uparrow(k) \neq \emptyset$$

and thus step k is not a write buffer step

$$s(k) \notin \Sigma_{WB,i}.$$

The claim follows

$$s(k) \in \Sigma_{P,i}.$$

□

We conclude that in an ordered schedule, the last step still fetches the correct values.

Lemma 298. *Let in a $k+1$-ordered and k-abstract schedule $s \in \mathrm{ORD}^-_{k+1} \cap \mathrm{ABS}_k$ which is semi-valid until k and IPI-valid until $k-1$*

$$\Gamma\Phi^k_*(s) \wedge \Delta^{k-1}_{IPI*}(s),$$

step k has the same core and still fetches the same values

$$m^k_\uparrow =_{C_\downarrow(k) \cup F_\downarrow(k)} m^k_\downarrow.$$

[5] Write buffer steps do not fetch and execute code and can thus be excluded as well.

Proof. By Lemma 267, at the end of the abstract portion of the schedule the low-level machine configuration still locally simulates the high-level configuration

$$c^k_\downarrow \doteq c^k_\uparrow$$

and by Lemma 261 we obtain that the steps agree on the core registers

$$m^k_\downarrow =_{C_\downarrow(k)} m^k_\uparrow.$$

It suffices now to show that the memories agree on the fetched registers

$$m^k_\downarrow \stackrel{!}{=}_{F_\downarrow(k)} m^k_\uparrow.$$

We show that the agree on the inclusive device closure of the fetched registers, which subsumes the fetched registers

$$m^k_\downarrow \stackrel{!}{=}_{idc(F_\downarrow(k))} m^k_\uparrow.$$

By Lemma 21 that set is closed

$$closed(idc(F_\downarrow(k))).$$

We apply Lemma 296, which reduces the claim to showing that no sequentially consistent processor is buffering writes to those addresses

$$\forall i.SC_{i\uparrow}(k) \rightarrow \neg hit(idc(F_\downarrow(k)), wb^k_\uparrow(i)).$$

Assume for the sake of contradiction that such a processor exists

$$SC_{i\uparrow}(k) \wedge hit(idc(F_\downarrow(k)), wb^k_\uparrow(i)).$$

By Lemma 127 there must be an issued timestamp t that put the write into the buffer

$$t \in issue^k_\uparrow(i) \wedge Dom(BW_\uparrow(t)) \dot{\cap} idc(F_\downarrow(k)).$$

By Lemma 142, the two machines fetch the same values and we obtain that the write must modify the fetched values of the high-level machine

$$Dom(BW_\uparrow(t)) \dot{\cap} idc(F_\uparrow(k)).$$

By Lemma 92, the buffered writes are outputs

$$Dom(BW_\uparrow(t)) \subseteq out_\uparrow(t)$$

and thus there is an intersection between outputs and the inclusive device closure of the fetched addresses

$$out_\uparrow(t) \dot{\cap} idc(F_\uparrow(k)).$$

The outputs are closed and by Lemma 15 we can drop the inclusive device closure for the fetched addresses

$$out_\uparrow(t) \dot{\cap} F_\uparrow(k)$$

and there is thus a code modification

$$CM_\uparrow(t,k).$$

By Lemma 297, step k must have been made by processor i

$$s(k) \in \Sigma_{P,i},$$

which is thus also done in strong memory mode

$$SC_{\uparrow}(k) = SC_{i\uparrow}(k) = 1.$$

By Condition CodeOrder, there is no hit in the write buffer for the inclusive device closure of the fetched addresses

$$\neg hit(idc(F_{\uparrow}(k)), wb^k_{\uparrow}(i)),$$

which was the claim. □

When an ordered schedule is abstract until k, validity can be transferred to the high-level machine until k.

Lemma 299. *Let in a $k+1$-semi-ordered and k-abstract schedule s which is semi-valid until k*

$$s \in \mathrm{ORD}^{-}_{k+1} \cap \mathrm{ABS}_k \wedge \Gamma\Phi^k_*(s) \wedge \Delta^{k-1}_{IPI*},$$

the schedule is valid at k in the high-level machine iff it is valid in the low-level machine

$$\Gamma_{\uparrow}(k) = \Gamma_{\downarrow}(k).$$

Proof. By Lemma 298, the machines see the same values for core and fetched registers

$$m^k_{\downarrow} =_{C_{\downarrow}(k) \cup F_{\downarrow}(k)} m^k_{\uparrow}$$

and by Lemma 143 the configurations have the same core and fetched registers, and thus also agree on the core and fetched registers of the other machine

$$m^k_{\downarrow} =_{C_{\uparrow}(k) \cup F_{\uparrow}(k)} m^k_{\uparrow}$$

By Lemma 267 the configurations have the same buffers

$$wb^k_{\downarrow} = wb^k_{\uparrow}.$$

By Lemma 180 the validity of the high-level machine implies the validity in the low-level machine

$$\Gamma_{\uparrow}(k) \rightarrow \Gamma_{\downarrow}(k)$$

and again by Lemma 180 the validity of the low-level machine implies the validity in the high-level machine

$$\Gamma_{\downarrow}(k) \rightarrow \Gamma_{\uparrow}(k)$$

and the claim follows.

□

We can now prove that while a processor buffers writes to an address, other units do not read from that address.

Lemma 300. *Let in a $k+1$-semi-ordered and k-abstract schedule s which is valid until k and IPI-valid until $k-1$*

$$s \in \mathrm{ORD}^{-}_{k+1} \cap \mathrm{ABS}_k \wedge \Gamma^k_{\downarrow}(s) \wedge \Delta^{k-1}_{IPI*}(s),$$

processor i in sequentially consistent mode

$$SC_{i\uparrow}(k)$$

have a buffered write from t at k

$$t \in issue^k_{\uparrow}(i).$$

Then there is no write-read race in memory with any other unit

$$WR_{\uparrow}(t,k) \rightarrow s(k) \in \Sigma_{P,i} \cup \Sigma_{WB,i}.$$

Proof. The schedule is also semi-valid in the low-level machine

$$\Gamma\Phi^k_{\downarrow}(s),$$

and by Lemma 271 also in the high-level machine

$$\Gamma\Phi^k_{\uparrow}(s).$$

By Lemma 299 the step k is valid in the high-level machine

$$\Gamma_{\uparrow}(k).$$

Assume now that there is a write-read race

$$WR_{\uparrow}(t,k).$$

By Lemma 168 we obtain that the buffered write was issued in sequentially consistent mode

$$SC_{\uparrow}(t) = SC_{i\uparrow}(t) = SC_{i\uparrow}(k) = 1.$$

Step k is semi-ordered

$$sord_*(k)$$

and by Lemma 283 we obtain that there is no shared write buffered for processor i in case step k is a shared read

$$\nexists l \in issue^k_{\uparrow}(i).l \geq t \wedge Sh_{\uparrow}(l) \wedge ShR_{\uparrow}(k).$$

By contraposition of Lemma 282 steps t and k must be far forward-synchronized

$$t \blacktriangleright\!\!\blacktriangleright k.$$

By Lemma 281 step k is a step of processor i or its write buffer

$$s(k) \in \Sigma_{P,i} \cup \Sigma_{WB,i},$$

which is the claim. □

We obtain that other processors in strong memory mode never buffer writes to a read address.

Lemma 301. *Let in a $k+1$-semi-ordered and k-abstract schedule s which is valid until k and IPI-valid until $k-1$*

$$s \in \mathrm{ORD}^{-}_{k+1} \cap \mathrm{ABS}_k \wedge \Gamma^k_{\downarrow}(s) \wedge \Delta^{k-1}_{IPI*}(s)$$

step k be made by processor i

$$s(k) \in \Sigma_{P,i}.$$

Then there is no other processor in strong memory mode that buffers a write to the inclusive device closure of the portions $X \in \{wba, bpa\}$ of the addresses read in the step

$$\forall j \neq i.SC_{j\uparrow}(k) \rightarrow \neg hit(idc(R_{\uparrow}(k).X), wb^k_{\uparrow}(j)).$$

Proof. Assume for the sake of contradiction that such a processor exists

$$j \neq i \wedge SC_{j\uparrow}(k) \wedge hit(idc(R_{\uparrow}(k).X), wb^k_{\uparrow}(j)).$$

By Lemma 127 there must be an issued timestamp t that put the write into the buffer

$$t \in issue^k_{\uparrow}(j) \wedge Dom(BW_{\uparrow}(t)) \dot{\cap} idc(R_{\uparrow}(k).X).$$

Clearly the domain of that buffered write is non-empty

$$Dom(BW_{\uparrow}(t)) \neq \emptyset.$$

With Lemma 168 we obtain that it was also made in strong memory mode

$$SC_{\uparrow}(t) = SC_{j\uparrow}(t) = SC_{j\uparrow}(k) = 1,$$

and thus by Lemma 92 the domain of the buffered write is part of the outputs

$$Dom(BW_{\uparrow}(t)) \subseteq out_{\uparrow}(t).$$

Thus the outputs intersect with the inclusive device closure of the read addresses

$$out_{\uparrow}(t) \dot{\cap} idc(R_{\uparrow}(k).X)$$

and because the outputs are closed under devices we can drop the inclusive device closure with Lemma 15

$$out_{\uparrow}(t) \dot{\cap} R_{\uparrow}(k).X.$$

Since the read addresses are inputs, there must be an intersection between outputs and inputs

$$out_{\uparrow}(t) \dot{\cap} in_{\uparrow}(k),$$

and thus a write-read race

$$WR_{\uparrow}(t,k).$$

By Lemma 300, step k must be made by processor j or its write buffer

$$s(k) \in \Sigma_{P,j} \cup \Sigma_{WB,j}.$$

Because oracle inputs are disjoint and step k is already a step of processor i, we obtain that i and j are the same

$$i = j,$$

which is a contradiction. □

Lemma 302. *A $k+1$-semi-ordered and k-abstract schedule s which is valid until k in the low-level machine*

$$s \in \mathrm{ORD}_{k+1}^{-} \cap \mathrm{ABS}_k \wedge \Gamma_{\downarrow}^{k}(s)$$

is also valid until k in the low-level machine

$$\Gamma_{\uparrow}^{k}(s).$$

Proof. By Lemma 271 the schedule is valid until $k-1$

$$\Gamma_{\uparrow}^{k-1}(s)$$

and by Lemma 299 also at k

$$\Gamma_{\uparrow}(k)$$

and the claim follows

$$\Gamma_{\uparrow}^{k}(s).$$

□

Lemma 303. *Let in a $k+1$-semi-ordered and k-abstract schedule s which is semi-valid until k and IPI-valid until $k-1$*

$$s \in \mathrm{ORD}_{k+1}^{-} \cap \mathrm{ABS}_k \wedge \Gamma_{\downarrow}^{k}(s) \wedge \Delta_{IPI*}^{k-1}(s)$$

step k be made by processor i

$$s(k) \in \Sigma_{P,i}.$$

Then there is no processor in strong memory mode that buffers a write to the inclusive device closure of the bypassing portion of the addresses read in the step

$$\forall j.SC_{j\uparrow}(k) \rightarrow \neg hit(idc(R_{\uparrow}(k).bpa), wb_{\uparrow}^{k}(j)).$$

Proof. Assume that processor j is in strong memory mode has a buffered write

$$SC_{j\uparrow}(k) \wedge hit(idc(R_{\uparrow}(k).bpa), wb_{\uparrow}^{k}(j)).$$

By contraposition of Lemma 301, processors i and j are the same

$$i = j.$$

Therefore the step is made in strong memory mode

$$SC_{\uparrow}(k) = SC_{j\uparrow}(k) = 1.$$

By Lemma 302 the schedule is valid until k

$$\Gamma_{\uparrow}^{k}(s).$$

By Condition WriteReadOrder, there is no hit

$$\neg hit(idc(R_{\uparrow}(k).bpa), wb_{\uparrow}^{k}(j)),$$

which is a contradiction. □

We conclude that if step k is also still ordered, the step reads the same values from memory.

Lemma 304. *Let in a $k+1$-semi-ordered and k abstract schedule s which is valid until k and IPI-valid until $k-2$*

$$s \in \mathrm{ORD}^{-}_{k+1} \cap \mathrm{ABS}_k \wedge \Gamma^k_{\downarrow}(s) \wedge \Delta^{k-1}_{IPI*}(s),$$

step k, if made by a processor, has the same core and still fetches the same values

$$s(k) \in \Sigma_{P,i} \rightarrow v_{\uparrow}(k) = v_{\downarrow}(k).$$

Proof. By Lemma 298 the step sees the same core and fetched register values

$$m^k_{\uparrow} =_{C_{\downarrow}(k) \cup F_{\downarrow}(k)} m^k_{\downarrow}.$$

By Lemma 143 we obtain that the step reads the same addresses

$$R_{\uparrow}(k) = R_{\downarrow}(k).$$

By Lemma 303 no processor has a hit for the inclusive device closure of the bypassing reads

$$\forall j.SC_{j\uparrow}(k) \rightarrow \neg hit(idc(R_{\uparrow}(k).bpa), wb^k_{\uparrow}(j)).$$

and by Lemma 296 the machines agree on the value of those addresses

$$m^k_{\uparrow} =_{idc(R_{\uparrow}(k).bpa)} m^k_{\downarrow},$$

and thus on the value of the bypassing reads

$$m^k_{\uparrow} =_{R_{\uparrow}(k).bpa} m^k_{\downarrow}.$$

By Lemma 301 we obtain that no other processor in strong memory mode has a hit for the inclusive device closure of the buffered reads

$$\forall j \neq i.SC_{j\uparrow}(k) \rightarrow \neg hit(idc(R_{\uparrow}(k).wba), wb^k_{\uparrow}(j)),$$

and thus by definition that memory region is up-to-date

$$utd_i(k, idc(R_{\uparrow}(k).wba)).$$

The proof now depends on the memory mode of processor i.

$SC_{i\uparrow}(k)$: By Lemma 295 we obtain that the memory of the high-level machine agrees with the perfect world forwarding in the low-level machine on the inclusive device closure of the buffered reads

$$m^k_{\uparrow} =_{idc(R_{\uparrow}(k).wba)} pfms_{i\downarrow}(k),$$

and thus also on the buffered reads

$$m^k_{\uparrow} =_{R_{\uparrow}(k).wba} pfms_{i\downarrow}(k).$$

By Lemma 173 we obtain that the step is not buffering hits to the device closure of the inputs

$$\neg hit(dc(in_{\uparrow}(k)), wb^k_{\uparrow}(i)),$$

which by Lemma 267 also holds for the low-level machine

$$\neg hit(dc(in_\uparrow(k)), wb^k_\downarrow(i)).$$

We obtain by Lemma 56 that perfect world forwarding and forwarding agree on the inputs in the low-level machine

$$\begin{aligned} pfms_{i\downarrow}(k) &= m^k_\downarrow \circledast wb^k_\downarrow(i) \\ &=_{in_\uparrow(k)} m^k_\downarrow \circledcirc wb^k_\downarrow(i) && \text{L 56} \\ &= fms_{i\downarrow}(k), \end{aligned}$$

and thus also on the buffered reads (which are a subset of the inputs)

$$pfms_{i\downarrow}(k) =_{R_\uparrow(k).wba} fms_{i\downarrow}(k).$$

We conclude that memory of the high-level machine and forwarding of the low-level machine step agree

$$\begin{aligned} m^k_\uparrow &=_{R_\uparrow(k).wba} pfms_{i\downarrow}(k) \\ &=_{R_\uparrow(k).wba} fms_{i\downarrow}(k) \\ &= fms_\downarrow(k). \end{aligned}$$

The step is in strong memory mode and thus not in low-level machine semantics in the high-level machine

$$SC_\uparrow(k) \wedge \neg LL_\uparrow(k),$$

and the claim follows

$$\begin{aligned} v_\uparrow(k) &= m^k_\uparrow\big|_{R_\uparrow(k)} \\ &= m^k_\uparrow\big|_{R_\uparrow(k).bpa} \cup m^k_\uparrow\big|_{R_\uparrow(k).wba} \\ &= m^k_\downarrow\big|_{R_\uparrow(k).bpa} \cup fms_\uparrow(k)\big|_{R_\uparrow(k).wba} \\ &= m^k_\downarrow\big|_{R_\downarrow(k).bpa} \cup fms_\uparrow(k)\big|_{R_\downarrow(k).wba} \\ &= v_\downarrow(k). \end{aligned}$$

$\neg SC_{i\uparrow}(k)$**:** With Lemma 295, we obtain that the memory of the machines agrees on the buffered reads

$$m^k_\uparrow =_{R_\uparrow(k).wba} m^k_\downarrow.$$

They agree thus on the core registers, the fetched registers, the bypassing reads, and the buffered reads

$$m^k_\uparrow =_{C_\uparrow(k) \cup F_\uparrow(k) \cup R_\uparrow(k).bpa \cup R_\uparrow(k).wba} m^k_\downarrow,$$

which together are the inputs

$$m^k_\uparrow =_{in_\uparrow(k)} m^k_\downarrow.$$

The step is being made in weak memory mode and thus uses low-level machine semantics

$$\neg SC_{\uparrow}(k) \wedge LL_{\uparrow}(k)$$

and thus the machine semantics of the machines are the same

$$LL_{\downarrow}(k) = 1 = LL_{\uparrow}(k).$$

By Lemma 267 the machines use the same buffer

$$wb^k_{\downarrow} = wb^k_{\uparrow},$$

in particular for the unit making the step

$$wb^k_{\downarrow} =_{u_{\downarrow}(k)} wb^k_{\uparrow},$$

and thus by Lemma 90 the buffers of the low-level machine subsume those of the high-level machine

$$bufS_{\downarrow}(s(k), c^k_{\downarrow}, c^k_{\uparrow}).$$

The claim is now Lemma 182.

□

The crucial result that semi-ordered schedules are abstract follows.

Lemma 305.

$$\Gamma^{k-1}_{\downarrow}(s) \wedge \Delta^{k-2}_{IPI\downarrow} \wedge s \in \mathrm{ORD}^{-}_{k} \rightarrow s \in \mathrm{ABS}_k$$

Proof. By induction on k. The base case is trivial. For $k \rightarrow k+1$, assume a k+1-semi-ordered schedule

$$s \in \mathrm{ORD}^{-}_{k+1}$$

that is valid until k and IPI-valid until $k-1$

$$\Gamma^{k}_{\downarrow}(s) \wedge \Delta^{k-1}_{IPI\downarrow}.$$

Clearly it is also valid until $k-1$, IPI-valid until $k-2$, and k-semi-ordered and by the inductive hypothesis it is abstract until k

$$s \in \mathrm{ABS}_k.$$

We have to show s is also $k+1$-abstract

$$s \stackrel{!}{\in} \mathrm{ABS}_{k+1},$$

for which it clearly suffices to show that the low-level machine simulates the high-level machine during step k

$$c^k_{\downarrow} \stackrel{!}{=}^{s(k)}_{\downarrow,\uparrow} c^k_{\uparrow}.$$

The proof distinguishes between processor and write buffer steps.

$s(k) \in \Sigma_{P,i}$: We unfold the definition of our claim and obtain three subclaims.

$m^k_{\downarrow} =_{C_{\downarrow}(k) \cup F_{\downarrow}(k)} m^k_{\uparrow}$: This is just Lemma 298.

$v_\downarrow(k) = v_\uparrow(k)$**:** This is just Lemma 304.

$\Delta_\downarrow(k) = \Delta_\uparrow(k)$**:** With Lemma 143 and what we have shown above we obtain that the configurations have the same core and fetch the same values

$$core_\downarrow(k) = core_\uparrow(k),\ fetch_\downarrow(k) = fetch_\uparrow(k).$$

By Lemma 267 the processors are in local simulation, and thus the write buffers are the same

$$wb^k_\downarrow = wb^k_\uparrow.$$

The claim follows

$$\begin{aligned}\Delta_\downarrow(k) &= \Delta_P(core_\downarrow(k), fetch_\downarrow(k), wb^k_\downarrow(i), s(k)) \\ &= \Delta_P(core_\uparrow(k), fetch_\uparrow(k), wb^k_\uparrow(i), s(k)) \\ &= \Delta_\uparrow(k).\end{aligned}$$

$s(k) \in \Sigma_{WB,i}$**:** We unfold the definition of our claim and obtain two subclaims.

$m^k_\downarrow =_{C_\downarrow(k)} m^k_\uparrow$**:** Follows directly from Lemma 298.

$hd(wb^k_\downarrow(i)) = hd(wb^k_\uparrow(i))$**:** Follows directly with Lemma 267.

□

4.12 Constructing an Ordered Schedule

In order to show that all schedules are equivalent to a reduced schedule, by Lemma 305 it suffices to show that every schedule can be reordered into an equivalent ordered schedule. We define a recursive reordering strategy. At each iteration, we start out with a t-ordered schedule and choose a candidate step $k_t \geq t$. We move that step to position t, keeping steps $[0:t)$ as they are. As a consequence, we obtain a growing prefix which is ordered. We later show that all steps of the original schedule eventually become part of the prefix. The idea is to unroll the schedule in the order of global steps. In order to obtain ordering, we may not allow a write to enter the buffer unless we can be sure that the next global step is the one committing the write to memory.

Our reordering only works if all buffered writes are eventually committed. Otherwise, in an infinite schedule with an infinite number of global steps, a buffered write might be pushed infinitely far away. To formalize this, we count after each processor step the *number of buffered writes of that processor* using the following recursive function $\#BW_i[s] : P, i(s) \to \mathbb{N}$

$$\begin{aligned}\#BW_i[s](0) &= 0, \\ \#BW_i[s](n+1) &= \#BW_i[s](n) + \begin{cases} 1 & BW_\downarrow[s](\#P, i \approx n(s)) \neq \emptyset \\ 0 & \text{o.w.} \end{cases}\end{aligned}$$

The number of writes buffered by each processor i after n steps of that processor is preserved by equivalence.

Lemma 306.

$$s \equiv_\downarrow r \to \#BW_i[s](n) = \#BW_i[r](n).$$

Proof. Note first that due to equivalence, the number of processor steps of i is the same

$$P,i(s) = P,i(r)$$

and thus the domains of the functions are the same.

The proof now is by induction on $n \in P,i(s)$. The base case is trivial.

In the step $n \to n+1$, we obtain by equivalence that the configurations during which processor i makes its n-th step strongly agree

$$c_\downarrow[s]^{\#P,i\approx n(s)} =_\downarrow^{s(\#P,i\approx n(s))} c_\downarrow[r]^{\#P,i\approx n(r)}$$

and the configurations are stepped with the same oracle input

$$s(\#P,i \approx n(s)) = r(\#P,i \approx n(r)).$$

We conclude with Lemma 144 that the steps agree on whether they buffered a write

$$\begin{aligned} BW_\downarrow[s](\#P,i \approx n(s)) &= BW_\downarrow(c_\downarrow[s]^{\#P,i\approx n(s)}, s(\#P,i \approx n(s))) \\ &= BW_\downarrow(c_\downarrow[r]^{\#P,i\approx n(r)}, s(\#P,i \approx n(s))) && \text{L 144} \\ &= BW_\downarrow(c_\downarrow[r]^{\#P,i\approx n(r)}, r(\#P,i \approx n(r))) \\ &= BW_\downarrow[r](\#P,i \approx n(r)). \end{aligned}$$

The claim now follows with the induction hypothesis

$$\begin{aligned} \#BW_i[s](n+1) &= \#BW_i[s](n) + \begin{cases} 1 & BW_\downarrow[s](\#P,i \approx n(s)) \\ 0 & \text{o.w.} \end{cases} \\ &= \#BW_i[r](n) + \begin{cases} 1 & BW_\downarrow[r](\#P,i \approx n(r)) \\ 0 & \text{o.w.} \end{cases} \\ &= \#BW_i[r](n+1). \end{aligned}$$

□

We say that a schedule is *balanced* when the set of reached numbers of buffered writes is equal to the set of reached number of write buffer steps

$$s \in bal \iff \forall i.\, \{\, \#BW_i[s](n) \mid n \in P,i(s) \,\} = WB,i(s).$$

We show that balance is preserved by equivalence.

Lemma 307.

$$s \equiv_\downarrow r \wedge s \in bal \to r \in bal.$$

Proof. Since the schedules are equivalent, processors and write buffers reach the same number of steps

$$P,i(s) = P,i(r) \wedge WB,i(s) = WB,i(r).$$

By Lemma 306 the number of buffered writes at each n are the same

$$\forall n \in P,i(s).\#BW_i[s](n) = \#BW_i[r](n).$$

By assumption, schedule s is balanced and thus the set of reached numbers of buffered writes is equal to the number of write buffer steps

$$\{\, \#BW_i[s](n) \mid n \in P, i(s) \,\} = WB, i(s),$$

and we immediately obtain the same for r

$$\{\, \#BW_i[r](n) \mid n \in P, i(r) \,\} = WB, i(r)$$

and the claim follows. □

Clearly, in a valid schedule, at each point in time t the number of buffered writes is at least the number of write buffer steps. In fact, the difference is exactly equal to the number of outstanding write buffer entries.

Lemma 308.

$$(\forall t'.\Gamma_{\downarrow}(t')) \rightarrow \#BW_i(\#P, i(t)) = \#WB, i(t) + |wb^t_{\downarrow}(i)|$$

Proof. By induction on t. The base case is trivial.

In the inductive step $t \rightarrow t+1$, we distinguish between the operator applied to the buffer of unit i.

$Op_{i\downarrow}(t) = push$**:** In this case step t is made by the processor of unit i and is buffering a write

$$s(t) \in \Sigma_{P,i} \wedge BW_{\downarrow}(t).$$

Thus the number of processor steps has increased

$$\begin{aligned} \#P, i(t+1) &= \#P, i(s[0:t]) \\ &= \#P, i(s[0:t-1]) + \#P, i(s(t)) \\ &= \#P, i(t) + 1, \end{aligned}$$

and step t is the is the step where the processor makes its $\#P, i(t)$-th step

$$\#P, i \approx (\#P, i(t))(s) = t.$$

Thus the number of buffered writes has increased

$$\begin{aligned} \#BW_i(\#P, i(t+1)) &= \#BW_i(\#P, i(t)+1) \\ &= \#BW_i(\#P, i(t)) + \begin{cases} 1 & BW_{\downarrow}(\#P, i \approx (\#P, i(t))(s)) \neq \emptyset \\ 0 & \text{o.w.} \end{cases} \\ &= \#BW_i(\#P, i(t)) + \begin{cases} 1 & BW_{\downarrow}(t) \neq \emptyset \\ 0 & \text{o.w.} \end{cases} \\ &= \#BW_i(\#P, i(t)) + 1. \end{aligned}$$

The number of write buffer steps has clearly not increased

$$\begin{aligned} \#WB, i(t+1) &= \#WB, i(s[0:t]) \\ &= \#WB, i(s[0:t-1]) + \#WB, i(s(t)) \\ &= \#WB, i(t) + 0, \end{aligned}$$

but the length of the write buffer has

$$\begin{aligned}|wb_{\downarrow}^{t+1}(i)| &= |wb_{\downarrow}^{t}(i) \circ BW_{\downarrow}(t)| \\ &= |wb_{\downarrow}^{t}(i)| + 1.\end{aligned}$$

The claim follows with the induction hypothesis

$$\begin{aligned}\#BW_i(\#P, i(t+1)) &= \#BW_i(\#P, i(t)) + 1 \\ &= (\#WB, i(t) + |wb_{\downarrow}^{t}(i)|) + 1 \\ &= \#WB, i(t) + 0 + |wb_{\downarrow}^{t}(i)| + 1 \\ &= \#WB, i(t+1) + |wb_{\downarrow}^{t+1}(i)|.\end{aligned}$$

$Op_{i\downarrow}(t) = pop$**:** In this case the step is a write buffer step

$$s(t) \in \Sigma_{WB,i}.$$

Thus the number of processor steps has not changed

$$\begin{aligned}\#P, i(t+1) &= \#P, i(s[0:t]) \\ &= \#P, i(s[0:t-1]) + \#P, i(s(t)) \\ &= \#P, i(t) + 0,\end{aligned}$$

but the number of write buffer steps has increased

$$\begin{aligned}\#WB, i(t+1) &= \#WB, i(s[0:t]) \\ &= \#WB, i(s[0:t-1]) + \#WB, i(s(t)) \\ &= \#WB, i(t) + 1.\end{aligned}$$

Since the schedule is valid, the drain condition holds

$$\Delta_{\downarrow}(t),$$

and thus the write buffer is non-empty

$$wb_{\downarrow}^{t}(i) \neq \varepsilon.$$

Its length is thus decreased by one

$$\begin{aligned}|wb_{\downarrow}^{t+1}(i)| &= |tl(wb_{\downarrow}^{t}(i))| \\ &= |wb_{\downarrow}^{t}(i)| - 1.\end{aligned}$$

The claim follows with the induction hypothesis

$$\begin{aligned}\#BW_i(\#P, i(t+1)) &= \#BW_i(\#P, i(t) + 0) \\ &= \#BW_i(\#P, i(t)) \\ &= \#WB, i(t) + |wb_{\downarrow}^{t}(i)| \\ &= (\#WB, i(t) + 1) + (|wb_{\downarrow}^{t}(i)| - 1) \\ &= \#WB, i(t+1) + |wb_{\downarrow}^{t+1}(i)|.\end{aligned}$$

$Op_{i,\downarrow}(t) = noop$**:** Step t is clearly not a write buffer step

$$s(t) \notin \Sigma_{WB,i}.$$

Neither the number of write buffer steps

$$\begin{aligned}\#WB,i(t+1) &= \#WB,i(s[0:t]) \\ &= \#WB,i(s[0:t-1]) + \#WB,i(s(t)) \\ &= \#WB,i(t) + 0,\end{aligned}$$

nor the length of the buffer have changed

$$|wb_{\downarrow}^{t+1}(i)| = |wb_{\downarrow}^{t}(i)|.$$

Thus the term on the right-hand side is unchanged

$$\#WB,i(t) + |wb_{\downarrow}^{t}(i)| = \#WB,i(t+1) + |wb_{\downarrow}^{t+1}(i)|.$$

We distinguish whether step t was a processor step or not.

$s(t) \in \Sigma_{P,i}$**:** The number of processor steps has increased

$$\begin{aligned}\#P,i(t+1) &= \#P,i(s[0:t]) \\ &= \#P,i(s[0:t-1]) + \#P,i(s(t)) \\ &= \#P,i(t) + 1,\end{aligned}$$

and step t is the is the step where the processor makes its $\#P,i(t)$-th step

$$\#P,i \approx (\#P,i(t))(s) = t.$$

However, since the operation is not a push, no write was buffered

$$BW_{\downarrow}(t) = \emptyset$$

and thus the number of buffered writes has not increased

$$\begin{aligned}\#BW_i(\#P,i(t+1)) &= \#BW_i(\#P,i(t)+1) \\ &= \#BW_i(\#P,i(t)) + \begin{cases}1 & BW_{\downarrow}(\#P,i \approx (\#P,i(t))(s)) \neq \emptyset \\ 0 & \text{o.w.}\end{cases} \\ &= \#BW_i(\#P,i(t)) + \begin{cases}1 & BW_{\downarrow}(t) \neq \emptyset \\ 0 & \text{o.w.}\end{cases} \\ &= \#BW_i(\#P,i(t)) + 0.\end{aligned}$$

The claim follows with the induction hypothesis

$$\begin{aligned}\#BW_i(\#P,i(t+1)) &= \#BW_i(\#P,i(t)) \\ &= \#WB,i(t) + |wb_{\downarrow}^{t}(i)| \\ &= \#WB,i(t+1) + |wb_{\downarrow}^{t+1}(i)|.\end{aligned}$$

$s(t) \notin \Sigma_{P,i}$**:** In this case the number of processor steps has also not increased

$$\begin{aligned}\#P,i(t+1) &= \#P,i(s[0:t]) \\ &= \#P,i(s[0:t-1]) + \#P,i(s(t)) \\ &= \#P,i(t) + 0,\end{aligned}$$

and the claim follows with the induction hypothesis

$$\begin{aligned}\#BW_i(\#P,i(t+1)) &= \#BW_i(\#P,i(t)) \\ &= \#WB,i(t) + |wb_{\downarrow}^{t}(i)| \\ &= \#WB,i(t+1) + |wb_{\downarrow}^{t+1}(i)|.\end{aligned}$$

□

Lemma 309. *In a balanced, valid schedule,*

$$s \in bal \wedge \forall t'.\Gamma_{\downarrow}(t')$$

a non-empty buffer will eventually make a step

$$wb_{\downarrow}^{t+1}(i) \neq \varepsilon \rightarrow \exists g > t.s(g) \in \Sigma_{WB,i}.$$

Proof. The length of the write buffer is greater than zero

$$|wb_{\downarrow}^{t+1}(i)| > 0$$

and by Lemma 308 the number of buffered writes is thus larger than the number of write buffer steps

$$\#BW_i(\#P,i(t+1)) = \#WB,i(t+1) + |wb_{\downarrow}^{t+1}(i)| > \#WB,i(t+1).$$

Since the schedule is balanced, the number of buffered writes is eventually matched

$$\#BW_i(\#P,i(t+1)) \in WB,i(s),$$

and thus there is a step k where the number of write buffer steps is equal to the number of buffered writes and thus larger than it is right now

$$\#WB,i(k) = \#BW_i(\#P,i(t+1)) > \#WB,i(t+1).$$

We obtain that the number of write buffer steps until t plus the number of write buffer steps between $t+1$ and $k-1$ is larger than the number of write buffer steps until t

$$\begin{aligned}\#WB,i(s[0:t]) + \#WB,i(s[t+1:k-1]) &= \#WB,i(s[0:k-1]) \\ &= \#WB,i(k) \\ &> \#WB,i(t) \\ &= \#WB,i(s[0:t]),\end{aligned}$$

and we conclude that there are additional write buffer steps between $t+1$ and k

$$\#WB,i(s[t+1:k-1]) > 0.$$

The claim follows

$$\exists g \in [t+1:k-1].s(t) \in \Sigma_{WB,i}.$$

□

Lemma 310. *In a balanced, valid schedule,*

$$s \in bal \wedge \forall t'.\Gamma_{\downarrow}(t')$$

each buffered write has its write buffer step

$$s(t) \in \Sigma_{P,i} \wedge BW_{\downarrow}(t) \neq \emptyset \rightarrow \exists g > t.s(g) \in \Sigma_{WB,i}.$$

Proof. By definition, step t is buffering a write

$$Op_{i\downarrow}(t) = push.$$

Thus the length of the write buffer at $t+1$ is greater than zero

$$|wb_{\downarrow}^{t+1}(i)| = |wb_{\downarrow}^{t}(i) \circ BW_{\downarrow}(t)| = |wb_{\downarrow}^{t}(i)| + 1 > 0.$$

The claim is now Lemma 309. □

Thus each global step and each processor step that buffers a write has a global step made by the same unit.

Lemma 311. *In a balanced, valid schedule,*

$$s \in bal \wedge \forall t'.\Gamma_{\downarrow}(t')$$

each buffered write and global step are eventually followed by a global step of the same unit

$$G_{\downarrow}(t) \vee BW_{\downarrow}(t) \neq \emptyset \rightarrow \exists g \geq t.u_{\downarrow}(g) = u_{\downarrow}(t) \wedge G_{\downarrow}(g).$$

Proof. We distinguish whether step t is global or buffering a write.

$G_{\downarrow}(t)$**:** Then $g := t$ solves the claim immediately.

$BW_{\downarrow}(t) \neq \emptyset$**:** Then the prepared buffered write is not empty

$$PW_{\downarrow}(t).wba = BW_{\downarrow}(t) \neq \emptyset$$

and thus by definition made by a processor

$$s(t) \in \Sigma_{P,i}.$$

We obtain from Lemma 310 that there is $g > t$ made by the write buffer of unit i. We choose $g := g$ and the first of part of the claim follows

$$u_{\downarrow}(g) = i = u_{\downarrow}(t),$$

and the second part is Lemma 136.

□

Lemma 312. *In a balanced, valid schedule,*

$$s \in bal \wedge \forall t'.\Gamma_{\downarrow}(t')$$

a non-empty sequence of issued writes will eventually make a step, and will only have grown until then

$$t' \in issue_{\downarrow}^{t+1}(i) \rightarrow \exists g > t, q.s(g) \in \Sigma_{WB,i} \wedge issue_{\downarrow}^{g}(i) = issue_{\downarrow}^{t}(i) \circ q.$$

Proof. By Lemma 123 the corresponding buffered write is in the buffer

$$BW_{\downarrow}(t') \in wb_{\downarrow}^{t+1}(i),$$

which is thus non-empty

$$wb_{\downarrow}^{t+1}(i) \neq \varepsilon.$$

By Lemma 309, there are write buffer steps of unit i after t

$$\exists g > t.s(g) \in \Sigma_{WB,i},$$

and we choose the first such step

$$g := \min \{\, g > t \mid s(g) \in \Sigma_{WB,i} \,\}.$$

Clearly there are no write buffer steps between t and g

$$\nexists t' \in [t : g).s(t') \in \Sigma_{WB,i}.$$

By Lemma 128, the tail of the sequence of issued writes has grown by some suffix q

$$issue_{\downarrow}^{g}(i) = issue_{\downarrow}^{t}(i) \circ q,$$

which is the claim. □

Thus, when unrolling the schedule in the order of global steps, we only have to worry that *other* steps might be pushed into infinity. Those steps are clearly local and thus not shared reads, and since the buffer is empty, the buffer is not dirty. We call these steps *pushable*, and give the formal simple definition

$$p_{\downarrow}(t) \equiv L_{\downarrow}(t) \wedge wb_{\downarrow}^{t}(u_{\downarrow}(t)) = \varepsilon \wedge BW_{\downarrow}(t) = \emptyset.$$

A non-pushable step is either global or leaves the buffer non-empty.

Lemma 313.

$$\neg p_{\downarrow}(t) \rightarrow G_{\downarrow}(t) \vee wb_{\downarrow}^{t+1}(u_{\downarrow}(t)) \neq \varepsilon.$$

Proof. Assume that the step is not global, i.e., local

$$L_{\downarrow}(t).$$

By Lemma 136, the step is a processor step of $i = u_{\downarrow}(t)$

$$s(t) \in \Sigma_{P,i}.$$

Thus the operation executed by step t is not a pop

$$Op_{i\downarrow}(t) \neq pop.$$

By definition of p, step t buffers a write or the write buffer is non-empty before the step

$$BW_{\downarrow}(t) \neq \emptyset \vee wb_{\downarrow}^{t}(i) \neq \varepsilon.$$

We distinguish between these two cases.

$BW_\downarrow(t) \neq \emptyset$**:** In this case the operation is a push and the claim follows

$$wb_\downarrow^{t+1}(i) = wb_\downarrow^t(i) \circ BW_\downarrow(t) \neq \varepsilon.$$

$wb_\downarrow^t(i) \neq \varepsilon$**:** By Lemma 42 the write buffer at t is a prefix of the write buffer at $t+1$, which thus must also be non-empty

$$wb_\downarrow^{t+1}(i) = wb_\downarrow^t(i) \circ q \neq \varepsilon.$$

That is the claim.

□

Lemma 314. *In a balanced, valid schedule,*

$$s \in bal \wedge \forall t'.\Gamma_\downarrow(t')$$

non-pushable steps are eventually followed by a global step

$$\neg p_\downarrow(t) \rightarrow \exists g \geq t.G_\downarrow(g).$$

Proof. By Lemma 313, the step is either global or has a non-empty write buffer

$$G_\downarrow(t) \vee wb_\downarrow^{t+1}(u_\downarrow(t)) \neq \varepsilon.$$

By Lemma 309 in the latter case there is a write buffer step after t

$$G_\downarrow(t) \vee \exists g > t.s(g) \in \Sigma_{WB,i}.$$

By contraposition of Lemma 136, write buffer steps are global thus the step g must be global

$$G_\downarrow(t) \vee \exists g > t.G_\downarrow(g)$$

and the claim follows. □

Lemma 315. *In a balanced, valid schedule,*

$$s \in bal \wedge \forall t'.\Gamma_\downarrow(t')$$

there is always a pushable or a global step

$$\exists g \geq t.G_\downarrow(g) \vee p_\downarrow(g).$$

Proof. By case distinction on whether step t is pushable.

$p_\downarrow(t)$**:** The claim follows with $g := t$.

$\neg p_\downarrow(t)$**:** The claim follows by Lemma 314.

□

We now recursively construct a sequence O_t of reorderings.
We begin with the original schedule

$$O_0 = \varepsilon.$$

For $t \to t+1$, we consider the next global or pushable step

$$g_t = \min \left\{ g \geq t \;\middle|\; G_{\downarrow} O_t(g) \vee p_{\downarrow} O_t(g) \right\}.$$

We wish to move that step to the front, but that is not always possible. We must never reorder an interrupt delivery step with steps of its victims, or the step that commits a write with the step that issued it. Therefore, if the next global step is made by one of those units, we have to find all steps that must not be reordered with the next global step (or each other). If such steps exist, we pick the first of these steps and move it to the front. If no such steps exist, we can simply take the next global step and move it directly.

Formally, we distinguish between write buffer steps in strong memory mode and other steps.

$SC_{\downarrow} O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i}$**:** Since we know that write buffer steps in strong memory mode do not interrupt other steps, we only need to make sure that the order with fences and with the step that issued the write is unchanged. Since the steps between the step that issued the write and step g_t (when the write is committed) can not possibly be fences, we only need to consider processor steps before that step. (cf Fig. 4.12)

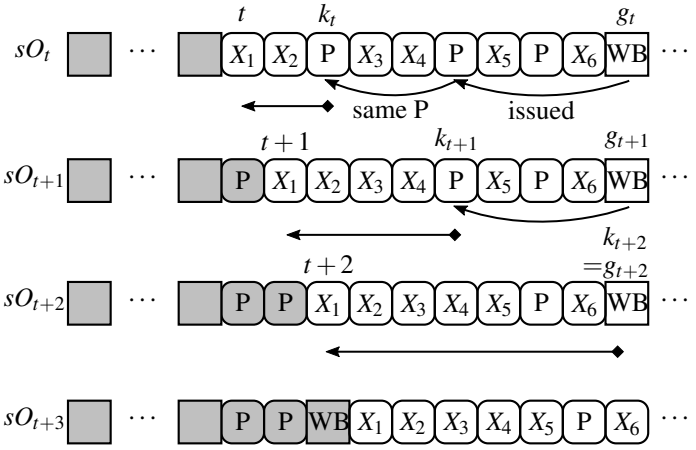


Figure 4.12: Reordering strategy in case step g_t is a write buffer step. Steps of the processor of the same unit are marked with P, steps of objects of other units with X_i. We move the step that issued the write to the front. This forces us to first move all steps by the same processor to the front. Steps of the same processor between the step that issued the write and the write buffer step are ignored.

$$k_t = \min \left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow} O_t^{g_t}(i)) \vee k = g_t \right\}.$$

$\neg(SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i})$**:** In this case we simply select the first step that is not unit-concurrent with the next global step[6] (cf. Fig. 4.13)

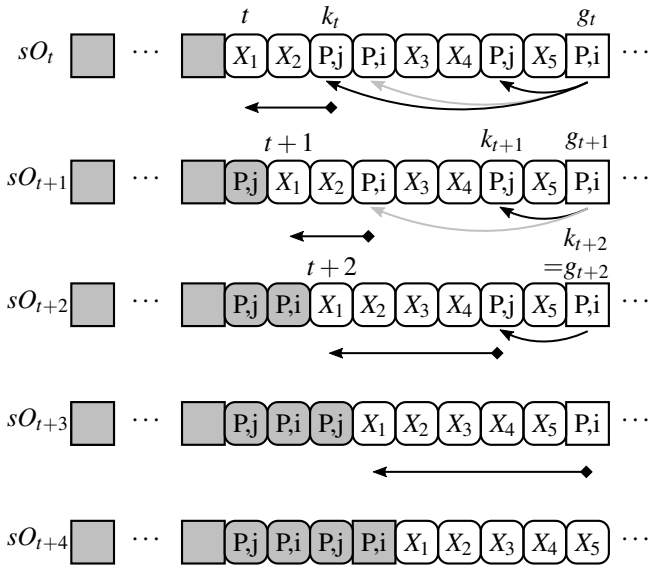


Figure 4.13: Reordering strategy in case step g_t made by processor i interrupts processor j. We first move all steps of the same processor (gray arrows) and steps that are interrupted by the step (black arrows) to the front.

$$k_t = \min\left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow}O_t(g_t,k) \right\}.$$

We define

$$O_{t+1} = O_t[t \leftarrow k_t].$$

Note that the above is only a definition when the schedule in iteration t is balanced and valid. We will only use g_t and k_t for such schedules.

We will now show that this reordering does indeed a) not destroy equivalence and b) creates an ordered schedule. In the process of proving this, we need to show that moving each k_t to position t is fine, under the induction hypothesis that the schedule is already t-ordered. Intuitively speaking, the following observations should help us to do this:

- Since g_t is the next global step and k_t is by definition not later than g_t, all the intermediate steps $t' \in [t : k_t)$ are local.
- Since all those local steps are also ordered, the schedule is k_t-ordered (not just t-ordered) and thus also k_t-abstract (by Lemma 305).
- Step k_t is only global if it is a write buffer step of a processor in strong memory mode (the first case in the definition above), or if it is unit-concurrent with all steps $t' \in [t : k_t)$.

[6]It suffices here to consider one direction of unit-concurrency, because interrupts are shared and thus none of the steps in the interval interrupt other steps.

Intuitively speaking, it should be easy to reorder a sequence of local steps with a global step, since races by assumption are shared, and shared steps are usually not local; therefore, there should not be any races between the sequence of local steps and the global step. This intuition turns out to be not quite correct, since local steps in the low-level machine may be global in the high-level machine, namely if they are made by a processor in strong memory mode which issues a write: in the low-level machine, the write goes to the buffer and is not executed, and thus the step is by definition local; in the high-level machine, the write is executed immediately, and thus the step is by definition global.

The proof is therefore more careful, and proceeds in three steps.

1. We move step k_t to position t. Because all intermediate steps are local in the low-level machine, they either have no effect in the low-level machine, or are not write-read racing with step k_t. Therefore in the low-level machine, step k_t can be executed at position t. This could still mess up all of the intermediate steps, in case step k_t modifies something other steps did not read.

2. To show that each intermediate step $l' \in [t : k_t)$ is not messed up after having moved k_t to t, and only for that reason, we temporarily push k_t (which is now at position t) as far back towards l' as possible in the high-level machine. If there are no races, it is possible to move to l', but in general we can only move until the first race, which occurs at some k'

   $$RW_\uparrow[t \leftarrow k_t][t \to k'](k', k'+1).$$

   As mentioned above, such a race is only possible if step $k'+1$ (formerly k') is a processor step in strong memory mode that issues a write

   $$\begin{aligned} & s[t \leftarrow k_t][t \to k'](k'+1) \in \Sigma_{P,i} \\ \wedge\ & SC_\uparrow[t \leftarrow k_t][t \to k'](k'+1) \\ \wedge\ & BW_\uparrow[t \leftarrow k_t][t \to k'](k'+1) \neq \emptyset. \end{aligned}$$

   Since local steps are not write buffer steps, that write is still buffered at $l'+1$ (formerly l')

   $$k'+1 \in issue^{l'}_\uparrow(i)$$

   and the memory mode of the unit is unchanged

   $$SC_{i\uparrow}[t \leftarrow k_t][t \to k'](l'+1) = SC_\uparrow[t \leftarrow k_t][t \to k'](k'+1).$$

   At this point we wish to apply Condition RMWRace to show that there is no visible write-read race between k' (formerly k_t) and $l'+1$ (formerly l'). At first, this only works if all of the steps t' between $k'+1$ and $l'+1$ are made by the same object.

3. To repeat the argument on schedules where the steps are not made by the same object, we simply sort the interval $[t : k_t)$ of local steps according to some strict weak order $\prec$ on objects, where that object P,i is the least object. To do this we show that two local steps in an abstract schedule never race with each other nor send interrupts, and thus can be reordered (unless they are made by the same object). As a result, all steps of that object are moved to the front, and with the previous argument we obtain that there are no such visible write-read races in this schedule. Using a slightly technical argument we obtain that the original, unsorted schedule also has no visible write-read races.

4. We conclude that no intermediate steps are messed up by the reordering, and thus the schedule sO_{t+1} after reordering k_t to position t is equivalent to schedule sO_t before the reordering.

We begin by showing in Section 4.12.1 that the interval of local steps can indeed be sorted in this way. Such a sorting results in a schedule where all steps at the beginning of the interval are made by the same object, which we therefore call sorted. We then show that these sorted schedules have no such visible write-read races in Section 4.12.2. We then show how to transfer these results to schedules which are not sorted in Section 4.12.3. We combine the results in Section 4.12.5.

4.12.1 Sorting a Local Segment

We turn to schedules where a sequence of steps is local. This case arises naturally in our construction of ordered schedules, where we look for the next global steps: all steps until that step are by definition local. At certain points, we need to sort these schedules in order to be able to apply Condition RMWRace, which requires that certain steps are made by the same unit. In this section we show that this is indeed possible.

A local step in an abstract schedule is unit-concurrent with steps made by other units.

Lemma 316.

$$\Gamma^t_*(s) \wedge L_\downarrow(t) \wedge s \in \mathrm{ABS}_{t+1} \wedge diffu(t,t') \rightarrow ucon_\uparrow(t,t').$$

Proof. Steps are unit-concurrent if they are made by different units and do not interrupt each other

$$ucon_\uparrow(t,t') \equiv diffu(t,t') \wedge \neg int_\uparrow(t,t').$$

Since the units are different by assumption, it suffices to show that there is no interrupt. Assume now for the sake of contradiction that step t interrupts step t'

$$int_\uparrow(t,t').$$

By definition, this means that the unit making step t' is a victim of step t

$$u_\uparrow(t') \in victims_\uparrow(t).$$

By Lemma 270 we obtain that this is also the case in the low-level machine

$$u_\uparrow(t') \in victims_\downarrow(t).$$

This contradicts Lemma 134. □

If the second step is a reduced local step, there is no race at all.

Lemma 317. *Two adjacent local reduced valid steps by different units*

$$\Gamma^{t+1}_*(s) \wedge L_\downarrow(t) \wedge L_\downarrow(t+1) \wedge s \in \mathrm{ABS}_{t+2} \wedge diffu(t,t+1)$$

do not race

$$\neg WR_\downarrow(t,t+1) \wedge \neg RW_\downarrow(t,t+1) \wedge \neg WW_\downarrow(t,t+1).$$

Proof. The steps are unit-concurrent in both directions due to Lemma 316

$$ucon_{\uparrow}(t,t+1) \wedge ucon_{\uparrow}(t+1,t).$$

Since the steps are local, they are not shared reads in the low-level machine

$$\neg ShR_{\downarrow}(t) \wedge \neg ShR_{\downarrow}(t+1),$$

and by Lemma 269 also not in the high-level machine

$$\neg ShR_{\uparrow}(t) \wedge \neg ShR_{\uparrow}(t+1).$$

By contraposition of Lemma 205 we obtain that there is no write-read race

$$\neg WR_{\uparrow}(t,t+1)$$

and by contraposition of Lemma 210 that there is no read-write race

$$\neg RW_{\uparrow}(t,t+1).$$

Assume now for the sake of contradiction that there is a race in the low-level machine

$$WR_{\downarrow}(t,t+1) \vee RW_{\downarrow}(t,t+1) \vee WW_{\downarrow}(t,t+1).$$

By Lemma 136 neither of the steps is a write buffer step

$$s(t), s(t+1) \notin \Sigma_{WB,i}$$

and by Lemma 265 the outputs of the low-level machine are subsumed by those in the high-level machine

$$out_{\downarrow}(t) \subseteq out_{\uparrow}(t),\ out_{\downarrow}(t+1) \subseteq out_{\uparrow}(t+1).$$

By Lemma 270 the inputs of the low-level machine are the inputs in the high-level machine

$$in_{\downarrow}(t) = in_{\uparrow}(t),\ in_{\downarrow}(t+1) = in_{\uparrow}(t+1).$$

We obtain that there is no write-read or read-write race in the low-level machine, either

$$\begin{aligned}
& WR_{\downarrow}(t,t+1) \vee RW_{\downarrow}(t,t+1) \\
\iff & out_{\downarrow}(t) \mathbin{\hat{\cap}} in_{\downarrow}(t+1) \vee in_{\downarrow}(t) \mathbin{\hat{\cap}} out_{\downarrow}(t+1) \\
\implies & out_{\uparrow}(t) \mathbin{\hat{\cap}} in_{\uparrow}(t+1) \vee in_{\uparrow}(t) \mathbin{\hat{\cap}} out_{\uparrow}(t+1) \\
\iff & WR_{\uparrow}(t,t+1) \vee RW_{\uparrow}(t,t+1) \\
\iff & 0,
\end{aligned}$$

leaving only a write-write race

$$WW_{\downarrow}(t,t+1).$$

We can similarly lift this race to the high-level machine

$$\begin{aligned}
WW_{\downarrow}(t,t+1) & \iff out_{\downarrow}(t) \mathbin{\hat{\cap}} out_{\downarrow}(t+1) \\
& \implies out_{\uparrow}(t) \mathbin{\hat{\cap}} out_{\uparrow}(t+1) \\
& \iff WW_{\uparrow}(t,t+1).
\end{aligned}$$

By Lemma 205, step t must be shared

$$Sh_{\uparrow}(t).$$

By Lemma 159, step t must be a memory write

$$mwrite_{\downarrow}(t)$$

and since it is local, it can not be shared

$$\neg Sh_{\downarrow}(t).$$

By Lemma 269, that is also true in the high-level machine

$$\neg Sh_{\uparrow}(t),$$

which is a contradiction. □

Consequently, local steps of different units can be reordered arbitrarily.

Lemma 318. *Two adjacent local reduced valid steps by different units*

$$\Gamma_{\downarrow}^{t+1}(s) \wedge L_{\downarrow}(t) \wedge L_{\downarrow}(t+1) \wedge s \in \mathrm{ABS}_{t+2} \wedge diffu(t,t+1)$$

can be reordered

$$s \equiv_{\downarrow} s[t \leftrightarrow t+1].$$

Proof. By Lemma 317 there is no race

$$\neg WR_{\downarrow}(t,t+1) \wedge \neg RW_{\downarrow}(t,t+1) \wedge \neg WW_{\downarrow}(t,t+1).$$

The claim follows with Lemma 167. □

This simple lemma allows us to take out the big guns and sort the local steps in the interval $[t:k)$ without destroying equivalence. In order for this sorting to be fully useful, we also need to maintain a few other key properties which are not generally stable under equivalence, such as IPI-validity[7].

Lemma 319. *Let $\prec$ be a total order over units. Then if in a k-ordered schedule that is valid and IPI-valid until $k-1$*

$$\Gamma_{\downarrow}^{k-1}(s) \wedge \Delta_{IPI\downarrow}^{k-1}(s) \wedge s \in \mathrm{ORD}_k$$

steps $[t:k)$ are local

$$\forall t' \in [t:k).L_{\downarrow}(t'),$$

those local steps can be arranged by a sequence of operations $\prec(t,k)$ such that

1. *The schedule and the rearranged schedule are equivalent*

$$s \prec (t,k) \equiv_{\downarrow} s,$$

[7]To see that IPI validity is not stable under equivalence, we change the order of the last write buffer step of a unit and a step that interrupts that unit. The schedules may well be equivalent but certainly are no longer IPI valid, as the write buffer is no longer empty during the interrupt.

2. *Only those local steps are reordered*

$$s \prec (t,k)[0:t-1] = s[0:t-1] \wedge s \prec (t,k)[k:\infty] = s[k:\infty]$$

3. *The local steps are only reordered among each other*

$$\#X{\approx}n(s) \in [t:k) \iff \#X{\approx}n(s \prec (t,k)) \in [t:k),$$

4. *The local steps remain local*

$$\forall t' \in [t:k).L_{\downarrow} \prec (t,k)(t'),$$

5. *The schedule remains valid and ordered*

$$\Gamma_{\downarrow}^{k-1}(s \prec (t,k)) \wedge \Delta_{IPI\downarrow}^{k-1}(s \prec (t,k)) \wedge s \prec (t,k) \in \mathrm{ORD}_k.$$

6. *After the reordered portion, the configurations are the same*

$$c_{\downarrow}^{k} = c_{\downarrow} \prec (t,k)^{k}.$$

7. *The local steps are now sorted by $\prec$, i.e., for $t' \in [t:k)$*

$$u(s \prec (t,k)(t')) \prec u(s \prec (t,k)(t'+1)) \vee t'+1 = k,$$

Proof. We simply sort the slice $s[t:k-1]$ with bubblesort [Knu98]. Bubblesort only uses swaps of adjacent values. We define

$$\prec (t,k)$$

to be the sequence of swaps performed by bubblesort. Claim 7 (that the resulting schedule is sorted) is satisfied by the correctness of bubblesort.

We prove the remaining claims by induction on the sequence of swaps in $\prec (t,k)$. The base case is trivial. In the inductive step $O \to O[t' \leftrightarrow t'+1]$, we show the claims individually. Note that we never swap steps of the same unit

$$diffu(t',t'+1)$$

and that we never swap steps outside of the reordered portion

$$t',t'+1 \in [t:k).$$

Let for the sake of brevity O' be the schedule after the additional swap

$$O' = O[t' \leftrightarrow t'+1].$$

$sO' \equiv_{\downarrow} s$: By the induction hypothesis, the sequence O preserved equivalence

$$sO \equiv_{\downarrow} s.$$

By the induction hypothesis, we also obtain that the schedule is valid and ordered

$$\Gamma_{\downarrow}^{k-1}(sO) \wedge \Delta_{IPI\downarrow}^{k-1}(sO) \wedge sO \in \mathrm{ORD}_k,$$

and by Lemma 305 thus abstract

$$sO \in \mathrm{ABS}_k.$$

By the induction hypothesis, we also obtain that the steps are still local

$$\forall t' \in [t:k).L_{\downarrow}O(t')$$

By Lemma 318 we can swap the steps

$$sO \equiv_{\downarrow} sO'$$

and the claim follows by transitivity

$$sO' \equiv_{\downarrow} s.$$

$sO'[0:t-1] = s[0:t-1] \wedge sO'[k:\infty] = s[k:\infty]$**:** By the induction hypothesis the schedule sO is the same as s on those portions

$$sO[0:t-1] = s[0:t-1] \wedge sO[k:\infty] = s[k:\infty]$$

The swap by definition only affects steps within the interval

$$sO'[0:t-1] = sO[0:t-1] \wedge sO'[k:\infty] = sO[k:\infty]$$

and the claim follows.

$\#X{\approx}n(s) \in [t:k) \iff \#X{\approx}n(sO') \in [t:k)$**:** When we swap two adjacent steps, we obtain by Lemma 117 that the position of those steps is swapped and the position of other steps is not moved. Therefore the position of steps stays within the interval in which the swaps are made.

$\forall t_S \in [t:k).L_{\downarrow}O'(t')$**:** Clearly t_S is the n-th step of some unit X

$$t_S = \#X{\approx}n(sO').$$

We have already shown that we reorder the local steps only among themselves. Therefore the original position t_O of the step was also in that interval

$$t_O = \#X{\approx}n(s) \in [t:k).$$

Steps in that interval were by assumption local

$$L_{\downarrow}(t_O).$$

Because the schedules are equivalent, the steps have strong agreement

$$c_{\downarrow}O'^{t_S} =_{\downarrow}^{sO'(t_S)} c_{\downarrow}^{t_O}$$

and execute the same step

$$sO'(t_S) = s(t_O).$$

The claim follows with Lemma 150:

$$L_{\downarrow}O'(t_S) = \neg G_{\downarrow}O'(t_S) = \neg G_{\downarrow}(t_O) = L_{\downarrow}(t_O) = 1.$$

$c_\downarrow^k = c_\downarrow O'^k$**:** By Lemma 317 there is no race

$$\neg WR_\downarrow O(t',t'+1) \wedge \neg RW_\downarrow O(t',t'+1) \wedge \neg WW_\downarrow O(t',t'+1).$$

By Lemma 166 the configurations after the swap are the same

$$c_\downarrow O^{t'+2} = c_\downarrow O'^{t'+2}.$$

By definition the schedules after the swap are the same

$$sO[t'+2:k-1] = sO[t' \leftrightarrow t'+1][t'+2:k-1],$$

and therefore the configurations at k are the same

$$c_\downarrow O^k = c_\downarrow O'^k.$$

The claim follows with the induction hypothesis

$$c_\downarrow^k = c_\downarrow O^k = c_\downarrow O'^k.$$

$\Gamma_\downarrow^{k-1}(sO) \wedge \Delta_{IPI\downarrow}^{k-1}(sO) \wedge sO' \in \mathrm{ORD}_k$**:** We have already shown that the first portion of the schedule is the same

$$sO'[0:t-1] = s[0:t-1]$$

and thus the independence of those steps can be taken from the original schedule

$$sO' \in \mathrm{ORD}_t.$$

The remaining steps $[t:k)$ are all local, and independence follows

$$sO' \in \mathrm{ORD}_k.$$

For validity, we obtain similarly that the validities for the first t steps still hold simply because the schedule is the same

$$\Gamma_\downarrow^t(sO') \wedge \Delta_{IPI\downarrow}^t(\text{`}sO).$$

For each remaining step $t_S \in [t:k)$ we first obtain that it was executed at step $t_O \in [t:k)$

$$t_S = \#X{\approx}n(sO'),\ t_O = \#X{\approx}n(s).$$

Step t_O was thus valid by assumption

$$\Gamma_\downarrow(t_O) \wedge \Delta_{IPI\downarrow}(t_O).$$

With equivalence we obtain strong agreement

$$c_\downarrow O'^{t_S} =_\downarrow^{s(t_S)} c_\downarrow^{t_O},$$

and by Lemma 150 we obtain that the step satisfies the guard condition

$$\Gamma_\downarrow O'(t_S) = \Gamma_\downarrow(t_O) = 1.$$

Note that the step is local and thus by Lemma 135 IPI-valid

$$\Delta_{IPI\downarrow} O'(t_S).$$

The claims follow, i.e., for $X \in \{\Gamma, \Delta_{IPI}\}$

$$\begin{aligned} X_{\downarrow}^{k-1}(sO') &= \forall k' < k.X_{\downarrow}O'(k') \\ &= (\forall k' < t.X_{\downarrow}O'(k')) \wedge (\forall t_S \in [t:k).X_{\downarrow}O'(t_S)) \\ &= X_{\downarrow}^{t}(sO') \wedge (\forall t_S \in [t:k).X_{\downarrow}O'(t_S)) \\ &= 1. \end{aligned}$$

□

4.12.2 Transferring Races in a Sorted Schedule

In this subsection we show that certain races in the low-level machine imply races in the high-level machine, which we can exclude with the conditions. The races are between one local step and one step which is not a step of a write buffer in strong memory mode; therefore, outputs of the steps can be easily transferred to the high-level with Lemma 265, which covers write-read, write-write, and read-write races. For visible write-read races we have to work a bit harder.

During the reordering, we will have to deal with three distinct types of situations:

1. We have to move a write buffer step in strong memory mode across a sequence of local steps,
2. we have to move a global step across a sequence of local steps made by different units,
3. we have to move a local step across a sequence of local steps made by different units.

All three situations have a few properties in common, which we bundle together in one definition. We say that schedule s has *a local tail from t to k* and write $\mathcal{L}[s](t,k)$ when all of the following hold.

1. The schedule is t-ordered, valid until k, and IPI-valid until $t-1$

$$\Gamma_{\downarrow}^{k}(s) \wedge \Delta_{IPI\downarrow}^{t-1}(s) \wedge s \in \mathrm{ORD}_t,$$

2. and all steps from t and before k are local

$$\forall t' \in [t:k).L_{\downarrow}[s](t').$$

Situations 1 and 2 are distinct because write buffer steps in strong memory mode have no effect in the high-level machine, and we can not use lemmas about races to deduce anything about them; but the write is already buffered, and we can use lemmas from Section 4.11.1 to show that there are no races. Additional complications arise from the fact that the steps in the local tail are not made by different units.

Situations 2 and 3 are distinct because local steps are always ordered, and thus placing a local step directly behind an ordered segment yields a longer ordered segment;

if we move a global step, however, the longer segment is only ordered if the earlier segment ended in a clean step.

We do our best to handle Situations 2 and 3 with the same Lemmas, by catching both of them with the following definition. We say that schedule s has *a local tail with an independent end from t to k* and write $\mathcal{L}^I[s](t,k)$ when all of the following hold.

1. The schedule has a local tail from t to k

$$\mathcal{L}[s](t,k),$$

2. the configuration at t is clean

$$clean_{\downarrow}[s](t),$$

3. step k is not a write buffer step in strong memory mode

$$\nexists i.SC_{\downarrow}(k) \wedge s(k) \in \Sigma_{WB,i},$$

4. and all steps from t and before k are unit-concurrent with k

$$\forall t' \in [t:k).ucon_{\downarrow}[s](k,t').$$

Lemma 320. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k),$$

it is k-ordered

$$s \in \mathrm{ORD}_k.$$

Proof. The schedule is t-ordered

$$s \in \mathrm{ORD}_t.$$

Steps from t and before k are local

$$\forall t' \in [t:k).L_{\downarrow}(t'),$$

and thus ordered

$$\forall t' \in [t:k).ord_{\downarrow}(t').$$

The claim follows

$$s \in \mathrm{ORD}_k.$$

□

Lemma 321. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k),$$

it is IPI-valid until $k-1$

$$\Delta^{k-1}_{IPI_{\downarrow}}(s).$$

Proof. The schedule is IPI-valid until $t-1$

$$\Delta_{IPI\downarrow}^{t-1}(s).$$

Steps from t and before k are local

$$\forall t' \in [t:k).L_{\downarrow}(t'),$$

and thus by Lemma 135 IPI-valid

$$\forall t' \in [t:k).\Delta_{IPI\downarrow}(t').$$

The claim follows

$$\Delta_{IPI\downarrow}^{k-1}(s).$$

□

Lemma 322. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k),$$

it is l-abstract for all $l \leq k$

$$s \in \mathrm{ABS}_l.$$

Proof. Since it has a local tail, the schedule is valid until k

$$\Gamma_{\downarrow}^{k}(s),$$

and by Lemmas 320 and 321 also k-ordered and IPI-valid until $k-1$

$$\Delta_{IPI\downarrow}^{k-1}(s) \wedge s \in \mathrm{ORD}_k.$$

By Lemma 305 the schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and thus also l-abstract

$$s \in \mathrm{ABS}_l.$$

□

Lemma 323. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k),$$

then the schedule is valid until $k-1$ in the high-level machine

$$\Gamma_{\uparrow}^{k-1}(s).$$

Proof. By Lemma 322 the schedule is abstract until k

$$s \in \mathrm{ABS}_k,$$

and since the schedule has a local tail it is valid in the low-level machine until k

$$\Gamma_{\downarrow}^{k}(s)$$

and thus also until $k-1$

$$\Gamma_{\downarrow}^{k-1}(s).$$

The claim is Lemma 271

$$\Gamma_{\uparrow}^{k-1}(s).$$

□

Lemma 324. *Step t after moving k to t is made by a write buffer iff step k was*

$$s[t \leftarrow k](t) \in \Sigma_{WB,i} \equiv s(k) \in \Sigma_{WB,i}.$$

Proof. The oracle inputs are the same

$$s[t \leftarrow k](t) = s(k)$$

and the claim follows. □

Lemma 325. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k)$$

then the sequence of issued writes at $l \in [t:k]$ is a prefix of the sequence at k

$$\exists q.issue^k_\downarrow(i) = issue^l_\downarrow(i) \circ q.$$

Proof. Since there is a local tail, the steps $t' \in [t:k)$ are local

$$\mathcal{L}_\downarrow(t'),$$

and by contraposition of Lemma 136 not write buffer steps of unit i

$$s(t') \notin \Sigma_{WB,i}.$$

The claim is now Lemma 128. □

Lemma 326. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k)$$

and the sequence of issued writes at $l \in [t:k]$ is non-empty

$$issue^l_\downarrow(i) \neq \varepsilon,$$

then the head of the write buffer at l is exactly the same as at k

$$hd(issue^l_\downarrow(i)) = hd(issue^k_\downarrow(i)).$$

Proof. By Lemma 325, step k has an additional sequence of writes q

$$issue^k_\downarrow(i) = issue^l_\downarrow(i) \circ q,$$

and the claim is Lemma 40

$$hd(issue^k_\downarrow(i)) = hd(issue^l_\downarrow(i)).$$

□

Lemma 327. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k)$$

and step k is a write buffer step of unit i issued before t

$$s(k) \in \Sigma_{WB,i} \wedge hd(issue^k_\downarrow(i)) < t,$$

then the mode of unit i at $l \in [t:k]$ is the same as at k

$$SC_{i\uparrow}(l) = SC_{i\uparrow}(k).$$

Proof. By Lemma 323 the schedule is valid until $k-1$

$$\Gamma_{\uparrow}^{k-1}(s).$$

Let now t' be the step when the write committed at k was issued

$$t' = hd(issue_{\downarrow}^{k}(i)).$$

By Lemma 322 the schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and with Lemma 267 we obtain that the sequence of issued writes is the same in both machines, and therefore t' is contained in both of them

$$t' \in issue_{\downarrow}^{k}(i) = issue_{\uparrow}^{k}(i).$$

By the monotonicity of the sequence of issued writes, the write was already buffered at l

$$t' \in issue_{\uparrow}^{l}(i),$$

and by applying twice Lemma 168 the mode is unchanged in the high-level machine

$$SC_{i\uparrow}(l) = SC_{i\uparrow}(t') = SC_{i\uparrow}(k),$$

which was the claim. □

Lemma 328. *If the schedule has a local tail from t to k*

$$\mathcal{L}(t,k)$$

and step k is a write buffer step of unit i issued before t

$$s(k) \in \Sigma_{WB,i} \wedge hd(issue_{\downarrow}^{k}(i)) < t,$$

then the mode of unit i at t is the same as at k

$$SC_{i\downarrow}(t) = SC_{i\downarrow}(k).$$

Proof. By Lemma 327 we obtain the claim for the high-level machine

$$SC_{i\uparrow}(t) = SC_{i\uparrow}(k).$$

By Lemma 322 the schedule is t-abstract

$$s \in \mathrm{ABS}_t.$$

The claim follows by applying twice Lemma 268

$$SC_{i\downarrow}[t \leftarrow k](t) = SC_{i\uparrow}[t \leftarrow k](t) = SC_{i\uparrow}(k) = SC_{i\downarrow}(k).$$

□

Lemma 329. *Let O be some reordering that moves $t'' < k$ to $t' < k$*

$$sO(t') = s(t'').$$

If the schedule and its reordering sO have a local tail from t to k

$$\mathcal{L}(t,k) \wedge \mathcal{L}O(t,k),$$

we have strong agreement with the low-level machine iff we have strong agreement with the high-level machine

$$c_\uparrow O^{t'} =_\uparrow^{sO(t')} c_\uparrow^{t''} \iff c_\downarrow O^{t'} =_\downarrow^{sO(t')} c_\downarrow^{t''}.$$

Proof. By Lemma 322 the schedules are $t''+1$- resp. $t'+1$-abstract

$$s \in \mathrm{ABS}_{t''+1} \wedge sO \in \mathrm{ABS}_{t'+1},$$

and the claim is Lemma 260. □

The outputs of local steps are always visible in the high-level machine.

Lemma 330.

$$\Gamma_*^t(s) \wedge s \in \mathrm{ABS}_{t+1} \wedge L_\downarrow(t) \to out_\downarrow(t) \subseteq out_\uparrow(t).$$

Proof. By contraposition of Lemma 136, step t is not a write buffer step

$$s(t) \notin \Sigma_{WB,i}$$

and therefore also not a sequentially consistent write buffer step

$$\neg(SC_\uparrow(t) \wedge s(t) \in \Sigma_{WB,i}).$$

The claim is just Lemma 265. □

If a local step is a shared write, it must be a processor step in strong memory mode that is buffering a write.

Lemma 331. *If in a t-valid, $t+1$-abstract schedule a local step executes a shared memory write in the high-level machine*

$$\Gamma_*^t(s) \wedge s \in \mathrm{ABS}_{t+1} \wedge L_\downarrow(t) \wedge Sh_\uparrow(t) \wedge mwrite_\uparrow(t),$$

the step is a processor step in strong memory mode that is buffering a write

$$\exists i.s(t) \in \Sigma_{P,i} \wedge SC_\uparrow(t) \wedge BW_\uparrow(t) \neq \emptyset.$$

Proof. The witness i is simply the unit making the step

$$i := u_\uparrow(t).$$

By contraposition of Lemma 136, the step is not made by a write buffer

$$s(t) \notin \Sigma_{WB,i},$$

and is thus made by the processor of the unit making the step

$$s(t) \in \Sigma_{P,i},$$

which is the first part of the claim.

By Lemma 269, the step is also shared in the low-level machine

$$Sh_{\downarrow}(t)$$

and since it is local it is not a memory write

$$\neg mwrite_{\downarrow}(t).$$

Thus the outputs of the step are a subset of the NPR in the low-level machine, but not in the high-level machine

$$out_{\downarrow}(t) \subseteq A_{NPR,i} \wedge out_{\uparrow}(t) \not\subseteq A_{NPR,i},$$

and we conclude the outputs are not the same

$$out_{\downarrow}(t) \neq out_{\uparrow}(t).$$

By contraposition of Lemma 264, the step must have been in sequentially consistent mode

$$SC_{\downarrow}(t),$$

which is the second part of the claim.

We further conclude with Lemma 270 that the domain of the prepared writes of the high-level machine are a subset of the NPR

$$\begin{aligned} Dom(PW_{\uparrow}(t).bpa) &= Dom(PW_{\downarrow}(t).bpa) && \text{L 270} \\ &\subseteq Dom(W_{\downarrow}(t)) \\ &\subseteq idc(Dom(W_{\downarrow}(t))) \\ &= out_{\downarrow}(t) \\ &\subseteq A_{NPR,i}. \end{aligned}$$

With Lemma 177 the third part of the claim follows

$$BW_{\uparrow}(t) \neq \emptyset.$$

□

Lemma 332. *If in an ordered, semi-valid schedule*

$$\Gamma\Phi^{k}_{\downarrow}(s) \wedge s \in \mathrm{ORD}_k$$

step $k-1$ *is a local step not made by step* k

$$L_{\downarrow}(k-1) \wedge diffu(k-1,k),$$

then step $k-1$ *does not modify the code of step* k *in the low-level machine*

$$\neg CM_{\downarrow}(k-1,k).$$

Proof. Assume for the sake of contradiction that there is a code modification

$$CM_{\downarrow}(k-1,k)$$

and thus a write-read race

$$WR_{\downarrow}(k-1,k).$$

By Lemma 159, step $k-1$ is a memory write

$$mwrite_{\downarrow}(k-1)$$

and because it is local it can not be shared

$$\neg Sh_{\downarrow}(k-1).$$

By Lemma 305 the schedule is k-abstract

$$s \in \mathrm{ABS}_k.$$

We apply Lemma 269 and obtain that step $k-1$ is also not shared in the high-level machine

$$\neg Sh_{\uparrow}(k-1).$$

By Lemma 316, steps $k-1$ and k are unit-concurrent

$$ucon_{\uparrow}(k-1,k),$$

and by contraposition of Condition CodeMod, there is no code modification in the high-level machine

$$\neg CM_{\uparrow}(k-1,k).$$

By Lemma 269, step k fetches the same addresses in the low-level machine

$$F_{\downarrow}(k) = F_{\uparrow}(k),$$

and by Lemma 330 step $k-1$ does not have fewer outputs in the high-level machine

$$out_{\downarrow}(k) \subseteq out_{\uparrow}(k).$$

We conclude that there is also no code modification in the low-level machine

$$\begin{aligned} \neg CM_{\uparrow}(k-1,k) &\iff out_{\uparrow}(k-1) \not\parallel F_{\uparrow}(k) \\ &\implies out_{\downarrow}(k-1) \not\parallel F_{\downarrow}(k) \\ &\implies \neg CM_{\downarrow}(k-1,k), \end{aligned}$$

which is a contradiction. □

Lemma 333. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

then each $t' \in [t:k]$ is made by a different unit than k

$$diffu(k,t').$$

Proof. Since the schedule has a local tail with an independent end, step k is unit-concurrent with all of those steps

$$ucon_{\downarrow}(k,t')$$

and thus by definition made by a different unit

$$diffu(k,t')$$

which is the claim. □

We now begin showing that step k in the low-level machine can be moved to t without messing up step k. We begin by showing that nothing changes except maybe for the read-results.

Lemma 334. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

we can move step k to position t without affecting its write buffers, core registers, fetched registers, and IPI-relevant registers

$$wb_\downarrow[t \leftarrow k]^t =_{u_\downarrow(k)} wb_\downarrow^k \wedge m_\downarrow[t \leftarrow k]^t =_{C_\downarrow(k) \cup F_\downarrow(k) \cup A_{IPI}(s(k))} m_\downarrow^k.$$

Proof. The configuration at t when moving k to t is the same as in the original schedule

$$c_\downarrow[t \leftarrow k]^t = c_\downarrow^t.$$

We use this and generalize the claim to $t' \in [t : k]$

$$wb_\downarrow^{t'} =_{u_\downarrow(k)} wb_\downarrow^k \stackrel{!}{\wedge} m_\downarrow^{t'} =_{C_\downarrow(k) \cup F_\downarrow(k) \cup A_{IPI}(s(k))} m_\downarrow^k.$$

We prove the claim by downwards induction on t', starting from k. The base case is trivial.

In the inductive step from t' to $t'-1$, we have by the induction hypothesis we now obtain that we can move step k to position t'

$$wb_\downarrow^{t'} =_{u_\downarrow(k)} wb_\downarrow^k \wedge m_\downarrow^{t'} =_{C_\downarrow(k) \cup F_\downarrow(k) \cup A_{IPI}(s(k))} m_\downarrow^k.$$

By Lemma 333 we obtain that step k and $t'-1$ we made by different units

$$diffu(k, t'-1),$$

and thus the units making those steps were different

$$u_\downarrow(k) \neq u_\downarrow(t'-1).$$

With Lemma 96 we obtain that the write buffer at $t'-1$ is the same as that at t', and therefore also as that at k

$$wb_\downarrow^{t'-1} =_{u_\downarrow(k)} wb_\downarrow^{t'} =_{u_\downarrow(k)} wb_\downarrow^k,$$

which is the first claim.

With Lemma 316 we obtain that steps $t'-1$ and k are unit-concurrent in the original schedule

$$ucon_\downarrow(t'-1, k)$$

and by contraposition of Lemma 104 step $t'-1$ does not modify the core registers of step k

$$out_\downarrow(t'-1) \not\upharpoonright C_\downarrow(k).$$

Note that step k is stepped at t' after moving k to t'

$$s[t' \leftarrow k](t') = s(k)$$

and we obtain with Lemma 152 that step t' after the move is feasible and has the same core and fetched registers

$$\begin{aligned}\Phi_\downarrow[t' \leftarrow k](t') &= \Phi_\downarrow(k) = 1,\\ C_\downarrow[t' \leftarrow k](t') &= C_\downarrow(k),\\ F_\downarrow[t' \leftarrow k](t') &= F_\downarrow(k).\end{aligned}$$

Since the steps before t' are not affected by moving k to t', we obtain that step $t'-1$ is still local

$$L_\downarrow[t' \leftarrow k](t'-1),$$

and that the steps until $t'-1$ are still valid and ordered

$$\Gamma_\downarrow^{t'-1}(s[t' \leftarrow k]) \wedge s[t' \leftarrow k] \in \mathrm{ORD}_{t'}.$$

Clearly the steps are also still made by different units

$$\begin{aligned}diffu[t' \leftarrow k](t'-1,t') &\equiv u(s[t' \leftarrow k](t'-1)) \neq u(s[t' \leftarrow k](t'))\\ &\equiv u(s(k)) \neq u(s(t'-1))\\ &\equiv diffu(k,t'-1).\end{aligned}$$

With Lemma 332 we obtain that step $t'-1$ does not modify code of step t' after moving k to t'

$$\neg CM_\downarrow[t' \leftarrow k](t'-1,t')$$

and since the fetched registers of step t' after the reordering are those at k in the original schedule, step $t'-1$ does not modify fetched registers of step k in the original schedule

$$\begin{aligned}\neg CM_\downarrow[t' \leftarrow k](t'-1,t') &\equiv out_\downarrow[t' \leftarrow k](t'-1) \mathbin{\dot{\not\cap}} F_\downarrow[t' \leftarrow k](t')\\ &\equiv out_\downarrow(t'-1) \mathbin{\dot{\not\cap}} F_\downarrow(k).\end{aligned}$$

By Lemma 137, step $t'-1$ does not modify the IPI-relevant registers

$$out_\downarrow(t'-1) \mathbin{\dot{\not\cap}} A_{IPI}(s(k)).$$

Therefore the step modifies neither core nor fetched nor IPI-relevant registers

$$out_\downarrow(t-1) \mathbin{\dot{\not\cap}} C_\downarrow(k) \cup F_\downarrow(k) \cup A_{IPI}(s(k))$$

The second claim follows with Lemma 138

$$m_\downarrow^{t-1} =_{C_\downarrow(k) \cup F_\downarrow(k) \cup A_{IPI}(s(k))} m_\downarrow^t =_{C_\downarrow(k) \cup F_\downarrow(k) \cup A_{IPI}(s(k))} m_\downarrow^k.$$

□

The guard condition is also still satisfied.

Lemma 335. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

we can move step k to position t without violating the guard condition

$$\Gamma_\downarrow[t \leftarrow k](t).$$

Proof. By Lemma 334 the step can be executed at the new position

$$wb_{\downarrow}[t \leftarrow k]^t =_{u_{\downarrow}(k)} wb_{\downarrow}^k \wedge m_{\downarrow}[t \leftarrow k]^t =_{C_{\downarrow}(k) \cup F_{\downarrow}(k) \cup A_{IPI}(s(k))} m_{\downarrow}^k.$$

The claim follows immediately with Lemma 180. □

Lemma 336. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

we can move step k to position t and the schedule is $t+1$-abstract and -ordered, and valid until t, and IPI-valid until $t-1$

$$\Delta_{IPI\downarrow}^{t-1}(s[t \leftarrow k]) \wedge \Gamma_{\downarrow}^t(s[t \leftarrow k]) \wedge s[t \leftarrow k] \in \mathrm{ABS}_{t+1} \cap \mathrm{ORD}_{t+1}.$$

Proof. The schedule is t-ordered, valid until k, and IPI-valid until $t-1$

$$\Gamma_{\downarrow}^k(s) \wedge \Delta_{IPI\downarrow}^{t-1}(s) \wedge s \in \mathrm{ORD}_t.$$

The reordered schedule is the same until $t-1$

$$s[0:t-1] = s[t \leftarrow k][0:t-1]$$

and thus also valid and IPI-valid until $t-1$

$$\Gamma_{\downarrow}^{t-1}(s[t \leftarrow k]) \wedge \Delta_{IPI\downarrow}^{t-1}(s[t \leftarrow k]),$$

which solves the first part of the claim.

By Lemma 335 step t is also still valid

$$\Gamma_{\downarrow}[t \leftarrow k](t)$$

and thus the schedule is valid until t

$$\Gamma_{\downarrow}^t(s[t \leftarrow k]) \equiv \Gamma_{\downarrow}^{t-1}(s[t \leftarrow k]) \wedge \Gamma_{\downarrow}[t \leftarrow k](t) \equiv 1,$$

which solves the second part of the claim.

Furthermore, steps before t are still ordered

$$s[t \leftarrow k] \in \mathrm{ORD}_t.$$

Since the steps leading to that configuration are the same, it still is clean

$$\begin{aligned}
&clean_{\downarrow}[t \leftarrow k](t) \\
&= \forall i. \neg dirty_{\downarrow}[t \leftarrow k](t,i) \\
&= \forall i. \neg (SC_{i\downarrow}[t \leftarrow k](t) \wedge \exists t' \in issue_{\downarrow}[t \leftarrow k]^t(i).Sh_{\downarrow}[t \leftarrow k](t')) \\
&= \forall i. \neg (SC_{i\downarrow}(t) \wedge \exists t' \in issue_{\downarrow}^t(i).Sh_{\downarrow}(t')) \\
&= \forall i. \neg dirty_{\downarrow}(t,i) \\
&= clean_{\downarrow}(t).
\end{aligned}$$

Thus step t is still executed in a clean state and by Lemma 273 also ordered

$$ord_{\downarrow}[t \leftarrow k](t).$$

Therefore the schedule is $t+1$-ordered

$$s[t \leftarrow k] \in \mathrm{ORD}_{t+1}.$$

By Lemma 305 the schedule is also $t+1$-abstract

$$s[t \leftarrow k] \in \mathrm{ABS}_{t+1}$$

and the claim follows. □

We have thus moved the step successfully to position t. We will now move it back towards k as far as possible, which means until we hit a read-write race. We can therefore maintain strong agreement, and the victims of the step are unchanged. With t and k fixed, we say that the step is *executable at* k' when strong agreement is maintained at k' and the step is unit-concurrent with all the remaining steps

$$\mathcal{E}(t,k,k') \equiv c_\uparrow[t \leftarrow k]^t =_\uparrow^{s[t\leftarrow k](t)} c_\uparrow[k' \leftarrow k]^{k'} \wedge \forall t' \in [k' : k).ucon_\uparrow[k' \leftarrow k](k', t'+1).$$

Lemma 337. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}'(t,k),$$

the step is executable at t

$$\mathcal{E}(t,k,t).$$

Proof. We have to show that the steps are in strong agreement and that the step is unit-concurrent with all the steps until $k-1$

$$c_\uparrow[t \leftarrow k]^t =_\uparrow^{s[t\leftarrow k](t)} c_\uparrow[t \leftarrow k]^t \stackrel{!}{\wedge} \forall t' \in [t : k).ucon_\uparrow[t \leftarrow k](t, t'+1).$$

The first part holds by reflexivity. For the second part, we obtain by Lemma 334 that the IPI-relevant registers are not changed by the move

$$m_\downarrow^k =_{A_{IPI}(s(k))} m_\downarrow[t \leftarrow k]^t,$$

And thus the victims are the same

$$victims_\downarrow(k) = victims_\downarrow[t \leftarrow k](t).$$

By Lemma 336 the schedule is $t+1$-abstract

$$s[t \leftarrow k] \in \mathrm{ABS}_{t+1},$$

and by Lemma 270 the victims are the same in the two machines

$$victims_\uparrow[t \leftarrow k](t) = victims_\downarrow[t \leftarrow k](t) = victims_\downarrow(k).$$

The unit making step $t'+1$ in the high-level machine after the move was the unit making step t' in the low level machine

$$u_\uparrow[t \leftarrow k](t'+1) = u(s[t \leftarrow k](t'+1)) = u(s(t')) = u_\downarrow(t').$$

The claim follows from the assumption that step k was unit-concurrent with step t' in the low-level machine

$$\begin{aligned}
&ucon_{\uparrow}[t \leftarrow k](t,t'+1)\\
\equiv\ &diffu[t \leftarrow k](t,t'+1) \wedge \neg int_{\uparrow}[t \leftarrow k](t,t'+1)\\
\equiv\ &u(s[t \leftarrow k](t)) \neq u(s[t \leftarrow k](t'+1)) \wedge u_{\uparrow}[t \leftarrow k](t'+1) \notin victims_{\uparrow}[t \leftarrow k](t)\\
\equiv\ &u(s(k)) \neq u(s(t')) \wedge u_{\downarrow}(t') \notin victims_{\downarrow}(k)\\
\equiv\ &diffu(k,t') \wedge \neg int_{\downarrow}(k,t')\\
\equiv\ &ucon_{\downarrow}(k,t').
\end{aligned}$$

□

Lemma 338. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

then the new schedule is valid until k' in the high-level machine

$$\Gamma_{\uparrow}^{k'}(s[k' \leftarrow k]).$$

Proof. By Lemma 336 the schedule where k is moved to t is valid in the low-level machine and abstract until t

$$\Gamma_{\downarrow}'(s[t \leftarrow k]) \wedge s[t \leftarrow k] \in \mathrm{ABS}_{t+1}.$$

By Lemma 271 it is also valid in the high-level machine

$$\Gamma_{\uparrow}'(s[t \leftarrow k]),$$

and in particular step t is valid

$$\Gamma_{\uparrow}[t \leftarrow k](t).$$

The step can be executed at k' and thus there is strong agreement between those configurations

$$c_{\uparrow}[t \leftarrow k]^{t} =_{\uparrow}^{s[t \leftarrow k](t)} c_{\uparrow}[k' \leftarrow k]^{k'}.$$

By Lemma 150, step k' is thus also valid when moving k to k'

$$\Gamma_{\uparrow}[k' \leftarrow k](k').$$

By Lemma 323 the original schedule is valid until $k-1$ in the high-level machine

$$\Gamma_{\uparrow}^{k-1}(s).$$

Since the original schedule and schedule where k is moved to k' are the same before k' and k' is less than k, the reordered schedule is also valid before k'

$$\Gamma_{\uparrow}^{k'-1}(s[k' \leftarrow k])$$

Since step k' is valid and the steps before k' are valid, the steps until k' are valid

$$\Gamma_{\uparrow}^{k'}(s[k' \leftarrow k]),$$

which was the claim. □

Lemma 339. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

then step $k'+1$ is valid and there is no write-read race between k' and $k'+1$ in the reordered schedule

$$\Gamma_\uparrow[k' \leftarrow k](k'+1) \wedge \neg WR_\uparrow[k' \leftarrow k](k',k'+1).$$

Proof. The configuration at k' is the same in both schedules

$$c_\uparrow^{k'} = c_\uparrow[k' \leftarrow k]^{k'}.$$

By contraposition of Lemma 104 we obtain that the core registers of step $k'+1$ are not modified

$$out_\uparrow[k' \leftarrow k](k') \not\parallel C_\uparrow[k' \leftarrow k](k'+1),$$

and by Lemma 138 the memory content is the same at k' in the original schedule

$$m_\uparrow[k' \leftarrow k]^{k'+1} =_{C_\uparrow[k' \leftarrow k](k'+1)} m_\uparrow[k' \leftarrow k]^{k'} = m_\uparrow^{k'}.$$

By assumption, all steps until k are local

$$\forall t' \in [t:k).L_\downarrow(t').$$

In particular k' was local

$$L_\downarrow(k')$$

and thus not a shared read

$$\neg ShR_\downarrow(k'),$$

and by Lemma 269 we obtain that the step was also not a shared read in the high-level machine

$$\neg ShR_\uparrow(k')$$

We apply twice Lemma 151 and obtain that the step is feasible not a shared read in its new position

$$\Phi_\uparrow[k' \leftarrow k](k'+1) = \Phi_\uparrow(k') = 1,$$
$$ShR_\uparrow[k' \leftarrow k](k'+1) = ShR_\uparrow(k') = 0.$$

By Lemma 338 the schedule is valid until k'

$$\Gamma_\uparrow^{k'}(s[k' \leftarrow k]),$$

and thus also semi-valid until $k'+1$

$$\Gamma\Phi_\uparrow^{k'+1}(s[k' \leftarrow k]).$$

Since the step is executable at k', it is unit-concurrent with $k'+1$

$$ucon_\uparrow[k' \leftarrow k](k',k'+1).$$

By contraposition of Condition CodeMod we obtain that there is no code modification

$$\neg CM_{\uparrow}[k' \leftarrow k](k'+1).$$

We conclude that the fetched registers are also not modified

$$out_{\uparrow}[k' \leftarrow k](k') \not\parallel F_{\uparrow}[k' \leftarrow k](k'+1),$$

and by Lemma 138 the memory content of the fetched registers is the same at k' in the original schedule

$$m_{\uparrow}[k' \leftarrow k]^{k'+1} =_{F_{\uparrow}[k' \leftarrow k](k'+1)} m_{\uparrow}[k' \leftarrow k]^{k'} = m_{\uparrow}^{k'}.$$

Combining this with the fact that the core registers are also unchanged we obtain that both are unchanged

$$m_{\uparrow}[k' \leftarrow k]^{k'+1} =_{C_{\uparrow}[k' \leftarrow k](k'+1) \cup F_{\uparrow}[k' \leftarrow k](k'+1)} m_{\uparrow}[k' \leftarrow k]^{k'}.$$

Since the step is unit-concurrent, it is also made by a different unit

$$diffu[k' \leftarrow k](k', k'+1)$$

and in particular the units making the steps are different

$$u_{\uparrow}[k' \leftarrow k](k') \neq u_{\uparrow}[k' \leftarrow k](k'+1).$$

By Lemma 96 we obtain that step k' after the move did not modify the write buffer of the unit making step $k'+1$ after the move

$$wb_{\uparrow}[k' \leftarrow k]^{k'+1} =_{u_{\uparrow}[k' \leftarrow k](k'+1)} wb_{\uparrow}[k' \leftarrow k]^{k'} = wb_{\uparrow}^{k'}$$

and with Lemma 180 we obtain that step $k'+1$ is valid

$$\Gamma_{\uparrow}[k' \leftarrow k](k'+1) = \Gamma_{\uparrow}(k'),$$

which is the first claim.

By contraposition of Lemma 205 we obtain that there is no write-read race

$$\neg WR_{\uparrow}[k' \leftarrow k](k'+1),$$

which is the second claim. □

Combining Lemmas 338 and 339 we obtain that the schedule is semi-valid until $k'+1$.

Lemma 340. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

then the new schedule is semi-valid until $k'+1$ in the high-level machine

$$\Gamma\Phi_{\uparrow}^{k'+1}(s[k' \leftarrow k]).$$

Proof. For the steps until k' we use Lemma 338, for step $k'+1$ we obtain with Lemma 339 that it is valid

$$\Gamma_\uparrow[k' \leftarrow k](k'+1)$$

and thus by definition of Γ feasible

$$\Phi_\uparrow[k' \leftarrow k](k'+1),$$

which proves the claim. □

Lemma 341. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

then step $k'+1$ after the reordering and k' before the reordering strongly agree

$$c_\uparrow[k' \leftarrow k](k'+1) =_\uparrow^{s[k' \leftarrow k](k'+1)} c_\uparrow^{k'}.$$

Proof. By Lemma 339 there is no write-read race

$$\neg WR_\uparrow[k' \leftarrow k](k', k'+1)$$

and the claim is Lemma 164. □

Lemma 342. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

then step $k'+1$ is shared in both of the following cases

1. *there is a valid read-write race between k' and $k'+1$ in the reordered schedule*

$$\Gamma_\uparrow[k' \leftarrow k](k'+1) \wedge RW_\uparrow[k' \leftarrow k](k', k'+1) \rightarrow Sh_\uparrow[k' \leftarrow k](k'+1)$$

2. *there is a valid write-write race between k' and $k'+1$ in the reordered schedule*

$$\Gamma_\uparrow[k' \leftarrow k](k'+1) \wedge WW_\uparrow[k' \leftarrow k](k', k'+1) \rightarrow Sh_\uparrow[k' \leftarrow k](k'+1)$$

Proof. By Lemma 338, the schedule is valid until k'

$$\Gamma_\uparrow^{k'}(s[k' \leftarrow k]).$$

Thus the schedule is in both cases valid until $k'+1$

$$\Gamma_\uparrow^{k'+1}(s[k' \leftarrow k]).$$

We distinguish between the two claims.

$RW_\uparrow[k' \leftarrow k](k', k'+1) \to Sh_\uparrow[k' \leftarrow k](k'+1)$**:** By Lemma 341 there is strong agreement between the step at $k'+1$ when k is moved to k' and the step at k'

$$c_\uparrow[k' \leftarrow k]^{k'+1} =_\uparrow^{s[k' \leftarrow k](k'+1)} c_\uparrow^{k'}.$$

In the original schedule, the step had no victims by Lemma 134

$$victims_\uparrow(k') = \emptyset,$$

and by Lemma 150 also not at its new position

$$victims_\uparrow[k' \leftarrow k](k'+1) = victims_\uparrow(k') = \emptyset.$$

We conclude it does not interrupt k' in the reordered schedule

$$\neg int_\uparrow[k' \leftarrow k](k'+1, k')$$

and since it is made by a different unit, is unit-concurrent with step k' in the reordered schedule

$$ucon_\uparrow[k' \leftarrow k](k'+1, k').$$

By Lemma 210 step $k'+1$ is shared or a shared read

$$Sh_\uparrow[k' \leftarrow k](k'+1) \vee ShR_\uparrow[k' \leftarrow k](k'+1),$$

and since shared reads are shared by Lemma 129, it is shared either way

$$Sh_\uparrow[k' \leftarrow k](k'+1),$$

which is the claim.

$WW_\uparrow[k' \leftarrow k](k', k'+1) \to Sh_\uparrow[k' \leftarrow k](k'+1)$**:**
The claim follows directly from Lemma 205

□

Lemma 343. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and the step is executable at some $k' \in [t : k)$

$$\mathcal{E}(t,k,k'),$$

and there is no read-write race between k' and $k'+1$ in the reordered schedule

$$\neg RW_\uparrow[k' \leftarrow k](k', k'+1),$$

the step is also executable at $k'+1$

$$\mathcal{E}(t,k,k'+1).$$

Proof. By Lemma 339, there is also no write-read race

$$\neg WR_{\uparrow}[k' \leftarrow k](k', k'+1),$$

and by Lemma 165 there is strong agreement between the step at k' when k is moved to k' and the step at $k'+1$ when k' is afterwards moved to to $k'+1$

$$c_{\uparrow}[k' \leftarrow k]^{k'} =_{\uparrow}^{s[k' \leftarrow k](k')} c_{\uparrow}[k' \leftarrow k][k' \leftrightarrow k'+1]^{k'+1}.$$

Obviously, the oracle input at k' after moving k to k' is the same as at t after moving k to t

$$s[k' \leftarrow k](k') = s[t \leftarrow k](t),$$

and moving k' to $k'+1$ after moving k to k' is the same as moving k to $k'+1$

$$s[k' \leftarrow k][k' \leftrightarrow k'+1] = s[k'+1 \leftarrow k].$$

We conclude that the configurations at k' when moving k to t and at $k'+1$ after moving k to k' strongly agree when stepped with the oracle input at t when moving k to t

$$c_{\uparrow}[k' \leftarrow k]^{k'} =_{\uparrow}^{s[t \leftarrow k](t)} c_{\uparrow}[k'+1 \leftarrow k]^{k'+1},$$

and by transitivity and the assumption we obtain that the configurations at t after moving k to t and at $k'+1$ after moving k to $k'+1$ also agree

$$\begin{aligned} c_{\uparrow}[t \leftarrow k]^{t} &=_{\uparrow}^{s[t \leftarrow k](t)} c_{\uparrow}[k' \leftarrow k]^{k'} \\ &=_{\uparrow}^{s[t \leftarrow k](t)} c_{\uparrow}[k'+1 \leftarrow k]^{k'+1}, \end{aligned}$$

which is the first part of the claim.

For the second part of the claim, let $t' \in [k'+1 : k)$. By assumption step k' was unit concurrent with step $t'+1$ when moving k to k'

$$ucon_{\uparrow}[k' \leftarrow k](k', t'+1).$$

Clearly $t'+1$ is after $k'+1$

$$t'+1 \geq (k'+1)+1 > k'+1,$$

and the position of the step is unchanged by swapping k' with $k'+1$

$$s[k' \leftarrow k][k' \leftrightarrow k'+1](t'+1) = s[k' \leftarrow k](t'+1).$$

By Lemma 207, step $k'+1$ after the additional swap is still unit concurrent with step $t'+1$

$$ucon_{\uparrow}[k' \leftarrow k][k' \leftrightarrow k'+1](k'+1, t'+1).$$

As argued before, this is the schedule where k was moved to $k'+1$, and the claim follows

$$ucon_{\uparrow}[k'+1 \leftarrow k](k'+1, t'+1).$$

□

We can thus move step k first to position t, and then move the step to the right until we hit a read-write race.

Lemma 344. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and there is no read-write race between t' and $t'+1$ in any schedule reordered by moving k to t' before some $k' \in [t:k)$

$$\forall t' \in [t:k').\neg RW_{\uparrow}[t' \leftarrow k](t',t'+1),$$

then for all $l \in [t:k']$, the step is executable at l

$$\mathcal{E}(t,k,l)$$

Proof. By induction on l, starting at t. The base case is Lemma 337.

In the inductive step $l \to l+1$ we have by the induction hypothesis that the step is executable at l

$$\mathcal{E}(t,k,l).$$

Since all steps before k' do not have read-write races, in particular step l does not have a read-write race

$$\neg RW_{\uparrow}[l \leftarrow k](l,l+1).$$

The claim is now Lemma 343. □

Not only step t can still be executed at k', but all the other steps can also be executed at their new position. To show this, we use Invariant $\mathcal{I}$ from page 178 to run the schedules in parallel. We use the fact that reorderings are "transitive", i.e., when moving a step t_1 to some position t_2, and then moving that step to another position t_3, it is the same as moving t_1 directly to t_3. We use this in the following instances

$$\begin{aligned} s[t \leftarrow k][t \to k'] &= s[k' \leftarrow k], \\ s[t \leftarrow k][t \to l] &= s[l \leftarrow k], \\ s[l \leftarrow k][t \leftarrow l] &= s[t \leftarrow k], \end{aligned}$$

where $t \le l \le k' < k$.

Lemma 345. *When the configurations at $l+1$ when moving k to t and l when moving t further to $k' \ge l$ are nearly the same*

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \to k'], l+1, l)$$

and the configurations at t and l strongly agree when moving k to t respectively k

$$c_{\uparrow}[t \leftarrow k]^{t} =^{s[t \leftarrow k](t)}_{\downarrow} c_{\uparrow}[l \leftarrow k]^{l},$$

then the configurations are also nearly the same at $l+1$ when moving k to l and $l+1$ when moving l further to t

$$\mathcal{I}^{l}_{\uparrow}[l \leftarrow k]([t \leftarrow l], l+1, l+1).$$

Proof. By Lemma 200 the configurations when moving t further to l are nearly the same

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \to l], l+1, l).$$

Clearly moving l back to t is the inverse operation for moving t to l

$$s[t \leftarrow k][t \rightarrow l][t \leftarrow l] = s[l \leftarrow k][t \leftarrow l] = s[t \leftarrow k],$$

and by Lemma 199 we obtain that the configurations at $l+1$ when moving k to t and then t to l, and $l+1$ when moving l back to t, are nearly the same

$$\mathcal{I}^{l}_{\uparrow}[t \leftarrow k][t \rightarrow l]([t \leftarrow l], l+1, l+1).$$

That is clearly the same as the claim

$$\mathcal{I}^{l}_{\uparrow}[l \leftarrow k]([t \leftarrow l], l+1, l+1).$$

□

Lemma 346. *When the configurations at $l+1$ when moving k to t and l when moving t further to $k' \geq l$ are nearly the same and the step can be executed at l*

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k'], l+1, l) \wedge \mathcal{E}(t,k,l),$$

then the configurations are also nearly the same at $l+1$ when moving k to l and $l+1$ when moving l further to t

$$\mathcal{I}^{l}_{\uparrow}[l \leftarrow k]([t \leftarrow l], l+1, l+1).$$

Proof. Since the step can be executed at l, the configurations at t and l strongly agree

$$c_{\uparrow}[t \leftarrow k]^{t} =^{s[t \leftarrow k](t)}_{\downarrow} c_{\uparrow}[l \leftarrow k]^{l},$$

and the claim is Lemma 345. □

Lemma 347. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and there is no read-write race between t' and $t'+1$ in any schedule reordered by moving k to t' before some $k' \in [t:k)$

$$\forall t' \in [t:k').\neg RW_{\uparrow}[t' \leftarrow k](t', t'+1),$$

then for all $l \in [t:k']$, the configurations at $l+1$ and l are nearly the same when moving until k'

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k'], l+1, l).$$

Proof. By induction on l, starting at t. The base case is Lemma 190.

In the inductive step $l \rightarrow l+1$ we have by the induction hypothesis that the configurations at $l+1$ and l are nearly the same when moving until k'

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k'], l+1, l).$$

By Lemma 344, the step can be executed at l

$$\mathcal{E}(t,k,l).$$

By Lemma 346, the configurations are also nearly the same at $l+1$ when moving k to l and $l+1$ when moving l further to t

$$\mathcal{I}^{l}_{\uparrow}[l \leftarrow k]([t \leftarrow l], l+1, l+1).$$

By Lemma 339 we obtain that there is no write-read race

$$\neg WR_{\uparrow}[l \leftarrow k](l, l+1),$$

and thus also no visible write-read race

$$\neg VR_{\uparrow}[l \leftarrow k](l, l+1),$$

and by Lemma 196 the configurations for $l+1$ strongly agree

$$c_{\uparrow}[l \leftarrow k]^{l+1} =^{s[l \leftarrow k](l+1)}_{\uparrow} c_{\uparrow}[l \leftarrow k][t \leftarrow l]^{l+1} = c_{\uparrow}[t \leftarrow k]^{l+1}.$$

By Lemma 150, the steps have the same inputs

$$in_{\uparrow}[l \leftarrow k](l+1) = in_{\uparrow}[t \leftarrow k](l+1).$$

Since the step can be executed at l, the configurations at t and at l strongly agree

$$c_{\uparrow}[t \leftarrow k]^{t} =^{s[t \leftarrow k](t)}_{\uparrow} c_{\uparrow}[l \leftarrow k]^{l}.$$

By Lemma 150, the steps have the same outputs

$$out_{\uparrow}[t \leftarrow k](t) = out_{\uparrow}[l \leftarrow k](l).$$

By Lemma 113, the visible outputs of step t at $l+1$ are a subset of the outputs at t

$$vout_{\uparrow}[t \leftarrow k](t, l+1) \subseteq out_{\uparrow}[t \leftarrow k](t).$$

Since there is no write-read race , there is no intersection between the outputs of step l and the inputs of step $l+1$

$$out_{\uparrow}[l \leftarrow k](l) \not\frown in_{\uparrow}[l \leftarrow k](l+1),$$

and we conclude that there is no such intersection between the visible outputs of step t at $l+1$ and the inputs of step $l+1$ either

$$vout_{\uparrow}[t \leftarrow k](t) \not\frown in_{\uparrow}[t \leftarrow k](l+1),$$

i.e., there is no visible write-read race in the schedule where k is moved to t

$$\neg VR_{\uparrow}[t \leftarrow k](t, l+1)$$

and the claim follows by Lemma 198 for $k := l = 1$, $k' := l$, and $O := [t \to k']$

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \to k'], l+2, l+1).$$

□

At this point we are in a good position to use Condition RMWRace, with which we can prove that subsequent steps of the same processor never have visible write-read races. We can thus run the original schedule and the schedule with the read-write race in parallel using again the invariant $\mathcal{I}$.

We use again the observation that we can treat $s[k' \leftarrow k]$ as the original and $s[k' \leftarrow k][k' \rightarrow k] = s$ as the reordered schedule. We define a simple invariant.We say that *the condition applies at l* and write

$$ca(l)$$

when all of the following are true.

1. The configurations at $l+1$ when k is moved to k' and at l when k is moved to k' and then k' is moved to k are nearly the same

$$\mathcal{I}_{\uparrow}^{k'}[k' \leftarrow k]([k' \rightarrow k], l+1, l),$$

2. the schedule is semi-valid until $l+1$

$$\Gamma\Phi_{\uparrow}^{l+1}(s[k' \leftarrow k]),$$

3. if step $l+1$ is step $k'+1$, it is buffering a write, and if it is after $k'+1$, it has a buffered write from $k'+1$

$$\begin{cases} BW_{\uparrow}[k' \leftarrow k](l+1) \neq \emptyset & k' = l \\ k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+1}(i) & k' < l. \end{cases}$$

4. step $l+1$ is a processor step in strong memory mode

$$s[k' \leftarrow k](l+1) \in \Sigma_{P,i} \wedge SC_{\uparrow}[k' \leftarrow k](l+1).$$

Recall that Condition RMWRace only applies when all steps since the read-write race are made by the same object. As we have mentioned before, sorting creates schedules where all steps at the beginning of the interval $[t:k)$ are made by the same object. We will for now focus mostly on such schedules.

Lemma 348. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

and there is a read-write race between k' and $k'+1$ in the reordered schedule

$$RW_{\uparrow}[k' \leftarrow k](k', k'+1),$$

and all steps after $k'+1$ until some step $l+1$ (where $l \in [k':k)$) are made by the same object

$$\forall t' \in [k':l].o_{\uparrow}[k' \leftarrow k](t'+1) = o_{\uparrow}[k' \leftarrow k](k'+1),$$

and the condition applies at l

$$ca(l),$$

then Condition RMWRace actually applies, i.e.,

1. *there is no visible write-read race*

$$\neg VR_{\uparrow}[k' \leftarrow k](k', l+1),$$

2. *a valid read-write race is marked as shared*

$$\Gamma_{\uparrow}[k' \leftarrow k](l+1) \wedge RW_{\uparrow}[k' \leftarrow k](k', l+1) \rightarrow Sh_{\uparrow}[k' \leftarrow k](l+1)$$

3. *a valid write-write race is marked as shared*

$$\Gamma_{\uparrow}[k' \leftarrow k](l+1) \wedge WW_{\uparrow}[k' \leftarrow k](k', l+1) \rightarrow Sh_{\uparrow}[k' \leftarrow k](l+1)$$

Proof. We distinguish between the cases where l is k' or after it.

$k' = l$: In this case we have by Lemma 339 that there is no write-read race at all between k' and $k'+1$

$$\neg WR_{\uparrow}[k' \leftarrow k](k', k'+1)$$

and thus also no visible write-read race

$$\neg VR_{\uparrow}[k' \leftarrow k](k', k'+1),$$

and the first claim follows with the fact that $l = k'$.

The other two claims are Lemma 342.

$k' < l$: In this case we have by assumption that the buffer at $l+1$ contains a write from step $k'+1$

$$k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+1}(i).$$

We also have that the schedule is semi-valid until $l+1$

$$\Gamma\Phi_{\uparrow}^{l+1}(s[k' \leftarrow k]),$$

and that step $l+1$ is a processor step in strong memory mode

$$s[k' \leftarrow k](l+1) \in \Sigma_{P,i} \wedge SC_{\uparrow}[k' \leftarrow k](l+1).$$

Since the step is executable at k', step k' is unit-concurrent with step $k'+1$

$$ucon_{\uparrow}[k' \leftarrow k](k', k'+1).$$

The claims are now Condition RMWRace.

□

Lemma 349. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

and there is a read-write race between k' and $k'+1$ in the reordered schedule

$$RW_{\uparrow}[k' \leftarrow k](k', k'+1),$$

then step $k'+1$ is a processor step in strong memory mode that is buffering a write

$$\exists i.s[k' \leftarrow k](k'+1) \in \Sigma_{P,i} \wedge SC_{\uparrow}[k' \leftarrow k](k'+1) \wedge BW_{\uparrow}[k' \leftarrow k](k'+1) \neq \emptyset.$$

Proof. By Lemma 322 the original schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and since k' is less than k, also $k'+1$-abstract

$$s \in \mathrm{ABS}_{k'+1}.$$

By Lemma 342 step $k'+1$ is shared

$$Sh_\uparrow[k' \leftarrow k](k'+1).$$

By Lemma 341, step $k'+1$ can be executed at k'

$$c_\uparrow[k' \leftarrow k]^{k'+1} =_\uparrow^{s[k' \leftarrow k](k'+1)} c_\uparrow^{k'}.$$

By Lemma 150 we obtain that that step k' is shared in the high-level machine in the original schedule

$$Sh_\uparrow(k').$$

A read-write race is a write-read race with parameters swapped, and thus step $k'+1$ has a write-read race with step k'

$$WR_\uparrow[k' \leftarrow k](k'+1,k').$$

Since the step can be executed at k', step k' is unit-concurrent with step $k'+1$

$$ucon_\uparrow[k' \leftarrow k](k',k'+1)$$

and thus made by a different unit

$$diffu[k' \leftarrow k](k',k'+1).$$

This is obviously symmetric, and thus steps $k'+1$ and k' are made by different units

$$diffu[k' \leftarrow k](k'+1,k'),$$

and we obtain that step $k'+1$ is a memory write by Lemma 159

$$mwrite_\uparrow[k' \leftarrow k](k'+1).$$

By Lemma 150 we obtain that that step k' is a memory write in the high-level machine in the original schedule

$$mwrite_\uparrow(k').$$

The step is thus a shared local memory write, and by Lemma 331 it is a processor step in strong memory mode that is buffering a write in the original schedule

$$\exists i.s(k') \in \Sigma_{P,i} \wedge SC_\uparrow(k') \wedge BW_\uparrow(k') \neq \emptyset.$$

The claim follows with Lemma 150

$$\exists i.s[k' \leftarrow k](k'+1) \in \Sigma_{P,i} \wedge SC_\uparrow[k' \leftarrow k](k'+1) \wedge BW_\uparrow[k' \leftarrow k](k'+1) \neq \emptyset.$$

□

Lemma 350. *When a step is a processor step, and it either is step $k'+1$ and buffering a write or it is after $k'+1$ and has buffered a write from $k'+1$*

$$s[k' \leftarrow k](l+1) \in \Sigma_{P,i} \wedge \begin{cases} BW_{\uparrow}[k' \leftarrow k](l+1) \neq \emptyset & k' = l \\ k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+1}(i) & k' < l, \end{cases}$$

then the next step certainly has buffered a write from $k'+1$

$$k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+2}(i).$$

Proof. We distinguish between the two cases.

$k' = l$: Then step $k'+1$ is by definition buffering the write

$$\begin{aligned} issue_{\uparrow}[k' \leftarrow k]^{l+2}(i) &= Op_{i\uparrow}[k' \leftarrow k](l+1)(issue_{\uparrow}[k' \leftarrow k]^{l+1}(i), l+1) \\ &= Op_{i\uparrow}[k' \leftarrow k](k'+1)(issue_{\uparrow}[k' \leftarrow k]^{k'+1}(i), k'+1) \\ &= push(issue_{\uparrow}[k' \leftarrow k]^{k'+1}(i), k'+1) \\ &= issue_{\uparrow}[k' \leftarrow k]^{k'+1}(i) \circ (k'+1), \end{aligned}$$

and it is certainly buffered in the next step

$$k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+2}(i).$$

$k' < l$: By assumption, the step is buffered at $l+1$

$$k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+1}(i)$$

and since step k' is a processor step, it does not perform a *pop* operation

$$Op_{i\uparrow}[k' \leftarrow k](l+1) \neq pop.$$

We distinguish between the remaining two operations.

$Op_{i\uparrow}[k' \leftarrow k](l+1) = push$: The buffer is increased by one element

$$issue_{\uparrow}[k' \leftarrow k]^{l+2}(i) = issue_{\uparrow}[k' \leftarrow k]^{l+1}(i) \circ \ldots$$

and thus all elements of the buffer at $l+1$, including $k'+1$, are still elements of the buffer at $l+2$

$$k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+2}(i)$$

which is the claim.

$Op_{i\uparrow}[k' \leftarrow k](l+1) = noop$: The list of issued writes is unchanged

$$issue_{\uparrow}[k' \leftarrow k]^{l+2}(i) = issue_{\uparrow}[k' \leftarrow k]^{l+1}(i)$$

and the claim follows immediately

$$k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+2}(i).$$

□

Lemma 351. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

and there is a read-write race between k' and $k'+1$ in the reordered schedule

$$RW_{\uparrow}[k' \leftarrow k](k',k'+1),$$

and all steps after $k'+1$ until some step $l+1$ (where $l \in (k':k)$) are made by the same object

$$\forall t' \in [k':l].o_{\uparrow}[k' \leftarrow k](t'+1) = o_{\uparrow}[k' \leftarrow k](k'+1),$$

and the condition applies at l

$$ca(l),$$

then the configurations at $l+1$ and l strongly agree and the reordered schedule $s[k' \leftarrow k]$ is valid until $l+1$

$$c_{\uparrow}[k' \leftarrow k]^{l+1} =_{\uparrow}^{s[k' \leftarrow k](l+1)} c_{\uparrow}^{l} \wedge \Gamma_{\uparrow}^{l+1}(s[k' \leftarrow k]).$$

Proof. By Lemma 348 we obtain that there is no visible write-read race

$$\neg VR_{\uparrow}[k' \leftarrow k](k',l+1).$$

Since the step can be executed at k', step k' is unit concurrent with step $l+1$

$$ucon_{\uparrow}[k' \leftarrow k](k',l+1)$$

and thus made by a different unit

$$diffu[k' \leftarrow k](k',l+1).$$

Since the condition applies at l, the configurations at $l+1$ and l are nearly the same

$$\mathcal{I}_{\uparrow}^{k'}[k' \leftarrow k]([k' \to k],l+1,l),$$

and with Lemma 196 we obtain that the configurations at $l+1$ and l strongly agree

$$c_{\uparrow}[k' \leftarrow k]^{l+1} =_{\uparrow}^{s[k' \leftarrow k](l+1)} c_{\uparrow}[k' \leftarrow k][k' \to k]^{l} = c_{\uparrow}^{l},$$

which is the first claim.

By Lemma 323 we obtain that the original schedule is valid until $k-1$

$$\Gamma_{\uparrow}^{k-1}(s)$$

and thus in particular step l was valid, which by Lemma 150 is now step $l+1$

$$\Gamma_{\uparrow}[k' \leftarrow k](l+1) = \Gamma_{\uparrow}(l) = 1.$$

The second claim follows

$$\Gamma_{\uparrow}^{l+1}(s[k' \leftarrow k]).$$

□

Lemma 352. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and the step is executable at some $k' \in [t:k)$

$$\mathcal{E}(t,k,k'),$$

and there is a read-write race between k' and $k'+1$ in the reordered schedule

$$RW_{\uparrow}[k' \leftarrow k](k',k'+1),$$

and all steps after $k'+1$ until some step $l+1$ (where $l \in [k':k)$) are made by the same object

$$\forall t' \in [k':l].o_{\uparrow}[k' \leftarrow k](t'+1) = o_{\uparrow}[k' \leftarrow k](k'+1),$$

then the condition applies at l

$$ca(l).$$

Proof. By induction on l, starting at k'. In the base case, the first part of the condition is Lemma 190 for $t := k'$ and $k := k$ and using $k = l$. The second part is Lemma 340. The third and fourth part follow directly by Lemma 349.

In the inductive step $l \to l+1$ we have that steps until $l+2$ are made by the same object

$$\forall t' \in [k':l+1].o_{\uparrow}[k' \leftarrow k](t'+1) = o_{\uparrow}[k' \leftarrow k](k'+1),$$

and thus clearly also those until $l+1$

$$\forall t' \in [k':l].o_{\uparrow}[k' \leftarrow k](t'+1) = o_{\uparrow}[k' \leftarrow k](k'+1).$$

We can thus apply the induction hypothesis and obtain that the configurations at $l+1$ and l are nearly the same

$$\mathcal{I}^{k'}_{\uparrow}[k' \leftarrow k]([k' \to k], l+1, l)$$

and the schedule is semi-valid until $l+1$

$$\Gamma\Phi^{l+1}_{\uparrow}(s[k' \leftarrow k])$$

and step $l+1$ is a processor step in strong memory mode

$$s[k' \leftarrow k](l+1) \in \Sigma_{P,i} \wedge SC_{\uparrow}[k' \leftarrow k](l+1)$$

that buffers or has a buffered write from $k'+1$

$$\begin{cases} BW_{\uparrow}[k' \leftarrow k](l+1) \neq \emptyset & k' = l \\ k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+1}(i) & k' < l. \end{cases}$$

Since the step can be executed at k', step k' is unit concurrent with steps $l+1$ and $l+2$

$$ucon_{\uparrow}[k' \leftarrow k](k', l+1) \wedge ucon_{\uparrow}[k' \leftarrow k](k', l+1)$$

and thus by definition of *ucon* made by a different unit than step $l+1$

$$diffu[k' \leftarrow k](k', l+1).$$

By Lemma 348 we obtain that there is no visible write-read race

$$\neg VR_\uparrow[k' \leftarrow k](k', l+1)$$

and with Lemma 198 for $k := l = 1$ and $k' := l$ we obtain the first claim

$$\mathcal{I}^{k'}_\uparrow[k' \leftarrow k]([k' \to k], l+2, l+1).$$

By Lemma 351 the steps until $l+1$ are valid

$$\Gamma^{l+1}_\uparrow(s[k' \leftarrow k]).$$

By Lemma 323 we obtain that the original schedule is valid until $k-1$

$$\Gamma^{k-1}_\uparrow(s)$$

and thus in particular step $l+1$ was valid

$$\Gamma_\uparrow(l+1)$$

and thus also feasible

$$\Phi_\uparrow(l+1).$$

By Lemma 194 for $k := l+2$ and $k' := l+1$ we obtain that the configurations at $k'+1$ and k' agree on the core registers

$$m_\uparrow[k' \leftarrow k]^{l+2} =_{C_\uparrow[k' \leftarrow k](l+2)} m_\uparrow[k' \leftarrow k][k' \to k]^{l+1} = m^{l+1}_\uparrow.$$

With Lemma 142 we obtain that step $l+2$ is feasible after the reordering

$$\Phi_\uparrow[k' \leftarrow k](l+2) = \Phi_\uparrow(l+1) = 1.$$

Therefore the reordered schedule is semi-valid until $l+2$

$$\Gamma\Phi^{l+2}_\uparrow(s[k' \leftarrow k]),$$

which is the second claim.

Since all the steps until $l+2$ are made by the same object, step $l+2$ also is made by processor i

$$s[k' \leftarrow k](l+2) \in \Sigma_{P,i}.$$

By Lemma 350 timestamp $k'+1$ is buffered at $l+2$

$$k'+1 \in issue_\uparrow[k' \leftarrow k]^{l+2}(i)$$

Note that $l+1$ is certainly greater than k', and the third claim follows.

By Lemma 168 with $k := l+1$ and $t := k'+1$ we conclude that the processor could not change its memory mode, and thus step $l+2$ is also in strong memory mode

$$\begin{aligned}
SC_\uparrow[k' \leftarrow k](l+2) &= SC_{i\uparrow}[k' \leftarrow k](l+2) \\
&= SC_{i\uparrow}[k' \leftarrow k](k'+1) && \text{L 168} \\
&= SC_{i\uparrow}[k' \leftarrow k](l+1) && \text{L 168} \\
&= SC_\uparrow[k' \leftarrow k](l+1),
\end{aligned}$$

and the fourth claim follows. □

It is now obvious that there are no races with the local steps in the low-level machine. On a very high level, the argument is that we have strong agreement essentially everywhere, and can thus reduce any race in the low-level machine to race in the high-level machine. Therefore the steps must be shared, which contradicts the fact that the steps are local.

We first show that the steps have to be shared no matter whether there is a read-write race that would prevent us from moving the step or not.

Lemma 353. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and all steps between t and $l \in [t:k)$ are made by the same object

$$\forall t' \in [t:l].o_{\uparrow}(t') = o_{\uparrow}(l),$$

and let k' be the first step with a read-write race (or l if none exists)

$$k' = \min\left\{ k' \geq t \;\middle|\; RW_{\uparrow}[k' \leftarrow k](k', k'+1) \vee k' = l \right\}.$$

then all of the following are true.

1. *The step can be executed at k'*
$$\mathcal{E}(t,k,k'),$$
2. *The configurations at $k'+1$ when k is moved to k' and at $k'+1$ when k is moved to t are nearly the same*
$$\mathcal{I}^{k'}_{\uparrow}[k' \leftarrow k]([t \leftarrow k'], k'+1, k'+1),$$
3. *The configurations at $l+1$ in the reordered and l in the original schedule strongly agree*
$$c_{\uparrow}[k' \leftarrow k]^{l+1} =^{s[k' \leftarrow k](l+1)}_{\uparrow} c^{l}_{\uparrow},$$
4. *There is no visible read-write race between k' and $l+1$*
$$\neg VR_{\uparrow}[k' \leftarrow k](k', l+1),$$
5. *If there is a read-write race, step $l+1$ is shared*
$$RW_{\uparrow}[k' \leftarrow k](k', l+1) \to Sh_{\uparrow}[k' \leftarrow k](l+1),$$
6. *If there is a write-write race, step $l+1$ is shared*
$$WW_{\uparrow}[k' \leftarrow k](k', l+1) \to Sh_{\uparrow}[k' \leftarrow k](l+1).$$

Proof. By definition of k' we know that all steps before it do not have read-write races

$$\forall t' \in [t:k').\neg RW_{\uparrow}[t' \leftarrow k](t', t'+1),$$

and by Lemma 344 the step is executable at k'

$$\mathcal{E}(t,k,k'),$$

which is the first claim.

By Lemma 347 with $l := k'$, we obtain that the configurations at $k'+1$ and k' are nearly the same

$$\mathcal{I}^t_\uparrow[t \leftarrow k]([t \to k'], k'+1, k'),$$

and the second claim is Lemma 346.

We now distinguish two cases: either k' is l, or less than l and there is a read-write race.

$k' = l$: The third claim is Lemma 341.

By Lemma 339 there is no write-read race

$$\neg WR_\uparrow[k' \leftarrow k](k', k'+1),$$

and thus also no visible write-read race

$$\neg VR_\uparrow[k' \leftarrow k](k', k'+1),$$

which is the fourth claim.

The remaining two claims are Lemma 342.

$k' < l \wedge RW_\uparrow[k' \leftarrow k](k', k'+1)$: Clearly in the reordered schedule, the steps from $k'+1$ until $l+1$ are made by the same unit

$$\forall t' \in [k' : l].o_\uparrow[k' \leftarrow k](t'+1) = o_\uparrow(t') = o_\uparrow(l) = o_\uparrow[k' \leftarrow k](l+1).$$

By Lemma 352, the conditions apply at l

$$ca(l).$$

The third claim is Lemma 351.

The fourth claim is just Lemma 348.

By Lemma 351 we obtain that the schedule is valid until $l+1$

$$\Gamma^{l+1}_\uparrow(s[k' \leftarrow k]).$$

In particular, step $l+1$ is valid

$$\Gamma_\uparrow[k' \leftarrow k](l+1),$$

and the remaining two claims are Lemma 348.

□

Lemma 354. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and all steps between t and $l \in [t : k)$ are made by the same object

$$\forall t' \in [t : l].o_\uparrow(t') = o_\uparrow(l),$$

and let k' be the first step with a read-write race (or l if none exists)

$$k' = \min\left\{ k' \geq t \;\middle|\; RW_\uparrow[k' \leftarrow k](k', k'+1) \vee k' = l \right\}.$$

then for each $l' \in [k' : l]$, the configurations at $l'+1$ are nearly the same in the two reordered schedules

$$\mathcal{I}^{k'}_\uparrow[k' \leftarrow k]([t \leftarrow k'], l'+1, l'+1),$$

Proof. By induction on l', starting at k'.

The base case is solved by Lemma 353.

In the inductive step $l' \to l'+1$, we have that the configurations at $l'+1$ are nearly the same

$$\mathcal{I}_{\uparrow}^{k'}[k' \leftarrow k]([t \leftarrow k'], l'+1, l'+1),$$

Since l' is by assumption greater than or equal to k', and $l'+1$ is less than or equal to l, k' must be less than l

$$k' \leq l' < l'+1 \leq l.$$

Thus k' has a read-write race and we obtain that k' is also the first step before l' with a read-write race

$$k' = \min\left\{ k' \geq t \;\middle|\; RW_{\uparrow}[k' \leftarrow k](k', k'+1) \vee k' = l' \right\}.$$

By Lemma 353 with $l := l'$, we obtain that there is no visible write-read race between k' and $l'+1$

$$\neg VR_{\uparrow}[k' \leftarrow k](k', l'+1),$$

and by the same Lemma, the step can be executed at k'

$$\mathcal{E}(t, k, k').$$

It is thus unit-concurrent with $l'+1$

$$ucon_{\uparrow}[k' \leftarrow k](k', l'+1)$$

and thus also made by a different unit

$$diffu_{\uparrow}[k' \leftarrow k](k', l'+1).$$

The claim is now Lemma 198 with $k := l'+1$ and $k' := l'+1$

$$\mathcal{I}_{\uparrow}^{k'}[k' \leftarrow k]([t \leftarrow k'], l'+2, l'+2).$$

□

Lemma 355. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t, k),$$

and all steps between t and $l \in [t : k)$ are made by the same object

$$\forall t' \in [t : l].o_{\uparrow}(t') = o_{\uparrow}(l),$$

and let k' be the first step with a read-write race (or l if none exists)

$$k' = \min\left\{ k' \geq t \;\middle|\; RW_{\uparrow}[k' \leftarrow k](k', k'+1) \vee k' = l \right\}.$$

then for each $l' \in [k' : l]$, the configurations at $l'+1$ strongly agree in the two reordered schedules

$$c_{\uparrow}[k' \leftarrow k]^{l'+1} =_{\uparrow}^{s[k' \leftarrow k](l+1)} c_{\uparrow}[t \leftarrow k]^{l'+1}.$$

Proof. By Lemma 354, the configurations at $l+1$ are nearly the same in the reordered schedules

$$\mathcal{I}_{\uparrow}^{k'}[k' \leftarrow k]([t \leftarrow k'], l+1, l+1).$$

By Lemma 353 with $l := l'$, we obtain that there is no visible write-read race between k' and $l'+1$

$$\neg VR_{\uparrow}[k' \leftarrow k](k', l'+1),$$

and by the same Lemma, the step can be executed at k'

$$\mathcal{E}(t,k,k').$$

It is thus unit-concurrent with $l'+1$

$$ucon_{\uparrow}[k' \leftarrow k](k', l'+1)$$

and thus also made by a different unit

$$diffu_{\uparrow}[k' \leftarrow k](k', l'+1).$$

By Lemma 196 with $k := l'+1$ and $k' := l'+1$, the steps at $l+1$ strongly agree in the reordered schedules

$$c_{\uparrow}[k' \leftarrow k]^{l+1} =_{\uparrow}^{s[k' \leftarrow k](l+1)} c_{\uparrow}[k' \leftarrow k][t \leftarrow k']^{l+1}.$$

We observe that moving k to k' and then k' to t is the same as moving k to t

$$s[k' \leftarrow k][t \leftarrow k'] = s[t \leftarrow k],$$

and the claim follows

$$c_{\uparrow}[k' \leftarrow k]^{l+1} =_{\uparrow}^{s[k' \leftarrow k](l+1)} c_{\uparrow}[t \leftarrow k]^{l+1}.$$

□

Lemma 356. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and all steps between t and $l \in [t:k)$ are made by the same object

$$\forall t' \in [t:l].o_{\uparrow}(t') = o_{\uparrow}(l),$$

then all steps $l \in [t:k)$ can be executed at their new position when moving k to t

$$c_{\uparrow}[t \leftarrow k]^{l+1} =_{\uparrow}^{s[t \leftarrow k](l+1)} c_{\uparrow}^{l}.$$

Proof. We move step t as far to the right as possible, i.e., either until we reach a read-write race, or l

$$k' = \min\left\{ k' \geq t \;\middle|\; RW_{\uparrow}[k' \leftarrow k](k', k'+1) \vee k' = l \right\}.$$

By Lemma 353, the configurations in that schedule and the original schedule strongly agree

$$c_{\uparrow}[k' \leftarrow k]^{l+1} =_{\uparrow}^{s[k' \leftarrow k](l+1)} c_{\uparrow}^{l}.$$

By Lemma 355, the configurations in that schedule and the schedule where k is moved to t also strongly agree

$$c_\uparrow[k' \leftarrow k]^{l+1} =_\uparrow^{s[k' \leftarrow k](l+1)} c_\uparrow[t \leftarrow k]^{l+1}.$$

Observing now that moving k to k' is the same as moving k to t and then t to k'

$$s[k' \leftarrow k] = s[t \leftarrow k][t \rightarrow k']$$

and that step $l+1$ is after k', we obtain that the latter operation has no effect on the oracle input at $l+1$

$$s[t \leftarrow k][t \rightarrow k'](l+1) = s[t \leftarrow k](l+1).$$

The claim follows by transitivity and symmetry of strong agreement

$$c_\uparrow[t \leftarrow k]^{l+1} =_\uparrow^{s[t \leftarrow k](l+1)} c_\uparrow[k' \leftarrow k]^{l+1} =_\uparrow^{s[t \leftarrow k](l+1)} c_\uparrow^l.$$

□

We now show the crucial result that in a sorted schedule, there is no visible write-read race between t (formerly k) and any intermediate step $l+1$ (formerly l).

Lemma 357. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and all steps between t and $l \in [t:k)$ are made by the same object

$$\forall t' \in [t:l].o_\uparrow(t') = o_\uparrow(l),$$

then all steps $l \in [t:k)$ have no visible write-read race from t when moving k to t

$$\neg VR_\uparrow[t \leftarrow k](t, l+1).$$

Proof. We move the step as far to the right as possible

$$k' = \min\left\{ k' \geq t \;\middle|\; RW_\uparrow[k' \leftarrow k](k', k'+1) \vee k' = l \right\}.$$

By Lemma 353, there is no visible write-read race in that schedule

$$\neg VR_\uparrow[k' \leftarrow k](k', l+1)$$

and the step is executable at k'

$$\mathcal{E}(t,k,k').$$

Thus by definition of $\mathcal{E}$ we have strong agreement between k' when moving to k' and t when moving to t

$$c_\uparrow[t \leftarrow k]^t =_\uparrow^{s[t \leftarrow k](t)} c_\uparrow[k' \leftarrow k]^{k'}$$

and by Lemma 144 the write set of those steps are the same

$$WS_\uparrow[t \leftarrow k](t) = WS_\uparrow[k' \leftarrow k](k').$$

By Lemma 355 all steps $l'+1$ until $l+1$ are also not affected

$$c_{\uparrow}[k' \leftarrow k]^{l'+1} =_{\uparrow}^{s[k' \leftarrow k](l'+1)} c_{\uparrow}[t \leftarrow k]^{l'+1}.$$

By Lemma 144 the outputs of those steps $l'+1 < l+1$ are the same

$$out_{\uparrow}[k' \leftarrow k](l'+1) = out_{\uparrow}[t \leftarrow k](l'+1)$$

and the inputs of step l are the same

$$in_{\uparrow}[k' \leftarrow k](l+1) = in_{\uparrow}[t \leftarrow k](l+1).$$

Thus the set of addresses overwritten in the interval $(k' : l+1)$ is the same

$$\begin{aligned} out_{\uparrow}[k' \leftarrow k]((k' : l+1)) &= \bigcup_{t' \in (k' : l+1)} out_{\uparrow}[k' \leftarrow k](t') \\ &= \bigcup_{l'+1 \in (k' : l+1)} out_{\uparrow}[k' \leftarrow k](l'+1) \\ &= \bigcup_{l'+1 \in (k' : l+1)} out_{\uparrow}[t \leftarrow k](l'+1) \\ &= out_{\uparrow}[t \leftarrow k]((k' : l+1)). \end{aligned}$$

Therefore the set of overwritten addresses in the interval $(t : l+1)$ when k is moved to t subsumes the set of addresses overwritten in the interval $(k' : l+1)$ when k is moved to k'

$$out_{\uparrow}[t \leftarrow k]((t : l+1)) \supseteq out_{\uparrow}[t \leftarrow k]((k' : l+1)) = out_{\uparrow}[k' \leftarrow k]((k' : l+1)).$$

Consequently, when moving k to t rather than k', less of its visible write-set remains

$$\begin{aligned} vws_{\uparrow}[t \leftarrow k](t, l+1) &= WS_{\uparrow}[t \leftarrow k](t) \setminus out_{\uparrow}[t \leftarrow k]((t : l+1)) \\ &\subseteq WS_{\uparrow}[k' \leftarrow k](k') \setminus out_{\uparrow}[k' \leftarrow k]((k' : l+1)) \\ &= vws_{\uparrow}[k' \leftarrow k](k', l+1). \end{aligned}$$

Therefore the step has fewer visible outputs

$$\begin{aligned} vout_{\uparrow}[t \leftarrow k](k', l+1) &= dc(WS_{\uparrow}[t \leftarrow k](t)) \cup vws_{\uparrow}[t \leftarrow k](t, l+1) \\ &\subseteq dc(WS_{\uparrow}[k' \leftarrow k](k')) \cup vws_{\uparrow}[k' \leftarrow k](k', l+1) \\ &= vout_{\uparrow}[k' \leftarrow k](k', l+1). \end{aligned}$$

To summarize: when moving k to t, the writing step has fewer visible outputs, and the reading step has the same inputs. The claim follows

$$\begin{aligned} \neg VR_{\uparrow}[k' \leftarrow k](k', l+1) &\iff vout_{\uparrow}[k' \leftarrow k](k', l+1) \not\parallel in_{\uparrow}[k' \leftarrow k](l+1) \\ &\implies vout_{\uparrow}[t \leftarrow k](t, l+1) \not\parallel in_{\uparrow}[t \leftarrow k](l+1) \\ &\iff \neg VR_{\uparrow}[t \leftarrow k](t, l+1). \end{aligned}$$

□

Lemma 358. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

then step t after moving k to t is not a write buffer step in strong memory mode

$$\nexists i.SC_{\uparrow}[t \leftarrow k](t) \wedge s[t \leftarrow k](t) \in \Sigma_{WB,i}.$$

Proof. Assume for the sake of contradiction that the step is made by a write buffer in strong memory mode

$$SC_{\uparrow}[t \leftarrow k](t) \wedge s[t \leftarrow k](t) \in \Sigma_{WB,i}.$$

By Lemma 324 step k is a write buffer step

$$s(k) \in \Sigma_{WB,i}.$$

By Lemma 336 the schedule is $t+1$-valid and thus also t-valid

$$s[t \leftarrow k] \in \mathrm{ABS}_t$$

and by Lemma 269 step t is also in strong memory mode in the low-level machine

$$SC_{\downarrow}[t \leftarrow k](t) = SC_{\uparrow}[t \leftarrow k](t) = 1.$$

By Lemma 334 the core registers are the same at t and k

$$m_{\downarrow}[t \leftarrow k]^t =_{C_{\downarrow}(k)} m_{\downarrow}^k$$

and by Lemma 151 we obtain that step k was made in strong memory mode

$$SC_{\downarrow}(k) = SC_{\downarrow}[t \leftarrow k](t) = 1.$$

Since the schedule has a local tail with an independent end from t to k, step k is not a write buffer step in strong memory mode

$$\nexists i.SC_{\downarrow}(k) \wedge s(k) \in \Sigma_{WB,i},$$

which is a contradiction. □

We also show that there is also no read-write race in the low-level machine; thus step k is not affected by being moved to t either. To do so we first show that the outputs of step l intersect with neither the inputs nor outputs of step t after moving k to t.

Lemma 359. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and all steps between t and $l \in [t:k)$ are made by the same object

$$\forall t' \in [t:l].o_{\uparrow}(t') = o_{\uparrow}(l),$$

then for $X \in \{in, out\}$ there is no intersection between the outputs of step l and the set X during step t after k is moved to t in the low-level machine

$$out_{\downarrow}(l) \not\cap X_{\downarrow}[t \leftarrow k](t).$$

Proof. Assume for the sake of contradiction the outputs of l and either inputs or outputs $X \in \{ in, out \}$ of step t after k is moved to t intersect

$$out_{\downarrow}(l) \dot{\cap} X_{\downarrow}[t \leftarrow k](t).$$

By Lemma 322, the schedule is k-abstract

$$s \in \mathrm{ABS}_k.$$

Since l is before k we obtain that the schedule is valid until l and $l+1$-abstract

$$\Gamma^{l}_{\uparrow}(s) \wedge s \in \mathrm{ABS}_{l+1},$$

and thus by Lemma 330, the outputs of step l in the low-level machine are a subset of those in the high-level machine

$$out_{\downarrow}(l) \subseteq out_{\uparrow}(l),$$

and by Lemma 269 the steps have the same annotation

$$Sh_{\uparrow}(l) = Sh_{\downarrow}(l),\ ShR_{\uparrow}(l) = ShR_{\downarrow}(l).$$

We now move step t as far to the right as possible, i.e., either until we reach a read-write race, or l

$$k' = \min \left\{ k' \geq t \;\middle|\; RW_{\uparrow}[k' \leftarrow k](k', k'+1) \vee k' = l \right\}.$$

By Lemma 353, step $l+1$ can be executed at l in the original schedule

$$c_{\uparrow}[k' \leftarrow k]^{l+1} =^{s[k' \leftarrow k](l+1)}_{\uparrow} c^{l}_{\uparrow}$$

and we conclude with Lemma 150 that the outputs of step $l+1$ subsume those at l in the low-level machine in the original schedule

$$out_{\downarrow}(l) \subseteq out_{\uparrow}(l) = out_{\uparrow}[k' \leftarrow k](l+1)$$

and also agree on the annotation

$$\begin{aligned} Sh_{\uparrow}[k' \leftarrow k](l+1) &= Sh_{\uparrow}(l) = Sh_{\downarrow}(l), \\ ShR_{\uparrow}[k' \leftarrow k](l+1) &= ShR_{\uparrow}(l) = ShR_{\downarrow}(l). \end{aligned}$$

By Lemma 336 step t in that schedule is abstract

$$s[t \leftarrow k] \in \mathrm{ABS}_{t+1}.$$

By Lemma 358 step t after moving k to t is not a write buffer step in strong memory mode

$$\nexists i. SC_{\uparrow}[t \leftarrow k](t) \wedge s[t \leftarrow k](t) \in \Sigma_{WB,i}.$$

By Lemma 270 has the same inputs in both machines

$$in_{\downarrow}[t \leftarrow k](t) = in_{\uparrow}[t \leftarrow k](t),$$

and by Lemma 265 more outputs in the high-level machine

$$out_{\downarrow}[t \leftarrow k](t) \subseteq out_{\uparrow}[t \leftarrow k](t).$$

We obtain in each case that both inputs and outputs $X \in \{ in, out \}$ in the low-level machine are subsumed by the inputs resp. outputs in the high-level machine

$$X_{\downarrow}[t \leftarrow k](t) \subseteq X_{\uparrow}[t \leftarrow k](t).$$

By Lemma 353, the step can be executed at k'

$$\mathcal{E}(t,k,k')$$

and thus the configurations at k' when k is moved to k' and at t when k is moved to t strongly agree

$$c_{\uparrow}[t \leftarrow k]^t =_{\uparrow}^{s[t \leftarrow k](t)} c_{\uparrow}[k' \leftarrow k]^{k'}$$

and by Lemma 150 for $t_1 := t$ and $t_2 := k'$, the value of $X \in \{ in, out \}$ is unchanged and thus still subsumes the value of X in the low-level machine

$$X_{\downarrow}[t \leftarrow k](t) \subseteq X_{\uparrow}[t \leftarrow k](t) = X_{\uparrow}[k' \leftarrow k](k').$$

By definition of $\mathcal{L}^I$, step k is unit-concurrent with step l

$$ucon_{\uparrow}(k,l)$$

and thus made by a different unit

$$u_{\downarrow}(l) \neq u_{\downarrow}(k) = u_{\downarrow}[t \leftarrow k](t).$$

By Lemma 93, the inputs resp. outputs of step t are accessible to the unit making the step

$$X_{\downarrow}[t \leftarrow k](t) \subseteq ACC_{u_{\downarrow}[t \leftarrow k](t)},$$

and by Lemma 121 step l is a memory write in both cases

$$mwrite_{\downarrow}(l)$$

and since it is local, not shared

$$\neg Sh_{\downarrow}(l).$$

We conclude that there is an intersection between the outputs of $l+1$ and the set X at k' in the reordered schedule

$$out_{\uparrow}[k' \leftarrow k](l+1) \cap X_{\uparrow}[k' \leftarrow k](k').$$

Thus there is either a read-write race or a write-write race (depending on whether X is the inputs or the outputs)

$$RW_{\uparrow}[k' \leftarrow k](k', l+1) \vee WW_{\uparrow}[k' \leftarrow k](k', l+1),$$

and by Lemma 353, step $l+1$ must be shared

$$Sh_{\uparrow}[k' \leftarrow k](l+1),$$

which is a contradiction. □

Lemma 360. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and all steps between t and $l \in [t:k)$ are made by the same object

$$\forall t' \in [t:l].o_\uparrow(t') = o_\uparrow(l),$$

then there is no write-read race between l and k in the low-level machine

$$\neg WR_\downarrow(l,k).$$

Proof. By Lemma 334 the configurations at t when k is moved to t and at k agree on the memory content of core and fetched registers

$$m_\downarrow[t \leftarrow k]^t =_{C_\downarrow(k) \cup F_\downarrow(k)} m_\downarrow^k.$$

By Lemma 152 those configurations agree on the inputs

$$in_\downarrow[t \leftarrow k](t) = in_\downarrow(k).$$

Assume now for the sake of contradiction that such a race exists, in which case there is an intersection between outputs and inputs

$$out_\downarrow(l) \hat{\cap} in_\downarrow(k)$$

and thus also when k is moved to t

$$out_\downarrow(l) \hat{\cap} in_\downarrow[t \leftarrow k](t).$$

That contradicts Lemma 359. □

4.12.3 Transferring Races in Unsorted Schedules

We wish to apply the Lemmas from the previous section in an unsorted schedule. We prove a key lemma which allows us to transfer all races of the unsorted schedule to a sorted schedule.

Lemma 361. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

then for all $l' \in [t:k)$ there is a sequence of operations O and some $l \in [t:k)$ such that all of the following hold

1. *The schedules before t, at k, and at l resp. l' are the same*

$$sO[0:t-1] = s[0:t-1] \wedge sO(k) = s(k) \wedge sO(l) = s(l'),$$

2. *the steps at k strongly agree*

$$c_\downarrow O^k =_\downarrow^{sO(k)} c_\downarrow^k,$$

3. *The reordered schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I O(t,k),$$

4. *the steps from t until l are made by the same object*

$$\forall t' \in [t:l].o_{\uparrow}O(t') = o_{\uparrow}O(l),$$

5. *and furthermore the steps l and l' strongly agree*

$$c_{\downarrow}O^{l} =_{\downarrow}^{sO(l)} c_{\downarrow}^{l'},$$

6. *the outputs from t until l are subsumed by those from t until l' in the original schedule*

$$out_{\uparrow}O([t:l)) \subseteq out_{\uparrow}([t:l'))$$

Proof. By Lemma 321 and 320 we obtain that the schedule is IPI valid until $k-1$ and k-ordered

$$\Delta_{IPI\downarrow}^{k-1}(s) \wedge s \in \mathrm{ORD}_k.$$

Using Lemma 319, we sort the steps from t until $k-1$, putting all steps of the unit making step l' to the very left by defining

$$u(s(l')) \prec i \qquad \forall i \neq u(s(l'))$$

and ordering all remaining units in an arbitrary way.

We define O to be the sequence of reorderings that sort the schedule

$$O := \prec(t,k)$$

and by Lemma 319 the sorted schedule is equivalent to the original schedule

$$sO \equiv_{\downarrow} s.$$

Let X be the object making step l'

$$X = o(s(l'))$$

and n be the number of steps made by X at l'

$$n = \#X(l').$$

Clearly l' is the time when object X makes its n-th step

$$l' = \#X \approx n(s) \in [t:l'+1).$$

We define l to be the n-th step of X in that schedule

$$l := \#X \approx n(sO)$$

which due to equivalence of the schedules uses the same oracle input

$$sO(l) = s(l').$$

The first claim follows with Lemma 319

$$sO[0:t-1] = s[0:t-1] \wedge sO(k) = s(k).$$

By Lemma 319, the configurations at k are the same

$$c_{\downarrow}O^k = c_{\downarrow}^k$$

and the second claim follows by reflexivity of strong agreement

$$c_{\downarrow}O^k =_{\downarrow}^{sO(k)} c_{\downarrow}^k.$$

By Lemma 319, the sorted schedule is k-ordered and valid and IPI-valid until $k-1$

$$\Gamma_{\downarrow}^{k-1}(sO) \wedge \Delta_{IPI\downarrow}^{k-1}(sO) \wedge sO \in \mathrm{ORD}_k.$$

Since the schedule and the configuration are the same at k, step k is still valid in the sorted schedule

$$\Gamma_{\downarrow}O(k) = \Gamma_{\downarrow}(c_{\downarrow}O^k, sO(k)) = \Gamma_{\downarrow}(c_{\downarrow}^k, s(k)) = \Gamma_{\downarrow}(k)$$

Thus the schedule is valid until k, and IPI-valid until $t-1$ and t-ordered

$$\Gamma_{\downarrow}^{k}(sO) \wedge \Delta_{IPI\downarrow}^{t-1}(sO) \wedge sO \in \mathrm{ORD}_t.$$

By Lemma 319 all steps before k are still local

$$\forall t' \in [t:k].L_{\downarrow}O(t'),$$

and thus has a local tail from t to k

$$\mathcal{L}O(t,k).$$

Because the steps before t are the same, the configuration at t is still the same

$$c_{\downarrow}O^t = c_{\downarrow}^t,$$

and also the sequence of issued writes

$$issue_{\downarrow}O^t(i) = issue_{\downarrow}^t(i),$$

and thus the configuration at t is still clean

$$clean_{\downarrow}O(t) = clean_{\downarrow}(t) = 1.$$

We wish to show that step k is still unit-concurrent with all steps $t' \in [t:k)$. By Lemma 144, the victims of step k have not changed

$$victims_{\downarrow}O(k) = victims_{\downarrow}(k).$$

Let now t' be some step in the interval, which is now the n'-th step of some object X'

$$X' = o(sO(t')) \wedge n' = \#X'O(t').$$

Let t'' be the position of the n'-th step of object X' in the original schedule

$$t'' = \#X' \approx n'(s)$$

which by Lemma 319 is also in the interval $[t:k)$

$$t'' \in [t:k).$$

Thus by assumption step k was unit-concurrent with step t''.

$$ucon_{\downarrow}(k,t'').$$

Clearly the steps are made by the same unit

$$u_{\downarrow}O(t') = u_{\downarrow}(t''),$$

and it is now easy to show that step k is unit-concurrent with step t'

$$\begin{aligned} ucon_{\downarrow}(k,t'') &\equiv diffu(k,t'') \wedge u_{\downarrow}(t'') \notin victims_{\downarrow}(k) \\ &\equiv u_{\downarrow}(k) \neq u_{\downarrow}(t'') \wedge u_{\downarrow}(t'') \notin victims_{\downarrow}(k) \\ &\equiv u_{\downarrow}O(k) \neq u_{\downarrow}O(t') \wedge u_{\downarrow}O(t') \notin victims_{\downarrow}O(k) \\ &\equiv ucon_{\downarrow}O(k,t'). \end{aligned}$$

Since the schedule and configurations are unchanged at k, we obtain that step k is still not made by a write buffer in strong memory mode

$$\nexists i.SC_{\uparrow}O(k) \wedge sO(k) \in \Sigma_{WB,i}.$$

We conclude that the sorted schedule also has a local tail with an independent end from t to k

$$\mathcal{L}^I O(t,k),$$

which is the third claim.

Since the schedule is sorted and the unit making step l' is by definition the minimal unit in the order, all steps from t until l are made by the same unit

$$\forall t' \in [t:l].u_{\uparrow}O(t') = u_{\uparrow}O(l).$$

All of those steps are local

$$\forall t' \in [t:l].L_{\downarrow}O(t').$$

Since by Lemma 136 the local steps are not write buffer steps, all steps of the same unit are also steps of the same object

$$\forall t' \in [t:l].o_{\uparrow}O(t') = P, u_{\uparrow}O(t') = P, u_{\uparrow}O(l') = o_{\uparrow}O(l),$$

which is the fourth claim.

The fifth claim follows from equivalence of the two schedules since l and l' are the n-th step of object X

$$c_{\downarrow}O^l =_{\downarrow}^{sO(l)} c_{\downarrow}^{l'}.$$

As we have shown above, all steps t' between t and l in the sorted schedule

$$t' \in [t:l)$$

are made by the same object as l, i.e., object X

$$o_{\uparrow}O(t') = o_{\uparrow}O(l) = X,$$

and thus t' is the n'-th step of unit X

$$t' = \#X \approx n'(sO).$$

Since step t' is before step l, and step counts are monotone, we obtain that n' is also before n

$$n' < n.$$

Let step t'' now be the n'-th step of object X in the original schedule

$$t'' = \#X \approx n'(s).$$

again since step counts are monotone, the n'-th step is also made before the n-th step in the original schedule

$$t'' = \#X \approx n'(s) < \#X \approx n(s) = l'.$$

Furthermore, since the step was in the sorted interval, by Lemma 319 it is still in the sorted interval

$$t'' \in [t : k)$$

and it is now between t and l'

$$t'' \in [t : l').$$

Since the schedules are equivalent, steps t', t'' have strong agreement in the low-level machine

$$c_{\downarrow} O^{t'} =_{\downarrow}^{sO(t')} c_{\downarrow}^{t''}$$

and the same oracle inputs

$$sO(t') = s(t'')$$

and by Lemma 329 the same holds in the high-level machine

$$c_{\uparrow} O^{t'} =_{\uparrow}^{sO(t')} c_{\uparrow}^{t''}.$$

By Lemma 150 the steps thus have the same outputs in the high-level machine

$$out_{\uparrow} O(t') = out_{\uparrow}(t'').$$

Thus for each $t' \in [t : l)$ there is a $t'' \in [t : l')$ with the same outputs. It immediately follows that the set of addresses overwritten in the interval from t to l' after sorting is a subset of the set of addresses overwritten in the interval from t to l in the original schedule

$$\begin{aligned} out_{\uparrow} O([t : l)) &= \bigcup_{t' \in [t:l)} out_{\uparrow} O(t') \\ &\subseteq \bigcup_{t'' \in [t:l')} out_{\uparrow}(t'') \\ &= out_{\uparrow}([t : l')), \end{aligned}$$

which is the sixth claim. □

We begin by moving the absence of write-read races of the low-level machine to the unsorted schedule.

Lemma 362. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

then there is no write-read race between any $l' \in [t : k)$ and k in the low-level machine

$$\neg WR_{\downarrow}(l',k).$$

Proof. By Lemma 361 we obtain O, l such that sO still has a local tail with an independent end from t to k

$$\mathcal{L}^I O(t,k),$$

and the schedule is sorted

$$\forall t' \in [t:l].o_\uparrow O(t') = o_\uparrow O(l),$$

and that steps k and l' can be executed at k and l, respectively

$$c_\downarrow O^k =_\downarrow^{sO(k)} c_\downarrow^k,$$
$$c_\downarrow O^l =_\downarrow^{sO(l)} c_\downarrow^{l'}.$$

By Lemma 360 there is no write-read race in sO

$$\neg WR_\downarrow O(l,k).$$

With Lemma 150 we obtain that the outputs resp. inputs are the same

$$in_\downarrow O(k) = in_\downarrow(k),$$
$$out_\downarrow O(l) = out_\downarrow(l'),$$

and thus there is also no write-read race in the sorted schedule

$$WR_\downarrow(l',k) \equiv out_\downarrow(l') \hat{\cap} in_\downarrow(k) \equiv out_\downarrow O(l) \hat{\cap} in_\downarrow O(k) \equiv WR_\downarrow O(l,k) \equiv 0,$$

which is the claim. □

Thus step k can be moved to step t without being affected at all.

Lemma 363. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

then k can be executed at t

$$c_\downarrow^k =_\downarrow^{s(k)} c_\downarrow[t \leftarrow k]^t.$$

Proof. We generalize the claim to $t' \in [t:k]$

$$c_\downarrow^k \stackrel{!}{=}_\downarrow^{s(k)} c_\downarrow[t' \leftarrow k]^{t'},$$

which we prove by downwards induction on t', starting from k.

In the inductive step $t' \to t'-1$ we have that step k can be executed at t'

$$c_\downarrow^k =_\downarrow^{s(k)} c_\downarrow[t' \leftarrow k]^{t'},$$

and by Lemma 362 that there is no write-read race between $t'-1$ and k in the original schedule

$$\neg WR_\downarrow(t'-1,k).$$

With Lemma 150 with $X := in$ we obtain that there is also no write-read race between $t'-1$ and t' in the reordered schedule

$$WR_\downarrow[t' \leftarrow k](t'-1,t') \equiv out_\downarrow[t' \leftarrow k](t'-1) \hat{\cap} in_\downarrow[t' \leftarrow k](t')$$

$$\equiv out_{\downarrow}(t'-1) \dot{\cap} in_{\downarrow}(k)$$
$$\equiv WR_{\downarrow}(t'-1,k).$$

By assumption the steps were unit-concurrent and thus made by different units

$$diffu_{\downarrow}(k,t'-1).$$

Clearly the steps are still made by different units

$$diffu_{\downarrow}[t' \leftarrow k](t'-1,t').$$

By Lemma 164 with $t := t'-1$ and $s := s[t' \leftarrow k]$, we can thus swap the order of $t'-1$ and t', which is the same as moving k to $t'-1$

$$\begin{aligned} c^k_{\downarrow} &=^{s(k)}_{\downarrow} c_{\downarrow}[t' \leftarrow k]^{t'} && \text{IH} \\ &=^{s[t' \leftarrow k](t')}_{\downarrow} c_{\downarrow}[t' \leftarrow k][t'-1 \leftrightarrow t']^{t'-1} && \text{L 164} \\ &= c_{\downarrow}[t'-1 \leftarrow k]^{t'-1} \end{aligned}$$

and since the oracle input at t' when k is moved to t' is the same as at k in the original schedule

$$s(k) = s[t' \leftarrow k](t')$$

the claim follows by transitivity

$$c^k_{\downarrow} =^{s(k)}_{\downarrow} c_{\downarrow}[t'-1 \leftarrow k]^{t'-1}.$$

□

We can now prove the absence of write-write races in sorted schedules.

Lemma 364. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

and all steps between t and $l \in [t:k)$ are made by the same object

$$\forall t' \in [t:l].o_{\uparrow}(t') = o_{\uparrow}(l),$$

then there is no write-write race between l and k in the low-level machine

$$\neg WW_{\downarrow}(l,k).$$

Proof. By Lemma 363 the configurations at t when k is moved to t and at k strongly agree

$$c^k_{\downarrow} =^{s(k)}_{\downarrow} c_{\downarrow}[t \leftarrow k]^t.$$

By Lemma 150 those configurations agree on the outputs

$$out_{\downarrow}[t \leftarrow k](t) = out_{\downarrow}(k).$$

Assume now for the sake of contradiction that such a race exists, in which case there is an intersection between outputs and outputs

$$out_{\downarrow}(l) \dot{\cap} out_{\downarrow}(k)$$

and thus also when k is moved to t

$$out_{\downarrow}(l) \dot{\cap} out_{\downarrow}[t \leftarrow k](t).$$

That contradicts Lemma 359. □

We transfer the absence of write-write races to the unsorted schedules.

Lemma 365. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

then there is no write-write race between $l' \in [t:k)$ and k in the low-level machine

$$\neg WW_{\downarrow}(l',k).$$

Proof. By Lemma 361 we obtain O,l such that sO still has a local tail with an independent end from t to k

$$\mathcal{L}^I O(t,k),$$

and the schedule is sorted

$$\forall t' \in [t:l].o_{\uparrow}O(t') = o_{\uparrow}O(l),$$

and that steps k and l' can be executed at k and l, respectively

$$c_{\downarrow}O^k =_{\downarrow}^{sO(k)} c_{\downarrow}^k,$$
$$c_{\downarrow}O^l =_{\downarrow}^{sO(l)} c_{\downarrow}^{l'}.$$

By Lemma 364 there is no write-write race in sO

$$\neg WW_{\downarrow}O(l,k).$$

With Lemma 150 we obtain that the outputs are the same

$$out_{\downarrow}O(k) = out_{\downarrow}(k),$$
$$out_{\downarrow}O(l) = out_{\downarrow}(l'),$$

and thus there is also no write-write race in the sorted schedule

$$WW_{\downarrow}(l',k) \equiv out_{\downarrow}(l') \dot{\frown} out_{\downarrow}(k) \equiv out_{\downarrow}O(l) \dot{\frown} out_{\downarrow}O(k) \equiv WW_{\downarrow}O(l,k) \equiv 0,$$

which is the claim. □

We wish to similarly transfer the visible write-read races, but this is more difficult: there is an inherent "direction" in the visible outputs, since we subtract all outputs behind the write until the read. Thus a visible write-read race between k and l' is somewhat meaningless, and we are focus instead on the visible write-read race between t and $l'+1$ when k is moved to t. As above, we wish to show that no such races exist; but to do so in the schedule where k is moved to t (rather than the original schedule s, as in the lemmas above) we would need to show that steps $[t:l')$ in the original schedule and $[t+1:l'+1)$ after moving k to t have the same outputs

$$out_{\uparrow}([t:l'))) \stackrel{!}{=} out_{\uparrow}[t \leftarrow k]([t:l')))$$

and that steps l' in the original schedule and $l'+1$ after moving k to t have the same inputs

$$in_{\uparrow}(l') \stackrel{!}{=} in_{\uparrow}[t \leftarrow k](l'),$$

which will only be possible after we have shown that such races do not exist.

Instead of attempting to prove these equalities and that there is no visible write-read race, we unfold the definition of a visible write-read race, substitute the write set of step k with the write step of step t after moving k to t (an equality that we can already prove)

$$WS_{\uparrow}(k) = WS_{\uparrow}[t \leftarrow k](t)$$

but keep the outputs of the intermediate steps and the inputs of l' in the original schedule. The resulting lemma is a little ugly and technical, but will do the job.

Lemma 366. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^I(t,k),$$

then for all $l' \in [t:k)$ in the high-level machine, the device closure of the write-set at t when moving k to t and the write-set at t when moving k to t minus the outputs from t to l' do not intersect the inputs of step l'

$$dc(WS_{\uparrow}[t \leftarrow k](t)) \cup (WS_{\uparrow}[t \leftarrow k](t) \setminus out_{\uparrow}([t:l'))) \not\parallel in_{\uparrow}(l').$$

Proof. By Lemma 361 we obtain O, l such that sO still has a local tail with an independent end from t to k

$$\mathcal{L}^I O(t,k),$$

and the schedule is sorted

$$\forall t' \in [t:l].o_{\uparrow}O(t') = o_{\uparrow}O(l),$$

and that the steps before t and at k and at l resp. l' are the same

$$sO[0:t-1] = s[0:t-1] \wedge sO(k) = s(k) \wedge sO(l) = s(l')$$

and that step l' can be executed at l

$$c_{\downarrow}O^l =_{\downarrow}^{sO(l)} c_{\downarrow}^{l'},$$

and that the outputs between t and l are a subset of those between t and l'

$$out_{\uparrow}O([t:l)) \subseteq out_{\uparrow}([t:l')).$$

By Lemma 357 we obtain that the sorted schedule has no visible write-read race

$$\neg VR_{\uparrow}O[t \leftarrow k](t, l+1).$$

Unfolding the definition of VR, then of $vout$, and then of vws, we obtain that there is no intersection between the the device closure of the write-set at t and the write-set at t minus the outputs from $t+1$ to $l+1$ and the inputs of step $l+1$

$$\begin{aligned}
& vout_{\uparrow}O[t \leftarrow k](t, l+1) \not\parallel in_{\uparrow}O[t \leftarrow k](l+1) \\
\iff\ & dc(WS_{\uparrow}O[t \leftarrow k](t)) \cup vws_{\uparrow}O[t \leftarrow k](t, l+1) \not\parallel in_{\uparrow}O[t \leftarrow k](l+1) \\
\iff\ & dc(WS_{\uparrow}O[t \leftarrow k](t)) \cup (WS_{\uparrow}O[t \leftarrow k](t) \setminus out_{\uparrow}O[t \leftarrow k]([t+1:l+1))) \\
& \not\parallel in_{\uparrow}O[t \leftarrow k](l+1). \qquad (4.23)
\end{aligned}$$

Since the oracle inputs at k are the same, so are the oracle inputs at t when k is moved to t

$$s[t \leftarrow k](t) = s(k) = sO(k) = sO[t \leftarrow k](t),$$

and since the steps before t are the same and not affected by moving k to t, they are also the same after that move

$$s[t \leftarrow k][0:t-1] = s[0:t-1] = sO[0:t-1] = sO[t \leftarrow k][0:t-1].$$

We conclude that all steps until t are the same after that move

$$s[t \leftarrow k][0:t] = sO[t \leftarrow k][0:t],$$

and thus the write-set at t is the same

$$WS_{\uparrow}[t \leftarrow k](t) = WS_{\uparrow}O[t \leftarrow k](t). \tag{4.24}$$

By Lemma 356 all the steps in the sorted schedule are executed the same before and after that move

$$\forall t' \in [t:k).c_{\uparrow}O[t \leftarrow k]^{t'+1} =_{\uparrow}^{sO[t \leftarrow k](t'+1)} c_{\uparrow}O^{t'}.$$

Thus by Lemma 150 we obtain in particular for the steps $t' \in [t:l)$ that the outputs are the same

$$\forall t' \in [t:l).out_{\uparrow}O[t \leftarrow k](t'+1) = out_{\uparrow}O(t'),$$

and thus the outputs from $t+1$ to $l+1$ after the move are the same as the outputs from t to l before the move

$$\begin{aligned} out_{\uparrow}O[t \leftarrow k]([t+1:l+1)) &= \bigcup_{t' \in [t+1:l+1)} out_{\uparrow}O[t \leftarrow k](t') \\ &= \bigcup_{t' \in [t:l)} out_{\uparrow}O[t \leftarrow k](t'+1) \\ &= \bigcup_{t' \in [t:l)} out_{\uparrow}O(t') \\ &= out_{\uparrow}O([t:l)) \end{aligned}$$

which as we know are a subset of the outputs from t to l' in the original schedule

$$\subseteq out_{\uparrow}([t:l')).$$

Thus the write-set minus the outputs in the sorted schedule after the move subsume the write-set minus the outputs in the original schedule

$$WS_{\uparrow}O[t \leftarrow k](t) \setminus out_{\uparrow}O[t \leftarrow k]([t+1:l+1)) \supseteq WS_{\uparrow}O[t \leftarrow k](t) \setminus out_{\uparrow}([t:l')). \tag{4.25}$$

Since steps l and l' are before k and agree in the low-level machine, by Lemma 329 we obtain that they also agree in the high-level machine

$$c_{\uparrow}O^{l} =_{\uparrow}^{sO(l)} c_{\uparrow}^{l'}.$$

Thus by Lemma 150 we obtain that step l in the sorted schedule has the same inputs as step l' in the original schedule

$$in_{\uparrow}O(l) = in_{\uparrow}(l').$$

By Lemma 356 step $l+1$ in the sorted schedule after the move is executed the same as step l in the sorted schedule

$$c_{\uparrow}O[t \leftarrow k]^{l+1} =_{\uparrow}^{sO[t \leftarrow k](l+1)} c_{\uparrow}O^{l}.$$

Thus by Lemma 150 we obtain that step $l+1$ after the move has the same inputs as step l' before the move

$$in_{\uparrow}O[t \leftarrow k](l+1) = in_{\uparrow}O(l) = in_{\uparrow}(l'). \tag{4.26}$$

Combining Statement (4.23) with Eqs. (4.24) to (4.26) we obtain the claim

$$\begin{aligned}
& dc(WS_{\uparrow}O[t \leftarrow k](t)) \\
& \cup (WS_{\uparrow}O[t \leftarrow k](t) \setminus out_{\uparrow}O[t \leftarrow k]([t+1 : l+1))) \\
& \quad \not\Cap in_{\uparrow}O[t \leftarrow k](l+1) \\
\Longrightarrow\ & dc(WS_{\uparrow}O[t \leftarrow k](t)) \cup (WS_{\uparrow}O[t \leftarrow k](t) \setminus out_{\uparrow}([t : l'))) \\
& \not\Cap in_{\uparrow}O[t \leftarrow k](l+1) && \text{E (4.25)} \\
\Longleftrightarrow\ & dc(WS_{\uparrow}[t \leftarrow k](t)) \cup (WS_{\uparrow}[t \leftarrow k](t) \setminus out_{\uparrow}([t : l'))) \not\Cap in_{\uparrow}(l') && \text{E (4.26), (4.24)}
\end{aligned}$$

□

Lemma 367. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

and after all steps t' before $l' \in [t : k)$ the configurations are nearly the same when moving k to t

$$\forall t' \in [t : l').\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \to k], t'+2, t'+1),$$

then the configurations until l' are nearly the same too

$$\forall t' \in [t : l'].\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \to k], l'+1, l').$$

Proof. By Lemma 190 the configuration at t is nearly the same

$$\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \to k], t+1, t)$$

With an index shift by one we obtain that the configurations from $t+1$ before $l'+1$ are also nearly the same

$$\forall t' \in [t+1 : l'+1).\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \to k], t'+1, t'),$$

and thus the configurations from $t+1$ until l'

$$\forall t' \in [t+1 : l'].\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \to k], t'+1, t'),$$

and the claim follows. □

We now obtain the crucial result that there are no visible write-read races in unsorted schedules.

Lemma 368. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

all of the following are true for each $l' \in [t : k)$ when moving k to t.

1. *In the high-level machine there is no visible write-read race with* $l'+1$

$$\neg VR_{\uparrow}[t \leftarrow k](t, l'+1),$$

2. *steps* t *and* $l'+1$ *are made by different units,*

$$diffu_{\uparrow}(t, l'+1),$$

3. *the configurations after* $l'+1$ *after moving* k *to* t *and after* l' *are nearly the same*

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k], l'+2, l'+1).$$

Proof. By strong induction on l' we have that all steps $t' \in [t : l')$ satisfy the the property

$$\forall t' \in [t : l').\neg VR_{\uparrow}[t \leftarrow k](t, t'+1) \wedge \mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k], t'+2, t'+1).$$

Note that this hypothesis may be vacuously true when $l' = t$. We use Lemma 367 to obtain that the configurations at l' are nearly the same anyways

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k], l'+1, l').$$

Clearly all of these steps are delayed exactly by one

$$\forall t' \in [t : l'].s[t \leftarrow k](t'+1) = s[t \leftarrow k][t \rightarrow k](t') = s(t').$$

By Lemma 337, the step can be executed at t

$$\mathcal{E}(t, k, t)$$

and is thus unit-concurrent with each step $t'+1$ where $t' \leq l'$

$$\forall t' \in [t : l'].ucon_{\uparrow}[t \leftarrow k](t, t'+1)$$

and thus also made by a different unit

$$\forall t' \in [t : l').diffu[t \leftarrow k](t, t'+1). \tag{4.27}$$

For the steps before l' we conclude by Lemma 196 with $k := t'+1$ and $k' := t'$ that they can be executed at their new position

$$\forall t' \in [t : l').c_{\uparrow}[t \leftarrow k]^{t'+1} =^{s[t \leftarrow k](t'+1)}_{\uparrow} c_{\uparrow}[t \leftarrow k][t \rightarrow k]^{t'} = c^{t'}_{\uparrow}.$$

By Lemma 150, the outputs of the steps are unchanged

$$\forall t' \in [t : l').out_{\uparrow}[t \leftarrow k](t'+1) = out_{\uparrow}(t'),$$

and thus also the combined outputs

$$\begin{aligned} out_{\uparrow}[t \leftarrow k]([t+1 : l'+1)) &= \bigcup_{t' \in [t+1 : l'+1)} out_{\uparrow}[t \leftarrow k](t') \\ &= \bigcup_{t' \in [t : l')} out_{\uparrow}[t \leftarrow k](t'+1) \end{aligned}$$

$$= \bigcup_{t' \in [t:l')} out_\uparrow(t')$$
$$= out_\uparrow([t:l')).$$

By Lemma 366 there is no intersection between the device closure of the write-set and the write-set minus the outputs from t to l' and the inputs

$$dc(WS_\uparrow[t \leftarrow k](t)) \cup (WS_\uparrow[t \leftarrow k](t) \setminus out_\uparrow([t:l'))) \not\pitchfork in_\uparrow(l')$$

and we rewrite with the equality we obtained for the outputs

$$dc(WS_\uparrow[t \leftarrow k](t)) \cup (WS_\uparrow[t \leftarrow k](t) \setminus out_\uparrow[t \leftarrow k]([t+1:l'+1))) \not\pitchfork in_\uparrow(l')$$

and fold the definition of the visible write-set

$$dc(WS_\uparrow[t \leftarrow k](t)) \cup vws_\uparrow[t \leftarrow k](t, l'+1) \not\pitchfork in_\uparrow(l')$$

and of the visible outputs

$$vout_\uparrow[t \leftarrow k](t, l'+1) \not\pitchfork in_\uparrow(l')$$

and have thus no intersection between the visible outputs in the reordered schedule and the inputs in the original schedule.

By Lemma 367 in particular the configurations at l' are nearly the same

$$\mathcal{I}^t_\uparrow[t \leftarrow k]([t \to k], l'+1, l').$$

By Lemma 191 the configurations agree on the memory content of the inputs

$$m_\uparrow[t \leftarrow k]^{l'+1} =_{in_\uparrow(l')} m_\uparrow[t \leftarrow k][t \to k]^{l'} = m^{l'}_\uparrow,$$

in particular the core and fetched registers

$$m_\uparrow[t \leftarrow k]^{l'+1} =_{C_\uparrow(l') \cup F_\uparrow(l')} m^{l'}_\uparrow.$$

By Lemma 152 we obtain that steps l' and $l'+1$ after the move have the same inputs

$$in_\uparrow[t \leftarrow k](l'+1) = in_\uparrow(l').$$

We conclude that there is no intersection between the visible outputs and the inputs after the move

$$vout_\uparrow[t \leftarrow k](t, l'+1) \not\pitchfork in_\uparrow[t \leftarrow k](l'+1)$$

and thus no visible write-read race

$$\neg VR_\uparrow[t \leftarrow k](t, l'+1),$$

which is the first claim.

Since by Statement (4.27) the steps until $l'+1$ are made by different units, in particular step $l'+1$ is made by a different unit

$$diffu_\uparrow[t \leftarrow k](t, l'+1),$$

which is the second claim.

The third claim is simply Lemma 198. □

We can finally prove that the intermediate steps are not affected by moving k to t.

Lemma 369. *If the schedule has a local tail with an independent end from t to k*

$$\mathcal{L}^{I}(t,k),$$

then in the high-level machine the configurations at $l' \in [t:k)$ strongly agree before and after the move

$$c_{\uparrow}[t \leftarrow k]^{l'+1} =_{\uparrow}^{s[t\leftarrow k](l'+1)} c_{\uparrow}^{l'}.$$

Proof. By Lemma 368 we obtain that there is no visible write-read race and the steps are made by different units

$$\neg VR_{\uparrow}[t \leftarrow k](t,l'+1) \wedge diffu[t \leftarrow k](t,l'+1),$$

and with Lemma 368 we obtain that after all t' before l' the configurations are nearly the same

$$\forall t' \in [t:l').\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \rightarrow k],t'+2,t'+1).$$

By Lemma 367 the configurations until l' are nearly the same, in particular the configuration at l'

$$\mathcal{I}_{\uparrow}^{t}[t \leftarrow k]([t \rightarrow k],l'+1,l').$$

The claim is now Lemma 196. □

4.12.4 Reordering Write Buffer Steps

So far we have mostly excluded write buffer steps in strong memory mode. We now show that all key properties (e.g., strong agreement between the steps) also hold when reordering those types of steps.

We say that schedule *s has a local tail from t to k that ends with with an issued write buffer step* and write $\mathcal{L}^{W}(t,k)$ when all of the following hold.

1. The schedule has a local tail from t to k

$$\mathcal{L}[s](t,k),$$

2. step k is a write buffer step in strong memory mode

$$s(k) \in \Sigma_{WB,i} \wedge SC_{\downarrow}(k)$$

3. the write committed at k was already buffered before t

$$hd(issue_{\downarrow}^{k}(i)) < t,$$

4. only unit i can have dirty buffers at t

$$dirty_{\downarrow}(t,j) \rightarrow i = j.$$

Lemma 370. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^{W}(t,k),$$

step k can be executed at t

$$c_{\downarrow}[t \leftarrow k]^{t} =_{\downarrow}^{s[t\leftarrow k](t)} = c_{\downarrow}^{k}.$$

Proof. By Lemma 328 the mode of unit $i = u_{\uparrow}(k)$ is the same

$$SC_{i\downarrow}(t) = SC_{i\downarrow}(k).$$

The only input of a write buffer step are the mode registers

$$in_{\downarrow}(k) = C_{\downarrow}(k) \cup F_{\downarrow}(k) \cup R_{\downarrow}(k) = A_{SC,i},$$

and thus the configurations agree on the content of inputs

$$m^t_{\downarrow} =_{in_{\downarrow}(k)} m^k_{\downarrow}.$$

The write committed at k was already buffered before t

$$hd(issue^k_{\downarrow}(i)) < t,$$

and by the monotonicity of the write buffer, the write was a buffered write at t

$$hd(issue^k_{\downarrow}(i)) \in issue^t_{\downarrow}(i).$$

Thus clearly there were buffered writes at t

$$issue^t_{\downarrow}(i) \neq \varepsilon.$$

By Lemma 326 the sequence of issued writes has the same head

$$hd(issue^k_{\downarrow}(i)) = hd(issue^t_{\downarrow}(i))$$

and by Lemma 123 the write buffers have the same head

$$\begin{aligned} hd(wb^k_{\downarrow}(i)) &= BW_{\downarrow}(hd(issue^k_{\downarrow}(i))) \\ &= BW_{\downarrow}(hd(issue^t_{\downarrow}(i))) \\ &= hd(wb^t_{\downarrow}(i)) \end{aligned}$$

and thus by definition the buffers of the configuration at k subsume those at t

$$bufS_{\downarrow}(s(k), c^k_{\downarrow}, c^t_{\downarrow}).$$

Since by hypothesis the schedule has a local tail, it is valid until k

$$\Gamma^k_{\downarrow}(s)$$

and in particular step k is valid

$$\Gamma_{\downarrow}(k).$$

By Lemma 187 the configurations at t and and k strongly agree when stepped with $s(k)$

$$c^t_{\downarrow} =^{s(k)}_{\downarrow} c^k_{\downarrow},$$

and the claim follows with the observation that the configuration t is unaffected by the reordering

$$c_{\downarrow}[t \leftarrow k]^t =^{s[t\leftarrow k](k)}_{\downarrow} c^k_{\downarrow}.$$

□

This proves that step k can be moved. For the other steps, we use again that the configurations are nearly the same; but we need extra machinery to deal with steps made by the same unit, i.e., by the processor of the unit making the write buffer step.

Lemma 371. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^W(t,k),$$

then the mode of step t after moving k to t is in strong memory mode in the high-level machine

$$SC_\uparrow[t \leftarrow k](t).$$

Proof. By Lemma 322 that the schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and thus t-abstract

$$s \in \mathrm{ABS}_t$$

and by Lemma 269 we obtain that step t was made in the same mode in both machines

$$SC_\uparrow[t \leftarrow k](t) = SC_\downarrow[t \leftarrow k](t).$$

By Lemma 370 we obtain that step k can be executed at t

$$c_\downarrow[t \leftarrow k]^t =_\downarrow^{s[t \leftarrow k](t)} = c_\downarrow^k,$$

and with Lemma 144 we obtain that step t in the high-level machine is made in strong memory mode

$$SC_\uparrow[t \leftarrow k](t) = SC_\downarrow[t \leftarrow k](t) = SC_\downarrow(k) = 1.$$

□

Lemma 372. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^W(t,k),$$

and at $l' \in [t : k)$ the configurations are nearly the same and the write buffer of unit $i = u_\uparrow(k)$ at $l'+1$ after the reordering is the tail of the write buffer at l'

$$\mathcal{I}_\uparrow^t[t \leftarrow k]([t \to k], l'+1, l') \wedge wb_\uparrow[t \leftarrow k]^{l'+1}(i) = tl(wb_\uparrow^{l'}(i)),$$

then the configurations at $l'+1$ and l' (in their respective schedule) strongly agree

$$c_\uparrow[t \leftarrow k]^{l'+1} =_\uparrow^{s[t \leftarrow k](l'+1)} c_\uparrow^{l'}.$$

Proof. We show the equivalent claim that the configurations strongly agree when stepped with $s(l')$

$$c_\uparrow[t \leftarrow k]^{l'+1} \stackrel{!}{=}_\uparrow^{s(l')} c_\uparrow^{l'}.$$

We obtain by Lemma 371 that the step at t is made in strong memory mode

$$SC_\uparrow[t \leftarrow k](t)$$

and thus not using low-level machine semantics

$$\neg LL_{\uparrow}[t \leftarrow k](t).$$

Therefore the step has no outputs

$$out_{\uparrow}[t \leftarrow k](t) = \emptyset$$

and by Lemma 192 we obtain that the memory is the same

$$m_{\uparrow}[t \leftarrow k]^{l'+1} = m_{\uparrow}^{l'},$$

in particular at the inputs

$$m_{\uparrow}[t \leftarrow k]^{l'+1} =_{in_{\uparrow}(l')} m_{\uparrow}^{l'}.$$

The proof now distinguishes between steps of unit i and other steps.

$u_{\uparrow}(l') = i$: Because l' is in the local tail, it is local

$$L_{\downarrow}(l').$$

By contraposition of Lemma 136, the step is not a write buffer step

$$s(l') \notin \Sigma_{WB,i},$$

and thus a processor step of unit i

$$s(l') \in \Sigma_{P,i}.$$

By Lemma 322 that the schedule is k-abstract

$$s \in \mathrm{ABS}_k.$$

With Lemmas 327 and 268 we obtain that the step is done in strong memory mode

$$\begin{aligned} SC_{\uparrow}[t \leftarrow k](l'+1) &= SC_{i\uparrow}[t \leftarrow k](l'+1) \\ &= SC_{i\uparrow}(l') \\ &= SC_{i\uparrow}(k) && \text{L 327} \\ &= SC_{i\downarrow}(k) && \text{L 268} \\ &= SC_{\downarrow}(k) = 1, \end{aligned}$$

and thus not using low-level machine semantics

$$\neg LL_{\uparrow}[t \leftarrow k](l'+1).$$

Therefore the forwarding inputs are empty

$$fin_{\uparrow}[t \leftarrow k](l'+1) = \emptyset.$$

There is thus no hit in the head of the write buffer

$$\neg hit(fin_{\uparrow}[t \leftarrow k](l'+1), hd(wb_{\uparrow}^{l'}(i))).$$

The buffer at l' is the head of the write buffer at l' followed by the buffer at $l'+1$ after the move

$$wb^{l'}_{\uparrow}(i) = hd(wb^{l'}_{\uparrow}(i)) \circ tl(wb^{l'}_{\uparrow}(i)) = hd(wb^{l'}_{\uparrow}(i)) \circ wb_{\uparrow}[t \leftarrow k]^{l'+1}(i)$$

and thus the buffers at l' before the reordering subsume those at $l'+1$ after the reordering

$$bufS_{\uparrow}(s(l'), c^{l'}_{\uparrow}, c_{\uparrow}[t \leftarrow k]^{l'+1}).$$

By Lemma 323 the schedule is valid in the high-level machine before k

$$\Gamma^{k-1}_{\uparrow}(s)$$

and since l' is before k, in particular step l' is valid

$$\Gamma_{\uparrow}(l').$$

By Lemma 187 the claim follows

$$c^{l'}_{\uparrow} =^{s(l')}_{\uparrow} c_{\uparrow}[t \leftarrow k]^{l'+1}.$$

$u_{\uparrow}(l') \neq i$**:** Since the configurations are nearly the same, the write buffers of that unit are the same

$$wb_{\uparrow}[t \leftarrow k]^{l'+1} =_{u_{\uparrow}[t \leftarrow k](l'+1)} wb^{l'}_{\uparrow}.$$

By Lemma 188 the configurations strongly agree

$$c_{\uparrow}[t \leftarrow k]^{l'+1} =^{s(l')}_{\uparrow} c^{l'}_{\uparrow}.$$

□

Lemma 373. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^{W}(t,k),$$

then at each $l' \in [t:k)$ the configurations are nearly the same and the write buffer of unit $i = u_{\uparrow}(k)$ at $l'+1$ after the reordering is the tail of the write buffer at l'

$$\mathcal{I}^{t}_{\uparrow}[t \leftarrow k]([t \rightarrow k], l'+1, l') \wedge wb_{\uparrow}[t \leftarrow k]^{l'+1}(i) = tl(wb^{l'}_{\uparrow}(i)).$$

Proof. The proof is by induction on l', starting at t.

In the base case, the first part of the claim is Lemma 190. By Lemma 324 step t after the move is made by the write buffer of unit i

$$s[t \leftarrow k](t) \in \Sigma_{WB,i}$$

which by definition executes a pop operation

$$Op_{i\uparrow}[t \leftarrow k](t) = pop.$$

The second part of the claim follows by definition and the observation that the configuration at t is unchanged by moving k to t

$$wb_{\uparrow}[t \leftarrow k]^{l'+1}(i) = Op_{i\uparrow}[t \leftarrow k](t)(wb_{\uparrow}[t \leftarrow k]^{t}(i), \ldots)$$

$$
\begin{aligned}
&= tl(wb_\uparrow[t \leftarrow k]^t(i)) \\
&= tl(wb_\uparrow^t(i)) \\
&= tl(wb_\uparrow^{l'}(i)).
\end{aligned}
$$

In the inductive step $l' \to l'+1$ we obtain by Lemma 372 that the configurations at l' and $l'+1$ strongly agree

$$c_\uparrow[t \leftarrow k]^{l'+1} =_\uparrow^{s[t \leftarrow k](l'+1)} c_\uparrow^{l'}.$$

The first part of the claim follows by Lemma 197. For the second part of the claim, we distinguish between steps of the same unit and steps of a different unit.

By Lemma 322 the schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and by Lemma 267 the head of the sequence of issued writes is the same

$$hd(issue_\uparrow^k(i)) = hd(issue_\downarrow^k(i)).$$

By definition of $\mathcal{L}^W$, that head is already buffered at t

$$hd(issue_\uparrow^k(i)) < t$$

and by the monotonicity of the sequence of issued writes, it is also buffered at $l' \in [t:k)$

$$hd(issue_\uparrow^k(i)) \in issue_\uparrow^{l'}(i),$$

and by Lemma 123 we obtain that the corresponding write is an element of the write buffer at l'

$$BW_\uparrow(\ldots) \in wb_\uparrow^{l'}(i),$$

which is thus non-empty

$$wb_\uparrow^{l'}(i) \neq \varepsilon.$$

By Lemma 45 we can change the order of operations, and by Lemma 150 both steps perform the same operation on the buffer

$$
\begin{aligned}
wb_\uparrow[t \leftarrow k]^{l'+2}(i) &= Op_{i\uparrow}[t \leftarrow k](l'+1)(wb_\uparrow[t \leftarrow k]^{l'+1}(i), BW_\uparrow[t \leftarrow k](l'+1)) \\
&= Op_{i\uparrow}(l')(wb_\uparrow[t \leftarrow k]^{l'+1}(i), BW_\uparrow(l')) && \text{L 150} \\
&= Op_{i\uparrow}(l')(tl(wb_\uparrow^{l'}(i)), BW_\uparrow(l')) && \text{IH} \\
&= tl(Op_{i\uparrow}(l')(wb_\uparrow^{l'}(i), BW_\uparrow(l'))) && \text{L 45} \\
&= tl(wb_\uparrow^{l'+1}(i)),
\end{aligned}
$$

and the second part of the claim follows. □

Lemma 374. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^W(t,k),$$

then for each $t' \in [t:k)$, the configurations at $l'+1$ and l' (in their respective schedule) strongly agree

$$c_\uparrow[t \leftarrow k]^{l'+1} =_\uparrow^{s[t \leftarrow k](l'+1)} c_\uparrow^{l'}.$$

Proof. By Lemma 373

$$\mathcal{I}^t_\uparrow[t \leftarrow k]([t \rightarrow k], l'+1, l') \wedge wb_\uparrow[t \leftarrow k]^{l'+1}(i) = tl(wb^{l'}_\uparrow(i)).$$

The claim is Lemma 372. □

Lemma 375. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^W(t,k),$$

then step k has no victims

$$victims_\downarrow(k) = \emptyset.$$

Proof. By Lemma 322, the schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and by Lemma 267 the machines have the same write buffer for unit $i = u_\uparrow(k)$

$$wb^k_\uparrow(i) = wb^k_\downarrow(i)$$

and by Lemma 269 the same memory mode

$$SC_\uparrow(k) = SC_\downarrow(k) = 1.$$

By Lemma 323, the schedule is valid until $k-1$ in the high-level machine

$$\Gamma^{k-1}_\uparrow(s).$$

Let now j be an arbitrary processor

$$j \in P$$

and we will show that j is not a victim, which proves the claim.

We obtain first by definition of $\mathcal{L}^W$ that the memory mode is raised in the high-level machine

$$SC_{i\uparrow}(k) = SC_\uparrow(k) = 1,$$

and then with Lemma 176 that there is no hit with processor registers in the high-level machine

$$\neg hit(A_{PR,j}, wb^k_\uparrow(i))$$

and thus also not in the low-level machine

$$\neg hit(A_{PR,j}, wb^k_\downarrow(i)).$$

Therefore the domain of the head of the write buffer does not intersect with processor registers

$$Dom(hd(wb^k_\downarrow(i))) \not\cap A_{PR,j}.$$

Since we are in the low-level machine we use low-level machine semantics

$$LL_\downarrow(k)$$

and the write buffer step commits the head of the write buffer to memory

$$W_\downarrow(k) = hd(wb^k_\downarrow(i))$$

and we conclude that step k is not modifying any processor registers

$$Dom(W_{\downarrow}(k)) = Dom(hd(wb^k_{\downarrow}(i))$$
$$\not\parallel A_{PR,j}.$$

By the definition of victims j is not a victim

$$j \notin victims_{\downarrow}(k)$$

which is the claim. □

Lemma 376. *If the schedule has a local tail from t to k that ends with with an issued write buffer step*

$$\mathcal{L}^W(t,k),$$

then there is no write-write race between $l' \in [t:k)$ and k in the low-level machine

$$\neg WW_{\downarrow}(l',k).$$

Proof. By assumption, step k is a write buffer step made by unit i in strong memory mode

$$s(k) \in \Sigma_{WB,i} \wedge SC_{\downarrow}(k)$$

and has been issued before t

$$hd(issue^k_{\downarrow}(i)) < t,$$

and by the monotonicity of the sequence of issued writes, was already issued at $l' \in [t:k)$

$$hd(issue^k_{\downarrow}(i)) \in issue^{l'}_{\downarrow}(i).$$

Assume now for the sake of contradiction that there is a write-write race between step l' and k, and thus an intersection between the outputs of step l' and step k

$$out_{\downarrow}(l') \dot{\cap} out_{\downarrow}(k)$$

and thus by definition with the inclusive device closure of the domain of the executed write in step k

$$out_{\downarrow}(l') \dot{\cap} idc(Dom(W_{\downarrow}(k))).$$

By Lemma 21 the outputs are closed, and with Lemma 15 we conclude that there is an intersection between the outputs at l' and the domain of the write at k

$$out_{\downarrow}(l') \dot{\cap} Dom(W_{\downarrow}(k)).$$

Step k uses low-level semantics

$$LL_{\downarrow}(k)$$

and thus the executed write is the head of the write buffer

$$W_{\downarrow}(k) = hd(wb^k_{\downarrow}(i)),$$

which by Lemma 123 is the write issued at the timestamp that is the head of the write buffer

$$hd(wb^k_{\downarrow}(i)) = BW_{\downarrow}(hd(issue^k_{\downarrow}(i))).$$

By Lemma 127 the outputs of step l' have a hit in the write buffer of unit i at l'

$$hit(out_{\downarrow}(l'), wb_{\downarrow}^{l'}(i)).$$

By Lemma 322, the schedule is l'-abstract

$$s \in \mathrm{ABS}_{l'}$$

and by Lemma 267 the write buffers at l' are the same in the two machines

$$wb_{\downarrow}^{l'} = wb_{\uparrow}^{l'},$$

and we conclude that there is also a hit in the high-level machine

$$hit(out_{\downarrow}(l'), wb_{\uparrow}^{l'}(i)).$$

Since the schedule has a local tail from t to k, we obtain that the schedule is valid until k

$$\Gamma_{\downarrow}^{k}(s),$$

and with Lemmas 320 and 321 we obtain that it is also k-ordered and IPI-valid until $k-1$

$$s \in \mathrm{ORD}_k \wedge \Delta_{IPI\downarrow}^{k-1}(s).$$

Since $l'+1 \leq k$ we obtain that the schedule is also $l'+1$-ordered, valid until l', and IPI-valid until $l'-1$

$$s \in \mathrm{ORD}_{l'+1} \wedge \Gamma_{\downarrow}^{l'}(s) \wedge \Delta_{IPI\downarrow}^{l'-1}(s),$$

and by Lemma 305 also $l'+1$-abstract

$$s \in \mathrm{ABS}_{l'+1}.$$

Since l' is in the local tail, it is local

$$L_{\downarrow}(l')$$

and by contraposition of Lemma 136 not a write buffer step, in particular not of unit i

$$s(l') \notin \Sigma_{WB,i}.$$

By Lemma 322, the schedule is k-abstract

$$s \in \mathrm{ABS}_k$$

and by Lemma 268 has the same mode in both machines

$$SC_{i\uparrow}(k) = SC_{i\downarrow}(k) = 1.$$

By Lemma 327, unit i has the same mode at l' as at k in the high-level machine

$$SC_{i\uparrow}(l') = SC_{i\uparrow}(k) = 1.$$

By Lemma 286 with $k := l'$, there is no hit of the outputs of step l' in the write buffer

$$\neg hit(out_{\downarrow}(l'), wb_{\uparrow}^{l'}(i)),$$

which is a contradiction. □

4.12.5 Combining the Results

We now combine the results of the previous two subsections. We first combine the two cases in one definition. We say that schedule *s has a local tail from t to k with a global end* if it has an independent end or ends with an issued write buffer step, i.e., there is $G \in \{ I, W \}$ such that

$$\mathcal{L}^G(t,k).$$

Clearly in each case the schedule actually has a local tail from t to k.

Lemma 377.

$$\mathcal{L}^G(t,k) \to \mathcal{L}(t,k)$$

Lemma 378. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

None of the steps in the local segment are made by its victims

$$\forall t' \in [t:k).u_{\downarrow}(t') \notin victims_{\downarrow}(k).$$

Proof. By case distinction on G.

$G = I$**:** Since the schedule has a local tail with an independent end, all of the steps $t' \in [t:k)$ are unit-concurrent

$$ucon_{\downarrow}(k,t')$$

and thus not interrupted

$$\neg int_{\downarrow}(k,t')$$

and thus not made by victims of step k

$$u_{\downarrow}(t') \notin victims_{\downarrow}(k).$$

$G = W$**:** The claim follows directly by Lemma 375.

□

Lemma 379. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then the configurations at k and t (in their respective schedule) strongly agree

$$c^k_{\downarrow} =^{s(k)}_{\downarrow} c_{\downarrow}[t \leftarrow k]^t.$$

Proof. By case distinction on G.

$G = I$**:** The claim is Lemma 363

$G = W$**:** The claim follows directly by Lemma 370.

□

Lemma 380. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then for each $l' \in [t:k)$, the configurations at $l'+1$ and l' (in their respective schedule) strongly agree

$$c_\uparrow[t \leftarrow k]^{l'+1} =_\uparrow^{s[t\leftarrow k](l'+1)} c_\uparrow^{l'}.$$

Proof. By case distinction on G.

$G = I$**:** The claim is Lemma 369.

$G = W$**:** The claim follows directly by Lemma 374.

□

Lemma 381. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then step t is still IPI-valid after moving k to t

$$\Delta_{IPI\downarrow}[t \leftarrow k](t).$$

Proof. By Lemma 379 we obtain that step k can be executed at t

$$c_\downarrow[t \leftarrow k]^t =_\downarrow^{s[t\leftarrow k](t)} = c_\downarrow^k,$$

and by Lemma 150 its victims are unchanged

$$victims_\downarrow[t \leftarrow k](t) = victims_\downarrow(k).$$

By Lemma 378 the steps $t' \in [t:k)$ are not made by victims

$$u_\downarrow(t') \notin victims_\downarrow(k).$$

Let unit i now be a victim of step k

$$i \in victims_\downarrow(k)$$

and we immediately observe that all steps $t' \in [t:k)$ are not made by unit i

$$u_\downarrow(t') \neq i.$$

By repeated application of Lemma 96 we conclude that none of those steps change the write buffer of unit i

$$wb_\downarrow^t =_i \ldots =_i wb_\downarrow^k.$$

Since the configuration at t is unaffected by moving k to t, the write buffers of victims of step k are thus the same as at k

$$\forall i \in victims_\downarrow(k).wb_\downarrow[t \leftarrow k]^t =_i wb_\downarrow^t =_i wb_\downarrow^k$$

and the claim follows

$$\Delta_{IPI\downarrow}[t \leftarrow k](t) \equiv \bigwedge_{i \in victims_\downarrow[t\leftarrow k](t)} wb_\downarrow[t \leftarrow k]^t(i) = \varepsilon$$

$$\equiv \bigwedge_{i \in victims_{\downarrow}(k)} wb_{\downarrow}[t \leftarrow k]^t(i) = \varepsilon$$
$$\equiv \bigwedge_{i \in victims_{\downarrow}(k)} wb^k_{\downarrow}(i) = \varepsilon$$
$$\equiv \Delta_{IPI\downarrow}(k).$$

□

Lemma 382. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then step t is ordered after moving k to t

$$ord[t \leftarrow k](t).$$

Proof. We have to show that if the buffer of some unit j is dirty and the step is global, it is made by the write buffer of unit j

$$dirty_{\downarrow}[t \leftarrow k](t,j) \wedge G_{\downarrow}[t \leftarrow k](t) \xrightarrow{!} s[t \leftarrow k](t) \in \Sigma_{WB,j}.$$

We prove this by case analysis on G.

$G = I$**:** Since the schedule had a local tail that ends with an independent step from t to k, the configuration at t was clean

$$clean_{\downarrow}(t).$$

The configuration at t is still clean after the reordering

$$clean_{\downarrow}[t \leftarrow k](t),$$

and the claim follows with Lemma 273

$$ord[t \leftarrow k](t).$$

$G = W$**:** Assume now hat the buffer was dirty

$$dirty_{\downarrow}[t \leftarrow k](t,j).$$

We conclude it was dirty in the original schedule

$$dirty_{\downarrow}(t,j),$$

and since the schedule had a local tail from t to k that ends with an issued write buffer step, we obtain first that step k was made by the write buffer of some unit i

$$s(k) \in \Sigma_{WB,i}$$

and second that that is the unit with the dirty buffer

$$j = i.$$

The claim follows with the observation that step k is now stepped at t

$$s[t \leftarrow k](t) = s(k) \in \Sigma_{WB,j}.$$

□

Lemma 383. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then all of the following are true

1. *the schedule where k is moved to t is valid until t*

$$\Gamma^t_\downarrow(s[t \leftarrow k])$$

2. *the schedule where k is moved to t is IPI-valid until t*

$$\Delta^t_{IPI\downarrow}(s[t \leftarrow k])$$

3. *the schedule where k is moved to t is $t+1$-ordered*

$$s[t \leftarrow k] \in \mathrm{ORD}_{t+1}.$$

Proof. Since the schedules are the same before t, we obtain

1. the schedule where k is moved to t is valid until $t-1$

$$\Gamma^{t-1}_\downarrow(s[t \leftarrow k])$$

2. the schedule where k is moved to t is IPI-valid until $t-1$

$$\Delta^{t-1}_{IPI\downarrow}(s[t \leftarrow k])$$

3. the schedule where k is moved to t is t-ordered

$$s[t \leftarrow k] \in \mathrm{ORD}_t.$$

For step t, we use for validity Lemmas 379 to obtain that the steps strongly agree

$$c^k_\downarrow =^{s(k)}_\downarrow c_\downarrow[t \leftarrow k]^t.$$

and by Lemma 150 that the step is still valid

$$\Gamma_\downarrow[t \leftarrow k](t) = \Gamma_\downarrow(k) = 1.$$

The first claim follows

$$\Gamma^t_\downarrow(s[t \leftarrow k]).$$

IPI-validity of step t is solved by Lemma 381

$$\Delta_{IPI\downarrow}[t \leftarrow k](t).$$

The second claim follows

$$\Delta^t_{IPI\downarrow}(s[t \leftarrow k]).$$

By Lemma 382, step t is ordered

$$ord_\downarrow[t \leftarrow k](t).$$

The third claim follows

$$s[t \leftarrow k] \in \mathrm{ORD}_{t+1}.$$

□

The schedule after the move is k+1-ordered, and each step can be executed at its new position.

Lemma 384. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then all of the following are true for each $l' \in [t:k]$.

1. *the schedule where k is moved to t is valid until k*

$$\Gamma^k_\downarrow(s[t \leftarrow k])$$

2. *if l' is after t, then step $l'-1$ can be executed at l'*

$$l' > t \rightarrow c_\downarrow[t \leftarrow k]^{l'} =^{s[t\leftarrow k](l')}_\downarrow c^{l'-1}_\downarrow,$$

3. *the schedule where k is moved to t is IPI-valid until l'*

$$\Delta^{k'}_{IPI\downarrow}(s[t \leftarrow k])$$

4. *the schedule where k is moved to t is l'+1-ordered*

$$s[t \leftarrow k] \in \mathrm{ORD}_{l'+1}.$$

Proof. The proof is by induction on l' starting at t.

The base case is Lemma 383.

In the inductive step $l' \rightarrow l'+1$, we have that the schedule is l'+1-ordered, valid and IPI-valid until l' for all $l' \in [t:k]$

$$\Gamma^{l'}_\downarrow(s[t \leftarrow k]) \wedge \Delta^{l'}_{IPI\downarrow}(s[t \leftarrow k]) \wedge s[t \leftarrow k] \in \mathrm{ORD}_{l'+1}.$$

By Lemma 305 it is l'+1-abstract

$$s[t \leftarrow k] \in \mathrm{ABS}_{l'+1}.$$

By Lemma 380 step l' can be executed at $l'+1$

$$c_\uparrow[t \leftarrow k]^{l'+1} =^{s[t\leftarrow k](l'+1)}_\uparrow c^{l'}_\uparrow.$$

By Lemma 377 the schedule has a local tail from t to k

$$\mathcal{L}(t,k).$$

By Lemma 322, schedule s is k-abstract

$$s \in \mathrm{ABS}_k$$

and by Lemma 259 we obtain that the low-level machine at l' strongly agrees with the high-level machine at $l'+1$ in the reordered schedule

$$c_\uparrow[t \leftarrow k]^{l'+1} =^{s[t\leftarrow k](l'+1)}_\uparrow c^{l'}_\downarrow.$$

Because step l' is in the local tail with an independent end, it is local

$$L_{\downarrow}(l')$$

and thus not a shared read

$$\neg ShR_{\downarrow}(l').$$

With Lemmas 144 and 269 we obtain that step $l'+1$ in the low-level machine is not a shared read either

$$\begin{aligned} ShR_{\downarrow}[t \leftarrow k](l'+1) &= ShR_{\uparrow}[t \leftarrow k](l'+1) && \text{L 269} \\ &= ShR_{\downarrow}(l') && \text{L 144} \end{aligned}$$

Thus step $l'+1$ is semi-ordered

$$sord_{\downarrow}[t \leftarrow k](l'+1)$$

and in fact the schedule is $l'+2$-semi-ordered

$$s[t \leftarrow k] \in \mathrm{ORD}^{-}_{l'+2}.$$

With the same lemmas we transfer feasibility

$$\begin{aligned} \Phi_{\downarrow}[t \leftarrow k](l'+1) &= \Phi_{\uparrow}[t \leftarrow k](l'+1) && \text{L 269} \\ &= \Phi_{\downarrow}(l') && \text{L 144} \end{aligned}$$

Thus the schedule is semi-valid until $l'+1$

$$\Gamma\Phi^{l'+1}_{\downarrow}(s[t \leftarrow k]).$$

and by Lemma 299 and Lemma 144 valid at $l'+1$

$$\begin{aligned} \Gamma_{\downarrow}[t \leftarrow k](l'+1) &= \Gamma_{\uparrow}[t \leftarrow k](l'+1) && \text{L 299} \\ &= \Gamma_{\downarrow}(l') && \text{L 144} \end{aligned}$$

The schedule is thus valid until $l'+1$

$$\Gamma^{l'+1}_{\downarrow}(s[t \leftarrow k]),$$

which is the first claim.

By Lemma 305 the schedule is $l'+2$-abstract

$$s[t \leftarrow k] \in \mathrm{ABS}_{l'+2}.$$

Therefore by Lemma 260 in the low-level machines step l' can be executed at $l'+1$

$$c_{\downarrow}[t \leftarrow k]^{l'+1} =^{s[t \leftarrow k](l'+1)}_{\downarrow} c^{l'}_{\downarrow},$$

which is the second claim, and by Lemma 150 step $l'+1$ is also local

$$L_{\downarrow}[t \leftarrow k](l'+1) = L_{\downarrow}(l') = 1.$$

By Lemma 135 step $l'+1$ is IPI-valid

$$\Delta_{IPI\downarrow}[t \leftarrow k](l'+1)$$

and thus the schedule is IPI-valid until $l'+1$

$$\Delta^{l'+1}_{IPI\downarrow}(s[t \leftarrow k]),$$

which is the third claim.

Furthermore, step $l'+1$ is local and thus ordered

$$ord_{\downarrow}[t \leftarrow k](l'+1),$$

and therefore the schedule is $l'+2$-ordered

$$s[t \leftarrow k] \in \mathrm{ORD}_{l'+2},$$

which is the fourth claim. □

Lemma 385. *If all steps $t' \in [t:k)$ are made by a different unit that step k*

$$\forall t' \in [t:k).diffu(k,t'),$$

and step k can be moved to $t \leq k$ and k can be executed at its new position

$$c_M[t \leftarrow k]^t =^{s[t\leftarrow k](t)}_M c^k_M$$

and all other steps can be executed at their new position

$$\forall t' \in [t:k).c_M[t \leftarrow k]^{t'+1} =^{s[t\leftarrow k](t'+1)}_M c^{t'}_M$$

and there is no write-write race between any of the other steps and step k

$$\forall t' \in [t:k).\neg WW_M(t',k),$$

then the configuration at $k+1$ is unchanged by the move

$$c^{k+1}_M = c_M[t \leftarrow k]^{k+1}.$$

Proof. The steps are indeed made by units other than unit $u_M(k)$

$$\forall t' \in [t:k).u_M(k) \neq u_M(t'),$$

and by repeated application of Lemma 96 we obtain that the buffers of the unit making step k are the same at t and at k

$$wb^t_M =_{u_M(k)} \ldots =_{u_M(k)} wb^k_M.$$

The buffers at t are unchanged by the reordering

$$wb_M[t \leftarrow k]^t = wb^t_M =_{u_M(k)} wb^k_M.$$

Since step k can be executed at t, by Lemma 150 it buffers the same write and executes the same operation, and thus the buffers after that step are also the same

$$\begin{aligned}
&wb_M[t \leftarrow k]^{t+1}(u_M(k)) \\
&= Op_{u_M(k)M}[t \leftarrow k](t)(wb_M[t \leftarrow k]^t(u_M(k)), BW_M[t \leftarrow k](t)) \\
&= Op_{u_M(k)M}(k)(wb_M[t \leftarrow k]^t(u_M(k)), BW_M(k)) && \text{L 150}
\end{aligned}$$

$$= Op_{u_M(k)M}(k)(wb_M^k(u_M(k)), BW_M(k))$$
$$= wb_M^{k+1}(u_M(k)).$$

The steps in the reordered schedule are still made by units other than $u_M(k)$

$$\forall t' \in [t:k).u_M(k) \neq u_M[t \leftarrow k](t'+1),$$

and by repeated application of Lemma 96 we obtain that the buffer of the unit making step k is not changed any further

$$wb_M[t \leftarrow k]^{t+1}(u_M(k)) =_{u_M(k)} \ldots =_{u_M(k)} wb_M[t \leftarrow k]^{k+1}(u_M(k)).$$

We consider now the sequence of writes from t to k, which form a virtual write buffer[8] wb

$$wb = W_M(t) \circ \ldots \circ W_M(k-1).$$

Note that we can treat this sequence of writes like a write buffer.

Since there is no write-write race with any of those steps

$$\neg WW_M(t',k),$$

the outputs of the steps do not intersect

$$out_M(t') \not\cap out_M(k)$$

and thus the inclusive device closure of the domain of the write at t' does not intersect with the outputs at k

$$idc(Dom(W_M(t'))) \not\cap out_M(k).$$

The outputs are closed by Lemma 21, and by Lemma 15 we can drop the inclusive device closure

$$Dom(W_M(t')) \not\cap out_M(k).$$

We conclude for every t' that the write of t' is not the hit of the outputs at k in the sequence of writes wb

$$W_M(t') \neq hit(out_M(k), wb)$$

and thus the hit can not be an element of the sequence of writes wb, which is made up exactly of those writes

$$hit(out_M(k), wb) \notin wb.$$

By contraposition of Lemma 47, there is no hit in wb for the outputs

$$\neg hit(out_M(k), wb),$$

and thus not for the inclusive device closure of the domain of the write at k

$$\neg hit(idc(Dom(W_M(k))), wb).$$

Clearly the memory at $k+1$ is obtained by applying the writes wb (using the overloaded definition of $\circledast$ from page 46) followed by the write at k to the memory at t

$$m_M^{k+1} = m_M^k \circledast W_M(k)$$

[8] Note that write buffers are sequences of writes on memory region BA whereas this is a sequence of writes on memory region $\bigcup_i ACC_i$. Still, all functions etc. we use are well-defined also for such sequences of writes.

$$
\begin{aligned}
&= \ldots \\
&= m_M^t \circledast W_M(t) \circledast \ldots \circledast W_M(k-1) \circledast W_M(k) \\
&= m_M^t \circledast (W_M(t) \circ \ldots \circ W_M(k-1)) \circledast W_M(k) \\
&= m_M^t \circledast wb \circledast W_M(k)
\end{aligned}
$$

and by Lemma 55 the writes wb and the write at k can be swapped

$$= m_M^t \circledast W_M(k) \circledast wb$$

and using the same arguments as before, we obtain that this is exactly the configuration at t after the reordering, updated by the write at t after the reordering

$$
\begin{aligned}
&= m_M[t \leftarrow k]^t \circledast W_M[t \leftarrow k](t) \circledast wb \\
&= m_M[t \leftarrow k]^{t+1} \circledast wb \\
&= m_M[t \leftarrow k]^{t+1} \circledast (W_M(t) \circ \ldots \circ W_M(k-1))
\end{aligned}
$$

and by Lemma 150 we obtain that all those writes are the writes executed one step later in the reordered schedule

$$
\begin{aligned}
&= m_M[t \leftarrow k]^{t+1} \circledast (W_M[t \leftarrow k](t+1) \circ \ldots \circ W_M[t \leftarrow k](k)) \\
&= m_M[t \leftarrow k]^{t+1} \circledast W_M[t \leftarrow k](t+1) \circledast \ldots \circledast W_M[t \leftarrow k](k) \\
&= m_M[t \leftarrow k]^{k+1}.
\end{aligned}
$$

It now remains to show that the buffers of the other units are the same after the reordering. By Lemma 190 we obtain that the configurations at t before and $t+1$ after the reordering are nearly the same

$$\mathcal{I}_M^t[t \leftarrow k]([t \rightarrow k], t+1, t),$$

and by repeatedly applying Lemma 197 we can push this until k

$$\mathcal{I}_M^t[t \leftarrow k]([t \rightarrow k], k+1, k).$$

We simply apply Lemma 199 and obtain that the configurations at $k+1$ are nearly the same

$$\mathcal{I}_M^t[t \leftarrow k][t \rightarrow k]([t \leftarrow k], k+1, k+1).$$

We observe that moving k to t and then t to k is a no-op

$$\mathcal{I}_M^t([t \leftarrow k], k+1, k+1).$$

We immediately obtain the claim

$$\forall i \neq u_M(k).wb_M^{k+1} =_i wb_M[t \leftarrow k]^{k+1}.$$

□

Lemma 386. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then there is no write-write race between $l \in [t:k)$ and k in the low-level machine

$$\neg WW_{\downarrow}(l,k).$$

Proof. By case distinction on G.

$G = I$**:** The claim is Lemma 365.

$G = W$**:** The claim is Lemma 376.

□

We can now easily show that the schedules are equivalent.

Lemma 387. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

then the configurations at $k+1$ are the same and the schedules are equivalent

$$c_\downarrow[t \leftarrow k]^{k+1} = c_\downarrow^{k+1} \wedge s \equiv_\downarrow s[t \leftarrow k].$$

Proof. By Lemma 379 step k can be moved to t

$$c_\downarrow^k =_\downarrow^{s(k)} c_\downarrow[t \leftarrow k]^t.$$

By Lemma 384, for all $t' \in (t : k]$, $t' - 1$ can be moved to t'

$$c_\downarrow[t \leftarrow k]^{t'} =_\downarrow^{s[t \leftarrow k](t')} c_\downarrow^{t'-1}.$$

By an index shift by one we obtain that all $t' \in [t : k)$ can be moved to $t' + 1$

$$c_\downarrow[t \leftarrow k]^{t'+1} =_\downarrow^{s[t \leftarrow k](t'+1)} c_\downarrow^{t'}.$$

By Lemma 386, there is no write-write race between k and any of those t'

$$\neg WW_\downarrow(t', k).$$

By Lemma 385, the final configuration at $k+1$ is the same in both schedules

$$c_\downarrow[t \leftarrow k]^{k+1} = c_\downarrow^{k+1}.$$

The claim follows with Lemma 118. □

Lemma 388. *If the schedule has a local tail from t to k with a global end*

$$\mathcal{L}^G(t,k),$$

and was valid and IPI-valid everywhere

$$\forall t'.\Delta_{IPI\downarrow}(t') \wedge \Gamma_\downarrow(t'),$$

the reordered schedule also is

$$\forall t'.\Delta_{IPI\downarrow}[t \leftarrow k](t') \wedge \Gamma_\downarrow[t \leftarrow k](t'),$$

Proof. For the steps before $k+1$, the claim is Lemma 384.

For the steps at and after $k+1$, we have by Lemma 387 that the configurations at $k+1$ are the same

$$c_{\downarrow}[t \leftarrow k]^{k+1} = c_{\downarrow}^{k+1}$$

and that the schedules starting from $k+1$ are the same

$$s[k+1:\infty] = s[t \leftarrow k][k+1:\infty],$$

and by straightforward induction all configurations $t' \geq k+1$ are the same

$$c_{\downarrow}[t \leftarrow k]^{t'} = c_{\downarrow}^{t'}.$$

Therefore all steps $t' \geq k+1$ are the same

$$X_{\downarrow}[t \leftarrow k](t') = X_{\downarrow}(c_{\downarrow}[t \leftarrow k]^{t'}, s[t \leftarrow k](t')) = X_{\downarrow}(c_{\downarrow}^{t'}, s(t')) = X_{\downarrow}(t')$$

and the claim follows. □

We wish to show that our reordering only reorders when there is a local tail from t to k_t with a global end. For this to work, we need to add as an invariant that whenever we allow a shared write to enter the buffer, it is committed at the next global step. Again, the next global step is only known to exist if the schedule is balanced and valid. In order to prove that there is a local tail, we also need IPI-validity. Since the hardware (hopefully) provides such a schedule, we say that s is a *hardware schedule* when those properties hold

$$s \in HW \iff s \in bal \wedge \forall t'.\Gamma_{\downarrow}[s](t') \wedge \Delta_{IPI\downarrow}[s](t').$$

Since hardware schedules are balanced, we can always use the definitions for constructing an ordered schedule from page 304, i.e., the next global or pushable step g_t and the next step k_t to be moved to the front.

We say that the schedule in iteration t is *ready* and write $\mathcal{R}[s](t)$ when each dirty buffer buffer contains only a single write, which is the write committed at the next global step

$$\mathcal{R}[s](t) \equiv \forall i.dirty_{\downarrow}O_t(t,i) \rightarrow sO_t(g_t) \in \Sigma_{WB,i} \wedge issue_{\downarrow}O_t^t(i) = hd(issue_{\downarrow}O_t^{g_t}(i)).$$

Since the sets oracle inputs are disjoint, this immediately implies that there is at most one dirty buffer: the one of the unit that will also commit it next. This ensures that we can always commit that write (i.e., move it to the front) without causing conflicts.

We show that the first schedule is ready.

Lemma 389. *The schedule in iteration* 0 *is ready*

$$\mathcal{R}(0).$$

Proof. Assume that some buffer is dirty

$$dirty_{\downarrow}O_0(0,i)$$

and thus has a write issued from t'

$$t' \in issue_{\downarrow}O_0(0).$$

This contradicts the fact that the sequence of issued writes is empty at 0

$$issue_{\downarrow} O_0(0) = \varepsilon.$$

□

We never move a step across the next global or pushable step.

Lemma 390.

$$sO_t \in HW \rightarrow k_t \leq g_t.$$

Proof. We distinguish whether step g_t is a write buffer step in strong memory mode or not.

$SC_{\downarrow} O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i}$**:** By definition k_t is the first step made by processor i before the head of the buffer at g_t, or it is equal to g_t

$$k_t = \min \left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow} O_t^{g_t}(i)) \vee k = g_t \right\}.$$

Obviously g_t is in that set

$$g_t \in \left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow} O_t^{g_t}(i)) \vee k = g_t \right\},$$

and the claim follows

$$k_t = \min \left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow} O_t^{g_t}(i)) \vee k = g_t \right\} \leq g_t.$$

$\neg(SC_{\downarrow} O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i})$**:** In this case we take k_t to be the first step that is not unit-concurrent

$$k_t = \min \left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow} O_t(g_t, k) \right\}.$$

The unit making step g_t is obviously the same as the unit making step g_t

$$u_{\downarrow} O_t(g_t) = u_{\downarrow} O_t(g_t),$$

from which we conclude that step g_t is not unit-concurrent with itself

$$\neg ucon_{\downarrow} O_t(g_t, g_t).$$

Thus g_t is in that set

$$g_t \in \left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow} O_t(g_t, k) \right\}$$

and the claim follows

$$k_t = \min \left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow} O_t(g_t, k) \right\} \leq g_t.$$

□

If k_t is global, it is the next global step.

Lemma 391.

$$sO_t \in HW \wedge G_{\downarrow} O_t(k_t) \rightarrow k_t = g_t.$$

Proof. By definition g_t is the first global or pushable step

$$g_t = \min \left\{ g \geq t \;\middle|\; G_{\downarrow} O_t(g) \vee p_{\downarrow} O_t(g) \right\}.$$

By Lemma 390, the step is before or equal to g_t

$$k_t \leq g_t,$$

and since k_t is by assumption global, it can not be earlier than g_t

$$k_t \geq g_t$$

and the claim follows

$$k_t = g_t.$$

□

When k_t is local, steps before k_t are made by a different unit than k_t.

Lemma 392.

$$sO_t \in HW \wedge L_{\downarrow} O_t(k_t) \rightarrow \forall t' \in [t : k_t).diffuO_t(k_t, t').$$

Proof. Assume for the sake of contradiction that step $t' \in [t : k_t)$ is made by the same unit

$$u_{\downarrow} O_t(t') = u_{\downarrow} O_t(k_t).$$

We distinguish between the cases in the definition of k_t.

$SC_{\downarrow} O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i}$**:** In this case k_t is the first step of a processor before or equal to the head of the sequence of issued writes, or equal to g_t

$$k_t = \min \left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow} O_t^{g_t}(i)) \vee k = g_t \right\},$$

and since write buffer step g_t is by Lemma 136 global

$$G_{\downarrow} O_t(g_t)$$

and k_t is not, we conclude that they are not the same

$$k_t \neq g_t$$

and thus k_t is the first processor step of unit i

$$sO_t(k_t) \in \Sigma_{P,i} \wedge \forall t'' \in [t : k_t).sO_t(t'') \notin \Sigma_{P,i}.$$

In particular, step t' is not made by processor i

$$sO_t(t') \notin \Sigma_{P,i}.$$

Since t' is made by the same unit as k_t, it must be made by the write buffer of unit i

$$sO_t(t') \in \Sigma_{WB,i}$$

and thus by Lemma 136 it must be a global step

$$G_{\downarrow} O_t(t'),$$

and thus must be after or equal to step g_t

$$t' \geq g_t.$$

Since it is before step k_t, step k_t must also be before g_t

$$k_t > t' \geq g_t,$$

but this contradicts Lemma 390.

$\neg(SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i})$**:** By definition, step k_t is the first step that is not unit concurrent with step g_t

$$k_t = \min\left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow}O_t(g_t,k) \right\}.$$

It is thus not unit-concurrent with step g_t, whereas t' is

$$\neg ucon_{\downarrow}O_t(g_t,k_t) \wedge ucon_{\downarrow}O_t(g_t,t').$$

Unfolding the definitions and using the fact that both steps are made by the same unit yields a contradiction

$$\begin{aligned}
\neg ucon_{\downarrow}O_t(g_t,k_t) &\equiv \neg(diffuO_t(g_t,k_t) \wedge \neg int_{\downarrow}O_t(g_t,k_t)) \\
&\equiv \neg(u_{\downarrow}O_t(g_t) \neq u_{\downarrow}O_t(k_t) \wedge \neg(u_{\downarrow}O_t(k_t) \in victims_{\downarrow}O_t(g_t)) \\
&\equiv \neg(u_{\downarrow}O_t(g_t) \neq u_{\downarrow}O_t(t') \wedge \neg(u_{\downarrow}O_t(t') \in victims_{\downarrow}O_t(g_t)) \\
&\equiv diffuO_t(g_t,t') \wedge \neg int_{\downarrow}O_t(g_t,t') \\
&\equiv \neg ucon_{\downarrow}O_t(g_t,t').
\end{aligned}$$

□

Lemma 393. *If the schedule is a hardware schedule and t-ordered*

$$sO_t \in HW \cap \mathrm{ORD}_t,$$

it has a local tail until g_t and until k_t

$$\mathcal{L}O_t(t,k_t) \wedge \mathcal{L}O_t(t,g_t).$$

Proof. By definition g_t is the first global or pushable step

$$g_t = \min\left\{ g \geq t \;\middle|\; G_{\downarrow}O_t(g) \right\} \vee p_{\downarrow}O_t(g).$$

Thus all steps between t and g_t are local

$$\forall t' \in [t : g_t).L_{\downarrow}O_t(t').$$

Since all steps are valid and IPI-valid, so are all steps until g_t resp. t

$$\Gamma_{\downarrow}^{g_t}(sO_t) \wedge \Delta_{IPI\downarrow}^{t}(sO_t),$$

and thus the schedule has a local tail until g_t

$$\mathcal{L}O_t(t,g_t).$$

By Lemma 390, k_t is before or equal to g_t

$$k_t \leq g_t$$

and thus all steps before k_t are also local

$$\forall t' \in [t : k_t).L_{\downarrow}O_t(t')$$

and all steps until k_t are valid

$$\Gamma_{\downarrow}^{k_t}(sO_t).$$

It follows that the schedule also has a local tail until k_t

$$\mathcal{L}O_t(t,k_t).$$

□

Lemma 394. *When the schedule is a hardware schedule*

$$sO_t \in HW$$

and the schedule is ready and has a local tail until g_t

$$\mathcal{R}(t) \wedge \mathcal{L}O_t(t,g_t),$$

then if the buffer of some unit is dirty, the next step is made by the write buffer of that unit and in strong memory mode.

$$dirty_{\downarrow}O_t(t,i) \rightarrow SC_{\downarrow}O_t(k_t) \wedge sO_t(k_t) \in \Sigma_{WB,i}.$$

Proof. Since the write buffer is dirty, it is in strong memory mode and has a shared write in the buffer from t'

$$SC_{i\downarrow}O_t(t) \wedge t' \in issue_{\downarrow}O_t^t(i) \wedge Sh_{\downarrow}O_t(t').$$

Since the schedule is ready, the next global or pushable step is made by a write buffer and the write buffer at t only contains the write to be committed

$$sO_t(g_t) \in \Sigma_{WB,i} \wedge issue_{\downarrow}O_t^t(i) = hd(issue_{\downarrow}O_t^{g_t}(i)).$$

Thus the write t' must be that write, which by the monotonicity of write buffers was issued before t

$$t' = hd(issue_{\downarrow}O_t^{g_t}(i)) < t.$$

By Lemma 328 with $k := g_t$ we obtain that the step uses the same memory mode and is thus in strong memory mode

$$SC_{\downarrow}O_t(g_t) = SC_{i\downarrow}O_t(g_t) = SC_{i\downarrow}O_t(t) = 1.$$

Thus g_t is a write buffer step in strong memory mode

$$SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i},$$

and by definition k_t is the first step of a processor before or equal to the head of the sequence of issued writes, or equal to g_t

$$k_t = \min\left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow}O_t^{g_t}(i)) \vee k = g_t \right\},$$

and since the head of the sequence of issued writes is before t, there are no such processor steps and k_t is g_t

$$k_t = g_t.$$

The claim immediately follows. □

Lemma 395. *When the schedule is a hardware schedule*

$$sO_t \in HW$$

and the schedule in iteration t is ready and has a local tail until k_t and g_t, then it has a local tail with a global end at k_t.

$$\mathcal{R}(t) \wedge \mathcal{LO}_t(t,k_t) \wedge \mathcal{LO}_t(t,g_t) \rightarrow \exists G \in \{W,I\}.\mathcal{L}^G O_t(t,k_t).$$

Proof. We distinguish whether k_t is a write buffer step in strong memory mode or not.

$SC_{\downarrow}O_t(k_t) \wedge sO_t(k_t) \in \Sigma_{WB,i}$**:** In this case we choose $G := W$, and show the four claims in the definition of $\mathcal{L}^W$.

By assumption the schedule has a local tail until k_t and step k_t is a write buffer step in strong memory mode, which solves the first two claims.

By Lemma 136, the step is global

$$G_{\downarrow}O_t(k_t)$$

and thus by Lemma 391 it is the next global step

$$k_t = g_t.$$

By definition k_t is the first step made by processor i at or before the head of the buffer at g_t, or it is equal to g_t

$$k_t = \min\left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow}O_t^{g_t}(i)) \vee k = g_t \right\},$$

and we conclude that there are no steps before g_t made by processor i at or before the head of the buffer at g_t

$$\nexists k \in [t : g_t).sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow}O_t^{g_t}(i)).$$

By the monotonicity of the sequence of issued writes, all writes, including the head element, entered the buffer before g_t

$$hd(issue_{\downarrow}O_t^{g_t}(i)) < g_t,$$

and since only processor steps issue writes we know that the head of the write buffer was issued by a processor step

$$sO_t(hd(issue_{\downarrow}O_t^{g_t}(i))) \in \Sigma_{P,i}.$$

We conclude that the head of the write buffer was not issued between t and g_t

$$hd(issue_{\downarrow}O_t^{g_t}(i)) \notin [t : g_t),$$

and since it was not issued after g_t either, it must have been issued before t

$$hd(issue_{\downarrow}O_t^{g_t}(i)) < t,$$

which solves the third claim.

For the fourth claim, assume that the buffer of some unit j is dirty

$$dirty_{\downarrow} O_t(t, j).$$

Since the schedule is ready, the next global or pushable step is made by the write buffer of unit j

$$sO_t(g_t) \in \Sigma_{WB,j}.$$

Step k_t is the same step, and thus step k_t is made by the write buffer of unit j

$$sO_t(k_t) \in \Sigma_{WB,j}.$$

Since we have already established that step k_t is made by the write buffer of unit i, the claim follows

$$i = j.$$

$\neg(SC_{\downarrow} O_t(k_t) \wedge sO_t(k_t) \in \Sigma_{WB,i})$**:** In this case we choose $G := I$, and show the four claims in the definition of $\mathcal{L}^I$.

The first claim is an assumption.

By contraposition of Lemma 394 there is no dirty buffer at t

$$\forall i. \neg dirty_{\downarrow} O_t(t, i)$$

and thus by definition the configuration is clean

$$clean_{\downarrow} O_t(t),$$

which is the second claim.

The third claim is an assumption.

For the fourth claim, we distinguish whether step k_t is global or local.

$G_{\downarrow} O_t(k_t)$**:** By Lemma 391, step k_t is the next global step

$$k_t = g_t.$$

Since k_t is not a write buffer step in strong memory mode, neither is g_t

$$\neg(SC_{\downarrow} O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i}),$$

and k_t is defined to be the first step that is not unit-concurrent with g_t

$$k_t = \min \left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow} O_t(g_t, k) \right\}.$$

We conclude that all steps before k_t are unit-concurrent with g_t

$$\forall t' \in [t : k_t). ucon_{\downarrow} O_t(g_t, t')$$

and thus also all steps before g_t, which is the claim

$$\forall t' \in [t : g_t). ucon_{\downarrow} O_t(g_t, t').$$

$L_\downarrow O_t(k_t)$**:** We obtain by Lemma 392 that all steps before step k_t are made by different units

$$\forall t' \in [t : k_t).diffuO_t(k_t, t'),$$

and by Lemma 134 that step k_t has no victims

$$victims_\downarrow O_t(k_t) = \emptyset.$$

Let now $t' \in [t : k_t)$ be one of the steps in question. Since step k_t has no victims, the unit making step t' is not among them

$$u_\downarrow O_t(t') \notin victims_\downarrow O_t(k_t)$$

and thus step t' is not interrupted by step k_t

$$\neg int_\downarrow O_t(k_t, t').$$

The claim follows by definition

$$ucon_\downarrow O_t(k_t, t').$$

□

Lemma 396. *When in an $t+1$-abstract schedule that is valid until t*

$$\Gamma^t_\downarrow(s) \wedge s \in \mathrm{ABS}_{t+1}$$

a step makes a buffer dirty, it is a shared buffered write in strong memory mode

$$\neg dirty^t_\downarrow(i) \wedge dirty^{t+1}_\downarrow(i) \rightarrow s(t) \in \Sigma_{P,i} \wedge SC_\downarrow(t) \wedge Sh_\downarrow(t) \wedge BW_\downarrow(t) \neq \emptyset.$$

Proof. By Lemma 271 the schedule is valid in the high-level machine

$$\Gamma^t_\uparrow(s).$$

Since the configuration is dirty at $t+1$, unit i is in strong memory mode and there is a shared write in the buffer

$$SC_{i\downarrow}(t+1) \wedge \exists t' \in issue^{t+1}_\downarrow(i).Sh_\downarrow(t').$$

Clearly the sequence of issued writes is non-empty

$$issue^{t+1}_\downarrow(i) \neq \varepsilon$$

and by Lemma 123 we obtain that the write buffer at $t+1$ is non-empty

$$wb^{t+1}_\downarrow(i) \neq \varepsilon.$$

By Lemma 267 we obtain that the buffer in the high-level machine is also non-empty

$$wb^{t+1}_\uparrow(i) = wb^{t+1}_\downarrow(i) \neq \varepsilon.$$

By contraposition of Condition Switch we obtain that step t did not change the mode in the high-level machine

$$SC_i \notin out_\uparrow(t)$$

and by Lemma 138 the mode is indeed unchanged

$$SC_{i\uparrow}(t) = SC_{i\uparrow}(t+1).$$

Using twice Lemma 268 we obtain that the mode was also unchanged in the low-level machine

$$SC_{i\downarrow}(t) = SC_{i\uparrow}(t) = SC_{i\uparrow}(t+1) = SC_{i\downarrow}(t+1) = 1.$$

Since the buffer however was not dirty at t, we conclude that no shared write was buffered at t

$$\nexists t' \in issue^t_{\downarrow}(i).Sh_{\downarrow}(t').$$

Since such a write is in the buffer at $t+1$ but not in t, we conclude that the buffer has grown and thus the operation at t was a push

$$Op_{i\downarrow}(t) = push$$

and thus step t was made by a processor and is buffering a write

$$s(t) \in \Sigma_{P,i} \wedge BW_{\downarrow}(t) \neq \emptyset.$$

We also conclude that the step was in strong memory mode

$$SC_{\downarrow}(t) = SC_{i\downarrow}(t) = 1.$$

Since step t is the only new element in the buffer, it must be the shared write

$$Sh_{\downarrow}(t),$$

and the claim follows. □

Lemma 397. *When the schedule is a hardware schedule*

$$sO_t \in HW$$

and ready and has a local tail from t to g_t and from t to k_t

$$\mathcal{R}(t) \wedge \mathcal{LO}_t(t, g_t) \wedge \mathcal{LO}_t(t, k_t),$$

and also in iteration $t+1$ from $t+1$ to g_{t+1}

$$\mathcal{LO}_{t+1}(t+1, g_{t+1}),$$

then it is also ready in iteration $t+1$

$$\mathcal{R}(t+1).$$

Proof. Assume that the buffer of some unit i is dirty at $t+1$

$$dirty_{\downarrow}O_{t+1}(t+1, i).$$

Note that step t in this schedule is k_t from the previous schedule

$$sO_{t+1}(t) = sO_t[t \leftarrow k](t) = sO_t(k_t).$$

We distinguish whether step g_t was a write buffer step in strong memory mode or not.

$SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i}$**:** By definition k_t is the first step made by processor i before the head of the buffer at g_t, or it is equal to g_t

$$k_t = \min\left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow}O_t^{g_t}(i)) \vee k = g_t \right\}.$$

We distinguish between those two cases.

$sO_t(k_t) \in \Sigma_{P,i} \wedge k_t \leq hd(issue_{\downarrow}O_t^{g_t}(i))$**:** We obtain that t is not after the head of the sequence of issued writes at g_t

$$t \leq k_t \leq hd(issue_{\downarrow}O_t^{g_t}(i))$$

Since write buffers are monotone, all writes in the buffer at t were issued before t, and thus the head of the sequence of issued writes at g_t can not be among them

$$hd(issue_{\downarrow}O_t^{g_t}(i)) \notin issue_{\downarrow}O_t^t(i).$$

We conclude that the head of the sequence of issued writes at g_t is not the head of the sequence of issued writes at t

$$hd(issue_{\downarrow}O_t^{g_t}(i)) \neq hd(issue_{\downarrow}O_t^t(i)).$$

By contraposition of Lemma 326 we obtain that the sequence of buffered writes at t was empty

$$issue_{\downarrow}O_t^t(i) = \varepsilon,$$

and since the schedule before t is unchanged by the reordering, the sequence is also empty at t in the reordered schedule

$$issue_{\downarrow}O_{t+1}^t(i) = issue_{\downarrow}O_t[t \leftarrow k_t]^t(i) = issue_{\downarrow}O_t^t(i) = \varepsilon.$$

However the list of issued writes at $t+1$ by assumption contains a shared write

$$\exists t' \in issue_{\downarrow}O_{t+1}^{t+1}(i).Sh_{\downarrow}O_{t+1}(t').$$

We conclude that the shared write in the buffers at $t+1$ was added by step t

$$issue_{\downarrow}O_{t+1}^{t+1}(i) = t \wedge Sh_{\downarrow}O_{t+1}(t).$$

By Lemma 326, the head of the write buffer at $t+1$ is the head of the write buffer at g_{t+1}

$$t = hd(issue_{\downarrow}O_{t+1}^{t+1}(i)) = hd(issue_{\downarrow}O_{t+1}^{g_{t+1}}(i)).$$

By Lemma 390, step k_t is before or equal to step g_t

$$k_t \leq g_t.$$

Since step k_t is a processor step and step g_t is a write buffer step, they are not the same

$$k_t \neq g_t,$$

and thus g_t is greater than k_t

$$g_t > k_t$$

and thus not moved and still a write buffer step

$$sO_{t+1}(g_t) = sO_t[t \leftarrow k_t](g_t) = sO_t(g_t) \in \Sigma_{WB,i}$$

and thus by Lemma 136 still global

$$G_{\downarrow}O_{t+1}(g_t).$$

By Lemma 395 the schedule had a local tail with a global end until k_t

$$\mathcal{L}^G O_t(t,k_t)$$

and by Lemma 384 with $l' : t'+1$, all steps $t' \in [t : k_t)$ can be executed at their new position

$$c_{\downarrow}O_t^{t'} =_{\downarrow}^{sO_t[t\leftarrow k](t'+1)} c_{\downarrow}O_t[t \leftarrow k]^{t'+1} = c_{\downarrow}O_{t+1}^{t'+1}$$

and by Lemma 150 are still local

$$L_{\downarrow}O_{t+1}(t'+1) = L_{\downarrow}O_t[t \leftarrow k](t'+1) = L_{\downarrow}O_t(t') = 1.$$

We conclude with an index shift by one that now all steps from $t+1$ before k_t+1 are still the same

$$\forall t' \in [t+1 : k_t+1).L_{\downarrow}O_{t+1}(t').$$

By Lemma 387 the configurations at k_t+1 are the same

$$c_{\downarrow}O_{t+1}^{k_t+1} = c_{\downarrow}O_t[t \leftarrow k]^{k_t+1} = c_{\downarrow}O_t^{k_t+1},$$

and since the steps after k_t are unchanged by the reordering

$$sO_{t+1}[k_t+1 : \infty] = sO_t[t \leftarrow k_t][k_t+1 : \infty] = sO_t[k_t+1 : \infty],$$

the configurations at $t' \in [k_t+1 : g_t)$ are also the same

$$c_{\downarrow}O_{t+1}^{t'} = c_{\downarrow}O_t^{t'}$$

and thus the steps are still local

$$L_{\downarrow}O_{t+1}(t') = L_{\downarrow}(c_{\downarrow}O_{t+1}^{t'}, sO_{t+1}(t')) = L_{\downarrow}(c_{\downarrow}O_t^{t'}, sO_t(t')) = L_{\downarrow}O_t(t') = 1.$$

Combining the two intervals, we obtain that all steps from $t+1$ before g_t are now local

$$\forall t' \in [t+1 : g_t).L_{\downarrow}O_{t+1}(t'),$$

and thus step g_t is still the first global step

$$g_{t+1} = g_t.$$

The claim follows

$$sO_{t+1}(g_{t+1}) = sO_t(g_t) \in \Sigma_{WB,i} \wedge issue_{\downarrow}O_{t+1}^{t+1}(i) = t = hd(issue_{\downarrow}O_{t+1}^{g_{t+1}}(i)).$$

$k_t = g_t$**:** In this case $sO_{t+1}(t)$ is a write buffer step

$$sO_{t+1}(t) = sO_t[t \leftarrow k_t](t) = sO_t(k_t) = sO_t(g_t) \in \Sigma_{WB,i},$$

which performs a pop operation

$$Op_{i\downarrow}O_{t+1}(t) = pop.$$

Thus the buffer at t contains all elements of the buffer at $t+1$, and one additional element

$$\begin{aligned} issue_{\downarrow}O^t_{t+1}(i) &= hd(issue_{\downarrow}O^t_{t+1}(i)) \circ tl(issue_{\downarrow}O^t_{t+1}(i)) \\ &= hd(issue_{\downarrow}O^t_{t+1}(i)) \circ issue_{\downarrow}O^{t+1}_{t+1}(i). \end{aligned}$$

The buffer at t is unchanged by moving k_t to t

$$\begin{aligned} issue_{\downarrow}O^t_t(i) &= issue_{\downarrow}O^t_{t+1}(i) \\ &= hd(issue_{\downarrow}O^t_{t+1}(i)) \circ issue_{\downarrow}O^{t+1}_{t+1}(i). \end{aligned}$$

Let t' be a write that makes the buffer at $t+1$ dirty

$$t' \in issue_{\downarrow}O^{t+1}_{t+1}(i) \wedge Sh_{\downarrow}O_{t+1}(t').$$

This write was therefore already buffered at t in the previous schedule

$$t' \in issue_{\downarrow}O^t_t(i),$$

and by the monotonicity of the sequence of issued writes it was issued before t

$$t' < t$$

and thus the step that issued it is unaffected by moving k_t to t, and thus the step was shared in the previous schedule

$$Sh_{\downarrow}O_t(t') = Sh_{\downarrow}O_t[t \leftarrow k_t](t') = Sh_{\downarrow}O_{t+1}(t').$$

Thus the buffer is dirty at t in the previous schedule

$$dirty_{\downarrow}O_t(t,i),$$

and since the schedule was ready, the buffer contained only one element

$$issue_{\downarrow}O^t_t(i) = hd(issue_{\downarrow}O^{g_t}_t(i)),$$

which contradicts the fact that it contained at least two elements.

$\neg(SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i})$**:** If the step is not sequentially consistent, it is not dirty by definition of $dirty$, and if the step is not a write buffer step, it is not dirty by contraposition of readiness at t. In each case, the buffer at t was not dirty

$$\neg dirty_{\downarrow}O_t(t,i).$$

Since the schedule before t is unaffected by the reordering, the buffer at t is not dirty in the reordered schedule either

$$\neg dirty_{\downarrow}O_{t+1}(t,i).$$

The schedule is a hardware schedule and thus valid, in particular until t

$$\Gamma^t_{\downarrow}(sO_{t+1}).$$

Since the reordered schedule has a local tail starting at $t+1$, we obtain with Lemma 322 that it is $t+1$-abstract

$$sO_{t+1} \in \mathrm{ABS}_{t+1}.$$

Since the buffer is dirty at $t+1$ but not at t, we conclude with Lemma 396 that step t is a shared processor step that buffered a write in strong memory mode

$$sO_{t+1}(t) \in \Sigma_{P,i} \wedge SC_{\downarrow}O_{t+1}(t) \wedge Sh_{\downarrow}O_{t+1}(t) \wedge BW_{\downarrow}O_{t+1}(t) \neq \emptyset.$$

By Lemma 271 the schedule is valid in the high-level machine until t

$$\Gamma^t_{\uparrow}(sO_{t+1}).$$

Using Lemmas 269 and 270 we obtain that the step is also a shared step buffering a write in the high-level machine

$$Sh_{\uparrow}O_{t+1}(t) \wedge BW_{\uparrow}O_{t+1}(t) \neq \emptyset.$$

Thus in particular the domain of the prepared buffered write is non-empty

$$Dom(PW_{\uparrow}O_{t+1}(t).wba) = Dom(BW_{\uparrow}O_{t+1}(t)) \neq \emptyset,$$

and with Condition AtomicWrite we obtain that step t did not prepare any bypassing writes (except to the normal processor registers)

$$Dom(PW_{\uparrow}O_{t+1}(t).bpa) \subseteq A_{NPR,i}.$$

By contraposition of Condition AtomicRMW we obtain further that step t is not a shared read

$$\neg ShR_{\uparrow}O_{t+1}(t).$$

Using Lemmas 269 and 270 we conclude the same for the low-level machine

$$Dom(PW_{\downarrow}O_{t+1}(t).bpa) \subseteq A_{NPR,i} \wedge \neg ShR_{\downarrow}O_{t+1}(t).$$

We conclude by Lemma 120 that the step is not a memory write in the low-level machine

$$\neg mwrite_{\downarrow}O_{t+1}(t),$$

and since it is also not shared, it is by definition not global

$$\neg G_{\downarrow}O_{t+1}(t).$$

Clearly the oracle input at t in the reordered schedule is the oracle input at k_t in the previous schedule

$$sO_{t+1}(t) = sO_t[t \leftarrow k_t](t) = sO_t(k_t),$$

which was thus also made by processor i

$$sO_t(k_t) \in \Sigma_{P,i}.$$

By Lemma 395 we obtain that the schedule has a local tail with a global end from t to k_t

$$\mathcal{L}^G O_t(t,k_t),$$

and by Lemma 379 we obtain that step k_t can be executed at its new position

$$c_\downarrow O_t^{k_t} =_\downarrow^{sO_t(k_t)} c_\downarrow O_t[t \leftarrow k_t]^t = c_\downarrow O_{t+1}^t$$

and from Lemma 150 we obtain that it was not a global step at its old position either, and also buffered a write

$$\neg G_\downarrow O_t(k_t) \wedge BW_\downarrow O_t(k_t) \neq \emptyset$$

and by definition of p it also was not a pushable step

$$\neg p_\downarrow O_t(k_t)$$

from which we conclude that it is not the next global or pushable step g_t

$$k_t \neq g_t.$$

Also by Lemma 150 we obtain that the step was also shared

$$Sh_\downarrow O_t(k_t) = Sh_\downarrow O_{t+1}(t) = 1.$$

By definition, the operation at k_t was a push

$$Op_{i\downarrow} O_t(k_t) = push,$$

and k_t was an issued write at $k_t + 1$

$$k_t \in issue_\downarrow O_t^{k_t+1}(i).$$

By Lemma 390, step k_t is before or equal to step g_t

$$k_t \leq g_t.$$

Since we already know that they are not equal, step k_t must have been before step g_t

$$k_t < g_t$$

and thus step $k_t + 1$ is before or equal to step g_t

$$k_t + 1 \leq g_t.$$

By Lemma 325 the sequence of issued writes at g_t has grown by some suffix q

$$issue_\downarrow O_t^{g_t}(i) = issue_\downarrow O_t^{k_t+1}(i) \circ q$$

and thus the issued write is still buffered at g_t

$$k_t \in issue_{\downarrow} O_t^{g_t}(i).$$

We conclude that the sequence is not empty

$$issue_{\downarrow} O_t^{g_t}(i) \neq \varepsilon,$$

and by Lemma 123 the write buffer is also not empty

$$wb_{\downarrow} O_t^{g_t}(i) \neq \varepsilon.$$

Thus by definition of p step g_t is not pushable

$$\neg p_{\downarrow} O_t(g_t)$$

and by definition of g_t it is global

$$G_{\downarrow} O_t(g_t).$$

Since the drain condition holds at g_t, unit i can thus not be a victim of step g_t

$$i \notin victims_{\downarrow} O_t(g_t).$$

Since step k_t is made by that unit

$$u_{\downarrow} O_t(k_t) = i,$$

step g_t does not interrupt step k_t

$$\neg int_{\downarrow} O_t(g_t, k_t).$$

By definition we take k_t to be the first step that is not unit-concurrent

$$k_t = \min \left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow} O_t(g_t, k) \right\},$$

from which we conclude that the step is not unit-concurrent

$$\neg ucon_{\downarrow} O_t(g_t, k_t).$$

Thus it is either made by the same unit as or interrupted by step g_t

$$u_{\downarrow} O_t(g_t) = u_{\downarrow} O_t(k_t) \vee int_{\downarrow} O_t(g_t, k_t).$$

Since we have already excluded the interrupt, step g_t must be made by the same unit

$$u_{\downarrow} O_t(g_t) = u_{\downarrow} O_t(k_t) = i.$$

With Lemma 322 we obtain that the previous schedule was g_t-abstract

$$sO_t \in \mathrm{ABS}_{g_t},$$

and with Lemma 269 we obtain that step g_t is feasible in the high-level machine

$$\Phi_{\uparrow} O_t(g_t) = \Phi_{\downarrow} O_t(g_t) = 1.$$

With Lemma 267 we obtain that the sequence of issued writes is the same in the two machines

$$issue_{\uparrow} O_t^{g_t}(i) = issue_{\downarrow} O_t^{g_t}(i),$$

and thus step k_t is buffered in the high-level machine as well

$$k_t \in issue_{\uparrow} O_t^{g_t}(i).$$

Furthermore, since the schedule is g_t-abstract and step g_t is after or equal to step $k_t + 1$, the schedule is also k_t+1-abstract

$$sO_t \in \mathrm{ABS}_{k_t+1},$$

and by Lemma 270, step k_t is also shared in the high-level machine

$$Sh_{\uparrow}(k_t).$$

The schedule is also k_t-abstract

$$sO_t \in \mathrm{ABS}_{k_t}.$$

Again by Lemmas 150 and 269, we obtain that the step was in strong memory mode

$$\begin{aligned} SC_{\uparrow} O_t(k_t) &= SC_{\downarrow} O_t(k_t) && \text{L 269} \\ &= SC_{\downarrow} O_{t+1}(t) && \text{L 150} \\ &= 1. \end{aligned}$$

Thus the unit was in strong memory mode at k_t

$$SC_{i\uparrow} O_t(k_t) = SC_{\uparrow} O_t(k_t) = 1.$$

By Lemma 323 the schedule is valid until $g_t - 1$

$$\Gamma_{\uparrow}^{g_t-1}(sO_t),$$

and since it is feasible at g_t it is semi-valid until g_t

$$\Gamma\Phi_{\uparrow}^{g_t}(sO_t).$$

Using Lemma 168, we obtain that the memory mode of unit i has not changed until g_t

$$SC_{i\uparrow} O_t(k_t) = SC_{i\uparrow} O_t(g_t),$$

and thus step g_t is made in strong memory mode

$$SC_{\uparrow}(g_t).$$

Since it is not made by a write buffer in strong memory mode, it is made by a processor; since it is made by the same unit as step k_t, it is made by the processor of unit i

$$sO_t(g_t) \in \Sigma_{P,i}.$$

Thus step g_t has a buffered shared write from k_t, and by Condition Flush we obtain that step g_t is not a shared read

$$\neg ShR_{\uparrow} O_t(g_t).$$

By Lemma 269 we obtain that it is also not a shared read in the low-level machine

$$\neg ShR_{\downarrow} O_t(g_t),$$

and we conclude that it is semi-ordered

$$sord_{\downarrow} O_t(g_t).$$

By Lemma 320 the schedule is g_t-ordered

$$sO_t \in \mathrm{ORD}_{g_t},$$

and it is thus g_t+1-semi-ordered

$$sO_t \in \mathrm{ORD}^{-}_{g_t+1}.$$

Since it is valid and IPI-valid, it is so in particular until g_t resp. $g_t - 1$

$$\Gamma^{g_t}_{\downarrow}(sO_t) \wedge \Delta^{g_t-1}_{IPI\downarrow}(sO_t),$$

and by Lemma 305 the schedule is g_t+1 abstract

$$sO_t \in \mathrm{ABS}_{g_t+1}.$$

We conclude with Lemma 271 that the schedule is valid until g_t in the high-level machine

$$\Gamma^{g_t}_{\uparrow}(sO_t).$$

Since step g_t is not a shared read but global, it is a shared memory write

$$mwrite_{\downarrow} O_t(g_t) \wedge Sh_{\downarrow} O_t(g_t).$$

The schedule is clearly g_t-abstract

$$sO_t \in \mathrm{ABS}_{g_t},$$

and by Lemma 269 step g_t is also shared in the high-level machine

$$Sh_{\uparrow} O_t(g_t).$$

By Lemma 120, the domain of the prepared bypassing writes is not contained in the normal processor registers

$$Dom(PW_{\downarrow} O_t(g_t).bpa) \not\subseteq A_{NPR,i}.$$

By Lemma 270 this also holds in the high-level machine

$$Dom(PW_{\uparrow} O_t(g_t).bpa) \not\subseteq A_{NPR,i},$$

And by Condition MessagePassing the buffer must be empty

$$wb_{\uparrow} O_t^{g_t}(i) = \varepsilon,$$

and by Lemma 123 so must the sequence of issued writes

$$issue_{\uparrow} O_t^{g_t}(i) = \varepsilon,$$

which contradicts the fact that k_t is an issued write.

□

We now prove that the reordering strategy indeed creates an ordered prefix.

Theorem 1. *If the original schedule is a hardware schedule*

$$s \in HW,$$

then in each iteration the following properties hold.

1. *the schedule is equivalent to the original schedule,*

$$sO_t \equiv_{\downarrow} s,$$

2. *the schedule is a hardware schedule*

$$sO_t \in HW,$$

3. *the schedule is t-ordered*

$$sO_t \in \mathrm{ORD}_t,$$

4. *the schedule is ready*

$$\mathcal{R}(t).$$

Proof. The proof is by induction on t.

In the base case, the first claim is trivial. The third claim is by assumption. The second claim is trivial. The fourth claim is Lemma 389.

In the inductive step $t \to t+1$, we obtain by Lemma 393 with the induction hypothesis that the schedule has a local tail until k_t and until g_t

$$\mathcal{L}O_t(t,k_t) \wedge \mathcal{L}O_t(t,g_t).$$

By Lemma 395 we obtain that the schedule has a local tail with a global end from t to k_t

$$\mathcal{L}^G O_t(t,k_t).$$

The first claim follows with Lemma 387

$$s \equiv_{\downarrow} sO_t \equiv_{\downarrow} sO_t[t \leftarrow k_t] = sO_{t+1}.$$

Since the previous schedule is a hardware schedule, it is balanced, valid, and IPI-valid

$$sO_t \in bal \wedge \forall t'.\Delta_{IPI\downarrow} O_t(t') \wedge \Gamma_{\downarrow} O_t(t').$$

By Lemma 307 the reordered schedule is still balanced

$$sO_{t+1} \in bal.$$

By Lemma 388 the schedule after the reordering is still valid and IPI-valid

$$\forall t'.\Delta_{IPI\downarrow} O_t[t \leftarrow k_t](t') \wedge \Gamma_{\downarrow} O_t[t \leftarrow k_t](t'),$$

which is the schedule in iteration $t+1$

$$\forall t'.\Delta_{IPI\downarrow} O_{t+1}(t') \wedge \Gamma_{\downarrow} O_{t+1}(t').$$

The third claim follows

$$sO_{t+1} \in HW.$$

By Lemma 383 the reordered schedule is $t+1$-ordered, and the third claim follows

$$sO_{t+1} = sO_t[t \leftarrow k_t] \in \mathrm{ORD}_{t+1}.$$

By Lemma 393 we obtain now that the reordered schedule has a local tail until g_{t+1}

$$\mathcal{L}O_{t+1}(t+1, g_{t+1}),$$

and the fourth claim is just Lemma 397. □

4.13 Infinite Schedules

The above method generates schedules where a finite prefix is reduced. We want to obtain an equivalent schedule which is completely reduced. Note that the above method never changes the finite reduced prefix.

Lemma 398.

$$t \leq t' \rightarrow sO_t[0:t-1] = sO_{t'}[0:t].$$

Proof. By induction on t' starting at t. The base case is trivial.

In the inductive step $t' \rightarrow t'+1$, we reorder only steps from t'

$$O_{t'+1} = O_{t'}[t' \leftarrow k_{t'}]$$

which does not change the steps before t' and the claim follows with the induction hypothesis

$$sO_{t'+1}[0:t-1] = sO_{t'}[t' \leftarrow k_{t'}][0:t-1] = sO_{t'}[0:t-1] = sO_t[0:t-1].$$

□

If we now raise t' towards infinity, we obtain a longer and longer prefix which is reduced. In the "limit"[9], we obtain a schedule s_∞

$$s_\infty(t) = sO_{t+1}(t).$$

Assume for the remainder of this section that the original schedule is a hardware schedule

$$s \in HW,$$

and thus by Theorem 1 also all finite reorderings are hardware schedules

$$sO_t \in HW.$$

[9] We do not use a formal meaning of limit here; we simply define a schedule s_∞ that we call the limit.

4.13.1 Fairness

We have to show that every step l of s is eventually added to s_∞. This is the case if in some iteration t, the new position of l *after* iteration t is equal to t.

We first show that the function mv which tracks the movement of steps also tracks the position of the n-th step of object X.

Lemma 399.

$$mv[t \leftarrow k_t](\#X{\approx}n(sO_t)) = \#X{\approx}n(sO_{t+1}).$$

Proof. Follows with Lemmas 77 and 117

$$\begin{aligned} mv[t \leftarrow k_t](\#X{\approx}n(sO_t)) &= \begin{cases} t & \#X{\approx}n(sO_t) = k_t \\ \#X{\approx}n(sO_t)+1 & \#X{\approx}n(sO_t) \in [t : k_t) \\ \#X{\approx}n(sO_t) & \text{o.w.} \end{cases} \\ &= \#X{\approx}n(sO_t[t \leftarrow k_t]) && \text{L 117} \\ &= \#X{\approx}n(sO_{t+1}). \end{aligned}$$

□

We show that these steps always strongly agree.

Lemma 400.

$$sO_t(k) = sO_{t+1}(mv[t \leftarrow k_t](k)) \wedge c_\downarrow O_t^k =_\downarrow^{sO_t(k)} c_\downarrow O_{t+1}^{mv[t \leftarrow k_t](k)}.$$

Proof. By Theorem 1 the schedules are equivalent

$$sO_t \equiv_\downarrow s \equiv_\downarrow sO_{t+1}.$$

Clearly step k is the n-th step of some object X

$$k = \#X{\approx}n(sO_t))$$

and by Lemma 399 step $mv[t \leftarrow k_t](k)$ is the n-th step of that object in the reordered schedule

$$mv[t \leftarrow k_t](k) = mv[t \leftarrow k_t](\#X{\approx}n(sO_t)) = \#X{\approx}n(sO_{t+1}).$$

The claims follow by definition of $\equiv_\downarrow$ for $t := k$ and $k := mv[t \leftarrow k_t](k)$. □

Using mvO_t we can track the position of the n-th step of X after t iterations of the reordering strategy, i.e., in schedule sO_t.

Lemma 401.

$$mvO_t(\#X{\approx}n(s)) = \#X{\approx}n(sO_t)$$

Proof. By induction on t.

The base case is easy

$$mvO_0(\#X{\approx}n(s)) = mv\varepsilon(\#X{\approx}n(s)) = \#X{\approx}n(s) = \#X{\approx}n(sO_0).$$

In the inductive step $t \to t+1$, the position is moved according to $mv[t \leftarrow k_t]$ and the claim follows with Lemma 399

$$\begin{aligned} mvO_{t+1}(\#X{\approx}n(s)) &= mvO_t[t \leftarrow k_t](\#X{\approx}n(s)) \\ &= mv[t \leftarrow k_t](mvO_t(\#X{\approx}n(s))) \\ &= mv[t \leftarrow k_t](\#X{\approx}n(sO_t)) && \text{IH} \\ &= \#X{\approx}n(sO_{t+1}). && \text{L 399} \end{aligned}$$

□

Lemma 402. *The steps have the same oracle input and strongly agree*

$$s(l) = sO_t(mvO_t(l)) \wedge c^l_\downarrow =^{s(l)}_\downarrow c_\downarrow O_t^{mvO_t(l)}.$$

Proof. By Theorem 1, the schedule sO_t is equivalent to the original schedule

$$sO_t \equiv_\downarrow s.$$

Obviously step l was the n-th step of some object X

$$l = \#X{\approx}n(s),$$

and by Lemma 401, the step $mvO_t(l)$ is the n-th step of that object in sO_t

$$mvO_t(l) = \#X{\approx}n(sO_t).$$

Since the schedules are equivalent, we obtain that these steps have the same oracle input

$$s(l) = sO_t(mvO_t(l))$$

and strongly agree

$$c^l_\downarrow =^{s(l)}_\downarrow c_\downarrow O_t^{mvO_t(l)},$$

which are the claims. □

Due to the monotonicity, buffered writes are never moved beyond the write buffer step that commits them. Those steps are global, and thus never delayed. It follows that buffered writes are never moved beyond the point at which they were committed in the original schedule.

In order to make this argument precise, we use invariant $\mathcal{J}$ from page 186 to show that the sequence of issued writes is reordered quite much in the same way as the individual steps.

In the steps before the reordering, nothing changes and nothing is moved.

Lemma 403.

$$k \le t \to issue_\downarrow O^k_{t+1}(i) = mv[t \leftarrow k_t](issue_\downarrow O^k_t(i)).$$

We show that the schedule only has a local tail that ends with an issued write buffer step of unit i, then both 1) step t in the reordered schedule executes a pop operation and 2) at all steps between t and k the sequence of issued writes in the original schedule of the unit is non-empty. This will allow us to use Lemma 46 to prove commutativity when we later apply Lemma 202.

Lemma 404.

$$k \in [t : k_t] \wedge \mathcal{L}^W O_t(t, k_t) \wedge i = u_{\downarrow} O_t(k_t) \rightarrow Op_{i\downarrow} O_{t+1}(t) = pop \wedge issue_{\downarrow} O_t^k(i) \neq \varepsilon.$$

Proof. By definition of $\mathcal{L}^W$, step k_t in the original schedule is a write buffer step

$$sO_t(k_t) \in \Sigma_{WB,i}.$$

By Lemmas 77 and 402 we obtain the same for step t in the reordered schedule

$$sO_{t+1}(t) = sO_{t+1}(mt[t \leftarrow k_t](k_t)) = sO_t(k_t) \in \Sigma_{WB,i},$$

which thus executes a pop operation

$$Op_{i\downarrow} O_{t+1}(t) = pop.$$

This proves the first part of the claim.

We also obtain by definition of $\mathcal{L}^W$ that the head of the write buffer at g_t has been issued before t

$$hd(issue_{\downarrow} O_t^{g_t}(i)) < t.$$

By the monotonicity of write buffers, it is also buffered at $k \in [t : g_t]$

$$hd(issue_{\downarrow} O_t^{g_t}(i)) \in issue_{\downarrow} O_t^k(i),$$

and the second part of the claim follows. □

Using the observation (with Lemmas 83 and 86) that we can treat the reordered schedule sO_{t+1} as the "original" schedule, and the original schedule sO_t as a "reordered" version of that schedule using the reordering $[t \rightarrow k_t]$

$$sO_t = sO_t[t \leftarrow k_t][t \rightarrow k_t] = sO_{t+1}[t \rightarrow k_t],$$

we can use for the steps in the moved interval the invariant $\mathcal{J}$ with the operation that is done in step t in the "original" schedule

$$f_i(l) = Op_i O_{t+1}(t)(l, t).$$

Lemma 405.

$$k \in [t : k_t] \rightarrow \mathcal{J}_{\downarrow}^{f_i} O_{t+1}([t \rightarrow k_t], k+1, k).$$

Proof. We prove this by induction on k, starting at t. The base case is just Lemma 201.

In the inductive step $k \rightarrow k+1$, we use Lemma 202, which reduces the claim to the following three subclaims.

$mv[t \rightarrow k_t](k+1) = k$**:** This is just a special case of Lemma 77.

Commutativity: We distinguish whether steps k and k_t are made by the same unit.

$diffuO_t(k_t, k)$**:** The claim is Lemma 100.

$\neg diffuO_t(k_t,k)$**:** We immediately obtain that steps k_t and k are not unit-concurrent

$$\neg ucon_{\downarrow}O_t(k_t,k)$$

and thus by definition of $\mathcal{L}^I$ the schedule does not have an independent end

$$\neg\mathcal{L}^I O_t(t,k_t).$$

With Theorem 1 and Lemmas 393 and 395 we obtain as before that the local tail has a global end

$$\exists G \in \{W,I\}.\mathcal{L}^G O_t(t,k_t),$$

and since it does not have an independent end, the tail ends with an issued write buffer step

$$\mathcal{L}^W O_t(t,k_t).$$

Let i be the unit that makes the two steps

$$i = u_{\downarrow}O_t(k) = u_{\downarrow}O_t(k_t)$$

and by Lemma 404, step t in the reordered ("original") schedule executes a pop operation and the sequence of issued writes in the original ("re-ordered") schedule is non-empty

$$Op_{i\downarrow}O_{t+1}(t) = pop \wedge issue_{\downarrow}O_t^k(i) \neq \varepsilon.$$

We turn the original schedule into the "reordered" schedule

$$issue_{\downarrow}O_{t+1}[t \to k_t]^k(i) = issue_{\downarrow}O_t^k(i) \neq \varepsilon$$

and obtain that the moved sequence of issued writes (which has the same length) is also non-empty

$$mv[t \to k]^{-1}(issue_{\downarrow}O_{t+1}[t \to k_t]^k(i)) \neq \varepsilon.$$

and with Lemma 46 we obtain that the operations done in steps t and k in the reordered schedule commute

$$\begin{aligned} &Op_{i\downarrow}O_{t+1}(t)(Op_{i\downarrow}O_{t+1}(k)(mv[t \to k]^{-1}(issue_{\downarrow}O_{t+1}[t \to k_t]^k(i)),k),t) \\ &= Op_{i\downarrow}O_{t+1}(k)(Op_{i\downarrow}O_{t+1}(t)(mv[t \to k]^{-1}(issue_{\downarrow}O_{t+1}[t \to k_t]^k(i)),t),k), \end{aligned}$$

which is the claim.

$c_{\downarrow}O_{t+1}^{k+1} =_{\downarrow}^{sO_{t+1}(k+1)} c_{\downarrow}O_{t+1}[t \to k_t]^k$**:** Using Lemma 402 and Lemma 77 we obtain that step $k+1$ in schedule sO_{t+1} is step k in schedule sO_t

$$sO_t(k) = sO_{t+1}(mv[t \leftarrow k_t](k)) = sO_{t+1}(k+1).$$

We rewrite the claim as follows with Lemma 77

$$c_{\downarrow}O_{t+1}^{mv[t\leftarrow k_t](k)} =_{\downarrow}^{sO_t(k)} c_{\downarrow}O_t^k.$$

This is just a special case of Lemma 402.

□

For the steps after k_t, no operation is missing

$$f_i'(l) = l.$$

Lemma 406.

$$k > k_t \to \mathcal{J}_{\downarrow}^{f_i'} O_{t+1}([t \to k_t], k, k).$$

Proof. By induction on k, starting at $k_t + 1$. The base case is just Lemma 203 and Lemma 405 with $k := k_t$.

In the inductive step $k \to k+1$, we use Lemma 202 which reduces the claim to the following three subclaims.

$mv[t \to k_t](k) = k$**:** This is just a special case of Lemma 77.

Commutativity: There is nothing to show because no operations are missing ($f_i'(l) = l$).

$c_{\downarrow} O_{t+1}^k =_{\downarrow}^{sO_{t+1}(k)} c_{\downarrow} O_{t+1}[t \to k_t]^k$**:** Using Lemma 402 and Lemma 77 we obtain that step $k+1$ in schedule sO_{t+1} is step k in schedule sO_t

$$sO_t(k) = sO_{t+1}(mv[t \leftarrow k_t](k)) = sO_{t+1}(k).$$

We rewrite the claim as follows with Lemma 77

$$c_{\downarrow} O_{t+1}^{mv[t \leftarrow k_t](k)} =_{\downarrow}^{sO_t(k)} c_{\downarrow} O_t^k.$$

This is just a special case of Lemma 402.

□

Lemma 407.

$$k \notin (t : k_t] \to issue_{\downarrow} O_{t+1}^k(i) = mv[t \leftarrow k_t](issue_{\downarrow} O_t^k(i)).$$

Proof. For the steps $k \leq t$ this is exactly Lemma 403 . For the remaining steps $k > k_t$, we obtain with Lemma 406 that the sequences of issued writes are nearly the same

$$\mathcal{J}_{\downarrow}^{f_i'} O_{t+1}([t \to k_t], k, k)$$

and we obtain that the "original" and "reordered" schedules are nearly the same

$$issue_{\downarrow} O_{t+1}^k(i) = f_i'(mv[t \to k_t]^{-1}(issue_{\downarrow} O_{t+1}[t \to k_t]^k(i))).$$

Rewriting with Lemma 83, $f_i'(l) = l$ and the fact that the "reordered" schedule is the original schedule we obtain the claim

$$issue_{\downarrow} O_{t+1}^k(i) = mv[t \leftarrow k_t](issue_{\downarrow} O_t^k(i)).$$

□

We observe that in any single iteration, global steps and pushable steps are not delayed.

Lemma 408.

$$G_{\downarrow}O_t(l) \vee p_{\downarrow}O_t(l) \rightarrow l \notin [t : k_t).$$

Proof. Assume for the sake of contradiction that the step is delayed

$$l \in [t : k_t).$$

With Lemma 390 we obtain that step $mvO_t(l)$ is before step g_t

$$l < k_t \leq g_t,$$

which is by definition the next global or pushable step

$$g_t = \min \left\{ g \geq t \;\middle|\; G_{\downarrow}O_t(g) \vee p_{\downarrow}O_t(g) \right\}.$$

It follows that step l is not global or pushable

$$\neg G_{\downarrow}O_t(l) \wedge \neg p_{\downarrow}O_t(l),$$

which is a contradiction. □

Pushable steps remain pushable.

Lemma 409.

$$p_{\downarrow}(l) \rightarrow p_{\downarrow}O_t(mvO_t(l)).$$

Proof. By induction on t. The base case is trivial.

In the inductive step $t \rightarrow t+1$, we have by the inductive hypothesis that the step is still pushable after t iterations

$$p_{\downarrow}O_t(mvO_t(l)).$$

Let i be the unit that makes step l

$$i = u_{\downarrow}(l)$$

(and by Lemma 402 also steps $mvO_t(l)$ and $mvO_{t+1}(l)$ in the respective schedules)

$$u_{\downarrow}O_t(mvO_t(l)) = u_{\downarrow}(l) = u_{\downarrow}O_{t+1}(mvO_{t+1}(l)) = i.$$

Because the step is pushable, it is local and does not buffer a write

$$L_{\downarrow}(l) \wedge BW_{\downarrow}(l) = \emptyset.$$

By Lemma 402, the steps have the same oracle inputs and strongly agree

$$s(l) = sO_{t+1}(mvO_{t+1}(l)) \wedge c^l_{\downarrow} =^{s(l)}_{\downarrow} c_{\downarrow}O^{mvO_{t+1}(l)}_{t+1}.$$

With Lemma 150 with $X := L, BW$ we obtain that the step is still local and does not buffer a write

$$L_{\downarrow}O_{t+1}(mvO_{t+1}(l)) \wedge BW_{\downarrow}O_{t+1}(mvO_{t+1}(l)) = \emptyset,$$

which are two parts of the definition of p.

Since step $mvO_t(l)$ is pushable, by definition of p, the write buffer at $mvO_t(l)$ is empty

$$wb_{\downarrow}O^{mvO_t(l)}_t(i) = \varepsilon.$$

We show now the third part of the definition of p, i.e., that the buffer at it is still empty after the reordering

$$wb_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i) \stackrel{!}{=} \varepsilon.$$

This is the same as showing that the write buffer has length zero

$$|wb_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i)| \stackrel{!}{=} 0.$$

With Lemma 123 it suffices to show that the length of the sequence of issued writes has not increased and is thus also zero

$$\begin{aligned} 0 = |wb_{\downarrow}O_{t}^{mvO_{t}(l)}(i)| &= |issue_{\downarrow}O_{t}^{mvO_{t}(l)}(i)| && \text{L 123} \\ &\stackrel{!}{\geq} |issue_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i)| \\ &= |wb_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i)| && \text{L 123} \\ &= 0, \end{aligned}$$

We distinguish whether step $mvO_t(l)$ is equal to k_t and thus moved to the front or not.

$mvO_t(l) = k_t$**:** In this case the step is moved to position t

$$mvO_{t+1}(l) = mv[t \leftarrow k_t](mvO_t(l)) = mv[t \leftarrow k_t](k_t) = t.$$

By Lemma 393 and Theorem 1 the schedule has a local tail

$$\mathcal{L}O_t(t, k_t)$$

and with Lemma 325 with $l := t$ we obtain that the sequence of issued writes at k_t has a suffix q not contained in the sequence of issued writes at t

$$\exists q.issue_{\downarrow}O_{t}^{k_t}(i) = issue_{\downarrow}O_{t}^{t}(i) \circ q$$

and is thus at least as long

$$|issue_{\downarrow}O_{t}^{k_t}(i)| \geq |issue_{\downarrow}O_{t}^{t}(i)|.$$

The claim follows as the reordering does not affect the sequence of issued writes at t

$$\begin{aligned} |issue_{\downarrow}O_{t}^{mvO_{t}(l)}(i)| &= |issue_{\downarrow}O_{t}^{k_t}(i)| \\ &\geq |issue_{\downarrow}O_{t}^{t}(i)| \\ &= |issue_{\downarrow}O_{t+1}^{t}(i)| \\ &= |issue_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i)|. \end{aligned}$$

$mvO_t(l) \neq k_t$**:** By Lemma 408 with $l := mvO_t(l)$, the step is not delayed in iteration t

$$mvO_t(l) \notin [t : k_t).$$

Since step $mvO_t(l)$ is also not moved to the front, it is not in the moved range

$$mvO_t(l) \notin [t : k_t]$$

and with Lemma 77 we obtain that it is thus not moved at all

$$mvO_{t+1}(l) = mv[t \leftarrow k_t](mvO_t(l)) = mvO_t(l).$$

Furthermore by Lemma 407 with $k := mvO_t(l)$ the sequence of issued writes is transformed by $mv[t \leftarrow k_t]$

$$issue_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i) = issue_{\downarrow}O_{t+1}^{mvO_t(l)}(i) = mv[t \leftarrow k_t](issue_{\downarrow}O_t^{mvO_t(l)}(i)).$$

We conclude they have the same length, and the claim follows

$$|issue_{\downarrow}O_{t+1}^{mvO_{t+1}(l)}(i)| = |mv[t \leftarrow k_t](issue_{\downarrow}O_t^{mvO_t(l)}(i))| = |issue_{\downarrow}O_t^{mvO_t(l)}(i)|.$$

□

Therefore global steps and pushable steps are never delayed at all.

Lemma 410.

$$G_{\downarrow}(l) \vee p_{\downarrow}(l) \rightarrow mvO_t(l) \notin [t : k_t).$$

Proof. With Lemma 410 we reduce the claim to showing that the step is still global or pushable

$$G_{\downarrow}O_t(mvO_t(l)) \vee p_{\downarrow}O_t(mvO_t(l)).$$

We distinguish whether the step was originally global or pushable.

$G_{\downarrow}(l)$**:** By Lemma 402, the steps have the same oracle inputs and strongly agree

$$s(l) = sO_t(mvO_t(l)) \wedge c_{\downarrow}^l =_{\downarrow}^{s(l)} c_{\downarrow}O_t^{mvO_t(l)}.$$

With Lemma 150 with $X := L$ we obtain that the step is still global

$$G_{\downarrow}O_t(mvO_t(l)) \equiv \neg L_{\downarrow}O_t(mvO_t(l)) \equiv \neg L_{\downarrow}(l) \equiv G_{\downarrow}(l),$$

and the claim follows.

$p_{\downarrow}(l)$**:** By Lemma 409 the step is still pushable

$$p_{\downarrow}O_t(mvO_t(l)),$$

and the claim follows.

□

Lemma 411.

$$G_{\downarrow}(l) \vee p_{\downarrow}(l) \rightarrow mvO_t(l) \leq l.$$

Proof. By induction on t. The base case is trivial.

In the inductive step $t \rightarrow t+1$, we have to show that the reordering in iteration t does not move $mvO_t(l)$ beyond l

$$mvO_{t+1}(l) = mv[t \leftarrow k_t](mvO_t(l)) \overset{!}{\leq} mvO_t(l) \leq l.$$

We distinguish between the three cases in the definition of $mv[t \leftarrow k_t]$

$mvO_t(l) = k_t$**:** By definition, step k_t is after or equal to step t

$$k_t \geq t,$$

and the claim follows

$$mv[t \leftarrow k_t](mvO_t(l)) = t \leq k_t = mvO_t(l).$$

$mvO_t(l) \in [t : k_t)$**:** This contradicts Lemma 410.

$mvO_t(l) \notin [t : k_t]$**:** By definition, the step is not moved, and the claim follows

$$mv[t \leftarrow k_t](mvO_t(l)) = mvO_t(l) \leq mvO_t(l).$$

□

It can now be shown that step that commits a buffered write never overtakes that buffered write.

Lemma 412.

$$l = hd(issue^k_\downarrow(i)) \wedge s(k) \in \Sigma_{WB,i} \rightarrow mvO_t(l) = hd(issue_\downarrow O_t^{mvO_t(k)}).$$

Proof. By induction on t. The base case is trivial.

In the inductive step $t \rightarrow t+1$, we distinguish between the three cases in the definition of $mv[t \leftarrow k_t]$ for the reordering of step k.

$mvO_t(k) = k_t$**:** Then the next position of step k is at t

$$mvO_{t+1}(k) = t.$$

By Lemma 402 we obtain that the step has the same oracle input as step k in the original schedule

$$sO_{t+1}(t) = sO_{t+1}(mvO_{t+1}(k)) = s(k)$$

and is thus made by the same unit, namely unit i

$$u_\downarrow O_{t+1}(t) = u_\downarrow(k) = i.$$

Since the steps before t are not changed in the reordering, we obtain that the sequence of issued writes at t in iteration t is the same sequence in iteration $t+1$ at the position of k in iteration $t+1$

$$\begin{aligned} issue_\downarrow O_t^t(i) &= issue_\downarrow O_t[t \leftarrow k_t]^t(i) \\ &= issue_\downarrow O_{t+1}^t(i) \\ &= issue_\downarrow O_{t+1}^{mvO_{t+1}(k)}(i). \end{aligned}$$

The schedule after the reordering is a hardware schedule and thus valid. In particular step t is valid

$$\Gamma_\downarrow O_{t+1}(t)$$

and thus by definition satisfies the drain condition

$$\Delta_{\downarrow} O_{t+1}(t)$$

and thus by definition has non-empty write buffer

$$wb_{\downarrow} O^{t}_{t+1}(i) \neq \varepsilon.$$

By Lemma 123 we obtain the same for the sequence of issued writes

$$issue_{\downarrow} O^{t}_{t+1}(i) \neq \varepsilon,$$

and since the steps before t are not changed in the reordering, the same holds in iteration t

$$issue_{\downarrow} O^{t}_{t}(i) \neq \varepsilon.$$

By Theorem 1 that schedule is t-ordered

$$sO_t \in \mathrm{ORD}_t,$$

and by Lemma 393 the schedule has a local tail from t to k_t

$$\mathcal{L} O_t(t, k_t).$$

By Lemma 326 and the induction hypothesis we obtain that the head at t is the same as at k_t, i.e., the current position of l

$$hd(issue_{\downarrow} O^{t}_{t}(i)) = hd(issue_{\downarrow} O^{k_t}_{t}(i)) = hd(issue_{\downarrow} O^{mvO_t(k)}_{t}(i)) = mvO_t(l).$$

We conclude from the monotonicity of the sequence of issued writes that the position of l is before t

$$mvO_t(l) < t$$

and is thus unchanged in the step

$$mvO_{t+1}(l) = mv[t \leftarrow k_t](mvO_t(l)) = mvO_t(l).$$

Thus the position of step l in iteration $t+1$ is the head of the sequence of issued writes t in iteration t

$$mvO_{t+1}(l) = hd(issue_{\downarrow} O^{t}_{t}(i)),$$

and the claim follows

$$mvO_{t+1}(l) = hd(issue_{\downarrow} O^{mvO_{t+1}(k)}_{t+1}(i)).$$

$mvO_t(k) \in [t : k_t)$**:** By Lemma 136 step k is global

$$G_{\downarrow}(k)$$

and this case contradicts Lemma 410.

$mvO_t(k) \notin [t : k_t]$**:** The position is unchanged

$$mvO_{t+1}(k) = mv[t \leftarrow k_t](mvO_t(k)) = mvO_t(k),$$

and by Lemma 407 the sequence of issued writes is moved by $mv[t \leftarrow k_t]$

$$\begin{aligned} issue_{\downarrow} O_{t+1}^{mvO_{t+1}(k)}(i) &= mv[t \leftarrow k_t](issue_{\downarrow} O_t^{mvO_{t+1}(k)}(i)) \\ &= mv[t \leftarrow k_t](issue_{\downarrow} O_t^{mvO_t(k)}(i)). \end{aligned}$$

The position of l is also moved by $mv[t \leftarrow k_t]$

$$mvO_{t+1}(l) = mv[t \leftarrow k_t](mvO_t(l))$$

and the claim follows with the induction hypothesis

$$\begin{aligned} mvO_{t+1}(l) &= mv[t \leftarrow k_t](mvO_t(l)) \\ &= mv[t \leftarrow k_t](hd(issue_{\downarrow} O_t^{mvO_t(k)}(i))) && \text{IH} \\ &= hd(mv[t \leftarrow k_t](issue_{\downarrow} O_t^{mvO_t(k)}(i))) \\ &= hd(issue_{\downarrow} O_{t+1}^{mvO_t(k)}(i))) \\ &= hd(issue_{\downarrow} O_{t+1}^{mvO_{t+1}(k)}(i)). \end{aligned}$$

□

We show that every write in a buffer is eventually committed

Lemma 413.

$$s \in bal \wedge l \in issue_{\downarrow}^{t+1}(i) \rightarrow \exists k > l.s(k) \in \Sigma_{WB,i} \wedge l = hd(issue_{\downarrow}^{k}(i)).$$

Proof. Clearly, the sequence can be split into a prefix p, step l, and then a suffix q

$$issue_{\downarrow}^{t}(i) = p \circ l \circ q.$$

The proof is by induction on p with t and q generalized.

In the base case $p = \varepsilon$, we obtain with Lemma 312 that there are g and q' such that the sequence has grown by q'

$$issue_{\downarrow}^{g}(i) = issue_{\downarrow}^{k}(i) \circ q' = l \circ q \circ q',$$

and step g is a write buffer step of unit i

$$s(g) \in \Sigma_{WB,i}.$$

Clearly the head of the sequence at g is step l

$$hd(issue_{\downarrow}^{g}(i)) = hd(l \circ q \circ q') = l,$$

and the claim follows with $k := g$.

In the inductive step $p \rightarrow t' \circ p$, we obtain with Lemma 312 that there are g and q' such that the sequence has grown by q'

$$issue_{\downarrow}^{g}(i) = issue_{\downarrow}^{k}(i) \circ q' = t' \circ p \circ l \circ q \circ q',$$

and step g is a write buffer step of unit i

$$s(g) \in \Sigma_{WB,i}.$$

Thus the operation at g is a pop operation

$$Op_{i\downarrow}(g) = pop$$

and the head of the buffer is removed, which by Lemma 41 is t'

$$\begin{aligned} issue_{\downarrow}^{g+1}(i) &= tl(issue_{\downarrow}^{g}(i)) \\ &= tl(t' \circ p \circ l \circ q \circ q') \\ &= tl(t') \circ p \circ l \circ q \circ q' && \text{L 41} \\ &= \varepsilon \circ p \circ l \circ q \circ q' \\ &= p \circ l \circ (q \circ q'). \end{aligned}$$

The claim is now the inductive hypothesis with $t := g+1$ and $q := q \circ q'$. □

Connecting these lemmas we obtain that each step is committed at some point.

Lemma 414.

$$\forall l. \exists t. mvO_{t+1}(l) \leq t.$$

Proof. We distinguish between global steps and pushable steps, and other steps.

$G_{\downarrow}(l) \vee p_{\downarrow}(l)$**:** We choose $t := l$. The claim is Lemma 411 with $t := l+1$.

$\neg G_{\downarrow}(l) \wedge \neg p_{\downarrow}(l)$**:** Since the step is not pushable, it is either not local or buffering a write

$$\neg L_{\downarrow}(l) \vee BW_{\downarrow}(l) \neq \emptyset.$$

Since it is not global, it is local, leaving only the possibility that it is buffering a write

$$BW_{\downarrow}(l) \neq \emptyset.$$

By contraposition of Lemma 136, step l is not not a write buffer step

$$s(l) \notin \Sigma_{WB,i}$$

and is thus a processor step

$$s(l) \in \Sigma_{P,i},$$

and thus by definition the operation performed by step l is a push

$$Op_{i\downarrow}(l) = push.$$

By definition, step l is an element of the sequence of issued writes at $l+1$

$$l \in issue_{\downarrow}^{l}(i) \circ l = issue_{\downarrow}^{l+1}(i).$$

By Lemma 413, step l eventually becomes the head of the write buffer at k, which is also a write buffer step

$$s(k) \in \Sigma_{WB,i} \wedge l = hd(issue_{\downarrow}^{k}(i)).$$

By Lemma 136, that step is global

$$G_{\downarrow}(k).$$

We choose $t := k$. By Lemma 412, the position in iteration $k+1$ is the head of the buffer in iteration $k+1$ at the position of k in iteration $k+1$

$$mvO_{k+1}(l) = hd(issue_{\downarrow} O_{k+1}^{mvO_{k+1}(k)}).$$

By the monotonicity of the sequence of issued writes, the position in iteration $k+1$ is less than the position of k in iteration $k+1$

$$mvO_{k+1}(l) < mvO_{k+1}(k),$$

and the claim follows with Lemma 411

$$mvO_{k+1}(l) < mvO_{k+1}(k) \leq k.$$

□

With this lemma at our disposal, it is easy to show that all steps are eventually added to the growing finite prefix and thus eventually end up in s_∞. We first show that a step that has become a part of the growing prefix also entered s_∞.

Lemma 415.

$$l < t \rightarrow sO_t(l) = sO_{l+1}(l).$$

Proof. By induction on t, starting at $l+1$. The base case is trivial.

In the inductive step, we simply observe that we only reorder steps at and after step t

$$sO_{t+1}(l) = sO_t[t \leftarrow k_t](l) = sO_t(l)$$

and the claim follows with the induction hypothesis

$$sO_t(l) = sO_{l+1}(l).$$

□

Thus each prefix of s_∞ corresponds to an iteration of the reordering.

Lemma 416.

$$s_\infty[0 : t-1] = sO_t[0 : t-1].$$

Proof. Let $l < t$ be one of the steps in question. The claim follows by definition and Lemma 415

$$s_\infty(l) = sO_{l+1}(l) = sO_t(l).$$

□

We can now show that our infinite reordering never drops (or adds) any steps.

Lemma 417.

$$X(s_\infty) = X(s).$$

Proof. We show that each set contains the other.

$X(s_\infty) \subseteq X(s)$**:** Let n be a step number reached by unit X at t

$$\#X[s_\infty](t) = n,$$

and we will show that it is also reached by unit X in s.

By Lemma 416, the schedule until $t-1$ is the same as some finite iteration

$$s_\infty[0:t-1] = sO_t[0:t-1],$$

and therefore also the step counts at t

$$\#X[sO_t](t) = \#X[s_\infty](t) = n.$$

Thus n steps are also reached by unit X in sO_t

$$n \in X(sO_t).$$

By Theorem 1, that iteration in the reordering is equivalent to s

$$s \equiv_\downarrow sO_t$$

and thus X reaches the same steps numbers in s, and the claim follows

$$n \in X(sO_t) = X(s).$$

$X(s_\infty) \supseteq X(s)$**:** Let n be a step number reached by unit X at t in s

$$\#X[s](t) = n,$$

and we will show that it is also reached by unit X in s_∞.

We distinguish two cases: either n is zero, or it the successor of some number n'.

$n = 0$**:** This step count is reached at the beginning of the schedule

$$\#X[s_\infty](0) = \#X(s_\infty[0:-1]) = \#X(\varepsilon) = 0,$$

and thus definitely reached

$$0 \in X(s_\infty).$$

$n = n' + 1$**:** By Lemma 116 there is a step at t' made by object X where object X has made n' steps

$$o(s(t')) = X \wedge \#X[s](t') = n.$$

By Lemma 115, that step is really the n'-th step of object X

$$\#X \approx n'(s) = t'.$$

By Lemma 414 there is an iteration $k+1$ in which the position of t' is before k

$$mvO_{k+1}(t') \leq k.$$

By Lemma 401, the n'-th step of X is at that position in iteration $k+1$

$$\#X \approx n'(sO_{k+1}) = mvO_{k+1}(t') \leq k,$$

and by Lemma 416, that schedule is the same until k as the limit

$$s_\infty[0:k] = sO_{k+1}[0:k].$$

We conclude that step counts of object X until the position of t' are the same

$$\#X[s_\infty](mvO_{k+1}(t')) = \#X[sO_{k+1}](mvO_{k+1}(t')) = n',$$

and that the step is still made by object X

$$o(s_\infty(mvO_{k+1}(t'))) = o(sO_{k+1}(mvO_{k+1}(t'))) = X,$$

and thus after the step, the step count has reached $n'+1$, i.e., n

$$\begin{aligned}\#X[s_\infty](mvO_{k+1}(t')+1) &= \#X[s_\infty](mvO_{k+1}(t')) + \#X(s_\infty(mvO_{k+1}(t'))) \\ &= \#X[s_\infty](mvO_{k+1}(t')) + 1 \\ &= n' + 1 \\ &= n\end{aligned}$$

and the claim follows

$$n \in X(s_\infty).$$

□

4.13.2 Correctness

We can finally show that the infinitely reordered schedule s_∞ is equivalent to s and reduced.

Theorem 2. *The infinitely reordered schedule is equivalent to the original schedule*

$$s_\infty \equiv_\downarrow s.$$

Proof. Let X be the object for which we have to show the claims. By Lemma 417, the object make the same steps

$$X(s_\infty) = X(s).$$

Let now t,k be the n-th step of object X in their respective schedule

$$t = \#X{\approx}n(s_\infty) \wedge k = \#X{\approx}n(s).$$

By Lemma 416, the limit has the same steps until t as the schedule in iteration $t+1$

$$s_\infty[0:t] = sO_{t+1}[0:t]$$

and thus step t is also the n-th step of object X in that schedule

$$t = \#X{\approx}n(sO_{t+1}).$$

Since by Theorem 1 the schedule in iteration $t+1$ and the original schedule are equivalent

$$sO_{t+1} \equiv_\downarrow s,$$

we obtain that the steps at t and k have the same oracle inputs and strongly agree

$$sO_{t+1}(t) = s(k) \wedge c_{\downarrow}[sO_{t+1}]^t =_{\downarrow}^{sO_{t+1}(t)} c_{\downarrow}[s]^k$$

and the claim follows

$$s_{\infty}(t) = sO_{t+1}(t) = s(k) \wedge c_{\downarrow}[s_{\infty}]^t = c_{\downarrow}[sO_{t+1}]^t =_{\downarrow}^{sO_{t+1}(t)} c_{\downarrow}[s]^k.$$

□

Theorem 3. *The infinitely reordered schedule is a hardware schedule*

$$s_{\infty} \in HW.$$

Proof. By Theorem 2 the schedules are equivalent

$$s_{\infty} \equiv_{\downarrow} s$$

and by Lemma 307 is still balanced

$$s_{\infty} \in bal.$$

Let now step t be a step for which we need to show that it is valid and IPI-valid in s_{∞}. By Lemma 416, the schedule in iteration $t+1$ is the same until t

$$s_{\infty}[0:t] = sO_{t+1}[0:t].$$

By Theorem 1, that schedule is a hardware schedule

$$sO_{t+1} \in HW$$

and thus step t is valid and IPI-valid

$$\Gamma_{\downarrow} O_{t+1}(t) \wedge \Delta_{IPI\downarrow} O_{t+1}(t).$$

Since the infinitely reordered schedule is the same until t, it immediately follows that step t is also valid and IPI-valid in that schedule

$$\Gamma_{\downarrow}[s_{\infty}](t) \wedge \Delta_{IPI\downarrow}[s_{\infty}](t).$$

□

Theorem 4. *In the infinitely reordered schedule, the low-level machine simulates the computation of the high-level machine*

$$\forall t. c_{\downarrow}[s_{\infty}]^t =_{\downarrow,\uparrow}^{s_{\infty}(t)} c_{\uparrow}[s_{\infty}]^t.$$

Proof. Let now step t be a step for which we need to show the simulation. By Lemma 416, the schedule in iteration $t+1$ is the same until t

$$s_{\infty}[0:t] = sO_{t+1}[0:t].$$

By Theorem 1, that schedule is $t+1$-ordered and a hardware schedule

$$sO_{t+1} \in \mathrm{ORD}_{t+1} \cap HW$$

and thus valid and IPI-valid, in particular until t and $t-1$

$$\Gamma^t_{\downarrow}(sO_{t+1}) \wedge \Delta^{t-1}_{IPI\downarrow}(sO_{t+1}).$$

By Lemma 305 it is therefore $t+1$-abstract

$$sO_{t+1} \in \mathrm{ABS}_{t+1},$$

and by definition step t is simulated correctly

$$c_{\downarrow} O^t_{t+1} =^{sO_{t+1}(t)}_{\downarrow,\uparrow} c_{\uparrow} O^t_{t+1}.$$

Since the schedules are the same until t, so are the configurations at t and the oracle input at t, and the claim follows

$$\forall t.c_{\downarrow}[s_\infty]^t =^{s_\infty(t)}_{\downarrow,\uparrow} c_{\uparrow}[s_\infty]^t.$$

□

We combine these results in a single write buffer reduction theorem.

Theorem 5. *If all schedules satisfy the conditions of Section 4.5 (which are formulated in terms of the high-level computation), then for every hardware schedule s there is an equivalent hardware schedule r which is reduced*

$$\forall s \in HW.\exists r \in HW.r \equiv_{\downarrow} s \wedge \forall t.c_{\downarrow}[r]^t =^{r(t)}_{\downarrow,\uparrow} c_{\uparrow}[r]^t.$$

Proof. We choose $r := s_\infty$ and the claims are Theorems 2 to 4. □

Chapter 5

Conclusion and Related Work

5.1 Review and Open Problems

We have introduced a powerful yet compact computational model for x86-like processors (Chapter 2) and have shown that it is strong enough to model an x86-like ISA from the literature (Chapter 3). We have extended the efficient software conditions of Cohen and Schirmer [CS10] to deal with the new features of this computational model, and have shown that the conditions are indeed sufficient to regain sequential consistency on the weak memory semantics of our processor model (Chapter 4).

There are several direct fruits of this work.

- First and foremost, it enables verification of complex software under the assumption of sequential consistency, even if the software is supposed to be run on x86-like processors.
- Secondly, it allows for a very simple compiler optimization: instead of compiling a fence behind every shared write, one can track the state of the write buffer – either using a static over-approximation, or dynamically in software – and thus ensure that there is a fence between every shared write and shared read on the same thread.
- Thirdly, it opens up the way for further optimizations of the hardware model such as by adding shared memory operations to a processor with write buffers, and draining in hardware all shared writes before a shared read is executed, or by introducing local read-modify-writes (e.g., non-interlocked fetch-and-add) that do not drain the buffer, or by introducing read and write barriers such as we have shown.
- Fourth, we have learned that direct-to-core IPIs, even though they are a natural way to formalize IPIs, are bad for sequential consistency; and similarly, one can use theorems like Theorem 5 to evaluate the effect of design decisions on sequential consistency when formalizing an instruction set architecture.

However, many questions remain open. We have done a tiny first step towards moving to weaker memory architectures than TSO by allowing bypassing writes in strong memory mode and while the buffer is non-empty. This immediately raises the question whether the order of local writes in the buffer matters at all (as long as they

can not be overtaken by shared writes). Allowing local writes to leave the buffer early – for example to clear a partial hit – or to be combined with earlier local and shared writes may be a valuable performance boost, and at least our work hints at the fact that this may be possible without changing the software discipline. Similar weakenings of the memory model with relation to speculative loads and of the visibility of writes in between processors are also interesting.

We consider all accesses to device memory as volatile and thus shared. That this is strictly necessary is not clear, for example, when some device memory is used only by a single processor. In this case, it might be sufficient to not issue a read from a device when a write to that device is being buffered by the same unit. The key statement that has to remain provable is that of Lemma 173.

In our work, we have not been able to handle autonomous steps of components closely related to the processor (such as the MMU or the INIT interrupt) in strong memory mode[1], because shared reads and shared bypassing writes of these units would introduce flushing obligations that the system programmer can not possibly satisfy. Kovalev, Chen, and Cohen [CCK14] have proposed and proven correct an extension of the Cohen Schirmer result that handles MMUs in strong memory mode by using a monotonicity property of the MMU (page 8 of [CCK14]), essentially showing that 1) for the sake of validity of steps of the same unit (CPU, MMU, etc.), MMU steps can always be moved forward because each MMU step makes more processor steps valid (this is monotonicity) and 2) MMU steps and CPU steps on the same unit can be reordered as long as the CPU steps do not access shared data structures with bypassing memory accesses, such as the PTO register or the TLB. A few simple flushing obligations then ensure that we never have to reorder MMU steps and CPU steps where the CPU executes such a bypassing access (e.g., by flushing before changing the PTO). As a result, they no longer need to flush the write buffer before the MMU executes a bypassing read to a buffered address, or before the MMU executes a bypassing shared store, or before the MMU executes a shared read (none of these can be realistically done by the programmer); instead they simply move the MMU step as if it was a global step of another unit. Whether a more general result exists for such autonomous components, or whether one would have to extend the model explicitly for MMUs that are allowed to run in strong memory mode, is not clear at all.

In many contexts it makes sense to use the fact that hardware write buffers have fixed length N, and thus if N write operations have been executed by a thread since its last shared write, that write has already left the write buffer and no shared writes are being buffered. It seems as if our reordering strategy maintains such limits of write buffers, i.e., every low-level computation is reordered to another low-level computation where at the beginning of the n-th step of object X, the write buffer of the unit that the object belongs to is not longer than the write buffer at the beginning of the n-th step of object X in the original schedule and thus also has no more than N elements; and also at every point in the proof where we use the software discipline on a reordered version of the original schedule, the write buffer of each unit has length N or less. One might thus strengthen our theorem by showing 1) that the software discipline only has to be obeyed in computations of the high-level machine where the write buffer length is bounded by N and 2) in the reordered schedule s_∞ write buffer length is bounded by N.

[1] Luckily in the case of MIPS, the MMU is disabled when the CPU is in kernel mode, and the page fault handler has to be written in kernel mode. This is also the case for multi-level MMUs, where the MMU of the hypervisor is disabled while the page fault handler of the hypervisor is running on the same core. Therefore the theorem presented here is strong enough for a MIPS operating system or hypervisor.

There are a few well-known code patterns that behave like a fence but are not well-understood by our discipline. For example, when a processor issues a store instruction and waits for a response by another processor, it is clear that the response can only come after the store has left the store buffer. Similarly, a processor that issues a store and then waits for the value to change will certainly wait until the store has reached the memory, as until then forwarding will ensure that the processor keeps seeing the same value. Finding ways to integrate such patterns into the reduction theorem – or prove a lower-level reduction theorem that shows that these situations behave like fences – may be useful, but this is not immediately clear, as we expect the performance penalty of additional fence instructions in these programs to be low.

There are other patterns that are not immediately sequentially consistent, but always eventually reach configurations reached by sequentially consistent executions. One of these is a barrier, where two threads wait for each other:

```
x.store(1);     || y.store(1);
while (!  y);   || while (!  x);
```

In TSO (but not in sequential consistency), there may be multiple failed attempts of both threads to leave the while-loop, but (as in sequential consistency) both threads will eventually leave the loop. By erasing the failed iterations of the loop, one obtains a sequentially consistent execution. One may thus call such an execution *quasi-SC*.

Compare this to a variant without loops, which is just plainly inconsistent (and often the textbook example of a sequentially inconsistent program)

```
x.store(1);  || y.store(1);
t1 = !  y;   || t2 = !  x;
```

It has been informally conjectured by Vafeiadis that in practice, "nobody programs like this". If that is the case, it should be possible to obtain a practical software discipline with no (or nearly no) memory barriers that only allows for quasi-SC executions; but this is still an open problem.

5.2 Related Work

Regaining sequential consistency on weak memory architectures is a real and thus very important problem. It is thus not surprising that a lot of very interesting research has been done on this field, and we only give a quick survey of the results that we find most important.

The only results, to the best of our knowledge, that come close to the hardware model we have presented here, is the result of Kovalev, Chen, and Cohen [CCK14], who considered TSO processors with word aligned accesses and memory management units that bypass the write buffer. The discipline is given in terms of ownership rather than races, and fences are introduced conservatively for communication between the MMU and the processor (e.g., when changing the page table origin or the mode). The result was extended to consider aligned sub-word accesses in the PhD thesis of Chen [Che16], but the definition of a partial hit was weakened to resolve the issue of races with RMW (cf. Condition RMWRace). In our model, we are only concerned about store buffer reduction while the memory management unit is not running, and we thus do not need to worry about interactions between the memory management unit and our processor. All of the fences inserted in the above results due to the MMU are replaced by a single fence when switching the memory mode. There are, to the best

of our knowledge, no other results for anything beyond simple user machines with aligned word accesses. From those, the closest to our work was done by Cohen and Schirmer [CS10], which serves as the basis of this work. Another interesting result in this class is given by Bouajjani et al. [BDM13], who describe a discipline which is optimal in the number of inserted fences, but is impractical. The key result is to look for executions which are sequentially consistent, but in TSO can be reordered easily into executions that are no longer sequentially consistent. Also worthy of mention is the work of Alglave et al. [AKNP14], who, unlike our result and most of the literature, consider considerably weaker memory models than TSO.

Bibliography

[ACHP10] Eyad Alkassar, Ernie Cohen, Mark Hillebrand, and Hristo Pentchev. Modular Specification and Verification of Interprocess Communication. In *Formal Methods in Computer Aided Design (FMCAD) 2010*. IEEE, 2010.

[AKNP14] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence. In *Computer Aided Verification*, pages 508–524. Springer, 2014.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *Programming Languages and Systems*, pages 533–553. Springer, 2013.

[BSDA14] Lennart Beringer, Gordon Stewart, Robert Dockins, and Andrew W Appel. Verified compilation for shared-memory c. In *European Symposium on Programming Languages and Systems*, pages 107–127. Springer Berlin Heidelberg, 2014.

[CA] Gordon Stewart Lennart Beringer Santiago Cuellar and Andrew W Appel. Compositional compcert.

[CCK14] Geng Chen, Ernie Cohen, and Mikhail Kovalev. Store buffer reduction with mmus. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools and Experiments*, volume 8471 of *Lecture Notes in Computer Science*, pages 117–132. Springer International Publishing, 2014.

[Che16] Geng Chen. *Store Buffer Reduction Theorem and Application*. PhD thesis, Saarland University, 2016.

[CL98] Ernie Cohen and Leslie Lamport. *Reduction in TLA*. Springer, 1998.

[CS10] Ernie Cohen and Bert Schirmer. From total store order to sequential consistency: A practical reduction theorem. pages 403–418, 2010.

[CV16] Soham Chakraborty and Viktor Vafeiadis. Validating optimizations of concurrent c/c++ programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 216–226. ACM, 2016.

[Deg11] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Saarland University, Saarbrücken, 2011.

[HIP05] Mark A. Hillebrand, Thomas In der Rieden, and Wolfgang Paul. Dealing with I/O Devices in the Context of Pervasive System Verification. In *Proceedings of the 2005 International Conference on Computer Design*, ICCD '05, pages 309–316, Washington, DC, USA, 2005. IEEE Computer Society.

[Hot12] Jenny Hotzkow. Store Buffers as an Alternative to Model Self Modifying Code. Bachelor's Thesis, Saarland University, Saarbrücken, 2012.

[Int] Intel Corporation. *Intel®64 and IA-32 Architectures, Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, 3C and 3D*.

[KMP14] Mikhail Kovalev, Silvia M Müller, and Wolfgang J Paul. *A Pipelined Multi-core MIPS Machine*. Springer, 2014.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Lut] Petro Lutsyk. Correctness of Multicore Processors with Operating System Support. PhD Thesis, to be published 2017 or 2018.

[Obe15] Jonas Oberhauser. *Verified Software: Theories, Tools, and Experiments: 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, chapter A Simpler Reduction Theorem for x86-TSO, pages 142–164. Springer International Publishing, Cham, 2015.

[Obe16] Jonas Oberhauser. *Verified Software: Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, Canada, July 17-18, 2016. Revised Selected Papers*, chapter Order Reduction for Interruptible Multi-Core Operating Systems. Springer International Publishing, Cham, 2016.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *European Conference on Object-Oriented Programming*, pages 478–503. Springer, 2010.

[Pau12] Wolfgang Paul. A Pipelined Multi Core MIPS Machine - Hardware Implementation and Correctness Proof. URL: `http://www-wjp.cs.uni-saarland.de/lehre/vorlesung/rechnerarchitektur2/ws1112/layouts/multicorebook.pdf`, 2012.

[PBLS16] Wolfgang J Paul, Christoph Baumann, Petro Lutsyk, and Sabine Schmaltz. *System Architecture: An Ordinary Engineering Discipline*. Springer, 2016.

[PLO] Wolfgang Paul, Petro Lutsyk, and Jonas Oberhauser. Multicore System Architecture. To be published as LNCS 9999.

[PSS98] A. Pnueli, M. Siegel, and F. Singerman. Translation validation. pages 151–166. Springer, 1998.

[Sch13] Sabine Schmaltz. Mips-86–a multi-core mips isa specification (november 4, 2013). Technical report, Saarland University, 2013.

[VN] VIKTOR VAFEIADIS and FRANCESCO ZAPPA NARDELLI. Compcerttso: A verified compiler for relaxed-memory concurrency.

[Zah16] Shahd Zahran. Implementing and debugging a pipelined mips machine with interrupts and multi-level address translation. Master's thesis, Saarland University, 2016.

Appendix A

Binary Arithmetic

In Chapter 3 and Appendix B we use notation for binary arithmetic from [PBLS16]. We reproduce the notation and definitions from that book, deviating from those definitions only slightly when this helps us speed things up.

The interpretation of a bit string $a \in \mathbb{B}^n$ as a binary number $\langle a \rangle$ is defined as follows

$$\langle a \rangle = \sum_{i<n} a_i \cdot 2^i.$$

The set of binary numbers that can be represented with n bits is denoted by B_n and can be defined as follows

$$B_n = [0 : 2^n).$$

Two integers are congruent modulo k if their difference is divisible by k

$$x \equiv y \mod k \iff \exists d \in \mathbb{Z}. x - y = k \cdot d.$$

Given an integer $x \in \mathbb{Z}$, we define its binary representation with n bits $bin_n(x)$ as the bit string of which the binary number interpretation is congruent to x modulo 2^n[1]

$$bin_n(x) = \varepsilon \{ a \in \mathbb{B}^n \mid \langle a \rangle \equiv x \mod 2^n \}.$$

We also use the following shorthand

$$x_n = bin_n(x).$$

The binary addition (substraction) of two strings $a, b \in \mathbb{B}^n$, denoted by $a +_n b$ ($a -_n b$), is the binary representation of the sum (difference) of the binary interpretations of the two bit strings

$$\circ \in \{ +, - \} \to a \circ_n b = (\langle a \rangle \circ \langle b \rangle)_n.$$

The interpretation of a bit string $a \in \mathbb{B}^n$ as a two's complement number $[a]$ is defined as follows

$$[a] = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2:0] \rangle.$$

[1] We differ here from the book in two ways: 1) we define the representation for all integers, and 2) we choose a representation that is congruent modulo 2^n. This way we can avoid defining functions *mod* and *tmod*, and simply use the binary representation directly when adding binary numbers. We do the same later for two's complement numbers.

The set of two's complement numbers that can be represented with n bits is denoted by T_n and can be defined as follows

$$T_n = [-2^{n-1} : 2^{n-1}).$$

The two's complement representation of a number $twoc_n(x)$ is easily defined as follows

$$twoc_n(x) = \varepsilon \{ a \in \mathbb{B}^n \mid [a] \equiv x \mod 2^n \}.$$

Note that binary and two's complement encoding are congruent modulo 2^n and thus the functions *twoc* and *bin* yield the same values

$$twoc_n(x) = bin_n(x).$$

We still have to define the functions for when they are used in Appendix B.

We lift operators $\vee, \wedge, \oplus$ normally defined on bits to bit strings of equal length $a, b \in \mathbb{B}^n$ by applying them pair wise

$$\circ \in \{ \vee, \wedge, \oplus \} \to (a \circ_{32} b)_i = a_i \circ b_i,$$

and similarly we lift negation to bit strings

$$(\neg a)_i = \neg(a_i).$$

For each bit string $a \in \mathbb{B}^n$ we define the zero-extension to $n' \geq n$ bits by filling the remaining bits with zeros

$$zxt_{n'}(a) = 0^{n'-n} \circ a,$$

and the sign-extension by filling the remaining bits with the most significant bit

$$sxt_{n'}(a) = a_{n-1}^{n'-n} \circ a.$$

Appendix B

MIPS86

The following is a verbatim copy of the technical report "MIPS 86" by Sabine Schmaltz; only formatting has been changed.

B.1 Formal Model of MIPS-86

This report provides a simple multi-core MIPS model we call MIPS-86. It aims at providing an overall specification for the reverse-engineered hardware models provided in [KMP14]. Essentially, we take the simple sequential MIPS processor model from [KMP14] and extend it with a memory and device model that resembles the one of modern x86 architectures. The model has the following features:

- Sequential processor core abstraction with atomic execute-and fetch transitions

  In order to justify modeling instruction execution by an atomic transition that combines fetching the instruction from memory and executing it, the absence of self-modifying code is a prerequisite. When instructions being fetched cannot be changed by the execution of other cores, these fetch cycles can be reordered to occur right before the corresponding execute cycle. This, in turn, means that the semantics of fetch and execute steps can be combined into single atomic steps.

- Memory-management unit (MMU) with translation-lookaside buffer (TLB)

  The memory-management unit considered performs a 2-level translation from virtual to physical addresses, caching partial and complete translations (which are called *walks*) in a *translation lookaside buffer* (TLB). The *page table origin*, i.e. the address of the first-level page table, is taken from the special purpose register pto. In order to allow an update of page-tables to be performed in a consistent manner, the machine is extended by two instructions: A *flush*-operation that empties the TLB of all walks, and an *invalidate-page*-operation that removes all walks to a certain virtual page address from the TLB.

- Store buffer (SB)

  In order to argue about store-buffer reduction, we provide a processor model with store-buffer. The store-buffer we consider is simple in the sense that it does not reorder or combine accesses but instead simply acts as a first-in-first-out queue for memory write operations to physical addresses. We provide two

serializing instructions: A *fence*-operation that simply drains the store-buffer, and a *compare-and-swap*-operation that atomically updates a memory word on a conditional basis while also draining the store-buffer.

- Processor-local advanced programmable interrupt controller (local APIC)

  In order to have a similar boot mechanism as the x86-architecture, we imitate the inter-processor-interrupt (IPI) mechanism of the x86-architecture. We extend our MIPS model by a strongly simplified local APIC for each processor. The local APIC provides interrupt signals to the processor and acts as a device for sending inter-processor-interrupts between processors. Local APIC ports are mapped into a processor's memory space by means of memory-mapped I/O.

- I/O APIC

  The I/O APIC is a component that is connected to the devices of the system and to the local APICs of the processors of the system. It provides the means to configure distribution of device interrupts to the processors of the multi-core system, i.e. whether a given device interrupt is masked and which processor will receive the interrupt. We do not consider edge-triggered interrupts, however, we do model the end-of-interrupt (EOI) protocol between local APIC and I/O APIC: After sending an interrupt signal to a local APIC, the I/O APIC will not sample a raised device interrupt again until the corresponding EOI message has been received from the local APIC.

- Devices

  We use a generic framework along the lines of the Verisoft device model [HIP05]: Device configurations are required to have a certain structure which can be instantiated, e.g. certain device transitions that specify side-effects associated with reading or writing device ports must be provided. Every device consists of ports and an interrupt line it may raise as well as some internal state that may be instantiated freely. Devices may interact with an external environment by receiving inputs and providing outputs.

In the following, we proceed by providing tables that give an overview over the instruction-set-architecture of MIPS-86, followed by operational semantics of the non-deterministic MIPS-86 model.

B.2 Instruction-Set-Architecture Overview and Tables

The instruction-set-architecture of MIPS-86 provides three different types of instructions: *I*-type instructions, *J*-type instructions and *R*-type instructions. *I*-type instructions are instructions that operate with two registers and a so-called *immediate constant*, *J*-type instructions are absolute jumps, and *R*-type instructions rely on three register operands.

B.2.1 Instruction Layout

The instruction-layout of MIPS-86 depends on the type of instruction. In the subsequent definition of the *MIPS-86* instruction layout, *rs*, *rt* and *rd* specify registers of the MIPS-86 machine.

***I*-Type Instruction Layout**

Bits	31 … 26	25 … 21	20 … 16	15 … 0
Field Name	opcode	*rs*	*rt*	immediate constant *imm*

***R*-Type Instruction Layout**

Bits	Field Name
31 … 26	opcode
25 … 21	rs
20 … 16	rt
15 … 11	rd
10 … 6	shift amount *sa*
5 … 0	function code fun

***J*-Type Instruction Layout**

Bits	31 … 26	25 … 0
Field Name	opcode	instruction index *iindex*

Effect of Instructions

A quick overview of available instructions is given in tables B.1 (for *I*-type), B.2 (for *J*-type), and B.3 (for *R*-type). Note that these tables – while giving a general idea what is available and what it approximately does – are not comprehensive. In particular, note that for all instructions whose mnemonic ends with ”u”, register values are interpreted as binary numbers whereas in all other cases they are interpreted as two’s-complement numbers. Note also that MIPS-86 is still incomplete in the sense that in order to accommodate the distributed cache model of the hardware construction, the architecture needs to be extended to allow proper management of cache-bypassing memory access (e.g. to devices). The abstract model provided here is one where caches are already abstracted into a view of a single coherent memory. The exact semantics of all instructions present in the provided instruction-set architecture tables is given later in the transition function of the MIPS-86 processor core.

B.2.2 Coprocessor Instructions and Special-Purpose Registers

Note that in contrast to most MIPS-architectures, in MIPS-86 coprocessor-instructions are provided as *R*-type instructions. Coprocessor instructions in MIPS-86 deal with moving data between special-purpose register file and general-purpose register file and exception return. The available special purpose registers of MIPS-86 are listed in table B.5.

Table B.1: *I*-Type Instructions of MIPS-86.

opcode	Mnemonic	Assembler-Syntax	d	Effect
Data Transfer				
100 000	lb	lb *rt rs imm*	1	rt = sxt(m[rs+imm])
100 001	lh	lh *rt rs imm*	2	rt = sxt(m[rs+imm])
100 011	lw	lw *rt rs imm*	4	rt = m[rs+imm]
100 100	lbu	lbu *rt rs imm*	1	rt = 0^{24}m[rs+imm]
100 101	lhu	lhu *rt rs imm*	2	rt = 0^{16}m[rs+imm]
101 000	sb	sb *rt rs imm*	1	m[rs+imm] = rt[7:0]
101 001	sh	sh *rt rs imm*	2	m[rs+imm] = rt[15:0]
101 011	sw	sw *rt rs imm*	4	m[rs+imm] = rt
Arithmetic, Logical Operation, Test-and-Set				
001 000	addi	addi *rt rs imm*		rt = rs + sxt(imm)
001 001	addiu	addiu *rt rs imm*		rt = rs + sxt(imm)
001 010	slti	slti *rt rs imm*		rt = (rs $<$ sxt(imm) ? 1 : 0)
001 011	sltiu	sltiu *rt rs imm*		rt = (rs $<$ sxt(imm) ? 1 : 0)
001 100	andi	andi *rt rs imm*		rt = rs $\wedge$ zxt(imm)
001 101	ori	ori *rt rs imm*		rt = rs $\vee$ zxt(imm)
001 110	xori	xori *rt rs imm*		rt = rs $\oplus$ zxt(imm)
001 111	lui	lui *rt imm*		rt = imm0^{16}

opcode	rt	Mnemonic	Assembler-S.	Effect
Branch				
000 001	00000	bltz	bltz *rs imm*	pc = pc + (rs $<$ 0 ? imm00 : 4)
000 001	00001	bgez	bgez *rs imm*	pc = pc + (rs $\geq$ 0 ? imm00 : 4)
000 100		beq	beq *rs rt imm*	pc = pc + (rs $=$ rt ? imm00 : 4)
000 101		bne	bne *rs rt imm*	pc = pc + (rs $\neq$ rt ? imm00 : 4)
000 110	00000	blez	blez *rs imm*	pc = pc + (rs $\leq$ 0 ? imm00 : 4)
000 111	00000	bgtz	bgtz *rs imm*	pc = pc + (rs $>$ 0 ? imm00 : 4)

B.2.3 Interrupts

Traditionally, hardware architectures provide an *interrupt* mechanism that allows the processor to react to events that require immediate attention. When an *interrupt signal* is raised, the hardware construction reacts by transferring control to an interrupt handler – on the level of hardware, this basically means that the program counter is set to the specific address where the hardware expects the interrupt handler code to be placed by the programmer and that information about the nature of the interrupt is provided in special registers. Since interrupts are mainly supposed to be handled by an operating system instead of by user processes, such a *jump to interrupt service routine* (JISR) step also tends to involve switching the processor to *system mode*.

Interrupts come in two major flavors: There are *internal interrupts* that are triggered by executing instructions, e.g. an overflow occuring in an arithmetic operation, a system-call instruction being executed (which is supposed to have the semantics of returning control to the operating system in order to perform some task requested by a user processor), or a page fault interrupt due to a missing translation in the page tables.

Table B.2: *J*-Type Instructions of MIPS-86

opcode	Mnemonic	Assembler-Syntax	Effect
Jumps			
000 010	j	j *iindex*	pc = bin_{32}(pc+4)[31:28]iindex00
000 011	jal	jal *iindex*	R31 = pc + 4, pc = bin_{32}(pc+4)[31:28]iindex00

In contrast, there are *external interrupts* which are triggered by an external source, e.g. the reset signal or device interrupts. Interrupts of lesser importance tend to be *maskable*, i.e. there is a control register that allows the programmer to configure that certain kinds of interrupts shall be ignored by the hardware.

The possible interrupt sources and priorities of MIPS-86 are listed in table B.6. Interrupts are either of type *repeat*, *abort*, or *continue*. Here *repeat* expresses that the interrupted instruction will be repeated after returning from the interrupt handler, *abort* means that the exception is usually so severe that the machine will be by default unable to return from the exception, and *continue* means that even though there is an interrupt, the execution of the interrupted execution will be completed before jumping to the interrupt-service-routine. In case of a *continue*-interrupt execution after exception return will proceed behind the interrupted execution. Note that the APIC mechanism is discussed in more detail in section B.8.1.

B.3 Overview of the MIPS-86-Model

B.3.1 Configurations

We define the set K of configurations of an abstract simplified multi-core MIPS machine in a top-down way. I.e., we first give a definition of the overall concurrent machine configuration which is composed of several subcomponent configurations whose definitions follow.

Definition 1 (Configuration of MIPS-86)**.** *A configuration*

$$c = (c.p, c.running, c.m, c.d, c.ioapic) \in K$$

of MIPS-86 is composed of the following components:

- *a mapping from processor identifier to processor configuration,* $c.p : [0 : np-1] \to K_p$*,*

  (np *is a parameter that describes the number of processors of the multi-core machine)*

- *a mapping from processor identifier to a flag that describes whether the processor is running,* $c.running : [0 : np-1] \to \mathbb{B}$*,*

  (If $c.running(i) = 0$*, this means that the processor is currently waiting for a startup-inter-processor-interrupt (SIPI))*

- *a shared global memory component* $c.m \in K_m$*,*

- *a mapping from device identifiers to device configurations,* $c.dev : [0 : nd-1] \to K_{dev}$*, and*

  (where $K_{dev} = \bigcup_{i=0}^{nd-1} K_{dev(i)}$ *is the union of individual device configurations, and* nd *is a parameter that describes the number of devices considered)*

- *an I/O APIC,* $c.ioapic \in K_{ioapic}$*.*

Processor

Definition 2 (Processor Configuration of MIPS-86)**.**

$$K_p = K_{core} \times K_{sb} \times K_{tlb} \times K_{apic}$$

A processor $p = (p.core, p.sb, p.tlb, p.apic) \in K_p$ *is subdivided into the following components:*

- *a processor core* $p.core \in K_{core}$*,*

  The processor core executes instructions according to the Instruction-Set-Architecture (ISA).

- *a store buffer* $p.sb \in K_{sb}$*,*

  A store-buffer buffers write accesses to the memory system local to the processor. If possible, read requests by the core are served by the store-buffer. Writes leave the store-buffer in the order they were placed (first-in-first-out).

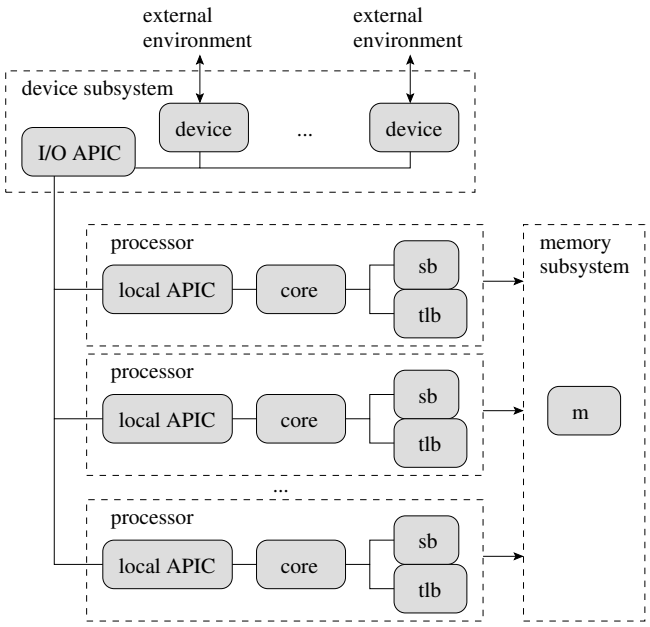


Figure B.1: Overview of MIPS-86 Model Components.

- *a TLB* $p.tlb \in K_{\mathbf{tlb}}$

  A translation-lookaside buffer (TLB) performs and caches address translations to be used by the processor in order to establish a virtual memory abstraction.

- *a local APIC* $p.apic \in K_{\mathbf{apic}}$

  A local APIC receives interrupt signals from the I/O APIC and provides them to the processor. Additionally, it acts as a processor-local device that can send inter-processor-interrupts (IPI) to other processors of the system.

Definitions of $K_{\mathbf{core}}, K_{\mathbf{sb}}, K_{\mathbf{tlb}}$, and $K_{\mathbf{apic}}$ are each given in the section that defines the corresponding component in detail.

Memory

Definition 3 (Memory Configuration of MIPS-86)**.** *For this abstract machine, we consider a simple byte-addressable shared global memory component*

$$K_{\mathbf{m}} \equiv \mathbb{B}^{32} \to \mathbb{B}^{8}$$

which is sequentially consistent.

Definition 4 (Reading Byte-Strings from Byte-Addressable Memory)**.** *For a memory* $m \in K_{\mathbf{m}}$ *and an address* $a \in \mathbb{B}^{32}$ *and a number* $d \in \mathbb{N}$ *of Bytes, we define*

$$m_d(a) = \begin{cases} m_{d-1}(a +_{32} 1_{32}) \circ m(a) & d > 0 \\ \varepsilon & d = 0 \end{cases}$$

B.3.2 Transitions

We define the semantics of the concurrent MIPS machine MIPS-86 as an automaton with a partial transition function

$$\delta : K \times \Sigma \rightharpoonup K$$

and an output function

$$\lambda : K \times \Sigma \rightharpoonup \Omega$$

where Σ is the set of inputs to the automaton and Ω is the set of outputs of the automaton. In particular, in order to be able to define the semantics of our system as a deterministic automaton, these inputs do include scheduling information, i.e. they determine exactly which subcomponent makes what step. Note that, in the following sections, we will first define the semantics of all individual components before we give the definition of δ and λ in section B.11.

Scheduling

We provide a model in which the execution order of individual components of the system is not known to the programmer. In order to prove correct execution of code, it is necessary to consider all possible execution orders given by the model. We model this non-deterministic behavior by deterministic automata which take, as part of their input, information about the execution order. This is done in such a way that, in every step, it is specified exactly which subcomponent of the overall system makes which particular step.

There are occasions where several components make a synchronous step. In such cases, our intuition is that one very specific subcomponent *actively* performs a step, while all other components make a *passive* step that merely responds to the active component in some way. The memory, in particular, is such a passive component. Also, devices can react passively to having their registers read by a processor, causing a side-effect on reading.

B.4 Memory

Definition 5 (Memory Transition Function)**.** *We define the memory transition function*

$$deltam : K_{\boldsymbol{m}} \times \Sigma_{\boldsymbol{m}} \rightharpoonup K_{\boldsymbol{m}}$$

where

$$sigmam = \mathbb{B}^{32} \times (\mathbb{B}^8)^* \cup \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{32}$$

Here,

- $(a,v) \in \mathbb{B}^{32} \times (\mathbb{B}^8)^*$ *– describes a write access to address a with value v, and*
- $(c,a,v) \in (\mathbb{B}^{32})^3$ *– describes a read-modify-write access to address a with compare-value c and value v to be written in case of success.*

We have

$$\delta_{\boldsymbol{m}}(m,in)(x) = \begin{cases} byte(\langle x\rangle - \langle a\rangle, v) & in = (a,v) \wedge 0 \leq \langle x\rangle - \langle a\rangle < \boldsymbol{len}(v)/8 \\ byte(\langle x\rangle - \langle a\rangle, v) & in = (c,a,v) \wedge m_4(a) = c \wedge 0 \leq \langle x\rangle - \langle a\rangle < 4 \\ m(x) & otherwise \end{cases}$$

Note that, in this memory model, we assume that read accesses cannot change the memory state – thus, any result of a read operation can simply be computed directly from a given memory configuration. In a more concrete model where caches are still visible, we need to consider memory reads as explicit inputs to the cache system.

B.5 TLB

B.5.1 Address Translation

Most processors, including MIPS-86 and x86-64, provide *virtual memory* which is mostly used for implementing process separation in the context of an operating system. By performing *address translation* from virtual memory addresses to physical memory addresses (i.e. regular memory addresses of the machine's memory system), the notion of a virtual memory is established – if this translation is injective, virtual memory has regular memory semantics (i.e. writing to an address affects only this single address and values being written can be read again later). Mostly, this is used to establish several virtual address spaces that are mapped to disjoint address regions in the physical memory of the machine. User processes of the operating system can then each be run by the operating system in their respective address spaces without any risk of user processes affecting each other or the operating system.

Processors tend to provide a mechanism to activate and deactivate address translation – usually by writing some special control register. In the case of MIPS-86, a special-purpose-register *mode* is provided which decides whether the processor is running in *system mode*, i.e. without address translation, or in *user mode*, i.e. with address translation.

The translation from virtual addresses to physical addresses is usually given at the granularity of *memory pages* – in the case of MIPS-86, a *memory page* consists of 2^{12} consecutive bytes. Since MIPS-86 is a 32-bit architecture, a *page address* thus consists of 20 Bits. Defining a particular translation from virtual addresses to phyiscal addresses is done by establishing *page tables* that describe the translation. In the most simple case, a single-level translation can be given by a single page table – which is a sequence of page table entries that each describe how – and if – a given *virtual page address* is translated to a corresponding *physical page address*. The translation tends to be partial, i.e. not every virtual page address has a corresponding physical page address, which is reflected in the page table entry by the *present bit*. Trying to access a virtual address in user mode that does not have a translation set up in the page table according to the present bit then results in a *page-fault* interrupt which returns the processor to system mode – e.g. allowing the operating system to set up a translation for the faulting virtual page address.

MIPS-86, like x86-64, applies a *multi-level page table hierarchy*: Instead of translating using a single page table that describes the virtual page address to physical page address translation, there are several levels of page tables. One advantage of this is that multi-level page tables tend to require less memory space: Instead of providing a page table entry for every virtual page address, the page tables now form a graph in such a way that every level of page tables effectively describes a part of the page address translation by linking to page tables belonging to the next level of translation. Since only a part of the translation is provided, these page tables are much smaller than a single-level translation page table. MIPS-86 provides 2 levels of address translation (in comparison, x86-64 makes use of 4 levels). The first level page table – also called *page*

directory – translates the first 10 Bits of a page address by describing where the page tables for translating the remaining 10 Bits can be found, i.e. they contain addresses of second level page tables. These *terminal page tables* then provide the physical page addresses. Note that, in defining a multi-level translation via page tables, page table entries can be marked as not present in early translation levels which essentially means that no further page tables need to be provided for the given virtual page address prefix – which effectively is the reason why multi-level translations tend to require less memory space.

The actual translation is performed in hardware by introducing a circuit called *memory management unit* (MMU) which serves translation requests from the processor by accessing the page tables residing in memory. In a naive hardware implementation of address translation, the processor running in user mode could simply issue translation requests to the MMU in order as needed for instruction execution and wait for the MMU circuit to respond with a translation. Such a synchronous implementation however, would mean that the processor is constantly waiting for the MMU to perform translations, limiting the speed of execution to that of the MMU performing as many memory accesses as needed to compute the translations needed for instruction fetch and execution. Fortunately, however, it can be observed that instruction fetch in user mode to a large degree tends to require translations for virtual addresses that lie in the same page (with an appropriate programming style this is also mostly true for memory accesses performed by instructions), thus, in order not to constantly have the MMU repeat a translation for the same virtual page address, it might be helpful to keep translations available to the processor in a special processor-local cache for translations. This cache is commonly called *translation lookaside buffer* (TLB) and is updated by the MMU whenever necessary in order to serve translation requests by the processor. Note that a hardware TLB may cache partial translations for virtual page address prefixes in order to reuse them later.

Since the operating system may modify the page tables, translations in the TLB may become outdated – removing or changing translations provided by the page tables can make the TLB inconsistent with the page tables. Thus, architectures with TLB tend to provide instructions that allow the processor to control the state of the TLB to some degree. The functionality needed in order to keep the TLB consistent with the page tables is in fact quite simple: In order to ensure that all translations present in the TLB are also given by the page tables, all we need is to be able to remove particular translations (or all translations) from the TLB. Both x86-64 and MIPS-86 provide such instructions – for MIPS-86, $invlpg$ invalidates a single virtual page address, while $flush$ removes all translations from the TLB.

B.5.2 TLB Configuration

When the MIPS-86 processor is running in user mode, all memory accesses are subject to address translation according to page tables residing in memory. In order to perform address translation, the MMU operates on the page tables to create, extend, complete, and drop walks. A complete walk provides a translation from a virtual address to a physical address of the machine that can in turn be used by the processor core. Our TLB offers *address space identifier*s – a tag that can be used to associate translations with particular users – which reduces the need for TLB flushes when switching between users.

Definition 6 (TLB Configuration of MIPS-86)**.** *We define the set of configurations of*

a TLB as

$$Ktlb = 2^{K_{walk}}$$

where the set of walks K_{walk} *is given by*

$$Kwalk = \mathbb{B}^{20} \times \mathbb{B}^{6} \times \{0,1,2\} \times \mathbb{B}^{20} \times \mathbb{B}^{3} \times \mathbb{B}$$

The components of a walk $w = (w.va, w.asid, w.level, w.ba, w.r, w.fault) \in K_{walk}$ *are the following:*

- $wva \in \mathbb{B}^{20}$ *– the virtual page address to be translated,*
- $wasid \in \mathbb{B}^{6}$ *– the* address space identifier *(ASID) the translation belongs to,*
- $wlevel \in \{0,1,2\}$ *– the current level of the walk, i.e. the number of remaining walk extensions needed to complete the walk,*
- $wbawalk \in \mathbb{B}^{20}$ *– the physical page address of the page table to be accessed next, or, if the walk is complete, the result of the translation,*
- $wr \in \mathbb{B}^{3}$ *– the accumulated access rights, and*

  Here, $r[0]$ *stands for write permission,* $r[1]$ *for user mode access, and* $r[2]$ *expresses execute permission.*
- $wfault \in \mathbb{B}$ *– a page fault indicator.*

B.5.3 TLB Definitions

In the following, we make definitions that describe the structure of page tables and the translation function specified by a given page table origin according to a memory configuration. Addresses are split in two page index components px_2, px_1 and a byte offset px_0 within a page:

$$a = a.px_2 \circ a.px_1 \circ a.px_0$$

Definition 7 (Page and Byte Index)**.** *Given an address* $a \in \mathbb{B}^{32}$*, we define*

- *the second-level page index* $apx2 = a[31:22]$*,*
- *the first-level page index* $apx1 = a[21:12]$*, and*
- *the byte offset* $apx0 = a[11:0]$

of a.

Definition 8 (Base Address (Page Address))**.** *The* base address *(also sometimes referred to as* page address*) of an address* $a \in \mathbb{B}^{32}$ *is then given by*

$$aba = a.px_2 \circ a.px_1.$$

Definition 9 (Page Table Entry)**.** *A* page table entry $pte \in \mathbb{B}^{32}$ *consists of*

- $ptebapte = pte[31:12]$ *– the base address of the next page table or, if the page table is a terminal one, the resulting physical page address for a translation,*
- $pteppte = pte[11]$ *– the present bit,*

- $pterpte = pte[10:8]$ – *the access rights for pages accessed via a translation that involves the page table entry,*
- $pteapte = pte[7]$ – *the accessed flag that denotes whether the MMU has already used the page table entry for a translation, and*

Definition 10 (Page Table Entry Address)**.** *For a base address* $ba \in \mathbb{B}^{20}$ *and an index* $i \in \mathbb{B}^{10}$*, we define the corresponding* page table entry address *as*

$$ptea(ba, i) = ba \circ 0^{12} +_{32} 0^{20} i00$$

The page table entry address needed to extend a given walk $w \in K_{walk}$ *is then defined as*

$$ptea = ptea(w.ba, (w.va \circ 0^{12}).px_{w.level})$$

Definition 11 (Page Table Entry for a Walk)**.** *Given a memory* $m \in K_{mem}$ *and a walk* $w \in K_{walk}$*, we define the* page table entry needed to extend a walk *as*

$$pte = m_4(ptea(w))$$

Definition 12 (Walk Creation)**.** *We define the function*

$$winit : \mathbb{B}^{20} \times \mathbb{B}^{20} \times \mathbb{B}^{6} \to K_{walk}$$

which, given a virtual base address $va \in \mathbb{B}^{20}$*, the base address* $pto \in \mathbb{B}^{20}$ *of the page table origin and an address space identifier* $asid \in \mathbb{B}^{6}$*, returns the* initial walk *for the translation of va.*

$$winit = w$$

is given by

$$w.va = va$$
$$w.asid = asid$$
$$w.level = 2$$
$$w.ba = pto$$
$$w.r = 111$$
$$w.fault = 0$$

Note that in our specification of the MMU, the initial walk always has full rights ($w.r = 111$). However, in every translation step, the rights associated with the walk can be restricted as needed by the translation request made by the processor core.

Definition 13 (Sufficient Access Rights)**.** *For a pair of access rights* $r, r' \in \mathbb{B}^3$*, we use*

$$le \quad \stackrel{def}{\Leftrightarrow} \quad \forall j \in [0:2] : r[j] \leq r'[j]$$

to describe that the access rights r are weaker than r'*, i.e. rights* r' *are sufficient to perform an access with rights r.*

Definition 14 (Walk Extension)**.** *We define the function*

$$wext : K_{walk} \times \mathbb{B}^{32} \times \mathbb{B}^3 \to K_{walk}$$

which extends a given walk $w \in K_{walk}$ using a page table entry $pte \in \mathbb{B}^{32}$ and access rights $r \in \mathbb{B}^3$ in such a way that

$$wext = w'$$

is given by

$$w'.va = w.va$$

$$w'.asid = w.asid$$

$$w'.level = \begin{cases} w.level - 1 & pte.p \\ w.level & otherwise \end{cases}$$

$$w'.ba = \begin{cases} pte.ba & pte.p \\ w.ba & otherwise \end{cases}$$

$$w'.r = \begin{cases} pte.r & pte.p \\ w.r & otherwise \end{cases}$$

$$w'.fault = \neg pte.p \vee \neg r \leq pte.r$$

Note that, in the original x86 model, in addition to restricting the rights according to the rights set in the page table entry used to extend the walk, there was the possibility to restrict the rights of a walk even further during walk extension. This has something to do with the fact that translation requests that do not need write rights would not need to set a dirty flag in the page table entry. In this model, however, we only model the accessed bit and not the dirty bit.

Definition 15 (Complete Walk)**.** *A walk $w \in K_{walk}$ with $w.level = 0$ is called a* complete walk*:*

$$complete \equiv w.level = 0$$

Definition 16 (Setting Accessed Flag of a Page Table Entry)**.** *Given a page table entry $pte \in \mathbb{B}^{32}$, we define the function*

$$set - ad = pte[a := 1]$$

which returns an updated page table entry in which the accessed bit is set.

Definition 17 (Translation Request)**.** *A translation request*

$$trq = (trq.asid, trq.va, trq.r) \in \mathbb{B}^6 \times \mathbb{B}^{32} \times \mathbb{B}^3$$

is a triple of

- *address space identifier $trqasidtrq \in \mathbb{B}^6$,*
- *virtual address $trqvatrq \in \mathbb{B}^{32}$, and*
- *access rights $trqrtrq \in \mathbb{B}^3$.*

Definition 18 (TLB Hit)**.** *When a walk w matches a translation request trq in terms of virtual address, address space identifier and access rights, we call this a* TLB hit*:*

$$hit \equiv w.va = trq.va[31:12] \wedge w.asid = trq.asid \wedge trq.r \leq w.r$$

Note, that a hit may be to an incomplete walk.

Definition 19 (Page-Faulting Walk Extension)**.** *A page fault for a given translation request can occur for a given walk when extending that walk would result in a fault: The page table entry needed to extend is not present or the translation would require more access rights than the page table entry provides. To denote this, we define the predicate*

$$\begin{aligned} faultwalk \equiv \quad & /complete(w) \wedge hit(trq, w) \\ & \wedge wext(w, pte(m, w), trq.r).fault \end{aligned}$$

which, given a memory m, a translation request trq and a walk w, is fulfilled when walk extension for walk w under translation request trq in memory configuration m page-faults.

Note that a page fault may occur at any translation level. However, the TLB will only store non-faulting walks (this is an invariant of the TLB) – page faults are always triggered by considering a faulting extension of a walk in the TLB.

How page faults are triggered is defined in the top-level transition function of MIPS-86 as follows: the processor core always chooses walks from the TLB non-deterministically to either obtain a translation, or, to get a page-fault when the chosen walk has a page faulting walk extension. Note that, when a page-fault for a given pair of virtual address and address space identifier occurs, MIPS-86 flushes all corresponding walks from the TLB. Another side-effect of page-faults in the pipelined hardware implementation is that the pipeline is drained. Since the MIPS-86 model provides a model of sequential instruction execution, draining the pipeline cannot be expressed on this level, however, this behavior is needed in order to be able to prove that the pipelined implementation indeed behaves as specified by MIPS-86.

Definition 20 (Transition Function of the TLB)**.** *We define the transition function of the TLB that states the passive transitions of the TLB*

$$deltatlb : K_{\mathit{tlb}} \times \Sigma_{\mathit{tlb}} \to K_{\mathit{tlb}}$$

where

$$sigmatlb = \{\textbf{\textit{flush}}\} \times \mathbb{B}^6 \times \mathbb{B}^{20} \cup \{\textbf{\textit{flush-incomplete}}\} \cup \{\textbf{\textit{add-walk}}\} \times K_{\mathit{walk}}$$

as a case distinction on the given input:

- *flushing a virtual address for a given address space identifier:*

$$\delta_{\mathit{tlb}}(tlb, (\textbf{\textit{flush}}, asid, va)) = \{w \in tlb \mid \neg(w.asid = asid \wedge w.va = va)\}$$

- *flushing all incomplete walks from the TLB:*

$$\delta_{\mathit{tlb}}(tlb, \textbf{\textit{flush-incomplete}}) = \{w \in tlb \mid complete(w)\}$$

- *adding a walk:*

$$\delta_{\mathit{tlb}}(tlb, (\textbf{\textit{add-walk}}, w)) = tlb \cup \{w\}$$

B.6 Processor Core

Definition 21 (Processor Core Configuration of *MIPS-86*)**.** *A* MIPS-86 *processor core configuration* $c = (c.pc, c.gpr, c.spr, c.HI, c.LO) \in Kcore$ *consists of*

- *a program counter:* $cpc \in \mathbb{B}^{32}$,
- *a general purpose register file:* $cgpr : \mathbb{B}^5 \to \mathbb{B}^{32}$,
- *a special purpose register file:* $cspr : \mathbb{B}^5 \to \mathbb{B}^{32}$, *and*
- *multiplication accumulator registers:* $cHI, c.LO \in \mathbb{B}^{32}$

Definition 22 (Processor Core Transition Function of *MIPS-86*)**.** *We define the processor core transition function*

$$deltacore : K_{core} \times \Sigma_{core} \rightharpoonup K_{core}$$

which takes a processor core input from

$$sigmacore = \Sigma_{instr} \times \Sigma_{eev} \times \mathbb{B} \times \mathbb{B}$$

where

$$sigmainstr = \mathbb{B}^{32} \times (\mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\})$$

is the set of inputs required for instruction execution, i.e. a pair of instruction word $I \in \mathbb{B}^{32}$ *and value* $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\}$ *read from memory (which is only needed for read or cas instructions), and*

$$sigmaeev = \mathbb{B}^{256}$$

is used to represent a vector $eev \in \mathbb{B}^{256}$ *of interrupt signals provided by the local APIC. Also, we explicitly pass the page fault fetch and page fault load/store signals* $pff, pfls \in \mathbb{B}$.

We define the processor core transition function

$$\delta_{core}(c, I, R, eev, pff, pfls) = \begin{cases} \delta_{jisr}(c, I, R, eev, pff, pfls) & jisr(c, I, eev, pff, pfls) \\ \delta_{rfe}(c) & eret(I) \wedge \neg jisr(c, I, eev, pff, pfls) \\ \delta_{instr}(c, I, R) & otherwise \end{cases}$$

as a case distinction on the jump-interrupt-service-routine-*signal* $jisr$ *(for definition, see B.6.3) which formalizes whether an interrupt is triggered in the current step of the machine and the* return-from-exception-*signal* rfe *which is active when the next instruction to be executed is* rfe.

In the definition above, we use the auxiliary transition functions

$$\delta_{\mathbf{instr}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{instr}} \rightharpoonup K_{\mathbf{core}}$$

which executes a non-interrupted instruction of the instruction set architecture (for definition, see section B.6.2),

$$\delta_{\mathbf{jisr}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{core}} \to K_{\mathbf{core}}$$

which is used to specify the state the core reaches when an interrupt is triggered (for definition, see section B.6.4), and

$$\delta_{\mathbf{rfe}} : K_{\mathbf{core}} \to K_{\mathbf{core}}$$

which specifies the return-from-exception transition (for definition, see section B.6.4).

B.6.1 Auxiliary Definitions for Instruction Execution

In the following, we make auxiliary definitions in order to define the processor core transitions that deal with instruction execution. In order to execute an instruction, the processor core needs to read values from the memory. Of relevance to instruction execution is the instruction word $I \in \mathbb{B}^{32}$ and, if the instruction I is a *read* or *cas* instruction, we need the value $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ read from memory.

Instruction Decoding

Definition 23 (Fields of the Instruction Layout)**.** *Formalizing the tables given in subsection B.2.1, we define the following shorthands for the fields of the MIPS-86 instruction layout:*

- instruction opcode
$$opc = I[31:26]$$
- instruction type
$$rtype \equiv opc(I) = 0^6 \vee opc(I) = 010^4 \vee opc(I) = 01^3 0^2$$
$$jtype \equiv opc(I) = 0^4 10 \vee opc(I) = 0^4 11$$
$$itype \equiv \overline{rtype(I) \vee jtype(I)}$$
- register addresses
$$rs = I[25:21]$$
$$rt = I[20:16]$$
$$rd = I[15:11]$$
- shift amount
$$sa = I[10:6]$$
- function code *(used only for R-type instructions)*
$$fun = I[5:0]$$
- immediate constants *(for I-type and J-type instructions, respectively)*
$$imm = I[15:0]$$
$$iindex = I[25:0]$$

Definition 24 (Instruction-Decode Predicates)**.** *For every MIPS-Instruction, we define a predicate on the MIPS-configuration which is true iff the corresponding instruction is to be executed next. The name of such an instruction-decode predicate is always the instruction's mnemonic (see MIPS ISA-tables at the beginning). Formally, the predicates check for the corresponding opcode and function code. E.g.*

$$lw(I) \equiv opc(I) = 100011$$

$$\ldots$$

$$add(I) \equiv rtype(I) \wedge fun(I) = 100000$$

The instruction-decode predicates are so trivial to formalize that we do not explicitly list all of them here.

Definition 25 (Illegal Opcode)**.** *Let*

$$ill = \neg(lw(I) \vee \ldots \vee add(I))$$

be the predicate that formalizes that the opcode of instruction I is illegal by negating the disjunction of all instruction-decode predicates.

Note that, encountering an illegal opcode during instruction execution, an illegal instruction interrupt will be triggered.

Arithmetic and Logic Operations

The *arithmetic logic unit* (ALU) of *MIPS-86* behaves according to the following table:

alucon[3:0]	i	alures	ovf
0 000	*	$a +_{32} b$	0
0 001	*	$a +_{32} b$	$[a] + [b] \notin T_{32}$
0 010	*	$a -_{32} b$	0
0 011	*	$a -_{32} b$	$[a] - [b] \notin T_{32}$
0 100	*	$a \wedge_{32} b$	0
0 101	*	$a \vee_{32} b$	0
0 110	*	$a \oplus_{32} b$	0
0 111	0	$\neg_{32}(a \vee_{32} b)$	0
0 111	1	$b[15:0]0^{16}$	0
1 010	*	$0^{31}([a] < [b]?1:0)$	0
1 011	*	$0^{31}(\langle a \rangle < \langle b \rangle ?1:0)$	0

Based on inputs $a, b \in \mathbb{B}^{32}, alucon \in \mathbb{B}^4$ and $i \in \mathbb{B}$, this table defines $alures \in \mathbb{B}^{32}$ and $ovf \in \mathbb{B}$.

Definition 26 (ALU Instruction Predicates)**.** *To describe whether a given instruction $I \in \mathbb{B}^{32}$ performs an arithmetic or logic operation, we define the following predicates:*

- *I-type ALU instruction:* $compi \equiv itype(I) \wedge I[31:29] = 001$
- *R-type ALU instruction:* $compr \equiv rtype(I) \wedge I[5:4] = 10$
- *any ALU instruction:* $alu \equiv compi(I) \vee compr(I)$

Definition 27 (ALU Operands of an Instruction)**.** *Following the instruction set architecture tables, we formalize the right and left operand of an ALU instruction $I \in \mathbb{B}^{32}$ based on a given processor core configuration $c \in K_{core}$ as follows:*

- *left ALU operand:* $lop = c.gpr(rs(I))$
- *right ALU operand:* $rop = \begin{cases} c.gpr(rt(I)) & rtype(I) \\ sxt_{32}(imm(I)) & /rtype(I) \wedge /I[28] \\ zxt_{32}(imm(I)) & otherwise \end{cases}$

Definition 28 (ALU Control Bits of an Instruction)**.** *We define the ALU control bits of an instruction* $I \in \mathbb{B}^{32}$ *as*

$$alucon[2:0] = \begin{cases} I[2:0] & rtype(I) \\ I[28:26] & otherwise \end{cases}$$

$$alucon(I)[3] \equiv rtype(I) \wedge I[3] \vee /I[28] \wedge I[27]$$

Definition 29 (ALU Compute Result)**.** *The ALU result of an instruction I executed in processor core configuration* $c \in K_{\boldsymbol{core}}$ *is then given by*

$$compres = alures(lop(c,I), rop(c,I), alucon(I), itype(I))$$

Multiplication Operations

The *multiplication unit* of *MIPS-86* takes inputs a, b, c, d, and $mulcon$ and behaves according to the following table, providing $mures(a,b,c,d,mulcon)$:

mulcon[2:0]	mures
000	$twoc_{64}([a] \cdot [b])$
001	$bin_{64}(\langle a \rangle \cdot \langle b \rangle)$
100	$cd +_{64} twoc_{64}([a] \cdot [b])$
101	$cd +_{64} bin_{64}(\langle a \rangle \cdot \langle b \rangle)$
110	$cd -_{64} twoc_{64}([a] \cdot [b])$
111	$cd -_{64} bin_{64}(\langle a \rangle \cdot \langle b \rangle)$

Definition 30 (Multiplication Control Bits)**.** *The multiplication control bits are*

$$mulcon \equiv (opc(I)[2] \oplus fun(I)[1]) \circ fun[2] \circ fun[0]$$

Definition 31 (Multiplication Result)**.** *We define the* multiplication result *as*

$$mres = mures(c.gpr(rs(I)), c.gpr(rt(I)), c.HI, c.LO, mulcon(I))$$

Definition 32 (Multiplication Accumulator Register Instructions)**.** *The predicate*

$$mulacc = mult(I) \vee multu(I) \vee madd(I) \vee maddu(I) \vee msub(I) \vee msubu(I)$$

is true whenever the machine is about to execute a multiplication operation that writes a result to the HI and LO registers.

Jump and Branch Instructions

Jump and branch instructions affect the program counter of the machine. The difference between branch instructions and jump instructions is that branch instructions perform conditional jumps based on some condition expressed over general purpose register values. The following table defines the branch condition result $bcres \in \mathbb{B}$, i.e. whether for the given parameters the branch will be performed or not, based on inputs $a, b \in \mathbb{B}^{32}$ and $bcon \in \mathbb{B}^4$:

bcon[3:0]	bcres(a, b, bcon)
001 0	$[a] < 0$
001 1	$[a] \geq 0$
100 *	$a = b$
101 *	$a \neq b$
110 *	$[a] \leq 0$
111 *	$[a] > 0$

Definition 33 (Branch Instruction Predicates)**.** *We define the following branch instruction predicates that denote whether a given instruction $I \in \mathbb{B}^{32}$ is a jump or successful branch instruction given configuration $c \in K_{core}$:*

- *branch instruction:* $b \equiv opc(I)[5:3] = 0^3 \wedge itype(I)$
- *jump instruction:* $jump \equiv j(I) \vee jal(I) \vee jr(I) \vee jalr(I)$
- *jump or branch taken:*

$$jbtaken \equiv jump(I) \vee b(I) \wedge bcres(c.gpr(rs(I)), c.gpr(rt(I)), opc[2:0]rt(I)[0])$$

Definition 34 (Branch Target)**.** *We define the target address of a jump or successful branch instruction $I \in \mathbb{B}^{32}$ in a given configuration $c \in K_{core}$ as*

$$btarget \equiv \begin{cases} c.pc +_{32} sxt_{30}(imm(I))00 & b(I) \\ c.gpr(rs(I)) & jr(I) \vee jalr(I) \\ (c.pc +_{32} 4_{32})[31:28]iindex(c)00 & j(I) \vee jal(I) \end{cases}$$

Shift Operations

Shift instructions perform shift operations on general purpose registers.

Definition 35 (Shift Results)**.** *For $a[n-1:0] \in \mathbb{B}^n$ and $i \in \{0,\ldots,n-1\}$ we define the following shift results ($\in \mathbb{B}^n$):*

- *shift left logical:* $sll = a[n-i-1:0]0^i$
- *shift right logical:* $srl = 0^i a[n-1:i]$
- *shift right arithmetic:* $sra = a^i_{n-1} a[n-1:i]$

Note that, for MIPS-86, we will use the aforementioned definitions only for $n = 32$.

Definition 36 (Shift Unit Result)**.** *We define the result of a shift operation based on inputs $a \in \mathbb{B}^n, i \in \{0,\ldots,n-1\}$, and $sf \in \mathbb{B}^2$ as follows:*

$$sures = \begin{cases} sll(a,i) & sf = 00 \\ srl(a,i) & sf = 10 \\ sra(a,i) & sf = 11 \end{cases}$$

Definition 37 (Shift Instruction Predicate)**.** *We define a predicate that, given an instruction $I \in \mathbb{B}^{32}$, expresses whether the instruction is a shift instruction by a simple disjunction of shift instruction predicates:*

$$su \equiv sll(I) \vee srl(I) \vee sra(I) \vee sllv(I) \vee srlv(I) \vee srav(I)$$

Definition 38 (Shift Operands)**.** *Given a shift instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{core}$, we define the following shift operands:*

- *shift distance:* $sdist = \begin{cases} \langle sa(I) \rangle \, \mathbf{mod} \, 32 & fun(I)[3] = 0 \\ \langle c.gpr(rs(I))[4:0] \rangle \, \mathbf{mod} \, 32 & fun(I)[3] = 1 \end{cases}$
- *shift left operand:* $slop = c.gpr(rt(I))$

Definition 39 (Shift Function)**.** *The shift function of a shift instruction $I \in \mathbb{B}^{32}$ is given by*

$$sf = I[1:0]$$

Memory Accesses

We define auxiliary functions that we need in order to define how values are read/written from/to the memory in the overall system's transition function.

Definition 40 (Effective Address and Access Width)**.** *Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{core}$, we define the effective address and access width of a memory access:*

- *effective address:* $ea = \begin{cases} c.gpr(rs(I)) +_{32} sxt_{32}(imm(I)) & itype(I) \\ c.gpr(rs(I)) & rtype(I) \end{cases}$
- *access width:* $d = \begin{cases} 1 & lb(I) \vee lbu(I) \vee sb(I) \\ 2 & lh(I) \vee lhu(I) \vee sh(I) \\ 4 & sw(I) \vee lw(I) \vee cas(I) \end{cases}$

The effective address is the first byte address affected by the memory address and the access width is the number of bytes which are read, or, respectively, written.

Definition 41 (Misalignment Predicate)**.** *For an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{core}$, we define the predicate*

$$\begin{aligned} mal \quad \equiv \quad & (lw(I) \vee sw(I) \vee cas(I)) \wedge ea(c,I)[1:0] \neq 00 \\ & \vee (lhu(I) \vee lh(I) \vee sh(I)) \wedge ea(c,I)[0] \neq 0 \end{aligned}$$

that describes whether the memory access is misaligned. To be correctly aligned, the effective address of the memory access must be divisible by the access width.

Note that misaligned memory access triggers the corresponding interrupt.

Definition 42 (Load/Store Instruction Predicates)**.** *In order to denote whether a given instruction $I \in \mathbb{B}^{32}$ is a load or store instruction, we define the following predicates:*

- *load instruction:* $load \equiv lw(I) \vee lhu(I) \vee lh(I) \vee lbu(I) \vee lb(I)$
- *store instruction:* $store \equiv sw(I) \vee sh(I) \vee sb(I)$

Definition 43 (Load Value)**.** *The value read from memory $R \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is given as an input to the transition function of the processor core. In order to write this value to a general purpose register, depending on the memory instruction used, we either need to sign-extend or zero-extend this value:*

$$lv = \begin{cases} zxt_{32}(R) & lbu(I) \vee lhu(I) \\ sxt_{32}(R) & \end{cases}$$

Definition 44 (Store Value)**.** *Given an instruction $I \in \mathbb{B}^{32}$ and a processor core configuration $c \in K_{core}$, the store value is given by the last $d(I)$ bytes taken from the general purpose register specified by $rt(I)$:*

$$sv = c.gpr(rt(I))[8 \cdot d(I) - 1 : 0]$$

General Purpose Register Updates

Definition 45 (General Purpose Register Write Predicate)**.** *The predicate*

$$gprw \equiv alu(I) \vee su(I) \vee lw(I) \vee cas(I) \vee jal(I) \vee jalr(I)$$

$$\vee movs2g(I) \vee mul(I) \vee mflo(I) \vee mfhi(I)$$

describes whether a given instruction $I \in \mathbb{B}^{32}$ results in a write to some general purpose register.

Definition 46 (General Purpose Register Result Destination)**.** *We define the result destination of an ALU/shift/coprocessor/memory instruction $I \in \mathbb{B}^{32}$ as the following general purpose register address:*

$$rdes = \begin{cases} rd(I) & rtype(I) \wedge /movs2g(I) \vee mul(I) \\ rt(I) & otherwise \end{cases}$$

Definition 47 (Written General Purpose Register)**.** *For an instruction $I \in \mathbb{B}^{32}$, the address of the general purpose register which is actually written to is defined as*

$$cad = \begin{cases} 1^5 & jal(I) \\ rdes(I) & otherwise \end{cases}$$

Definition 48 (General Purpose Register Input)**.** *We define the value written to the general purpose register specified above based on the instruction $I \in \mathbb{B}^{32}$ and a given processor core configuration $c \in K_{core}$ as*

$$gprdin = \begin{cases} c.pc +_{32} 4_{32} & jal(I) \vee jalr(I) \\ lv(R) & load(I) \vee cas(I) \\ c.spr(rd(I)) & movs2g(I) \\ c.HI & mfhi(I) \\ c.LO & mflo(I) \\ alures(lop(c,I), rop(c,I), alucon(I)) & alu(I) \\ mres(c,I)[31:0] & mul(I) \\ sures(slop(c,I), sdist(c,I), sf(I)) & su(I) \end{cases}$$

B.6.2 Definition of Instruction Execution

Based on the auxiliary functions defined in the last subsection, we give the definition of instruction execution in closed form:

Definition 49 (Non-Interrupted Instruction Execution)**.** *We define the transition function for non-interrupted instruction execution*

$$\delta_{instr} : K_{core} \times \Sigma_{instr} \rightharpoonup K_{core}$$

where

$$\Sigma_{instr} = \mathbb{B}^{32} \times (\mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32} \cup \{\bot\})$$

as

$$\delta_{instr}(c,I,R) = \begin{cases} undefined & (load(I) \vee cas(I)) \wedge R \notin \mathbb{B}^{8 \cdot d(I)} \\ c' & otherwise \end{cases}$$

where

- $c'.pc = \begin{cases} btarget(c,I) & jbtaken(c,I) \\ c.pc +_{32} 4_{32} & otherwise \end{cases}$
- $c'.gpr(x) = \begin{cases} gprdin(c,I,R) & x = cad(I) \wedge gprw(I) \\ c.gpr(x) & otherwise \end{cases}$
- $c'.spr(x) = \begin{cases} c.gpr(rt(I)) & rd(I) = x \wedge movg2s(I) \\ c.spr(x) & otherwise \end{cases}$
- $c'.HI = \begin{cases} mres(c,I)[63:32] & mulacc(I) \\ c.gpr(rs(I)) & mthi(I) \\ undefined & mul(I) \\ c.HI & otherwise \end{cases}$
- $c'.LO = \begin{cases} mres(c,I)[31:0] & mulacc(I) \\ c.gpr(rs(I)) & mtlo(I) \\ undefined & mul(I) \\ c.HI & otherwise \end{cases}$

B.6.3 Auxiliary Definitions for Triggering of Interrupts

MIPS-86 provides the following interrupt types which are ordered by their priority (interrupt level):

interrupt level	shorthand	internal /external	type	maskable	
0	reset	eev	abort	0	reset
1	I/O	eev	repeat	1	devices
2	ill	iev	abort	0	illegal instruction
3	mal	iev	abort	0	misaligned
4	pff	iev	repeat	0	page fault fetch
5	pfls	iev	repeat	0	page fault load/store
6	sysc	iev	continue	0	system call
7	ovf	iev	continue	1	overflow

Note that the all continue interrupts are either triggered by execution of ALU operations with overflow or execution of the sysc-Instruction.

While external event signals are provided by the local APIC as input $eev \in \mathbb{B}^{256}$ to the processor core transition function, the internal event signals $iev \in \mathbb{B}^8$ are defined by the following table that uses the page-fault signals $pff, pfls \in \mathbb{B}$ which are provided by the MMU of the processor to the processor core transition function.

internal event signal	defined by
$iev(c,I,pff,pfls)[2]$	$\equiv ill(I) \vee c.mode[0] = 1 \wedge (movg2s(I) \vee movs2g(I))$
$iev(c,I,pff,pfls)[3]$	$\equiv mal(c,I)$
$iev(c,I,pff,pfls)[4]$	$\equiv pff$
$iev(c,I,pff,pfls)[5]$	$\equiv pfls$
$iev(c,I,pff,pfls)[6]$	$\equiv sysc(I)$
$iev(c,I,pff,pfls)[7]$	$\equiv ovf(lop(c,I),rop(c,I),alucon(I),itype(I))$

Note that even though, from the view of the processor core, the page-fault signals appear just as external as the external event vector provided by the local APIC, the difference is that the external interrupts provided by the local APIC originate from devices while the page-fault signals originate from the MMU belonging to processor itself. This justifies classifying them as internal event signals.

When an interrupt occurs, information about the type of interrupt is stored in a special purpose register to allow the programmer to discover the reason, i.e. the cause, for the interrupt.

Definition 50 (Cause and Masked Cause of an Interrupt)**.** *We define the cause $ca \in \mathbb{B}^8$ of an interrupt and masked cause $mca \in \mathbb{B}^8$ of an interrupt based on the current processor core configuration $c \in K_{core}$, the instruction $I \in \mathbb{B}^{32}$ to be executed, the external event vector $eev \in \mathbb{B}^{256}$ and the page-fault signals $pff,pfls \in \mathbb{B}$ as follows:*

- *cause of interrupt:*

$$ca[j] = \begin{cases} iev(c,I,pff,pfls)[j] & j \in [2:7] \\ \bigvee_{i=0}^{255} eev[i] & j = 1 \\ 0 & otherwise \end{cases}$$

- *masked cause:*

$$mca[j] = \begin{cases} ca(c,I,eev,pff,pfls)[j] & j \notin \{1,7\} \\ ca(c,I,eev,pff,pfls)[j] \wedge c.spr(sr)[j] & j \in \{1,7\} \end{cases}$$

Only interrupt levels 1 and 7 are maskable; the corresponding mask can be found in special purpose register sr (status register) and is applied to the cause of interrupt to obtain the masked cause.

Definition 51 (Jump-to-Interrupt-Service-Routine Predicate)**.** *To denote that in a given configuration $c \in K_{core}$ for a given instruction $I \in \mathbb{B}^{32}$, external event signals $eev \in \mathbb{B}^{256}$, and page-fault signals $pff,pfls \in \mathbb{B}$ an interrupt is triggered, we define the predicate*

$$jisr \equiv \bigvee_j mca(c,I,eev,pff,pfls)[j]$$

Definition 52 (Interrupt Level of the Triggered Interrupt)**.** *To determine the interrupt level of the triggered interrupt, we define the function*

$$il = min\{j \mid mca(c,I,eev,pff,pfls)[j] = 1\}$$

Definition 53 (Continue-Type Interrupt Predicate)**.** *The predicate*

$$continue \equiv il(c,I,R,eev) \in \{6,7\}$$

denotes whether the triggered interrupt is of continue type.

B.6.4 Definition of Handling

Definition 54 (Interrupt Execution Transition Function)**.** *We define*

$$\delta_{jisr}(c, I, R, eev, pff, pfls) = c'$$

where $I \in \mathbb{B}^{32}$ is the instruction to be executed and $eev \in \mathbb{B}^{256}$ are the event signals received from the local APIC and $pff, pfls \in \mathbb{B}$ are the page-fault signals provided by the processor's MMU.

Let $k = min\{j \mid eev[j] = 1\}$.

- $c'.pc = 0^{32}$
- $c'.spr(x) = \begin{cases} 0^{32} & x = sr \\ 0^{32} & x = mode \\ c.sr & x = esr \\ zxt_{32}(mca(c, I, eev, pff, pfls)) & x = eca \\ c.pc & x = epc \\ & \wedge\, /continue(c, I, eev, pff, pfls) \\ \delta_{\textbf{\textit{instr}}}(c, I, R).pc & x = epc \wedge continue(c, I, eev, pff, pfls) \\ ea(c, I) & x = edata \wedge il(c, I, eev, pff, pfls) = 5 \\ bin_{32}(k) & x = edata \wedge il(c, I, eev, pff, pfls) = 1 \\ c.spr(x) & otherwise \end{cases}$
- $c'.gpr = \begin{cases} c.gpr & /continue(c, I, eev, pff, pfls) \\ \delta_{\textbf{\textit{instr}}}(c, I, R).gpr & otherwise \end{cases}$

Definition 55 (Return From Exception Transition Function)**.** *We define $deltarfe(c) = c'$.*

- $c'.pc = c.spr(epc)$
- $c'.spr(x) = \begin{cases} 0^{31}1 & x = mode \\ c.spr(esr) & x = sr \\ c.spr(x) & otherwise \end{cases}$
- $c'.gpr = c.gpr$

B.7 Store Buffer

Store buffers are, in their simplest form, first-in-first-out queues for write accesses that reside between processor core and memory. In a processor model with store-buffer, servicing memory reads is done by finding the newest store-buffer entry for the given address if one is available – otherwise the read is serviced by the memory subsystem. Essentially, this means that read accesses that rely on values from preceeding write accesses can be serviced even before they reach the caches. The benefit of store-buffers implemented in hardware is that instruction execution can proceed while the memory is still busy servicing previous write accesses.

In order to allow the programmer to obtain a sequentially consistent view of memory in the presence of store-buffers, architectures whose abstract model contains store-buffers tend to provide instructions that have an explicit effect on the store-buffer, e.g. by draining the pipeline. *MIPS-86* offers a memory fence instruction $fence$ that simply drains the store buffer and a read-modify-write operation *rmw* that performs an atomic conditional memory update with the side-effect of draining the store-buffer.

Note that, even in a machine that has no store-buffer in hardware, pipelining of instruction execution may introduce a store-buffer to the abstract machine model. We discuss this in the next subsection before we give a definition of the store-buffer of MIPS-86.

B.7.1 Instruction Pipelining May Introduce a Store-Buffer

The term *pipelining* used in the context of gate-level circuit design can be used to describe splitting up a hardware construction (e.g. of a processor) that computes some result in a single hardware cycle (e.g. executes an instruction) into n smaller components which are called *pipeline stages* whose outputs are always inputs of the next one and which each computes a part of the final result in its own registers – in such a way that, initially, after n cycles, the first result is provided by the nth component and then, subsequently, every following cycle a computation finishes. The reason why this is efficient lies in the fact that, in terms of electrophysics, smaller circuits require less time for input signals to propagate and for output signals to stabilize, thus, smaller circuits can be clocked faster than larger ones. Note that the increase in delay for inserting additional registers in the subcomponents tends to be less than the delay saved by splitting the construction into pipeline stages, resulting in an overall faster computation due to the achieved parallelization.

A common feature to be found in processors is *instruction pipelining*. For a basic RISC machine (like *MIPS-86*), the common five-stage pipeline is given by the following five stages: IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, and WB = Register write back. Note that a naive hardware implementation where all that is changed from the one-cycle hardware construction is that additional registers are inserted will, in general, behave differently than the original construction: Execution of the next instruction may depend on results from the execution of previous instructions which are still in the instruction pipeline. The occurrence of such a dependency where the result of a computation in the naively pipelined machine does not match the result of the sequential machine is referred to as a *hazard*. One way to circumvent this is by software: If the programmer/compiler ensures that the machine code executed by the pipelined machine does not cause any hazard (e.g. by inserting NOOPs (no operations, i.e. instructions that do not have any effect other than incrementing the program counter) or by reordering instructions). This, however, by requiring a much more conservative style of programming, reduces the speedup gained by introducing pipelining in the first place.

In fact, instead of leaving hazard detection and handling exclusively to the programmer of the machine, modern architectures implement proper handling of most hazards in hardware. When a *data hazard* is detected (i.e. an instruction depends on some value computed by an earlier instruction that is still in the pipeline), the hardware *stalls* execution of the pipeline on its own until the required result has been computed. Additional hardware then *forwards* the result from the later pipeline stage directly to the waiting instruction that is stalled in an earlier pipeline stage. Note that even though many hazards can be detected and resolved efficiently in hardware, it is not necessarily

the best thing to prevent all hazards in hardware – overall performance of a system may be better when minor parts of hazard handling are left to the programmer/compiler. In fact, for modern pipelined architectures, it is common practice to allow slight changes to the abstract hardware model at ISA level which allow for a less strict but more performant treatment of hazards.

When a memory write is forwarded to a subsequent memory read instruction to the same address (or to the instruction fetch stage, possibly), this can be modeled by introducing a *store-buffer* between processor and memory system – even when there is no physical store-buffer present in the hardware implementation [Hot12]. For a single-core architecture, it is not overly hard to prove that a processor model with store-buffer is actually equivalent to the processor model without store-buffer. For a multi-core architecture however, it is more involved to prove that store-buffers become invisible on higher layers of abstraction: Since every processor has its own store-buffer and the values from store-buffers are not forwarded to other processors, any two processors may have different values for the same address present in their store-buffers. For an in-detail treatment of hardware construction and correctness for a pipelined simple MIPS machine, see [Pau12].

B.7.2 Configuration

Definition 56 (Store Buffer Configuration)**.** *The set of store buffer entries is given by*

$$K_{sbe} \equiv \{(a,v) \mid a \in \mathbb{B}^{32} \wedge v \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}\}$$

while the set of store buffer configurations is defined as follows:

$$K_{sb} \equiv K_{sbe}^*$$

We consider a store buffer modeled by a finite sequence of store buffer write accesses (a,v) where $a \in \mathbb{B}^{32}$ is an address and $v \in \mathbb{B}^8 \cup \mathbb{B}^{16} \cup \mathbb{B}^{32}$ is the value to be written starting at memory address a.

B.7.3 Transitions

A step of the store buffer produces the memory write specified by the oldest store-buffer entry – in order to be sent to the memory subsystem. When the store buffer is empty ($c.sb = \varepsilon$), it cannot make a step. We always append at the front and remove at the end of the list. Transitions of the store buffer are formalized in the overall transition relation – we do not provide an individual transition relation for the store buffer.

B.7.4 Auxiliary Definitions

We define some auxiliary functions for use in the definition of the system's transition function.

Definition 57 (Store Buffer Entry Hit Predicate)**.** *Given a store buffer entry $(a,w) \in K_{sbe}$ and a byte address $x \in \mathbb{B}^{32}$, we define the predicate*

$$sbehit \equiv \langle x -_{32} a \rangle < |w|/8$$

which denotes that there is a store buffer hit for the given entry and address, i.e. the address is written by the write access represented by the store buffer entry.

Definition 58 (Store Buffer Hit Predicate)**.** *Given a store buffer configuration $sb \in K_{sb}$ and a byte address $x \in \mathbb{B}^{32}$, the predicate*

$$sbhit \equiv \exists j : sbehit(sb[j], x)$$

denotes whether there is a store buffer hit for the given address in the given store buffer.

Definition 59 (Newest Store Buffer Hit Index)**.** *Given a store buffer configuration $sb \in K_{sb}$ and a byte address $x \in \mathbb{B}^{32}$, we define the function*

$$maxsbhit \equiv \begin{cases} \max\{j \mid sbehit(sb[j], x)\} & sbhit(sb, x) \\ \bot & otherwise \end{cases}$$

which computes the index of the newest entry of the store buffer for which there is a hit or returns the special value $\bot$ if there is no such index.

Definition 60 (Store Buffer Value)**.** *We define the function*

$$sbv : K_{sb} \times \mathbb{B}^{32} \rightharpoonup \mathbb{B}^8$$

which, given a store buffer configuration sb and a byte-address x computes the value forwarded from the store buffer for the given address, if defined:

$$sbv = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & sb[maxsbhit(sb, x)] = (a, v) \\ undefined & otherwise \end{cases}$$

Definition 61 (Store Buffer Hazard)**.** *When the processor tries to perform a read access which causes a store-buffer hit but at the same time cannot be serviced by the single newest store buffer entry (e.g. when we have a partial hit for a halfword or word access), we have a* store-buffer hazard[1]*: Given a store-buffer configuration sb, a byte-address a and an access width $d \in \mathbb{N}$, we have*

$$sbhazard \equiv$$

$$\exists i, j : 0 \leq i < j < d \wedge maxsbhit(sb, a +_{32} i_{32}) \neq maxsbhit(sb, a +_{32} j_{32})$$

B.8 Devices

B.8.1 Introduction to Devices, Interrupts and the APIC Mechanism

Interesting hardware tends to include devices of some kind, e.g. screens, input devices, timers, network adapters, storage devices, or devices that control factory robots, conveyor belts, nuclear power plants, radiation therapy units, etc. In order to trigger interrupts in a processor core, devices tend to provide *interrupt request signals*. Commonly, device interrupt request signals are distinguished in *edge-triggered* signals, i.e. the device signals an interrupt by switching the state of its interrupt request signals, and *level-triggered* signals, i.e. the interrupt request signal is raised when an interrupt is currently pending (i.e. the interrupt request signal has the digital value 1). In a simple

[1] In the overall transition relation, the processor core has to wait until the write causing the store-buffer hazard leaves the store-buffer.

hardware implementation of a single-core architecture, level-triggered interrupt request signals originating from devices basically just need to be connected to the processor as external interrupt signals while edge-triggered signals need to be sampled properly and then provided to the processor until it can accept them. Note that in MIPS-86, we restrict interrupt request signals to level-triggered signals.

The processor tends to communicate with devices either by means of *memory mapped input/output* (memory mapped I/O), i.e. device *ports* (which can essentially be considered user-visible registers of a device) are mapped into the memory space of the processor and accessed with regular memory instructions, or by using special I/O instructions that operate on a seperate port address space where device ports reside. Reading or writing device ports tends to have side-effects, e.g. such as disabling the interrupt request signal raised by the device, allowing the processor to acknowledge that it has received the interrupt, or causing the device to initiate some interaction with the external world.

For a multi-core architecture, device interrupts become more interesting: Is it desirable to interrupt all processors when a device raises an interrupt signal? In many cases, the answer is no: It is fully sufficient to let one processor handle the interrupt. The main question is mostly which device interrupt is supposed to go to which processor. In x86 architectures this is resolved in hardware by providing each processor with a *local advanced programmable interrupt controller* (local APIC) that receives *interrupt messages* from a global *input/output advanced programmable interrupt controller* (I/O APIC) which collects and distributes the interrupt request signals of devices. In order not to transmit a single interrupt to a processor more often than necessary, there is a protocol between I/O APIC and the processor (via the local APIC) in which the processor has to acknowledge the handling of the interrupt by sending an *end of interrupt* (EOI) message via the local APIC. Only after such an EOI message has been received will the I/O APIC sample the corresponding interrupt vector again. Essentially, in the abstract hardware model, both local APIC and I/O APIC can be seen as a special kind of device that does not raise interrupts on its own but can be accessed by the processor by means of memory mapped I/O just like a regular device. Since MIPS-86 implements a greatly simplified version of the x86 APIC mechanism, we will not discuss the detailed x86 APIC mechanism in the following and focus on MIPS-86 in the following.

How exactly the I/O APIC distributes device interrupt signals to the individual processor cores is specified by the *redirect table* – which can be accessed through the I/O APIC ports. This redirect table – which must be set up by the programmer of the machine – specifies the following for each device interrupt request signal: The destination processor, whether the interrupt signal is masked already at the I/O APIC, and the *interrupt vector* to be triggered. The interrupt vector of an interrupt is used to provide information about the cause of the interrupt to the processor. Device interrupt signals are sampled at the I/O APIC and subsequently sent to the destination processor's local APIC over a common bus that connects all local APICs and the I/O APIC. The local APIC associated with a processor core receives interrupt messages from the I/O APIC, collecting interrupt vectors which are then passed to the processor core by raising external interrupt signals at the processor core.

In addition to providing a means of distributing device interrupts, the APIC mechanism offers processor cores of the multi-core system the opportunity to send *inter-processor interrupts* (IPIs). This can, for example, be useful to implement communication between different processors in the context of an operating system. Sending of an inter-processor interrupt is triggered by writing a particular control register belonging to the ports of the local APIC of the processor. The content of this control register

describes the destination processors, delivery status, delivery mode and interrupt vector of the inter-processor interrupt. The IPI mechanism is particularly important for booting the machine: Initially, after power on, only the *bootstrap processor* (BSP) is running while all other processors are in a halted state with their local APICs waiting for an initialization inter-processor interrupt (INIT-IPI) and a subsequent startup inter-processor interrupt (SIPI). Effectively, booting the multi-core system can be done by the bootstrap processor in a sequential fashion until it initializes and starts the other processor cores of the system via the IPI mechanism.

B.8.2 Configuration

Definition 62 (Device Port Address Length)**.** *We assume a function*

$$dev_{\boldsymbol{palen}} : [0 : nd - 1] \to \mathbb{N}$$

to be given that specifies the address length of port addresses of all $nd \in \mathbb{N}$ devices of the system in bits.

Definition 63 (Device Configuration)**.** *The configuration $d \in K_{\boldsymbol{devi}}$ of device i is given by*

- *I/O ports $d.ports : \mathbb{B}^{dev_{\boldsymbol{palen}}(i)} \to \mathbb{B}^8$,*
- *an interrupt request signal $d.irq \in \mathbb{B}$, and*
- *internal state $d.internal \in \mathcal{D}_i$*

Note that the $\mathcal{D}_i$ have to be defined by users of our model to describe the devices they want to argue about.

B.8.3 Transitions

Devices react to external inputs provided to them and they have side-effects that occur when their ports are read, or, respectively, written. Note that we currently do not model read-modify-write accesses to devices and we only consider word-accesses on device ports.

Definition 64 (Device Transition Function)**.** *For every device, we assume a transition function to be given of the form*

$$\delta_{\boldsymbol{dev(i)}} : K_{\boldsymbol{dev(i)}} \times \Sigma_{\boldsymbol{dev(i)}} \rightharpoonup K_{\boldsymbol{dev(i)}}$$

with

$$\Sigma_{\boldsymbol{dev(i)}} = \Sigma_{\boldsymbol{ext(i)}} \cup \mathbb{B}^{dev_{\boldsymbol{palen}}} \cup \mathbb{B}^{dev_{\boldsymbol{palen}}} \times (\mathbb{B}^8)^* \cup \mathbb{B}^{32} \times \mathbb{B}^{dev_{\boldsymbol{palen}}} \times \mathbb{B}^{32}$$

where the input to the transition function $in \in \Sigma_{\boldsymbol{dev(i)}}$ is either

- *an external input for device i: $in = ext \in \Sigma_{\boldsymbol{ext(i)}}$,*
- *an external input for device i: $in = a \in \mathbb{B}^{dev_{\boldsymbol{palen}}(i)}$, or*
- *a word write-access to port address a with value v: $in = (a, v) \in \mathbb{B}^{dev_{\boldsymbol{palen}}(i)} \times \mathbb{B}^{32}$.*

Note that all active steps of a device are modeled via external inputs, i.e. every active step of the device should be modeled by an input from $\Sigma_{\mathbf{ext(i)}}$ that triggers the corresponding step. Further, $\Sigma_{\mathbf{ext(i)}}$ can be used to model how the device reacts to the external world.

Depending on the device in question, reading or writing port addresses may have side-effects – for example, deactivating the interrupt request when a specific port is read. This needs to be specified individually for the given device in its transition function. One restriction we make in this model is that even though reading ports may have side-effects, the value being read is always the one that is given in the I/O ports component of the device. This is reflected in the next section when an overall view of memory with device I/O ports mapped into the physical address space of the machine is defined.

B.8.4 Device Outputs

We allow devices to provide an output function

$$\lambda_{\mathbf{dev(i)}} : K_{\mathbf{ext(i)}} \times \Sigma_{\mathbf{dev(i)}} \rightharpoonup \Omega_{\mathbf{dev(i)}}$$

in order to allow interaction with some external world. This is a partial function, since a device does not need to produce an output for every given external input in a given configuration.

B.8.5 Device Initial State

To define a set of acceptable initial states of a device after reset, the predicate

$$initial_{\mathbf{state(i)}} : K_{\mathbf{dev(i)}} \rightarrow \mathbb{B}$$

shall be defined.

B.8.6 Specifying a Device

To specify a particular device of interest, we always need to define the following:

- $\mathcal{D}_i$ – the internal state of the device,
- $\Sigma_{\mathbf{ext(i)}}$ – the external inputs the device reacts to,
- $\Omega_{\mathbf{dev(i)}}$ – the possible outputs provided by the device,
- $dev_{\mathbf{palen}}(i)$ – the length of port addresses of the device,
- $\delta_{\mathbf{dev(i)}}$ – the transition function of the device,
- $\lambda_{\mathbf{dev(i)}}$ – the output function of the device, and
- $initial_{\mathbf{state(i)}}$ – the set of acceptable initial states of the device.

B.9 Local APIC

The local APIC receives device and inter-processor interrupts sent over the interrupt bus of the system and provides these interrupts to the processor core it is associated with. While the local APIC shares some behavior with devices (i.e. it is accessed by means of memory-mapped I/O) some of its behavior differs significantly from that of devices (i.e. communicating over the interrupt bus instead of raising an interrupt request signal, providing interrupt signals directly to the processor core).

The x86-64 model of [Deg11] provides a local APIC model that describes sending of inter-processor interrupts but ignores devices. While already simplified somewhat compared to the informal architecture definitions, this model is still quite complex. Thus, Hristo Pentchev provides a simplified version of the x86-64 local APIC model in his upcoming dissertation in order to prove formal correctness of an inter-processor interrupt protocol implementation [ACHP10]. On the one hand, the local APIC model we present in the following is even further simplified – mostly by expressing APIC system transitions atomically instead of in terms of many intermediate steps and by reducing the possible interrupt types and target modes. On the other hand, the model provided here is more powerful in the sense that device interrupts and I/O APIC are modeled.

We have the following simplifications over x86-64:

- We only consider level-triggered interrupts.
- We reduce IPI-delivery to Fixed, INIT and Startup interrupts. The I/O APIC only delivers Fixed interrupts.
- We only model physical destination mode where IPIs are addressed to a local APIC ID (or to a shorthand). We don't consider logical destination mode.
- We do not consider the error-status-register which keeps track of errors encountered when trying to deliver interrupts.

B.9.1 Configuration

Definition 65 (Local APIC Configuration)**.** *The configuration of a local APIC*

$$apic = (apic.ports, apic.initrr, apic.sipirr, apic.sipivect, apic.eoipending) \in K_{apic}$$

consists of

- *I/O-ports* $apic.ports : \mathbb{B}^7 \to \mathbb{B}^8$,
- *INIT-request register* $apic.initrr \in \mathbb{B}$,

  (a flag that denotes whether an INIT-request is pending to be delivered to the processor)
- *SIPI-request register* $apic.sipirr \in \mathbb{B}$,

  (a flag that denotes whether a SIPI-request is pending to be delivered to the processor)
- *SIPI-vector register* $apic.sipivect \in \mathbb{B}^8$,

  (the start address for the processor to execute code after receiving SIPI)

- *EOI-pending register* $apic.eoipending \in \mathbb{B}^{256}$

  (a register that keeps track of all interrupt vectors for which an EOI message is to be sent to the I/O APIC)

The I/O ports of the local apic can be accessed by the processor by means of memory mapped I/O. All other local APIC components cannot be accessed by other components. This is reflected in the overall transition relation of the system.

Local APIC ports

Let us define a few shorthands for specific regions in the local APIC ports:

- **APIC ID Register**

$$apic.\mathbf{APIC\_ID} = apic.ports_4(0_7)$$

Bits	description
31-24	local APIC ID
23-0	reserved

This register contains the local APIC ID of the local APIC. This ID is used when addressing inter-processor-interrupts to a specific local APIC.

- **Interrupt Command Register (ICR)**

$$apic.\mathbf{ICR} = apic.ports_8(4_7) \in \mathbb{B}^{64}$$

Bits	abbreviation	description
63-56	*dest*	destination field
55-20		reserved
19-18	*dsh*	destination shorthand
		00b = no shorthand, 01b = self 10b = all including self, 11b = all excluding self
17-13		reserved
12	*ds*	delivery status 0b = idle, 1b = send pending
11	*destmode*	destination mode 0b = physical
10-8	*dm*	delivery mode 000b = Fixed, 101b = INIT, 110b = Startup
7-0	*vect*	vector

This register is used to issue a request for sending an inter-processor interrupt to the local APIC.

- **End-Of-Interrupt Register**

$$apic.\mathbf{EOI} = apic.ports_4(12_7) \in \mathbb{B}^{32}$$

Writing to this register is used to signal to the local APIC that the interrupt-service-routine has finished. This has the effect that the local APIC will eventually send an end-of-interrupt acknowledgement to the I/O-APIC.

- **In-Service Register**

$$apic.\mathbf{ISR} = apic.ports_{32}(16_7) \in \mathbb{B}^{256}$$

This register keeps track of which interrupt vectors are currently being serviced by an interrupt-service-routine. For our simple processor, maskable interrupts (to which device interrupts belong) are by default masked in the processor core when an interrupt is triggered. However, when the programmer explicitly unmasks device interrupts during the interrupt handler run, it can happen that a higher-priority interrupt provided by the local APIC may trigger another interrupt, resulting in several interrupt vectors being in service at the same time.

- **Interrupt Request Register**

$$apic.\mathbf{IRR} = apic.ports_{32}(48_7) \in \mathbb{B}^{256}$$

This register keeps track for which interrupt vectors there is currently a request pending. These requests are provided to the processor as external event signals. In this process, all interrupt requests of lower priority than the ones currently in service are masked by the local APIC.

Definition 66 (Processor Core External Event Signals)**.** *We define the external event vector* $eev \in \mathbb{B}^{256}$ *provided by the local APIC* $apic \in K_{\mathbf{apic}}$ *to the processor core as*

$$eev(apic)[j] = \begin{cases} 0 & \exists k \leq j : apic.\mathbf{ISR}[k] = 1 \\ apic.\mathbf{IRR}[j] & otherwise \end{cases}$$

B.9.2 Transitions

We simplify device accesses in such a way that we expect only aligned word-accesses to occur on device ports, i.e. halfword and byte accesses on devices are not modeled.

For all passive steps of the local APIC, we define a transition function

$$\delta_{\mathbf{apic}} : K_{\mathbf{apic}} \times \Sigma_{\mathbf{apic}} \to K_{\mathbf{apic}}$$

where

$$\Sigma_{\mathbf{apic}} \equiv \mathbb{B}^7 \times \mathbb{B}^{32} \cup \{\mathbf{Fixed}, \mathbf{INIT}, \mathbf{SIPI}\} \times \mathbb{B}^8 \cup \{\mathbf{jisr}, \mathbf{rfe}\}$$

A passive step of a local APIC is a write access to its ports, a receive-interrupt step, the reaction to a *jisr*-step of the processor core, or the reaction to a *rfe*-step of the processor. We define

$$\delta_{\mathbf{apic}}(apic, in) = apic'$$

by a case-distinction:

- write without side-effects:

$$in = (a, v) \wedge a \neq 12_7$$

$$apic'.ports(x) = \begin{cases} byte(\langle x \rangle - \langle a \rangle, v) & in = (a, v) \wedge 0 \leq \langle x \rangle - \langle a \rangle < 4 \\ apic.ports(x) & otherwise \end{cases}$$

- write with side effects (to EOI-register):

$$in = (a, v) \wedge a = 12_7$$

$apic'.\mathbf{EOI} = v$

$$apic'.\mathbf{ISR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\mathbf{ISR}[k] = 1\} \\ apic.\mathbf{ISR}[j] & otherwise \end{cases}$$

All other local APIC ports stay unchanged.

$$apic'.eoipending[j] = \begin{cases} 1 & j = min\{k \mid apic.\mathbf{ISR}[k] = 1\} \\ apic.eoipending[j] & otherwise \end{cases}$$

Writing to the EOI-Register puts the local APIC in a state where it will send an EOI-message over the interrupt bus in order to acknowledge handling of the highest-priority interrupt to the I/O APIC.

- receiving an interrupt:
  - $in = (\mathbf{Fixed}, vect)$

    $$apic'.\mathbf{IRR}[j] = \begin{cases} 1 & j = \langle vect \rangle \\ apic.\mathbf{IRR}[j] & otherwise \end{cases}$$

    All other ports unchanged.
  - $in = (\mathbf{SIPI}, vect)$

    Receiving a startup-interrupt is successful when there is currently no startup-interrupt pending: If $apic.sipirr \neq 0$, $apic' = apic$ (the local APIC will discard the interrupt), otherwise

    $apic'.ports = apic.ports$

    $apic'.sipirr = 1$

    $apic'.sipivect = vect$

    This records a SIPI which can in turn be used by the local APIC to set the *running* flag of the corresponding processor, effectively starting it.
  - $in = (\mathbf{INIT}, vect)$

    Receiving an INIT-interrupt is successful when there is currently no INIT-interrupt pending: If $apic.initrr \neq 0$, $apic' = apic$, otherwise

    $apic'.ports = apic.ports$

    $apic'.initrr = 1$

    When the local APIC received an INIT-IPI, it will force a reset on the corresponding processor.
- reaction to a *jisr*-step $in = \mathbf{jisr}$:

$$apic'.\mathbf{IRR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\mathbf{IRR}[k] = 1\} \\ apic.\mathbf{IRR}[j] & otherwise \end{cases}$$

$$apic'.\mathbf{ISR}[j] = \begin{cases} 1 & j = min\{k \mid apic.\mathbf{IRR}[k] = 1\} \\ apic.\mathbf{ISR}[j] & otherwise \end{cases}$$

- reaction to a *rfe*-step $in = \mathbf{rfe}$:

$$apic'.\mathbf{ISR}[j] = \begin{cases} 0 & j = min\{k \mid apic.\mathbf{ISR}[k] = 1\} \\ apic.\mathbf{ISR}[j] & otherwise \end{cases}$$

All components not explicitly mentioned stay unchanged between $apic$ and $apic'$.

The active steps of the local APIC (i.e. sending IPIs, sending EOI messages, applying SIPI and INIT-IPI to the processor) are treated in the overall transition relation of the system.

B.10 I/O APIC

The I/O APIC samples the interrupt request signals of devices and distributes the interrupts to the local APICs according to its redirect table by sending interrupt messages over the interrupt bus. Device interrupts can be masked directly at the I/O APIC.

B.10.1 Configuration

Definition 67 (I/O APIC Configuration)**.** *The configuration of an I/O APIC*

$$ioapic = (ioapic.ports, ioapic.redirec) \in K_{\boldsymbol{ioapic}}$$

is given by

- *I/O-ports* $ioapic.ports : \mathbb{B}^3 \to \mathbb{B}^8$*, and*
- *a redirect table* $ioapic.redirect : [0:23] \to \mathbb{B}^{32}$

I/O APIC Ports

Shorthands to the ports of the I/O APIC are

- select register $ioapic.ioregsel = ioapic.ports_4(0_3)$
- data register $ioapic.iowin = ports_4(4_3)$

Note a pecularity about the I/O APIC: instead of mapping the redirect table into the processor's memory, only a select and a data register are provided. Writing the select register has the side-effect of fetching the redirect table entry specified to the data register. Writing the data register has the side effect of also writing the redirect table entry specified by the select register. Reading from the I/O APIC ports does not have any side-effects.

Format of the Redirect Table

A redirect table entry $e \in \mathbb{B}^{32}$ has the following fields:

Bits	Short	Name	Description
24-31	*dest*	Destination	Local APIC ID of destination local APIC
17-24		reserved	
16	*mask*	Interrupt Mask	masked if set to 1
15		reserved	
14	*rirr*	Remote IRR	0b = EOI received 1b = interrupt was received by local APIC
13		reserved	
12	*ds*	Delivery Status	1b = Interrupt needs to be delivered to local APIC
11		reserved	
8-10	*dm*	Delivery Mode	000b = Fixed
0-7	*vect*	Interrupt Vector	

B.10.2 Transitions

We define a transition function for the passive steps of the I/O APIC

$$\delta_{\textbf{ioapic}} : K_{\textbf{ioapic}} \times \Sigma_{\textbf{ioapic}} \rightharpoonup K_{\textbf{ioapic}}$$

with

$$\Sigma_{\textbf{ioapic}} = \mathbb{B}^3 \times \mathbb{B}^{32} \cup \mathbb{B}^8$$

where $in \in \Sigma$ is either

- $in = (a, v) \in \mathbb{B}^3 \times \mathbb{B}^{32}$ – a write access to port address a with value v,
- $in = vect \in \mathbb{B}^8$ – receiving an EOI message for interrupt vector *vect*

We define $\delta_{\textbf{ioapic}}(ioapic, in) = ioapic'$ by case distinction on *in*:

- $in = (a, v) \in \mathbb{B}^3 \times \mathbb{B}^{32}$ – write access to the I/O APIC ports
  $\delta_{\textbf{ioapic}}(ioapic, in)$ is undefined iff $a \notin \{0_3, 4_3\}$. Otherwise
  - **Case** $a = 0_3$:
    $ioapic'.\textbf{IOREGSEL} = v$
    $ioapic.\textbf{IOWIN} = ioapic.redirect(\langle v \rangle)$
  - **Case** $a = 4_3$:
    $ioapic'.\textbf{IOWIN} = v$
    $$ioapic'.redirect(i) = \begin{cases} v & i = \langle ioapic.\textbf{IOREGSEL} \rangle \\ ioapic.redirect(i) & otherwise \end{cases}$$
- $in = vect \in \mathbb{B}^8$ – receiving an EOI message for interrupt vector *vect*

$$ioapic'.redirect(i).rirr = \begin{cases} 0 & ioapic.redirect(i).vect = vect \\ ioapic.redirect(i).rirr & otherwise \end{cases}$$

Receiving an EOI message for interrupt vector *vect* resets the corresponding remote interrupt request signal associated with all redirect table entries associated with the interrupt vector. Note that it is adviseable to configure the system in such a way that interrupt vectors assigned to devices are unique.

All components not explicitly mentioned stay unchanged.

Active transitions of the I/O APIC can be found in the definition of the overall transition relation.

B.11 Multi-Core MIPS

Transitions of the abstract machine are defined as

$$\delta : K \times \Sigma \rightharpoonup K$$

Inputs of the system specify which processor component makes what kind of step and are defined below. On the level of abstraction provided, we assume that the memory subsystem does not make steps on its own, thus it may neither receive external inputs nor be scheduled to make an active step.

The transition functions of the subcomponents are given by

- memory transitions : $\delta_{\mathbf{m}} : K_{\mathbf{m}} \times \Sigma_{\mathbf{m}} \to K_{\mathbf{m}}$ (always passive, section B.4),
- processor core transitions : $\delta_{\mathbf{core}} : K_{\mathbf{core}} \times \Sigma_{\mathbf{core}} \rightharpoonup K_{\mathbf{core}}$ (section B.6),
- passive TLB transitions : $\delta_{\mathbf{tlb}} : K_{\mathbf{tlb}} \times \Sigma_{\mathbf{tlb}} \to K_{\mathbf{tlb}}$ (section B.5, active transitions are given explicitly in the top level transition function),
- store-buffer transitions which are stated explicitly in the top level transition function,
- passive local APIC transitions : $\delta_{\mathbf{apic}} : K_{\mathbf{apic}} \times \Sigma_{\mathbf{apic}} \to K_{\mathbf{apic}}$ (see section B.9),
- passive I/O APIC transitions: $\delta_{\mathbf{ioapic}} : K_{\mathbf{ioapic}} \times \Sigma_{\mathbf{ioapic}} \rightharpoonup K_{\mathbf{ioapic}}$ (section B.10), and
- device transitions which are given by : $\delta_{\mathbf{dev(i)}} : K_{\mathbf{dev(i)}} \times \Sigma_{\mathbf{dev(i)}} \to K_{\mathbf{dev(i)}}$ (see section B.8).

Additionally, we have an output-function

$$\lambda : K \times \Sigma \rightharpoonup \Omega$$

where

$$\Omega = \bigcup_{i=0}^{nd-1} \Omega_{\mathbf{dev(i)}}$$

that allows devices to interact in some way with the external world.

B.11.1 Inputs of the System

We define

$$\Sigma = \Sigma_{\mathbf{p}} \times [0 : np-1] \cup \Sigma_{\mathbf{ioapic+dev}}$$

as the union of processor inputs and I/O APIC and device inputs. In the following, we define both of them.

Definition 68 (Processor Inputs)**.** *We have*

$$\begin{aligned}\Sigma_{p} = {} & \{\textbf{\textit{core}}\} \times K_{\textbf{\textit{walk}}} \times K_{\textbf{\textit{walk}}} \cup \{\textbf{\textit{tlb-create}}\} \times \mathbb{B}^{20} \cup \{\textbf{\textit{tlb-extend}}\} \times K_{\textbf{\textit{walk}}} \times \mathbb{B}^{3} \\ & \cup \{\textbf{\textit{tlb-accessed}}\} \times K_{\textbf{\textit{walk}}} \cup \{\textbf{\textit{sb}}\} \\ & \cup \{\textbf{\textit{apic-sendIPI}}, \textbf{\textit{apic-sendEOI}}, \textbf{\textit{apic-initCore}}, \textbf{\textit{apic-startCore}}\}\end{aligned}$$

Note that the processor and the store buffer are both deterministic, i.e. they have only one active step they can perform. In contrast, the TLB and the local APIC are non-deterministic, i.e. there are several steps that can be performed, thus, the scheduling part of the system's input specifies which step is made.

Definition 69 (I/O APIC and Device Inputs)**.** *We have*

$$\begin{aligned}\Sigma_{\textbf{\textit{ioapic+dev}}} = {} & \{\textbf{\textit{ioapic-sample}}, \textbf{\textit{ioapic-deliver}}\} \times [0 : nd-1] \\ & \cup \{\textbf{\textit{device}}\} \times [0 : nd-1] \times \Sigma_{\textbf{\textit{ext}}}\end{aligned}$$

where

$$\Sigma_{\textbf{\textit{ext}}} = \bigcup_{i \in [0:nd-1]} \Sigma_{\textbf{\textit{ext(i)}}}$$

is the union of all external inputs to devices.

B.11.2 Auxiliary Definitions

In order to define the overall transition relation, we need a view of the memory that can serve read requests of the processor in the way we expect: depending on the address, a read request can go to a local apic, to the I/O-apic, to a device, to the store-buffer, or, if none of the aforementioned apply, to the memory. Depending on whether the machine is running in user mode or system mode, memory accesses are subject to address translation performed using the TLB component of the machine.

Definition 70 (Local APIC Base Address)**.** *The local APIC ports in this machine are mapped to address*

$$apic_{\textbf{\textit{base}}} \equiv 1^{20}0^{12}$$

Definition 71 (Local APIC Addresses)**.** *The set of byte-addresses covered by local APIC ports is given by*

$$A_{\textbf{\textit{apic}}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - apic_{\textbf{\textit{base}}} < 128\}$$

Definition 72 (I/O APIC Base Address)**.** *The I/O APIC ports in this machine are always mapped to address*

$$ioapic_{\textbf{\textit{base}}} \equiv 1^{19}0^{13}$$

Definition 73 (I/O APIC Addresses)**.** *The set of byte-addresses covered by the I/O APIC ports is*

$$A_{\textbf{\textit{ioapic}}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - ioapic_{\textbf{\textit{base}}} < 8\}$$

Definition 74 (Device Base Addresses)**.** *We assume a function*

$$dev_{\textbf{\textit{base}}} : [0 : nd-1] \to \mathbb{B}^{32}$$

to be given that specifies the base address of the ports region of all devices.

Definition 75 (Device Addresses)**.** *The set of addresses covered by device i's ports is given by*

$$A_{\mathbf{dev(i)}} = \{a \in \mathbb{B}^{32} \mid 0 \leq \langle a \rangle - dev_{\mathbf{base}}(i) < 2^{dev_{\mathbf{palen}}}\}.$$

The set of all byte-addresses covered by device, local APIC and I/O APIC ports is defined as

$$A_{\mathbf{dev}} = \bigcup_{i=0}^{nd-1} A_{\mathbf{dev}}(i) \cup A_{\mathbf{apic}} \cup A_{\mathbf{ioapic}}$$

Definition 76 (Port Address)**.** *Given a memory address x, the corresponding port addresses of devices, local APIC and I/O APIC are computed as*

$$dev_{\mathbf{adr}}(i,x) = (x -_{32} dev_{\mathbf{base}}(i))[dev_{\mathbf{palen}}(i) - 1 : 0]$$

$$apic_{\mathbf{adr}}(x) = (x -_{32} apic_{\mathbf{base}}(i))[6 : 0]$$

$$ioapic_{\mathbf{adr}}(x) = (x -_{32} ioapic_{\mathbf{base}}(i))[2 : 0]$$

Definition 77 (Memory System)**.** *The results of read accesses performed by the processor core are described in terms of a memory system that takes into account device ports, I/O APIC ports, local APIC ports, the store buffer and the memory. We define a function ms that, given these components, returns the merged memory view seen by the processor core:*

$$ms(dev, ioapic, apic, sb, m)(x) =$$

$$\begin{cases} sbv(sb,x) & sbhit(sb,x) \\ apic.ports(apic_{\mathbf{adr}}(x)) & \neg sbhit(sb,x) \wedge x \in A_{\mathbf{apic}} \\ ioapic.ports(ioapic_{\mathbf{adr}}(x)) & \neg sbhit(sb,x) \wedge x \in A_{\mathbf{ioapic}} \\ dev(i).ports(dev_{\mathbf{adr}}(i,x)) & \neg sbhit(sb,x) \wedge x \in A_{\mathbf{dev}}(i) \\ m(x) & otherwise \end{cases}$$

Note that, in order to have a meaningful memory system, the machine must be configured in such a way that address ranges of devices, I/O APIC and local APIC are pairwise disjoint.

Definition 78 (Current Address Space Identifier)**.** *The current address space identifier is given by the last* 6 *bits of the special purpose register asid:*

$$asid(core) = core.spr(asid)[5 : 0]$$

B.11.3 Transitions of the Multi-Core MIPS

Let us define the transition function δ and the output function λ of MIPS-86 by a case distinction on the given input a:

$$\delta(c,a) = c'$$

Any subcomponent of configuration c' that is not listed explicitly in the following has the same value as in configuration c.

- $a = (\mathbf{core}, w_I, w_R, i)$ – processor core i performs a step (using walks w_I and w_R if running in translated mode; in system mode, w_I and w_R are ignored)

  In order to formalize a processor core step of processor i, we define the following shorthands:

- $c_i = c.p(i).core$ – the processor core configuration of processor i,
- $ms(i) = ms(c.dev, c.ioapic, c.p(i).apic, c.p(i).sb, m)$ – the memory view of processor i,
- $mode_i = c_i.spr(mode)[0]$ – the execution mode of processor i,
- $trqI = (asid(c_i), c_i.pc, 011)$ – the translation request for instruction execution if processor core i is running in user mode,
- $pff \equiv mode_i = 1 \wedge fault(c.m, trqI, w_I)$ – signals whether there is a page-fault-on-fetch for the given walk w_I and the translation request $trqI$, and
- $pmaI = \begin{cases} w_I.pa \circ c_i.pc[11:0] & mode_i = 1 \\ c_i.pc & mode_i = 0 \end{cases}$
  – the physical memory address for instruction fetch of processor core i (which is only meaningful if no page-fault on instruction fetch occurs),
- $I = ms(i)_4(pmaI)$
  – the instruction fetched from memory for processor core i (in case of a page-fault-on-fetch the value of I has no further relevance),
- $trqEA = (asid(c_i), ea(c_i, I), (store(I) \vee cas(I)) \circ 10)$ – the translation request for the effective address if processor core i is running in user mode,
- $pfls \equiv mode_i = 1 \wedge fault(c.m, trqEA, w_R)$
  $\wedge / pff \wedge (store(I) \vee load(I) \vee cas(I))$
  – the page-fault-on-load-store signal for processor core i.
- $pmaEA = \begin{cases} w_R.pa \circ ea(c_i, I)[11:0] & mode_i = 1 \\ ea(c_i, I) & mode_i = 0 \end{cases}$
  – the physical memory address for the effective address of processor core i,
- $R = \begin{cases} \perp & pff \vee pfls \\ ms(i)_{d(I)}(pmaEA) & otherwise \end{cases}$
  – the value read from memory for a *read* or *cas* instruction of processor core i,
- $eev = eev(c.p(i).apic)$ – the external event vector provided to the processor core by its apic,

$\delta(c, a)$ is defined iff all of the following hold:

- $mode_i = 1 \Rightarrow w_I \in c.p(i).tlb$ – in translated mode, walk w_I must be a walk from the TLB,
- $mode_i = 1 \wedge (cas(I) \vee store(I) \vee load(I)) \Rightarrow w_R \in c.p(i).tlb$ – in translated mode, in case the instruction causes a memory access, the walk w_R must be a walk from the TLB
- $mode_i = 1 \Rightarrow hit(w_I, trqI)$ – running in translated mode, the walk w_I must match the translation request for instruction fetch, and
- $mode_i = 1 \Rightarrow ((store(I) \vee load(I) \vee cas(I)) \wedge \neg pff \Rightarrow (hit(w_R, trqEA)))$ – running in translated mode, if there is a read or write instruction and no page-fault on fetch has occurred, the walk w_R must match the translation request for the effective address, and

- $\neg pff \Rightarrow complete(w_I)$ – if there is no page-fault on fetch, walk w_I is complete, and thus, provides a translation from virtual to physical address, and
- $\neg pfls \Rightarrow complete(w_R)$ – if there is no page-fault on load/store, walk w_R is complete, and thus, provides a translation from virtual to physical address, and
- $(cas(I) \vee mfence(I)) \Rightarrow c.sb = \varepsilon$ – a compare-and-swap or a fence instruction can only be executed when the store-buffer is empty, and
- $load(I) \Rightarrow \neg sbhazard(c.p(i).sb, pmaEA, d(I))$ – there is no store-buffer hazard for the read access the processor tries to perform
- $pmaI \notin A_{\mathbf{dev}}$ – we do not fetch instructions from device ports, and
- $(cas(I) \vee d(I) \neq 4 \wedge (load(I) \vee store(I))) \Rightarrow pmaEA \notin A_{\mathbf{dev}}$ – we exclude compare-and-swap accesses and byte/halfword accesses to device ports, and
- $c.running(i)$ – only processors that are not waiting for a SIPI can execute.

Then,

$$c'.p(j).core = \begin{cases} \delta_{\mathbf{core}}(c_i, I, R, eev, pff, pfls) & i = j \wedge (load(I) \vee cas(I)) \\ \delta_{\mathbf{core}}(c_i, I, \bot, eev, pff, pfls) & i = j \wedge (\neg load(I) \wedge \neg cas(I)) \\ c.p(j).core & otherwise \end{cases}$$

$$c'.p(j).sb = \begin{cases} (pmaEA, sv(c_i, I)) \circ c.p(i).sb & i = j \wedge store(I) \\ & \wedge / jisr(c_i, I, eev, pff, pfls) \\ c.p(j).sb & otherwise \end{cases}$$

$$c'.p(j).apic = \begin{cases} \delta_{\mathbf{apic}}(c'.p(i).apic, \mathbf{jisr}) & j = i \wedge jisr(c, I, eev, pff, pfls) \\ & \wedge il(c, I, eev, pff, pfls) = 1 \\ \delta_{\mathbf{apic}}(c'.p(i).apic, \mathbf{rfe}) & j = i \wedge eret(I) \\ & \wedge / jisr(c_i, I, eev, pff, pfls) \\ c.p(j).apic & otherwise \end{cases}$$

$$c'.p(j).tlb = \begin{cases} \emptyset & i = j \wedge flush(I) \\ tlb' & i = j \wedge invlpg(I) \\ \delta_{\mathbf{tlb}}(c.p(i).tlb, (\mathbf{flush}, asid(c_i), c_i.pc.ba)) & i = j \wedge pff \\ \delta_{\mathbf{tlb}}(c.p(i).tlb, (\mathbf{flush}, asid(c_i), ea(c_i, I).ba)) & i = j \wedge / pff \wedge pfls \\ c.p(j).tlb & otherwise \end{cases}$$

where

$$tlb' = \delta_{\mathbf{tlb}}(\delta_{\mathbf{tlb}}(c.p(i).tlb, (\mathbf{flush}, c_i.gpr(rs(I))[5:0], c_i.gpr(rd(I)).ba)), \\ \mathbf{flush\text{-}incomplete})$$

$$c'.m = \begin{cases} \delta_{\mathbf{m}}(c.m, (c_i.gpr(rd(I)), pmaEA, sv(c_i, I))) & cas(I) \wedge pmaEA \notin A_{\mathbf{dev}} \\ & \wedge / jisr(c_i, I, eev, pff, pfls) \\ c.m & otherwise \end{cases}$$

$$c'.dev(j) = \begin{cases} \delta_{\mathbf{dev(j)}}(c.dev(j), dev_{\mathbf{adr}}(j, pmaEA)) & lw(I) \wedge pmaEA \in A_{\mathbf{dev(j)}} \\ c.dev(j) & otherwise \end{cases}$$

The flag *running* cannot be modified by a processor core step; it can only be modified by the corresponding local APIC. Local APIC and I/O APIC configurations are never modified by a processor core step since neither local APICs nor I/O APIC have side-effects on reads and we do not allow compare-and-swap accesses to devices – writes to devices always go through the store buffer, thus, any side-effects on device writes are triggered when the write leaves the store buffer.

Performing a processor core step of core i, we apply the processor core transition function to the current processor core configuration, providing the instruction word I read from memory, the read value R (if needed), the external event signals $eev(c.p(i).apic)$ provided by the local APIC belonging to processor i, and the page-fault signals *pff*, and *pfls* given above. In case of a *store*-instruction, the corresponding write access enters processor i's store buffer. If there is a page-fault, the TLB reacts by flushing all translations for the page-faulting address – this is necessary in our model in order to allow the MMU to rewalk the page-tables after interrupt handling without triggering the old page-fault. Only in case

of a compare-and-swap instruction, the memory component reacts directly to the compare-and-swap access (since, in all other cases, the store-buffer receives any write requests). Last, in case there is a read-access to a device, the corresponding device transition function is triggered: It specifies how the device reacts to the read-access by specifying appropriate side-effects on reading for the device.

$\lambda(c,a)$ undefined: The processor core does not interact with the external environment, this is exclusive to devices.

Note that continue interrupts can only be caused by execution of instructions that affect only the processor core – thus, continue interrupts are covered in an adequate way in the definitions given here. That is, we do not need to consider changes to other components than the processor core in the case of a continue interrupt.

- $a = (\mathbf{sb}, i)$ – a memory write leaves store buffer i

  $\delta(c,a)$ is defined iff $c.p(i).sb \neq \varepsilon$ – the store buffer can only make a step when it is not empty. Then,

$$c'.p(j).sb = \begin{cases} c.p(i).sb[|c.p(i).sb|-1:1] & i = j \\ c.p(j).sb & i \neq j \end{cases}$$

$$c'.p(j).apic = \begin{cases} \delta_{\mathbf{apic}}(c.p(i).apic, (apic_{\mathbf{adr}}(a), v)) & i = j \wedge c.p(i).sb[0] = (a,v) \\ & \wedge\, a \in A_{\mathbf{apic}} \\ c.p(j).apic & otherwise \end{cases}$$

$$c'.m = \begin{cases} \delta_{\mathbf{m}}(c.m, c.p(i).sb[0]) & c.p(i).sb[0] = (a,v) \wedge a \notin A_{\mathbf{dev}} \\ c.m & otherwise \end{cases}$$

$$c'.ioapic = \begin{cases} \delta_{\mathbf{ioapic}}(c.ioapic, (ioapic_{\mathbf{adr}}(a), v)) & c.p(i).sb[0] = (a,v) \\ & \wedge a \in A_{\mathbf{ioapic}} \\ c.ioapic & otherwise \end{cases}$$

$$c'.dev(j) = \begin{cases} \delta_{\mathbf{dev(j)}}(c.dev(j), (dev_{\mathbf{adr}}(j,a), v)) & c.p(i).sb[0] = (a,v) \\ & \wedge\, a \in A_{\mathbf{dev(j)}} \\ c.dev(j) & otherwise \end{cases}$$

  Store buffer steps never change processor core configurations, TLB configurations or the *running* flag. The oldest write in the store buffer is handled by the component the address belongs to. Note that here, we rely on the correct alignment of accesses, since otherwise, write accesses might partially cover ports and memory at the same time.

  $\lambda(c,a)$ undefined

- $a = (\mathbf{tlb\text{-}create}, va, i)$ – a new walk for virtual address va is created in TLB i

  $\delta(c,a)$ is defined iff

  - $c.p(i).spr(mode)[0] = 1$ – the TLB will only create walks when the processor is running in user mode, and
  - $c.running(i)$ – the TLB will only create walks when the processor is not waiting for SIPI.

– $c.running(i)$ – only when the processor is not waiting for a SIPI, the MMU will start translations.

Then,

$$c'.p(j).tlb = \begin{cases} c.p(i).tlb \cup winit(va, c_i.spr(pto).ba, asid(c.p(i)) & i = j \\ c.p(j).tlb & otherwise \end{cases}$$

Creating a new walk in the TLB is a step that affects only the TLB.

$\lambda(c,a)$ undefined

- $a = (\textbf{tlb-set-accessed}, w, i)$ – accessed bit of the page table entry needed to extend walk w in TLB i is set

  $\delta(c,a)$ is defined iff

  – $c.p(i).spr(mode)[0] = 1$ – page table entry flags can only be set in translated mode,
  – $w \in c.p(i).tlb \wedge \neg complete(w) \wedge w.asid = asid(c.p(i))$ – we only set the accessed bit for incomplete walks of the current address space identifier, and
  – $pte(c.m, w).p = 1$ – the MMU can only set accessed flags for page table entries which are actually present.

  Then,

  $c'.m = \delta_{\mathbf{m}}(c.m, (ptea(w), set\text{-}a(pte(c.m, w))))$

  Setting the page table entry flags only affects the corresponding page table entry in memory. In this model, the MMU non-deterministically sets the accessed flag – enabling walk extension using the given page table entriy.

  $\lambda(c,a)$ undefined

- $a = (\textbf{tlb-extend}, w, i)$ – an existing walk in TLB i is extended

  $\delta(c,a)$ is defined iff

  – $w \in c.p(i).tlb$ – the walk is to be extended is contained in the TLB, and
  – $\neg complete(w)$ – the walk is not yet complete, and
  – $w.asid = asid(c.p(i))$ – the walk is for the current ASID, and
  – $pte(c.m, w).a$ – the accessed flag is set appropriately, and
  – $\neg wext(w, pte(c.m, w), 000).fault$ – the walk extension does not fault result in a faulty walk, and
  – $c.running(i)$ – the TLB will only extend walks when the processor is not waiting for SIPI.

  Then,

  $$c'.p(j).tlb = \begin{cases} \delta_{\mathbf{tlb}}(c.p(i).tlb, \\ \quad \textbf{add-walk}(wext(w, pte(c.m, w), 000))\} & i = j \\ c.p(j).tlb & otherwise \end{cases}$$

  Walk extension only affects the TLB, note however, that in order to perform walk extension, the corresponding page-table entry is read from memory.

  $\lambda(c,a)$ undefined

- $a = (\textbf{apic-sendIPI}, i)$ – local APIC i sends a pending inter-processor-interrupt to all target local APICs

  $\delta(c,a)$ is defined iff

  - $c.p(i).apic.\textbf{ICR}.ds = 1$ – there is currently an inter-processor-interrupt to be delivered, and
  - $c.p(i).apic.\textbf{ICR}.destmode \neq 0$ – the destination mode is set to something other than 0

  Then,

$$c'.p(j).apic = \begin{cases} \delta_{\textbf{apic}}(apic', (type, vect)) & i = j \wedge \textit{self-target} \\ apic' & i = j \wedge \neg \textit{self-target} \\ \delta_{\textbf{apic}}(c.p(j).apic, (type, vect)) & i \neq j \wedge (t = \textbf{ID} \\ & \quad \wedge c.p(j).apic.\textbf{APIC\_ID} \\ & \quad\quad = c.p(i).apic.\textbf{ICR}.dest \\ & \quad \vee t = \textbf{ALL-BUT-SELF} \vee t = \textbf{ALL}) \\ c.p(j).apic & \textit{otherwise} \end{cases}$$

  where

$$vect = c.p(i).apic.\textbf{ICR}.vect[7:0]$$

  is the interrupt vector that is sent over the interrupt bus,

$$type = \begin{cases} \textbf{Fixed} & c.p(i).apic.\textbf{ICR}.dm = 0^3 \\ \textbf{INIT} & c.p(i).apic.\textbf{ICR}.dm = 101 \\ \textbf{SIPI} & c.p(i).apic.\textbf{ICR}.dm = 110 \end{cases}$$

  is the type of interrupt as specified by the command register of the sending local APIC,

$$t = \begin{cases} \textbf{ALL} & c.p(i).apic.\textbf{ICR}.dsh = 10 \\ \textbf{ALL-BUT-SELF} & c.p(i).apic.\textbf{ICR}.dsh = 11 \\ \textbf{SELF} & c.p(i).apic.\textbf{ICR}.dsh = 01 \\ \textbf{ID} & c.p(i).apic.\textbf{ICR}.dsh = 00 \end{cases}$$

  describes the target mode of the requested inter-processor interrupt,

$$\begin{aligned} \textit{self-target} \equiv\ & t = \textbf{SELF} \vee t = \textbf{ALL} \\ & \vee (t = \textbf{ID} \wedge c.p(i).apic.\textbf{APIC\_ID} = c.p(i).apic.\textbf{ICR}.dest) \end{aligned}$$

  expresses whether the sending local APIC is also a target of the inter-processor interrupt, and $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.\textbf{ICR}.ds = 0$.

  Sending an inter-processor-interrupt only affects local APIC configurations – both of the sending local APIC and the receiving ones. All receiving local APICs perform a passive receive-interrupt transition.

  $\lambda(c,a)$ undefined

- $a = (\textbf{apic-sendEOI}, i)$ – local APIC i sends an EOI message to the I/O APIC

  $\delta(c,a)$ is defined iff

  - $\bigvee_i c.p(i).apic.eoipending[i] = 1$ – there is currently an end-of-interrupt message pending

  Then,

  $$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

  where $apic'$ is identical to $c.p(i).apic$ everywhere except

  $$apic'.eoipending[k] = \begin{cases} 0 & k = \langle vect \rangle \\ c.p(i).apic.eoipending[k] & otherwise \end{cases}$$

  and $vect = min\{l \mid c.p(i).apic.eoipending[l] = 1\}_8$

  $c'.ioapic = \delta_{\textbf{ioapic}}(c.ioapic, vect)$

  Transmitting an EOI message affects only the sending local APIC and the I/O APIC. When a local APIC sends an EOI message it is always for the smallest interrupt vector for which an EOI message is pending. The I/O APIC receives the EOI message and reacts with the passive transition given by $\delta_{\textbf{ioapic}}$ that resets the remote interrupt request flag in its redirect table, re-enabling the I/O APIC to sample the corresponding device interrupt.

  $\lambda(c,a)$ undefined

- $a = (\textbf{apic-initCore}, i)$ – local APIC i applies a pending INIT-IPI to processor core i

  $\delta(c,a)$ is defined iff $c.p(i).apic.initrr = 1$ – there is currently an INIT-IPI pending in the local APIC of processor i. Then,

  $$c'.p(j).core = \begin{cases} core' & i = j \\ c.p(j).core & otherwise \end{cases}$$

  where

  $core'$ is identical to $c.p(i).core$ except for $core'.pc = 0^{32}$, $core'.spr(mode) = 0^{32}$ and $core'.spr(eca) = 0^{32}$.

  $$c'.running(j) = \begin{cases} 0 & i = j \\ c.running(j) & otherwise \end{cases}$$

  $$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

  where $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.initrr = 0$.

  When the local APIC applies an INIT-IPI to the corresponding processor core, the store buffer and TLB of that processor as well as the global memory and all other devices are not affected. The INIT-IPI effectively acts as a warm reset to the processor core.

  $\lambda(c,a)$ undefined

- $a = (\textbf{apic-startCore}, i)$ – local APIC i applies a pending SIPI interrupt to processor core i

  $\delta(c,a)$ is defined iff

  - $c.p(i).apic.sipirr = 1$ – there is currently a SIPI pending in the local APIC of processor i, and
  - $c.running(i) = 0$ – the processor is currently waiting for SIPI.

  Then,

  $$c'.p(j).core = \begin{cases} core' & i = j \\ c.p(j).core & otherwise \end{cases}$$

  where

  $core'$ is identical to $c.p(i).core$ except for $core'.pc = c.p(i).apic.sipivect \circ 0^{24}$.

  $$c'.running(j) = \begin{cases} 1 & i = j \\ c.running(j) & otherwise \end{cases}$$

  $$c'.p(j).apic = \begin{cases} apic' & i = j \\ c.p(j).apic & otherwise \end{cases}$$

  where $apic'$ is identical to $c.p(i).apic$ everywhere except $apic'.sipirr = 0$.

  Similar to the INIT-IPI, the store buffer and TLB of that processor as well as the global memory and all other devices are not affected. The interrupt vector of the SIPI is used to initialize the program counter of the processor core and the flag $c.running(i)$ is set in order to allow the processor core to perform steps.

  $\lambda(c,a)$ undefined

- $a = (\textbf{ioapic-sample}, k)$ – the I/O APIC samples the raised interrupt of device k

  $\delta(c,a)$ is defined iff

  - $c.dev(k).irq = 1$ – device k does currently have an interrupt signal activated, and
  - $c.ioapic.redirect(k).mask = 0$ – the interrupt of device k is not masked, and
  - $c.ioapic.redirect(k).rirr = 0$ – handling of any previous interrupt for device k has been acknowledged by an EOI message from the corresponding local APIC.

  Then,

  $c'.ioapic$ is identical to $c.ioapic$ except for

  $$c'.ioapic.redirect(i).ds = \begin{cases} 1 & i = k \\ c.ioapic.redirect(i).ds & otherwise \end{cases}$$

  Sampling a device interrupt only affects the state of the I/O APIC. The delivery status bit of the corresponding redirect table entry is set in order to allow a subsequent **ioapic-deliver** transition.

  $\lambda(c,a)$ undefined

- $a = (\textbf{ioapic-deliver}, k)$ – the I/O APIC delivers a pending interrupt from device k to the target local APIC

  $\delta(c,a)$ is defined iff

  - $c.ioapic.redirect(k).ds = 1$ – there is currently an interrupt pending to be delivered for device k

  Then,

  $$c'.p(j).apic = \begin{cases} \delta_{\textbf{apic}}(c.p(j).apic, (\textbf{Fixed}, v)) & c.p(j).apic.\textbf{APIC\_ID} \\ & \quad = c.ioapic.redirect(k).dest \\ c.p(j).apic & otherwise \end{cases}$$

  where $v = c.ioapic.redirect(k).vect$

  and $c'.ioapic$ is identical to $c.ioapic$ except for the following:

  - $c'.ioapic.redirect(k).ds = 0$
  - $c'.ioapic.redirect(k).rirr = 1$

  Delivering a pending device interrupt only affects the I/O APIC and the target local APIC specified in the redirect table of the I/O APIC. The I/O APIC sets the remote interrupt request flag in order to prevent sampling the same device interrupt several times. Only after the corresponding local APIC acknowledges handling of the interrupt vector to the I/O APIC by sending an EOI message, sampling the device interrupt is possible again.

  $\lambda(c,a)$ undefined

- $a = (\textbf{device}, k, ext)$ – device k performs a step under given external input ext

  $\delta(c,a)$ is defined iff

  - $ext \in \Sigma_{\textbf{dev(k)}}$ – the external input belongs to device k.

  Then,

  $$c'.dev(j) = \begin{cases} \delta_{\textbf{dev(k)}}(c.dev(k), ext) & j = k \\ c.dev(j) & otherwise \end{cases}$$

  Only device k is affected. Execution proceeds as specified by the device transition function.

  $\lambda(c,a) = \lambda_{\textbf{dev(k)}}(c.dev(k), ext)$

  The device may provide an output to the external world as specified by its output function.

Table B.3: *R*-Type Instructions of MIPS-86.

opcode	fun	Mnemonic	Assembler-Syntax	Effect
Shift Operation				
000000	000 000	sll	sll *rd rt sa*	rd = sll(rt,sa)
000000	000 010	srl	srl *rd rt sa*	rd = srl(rt,sa)
000000	000 011	sra	sra *rd rt sa*	rd = sra(rt,sa)
000000	000 100	sllv	sllv *rd rt rs*	rd = sll(rt,rs)
000000	000 110	srlv	srlv *rd rt rs*	rd = srl(rt,rs)
000000	000 111	srav	srav *rd rt rs*	rd = sra(rt,rs)
Multiplication Unit Instructions				
000000	010 000	mfhi	mfhi *rd*	rd = HI
000000	010 001	mthi	mthi *rs*	HI = rs
000000	010 010	mflo	mflo *rd*	rd = LO
000000	010 011	mtlo	mtlo *rs*	LO = rs
000000	011 000	mult	mult *rs rt*	HI,LO = rs $\cdot$ rt
000000	011 001	multu	multu *rs rt*	HI,LO = rs $\cdot$ rt
Arithmetic, Logical Operation				
000000	100 000	add	add *rd rs rt*	rd = rs $+$ rt
000000	100 001	addu	addu *rd rs rt*	rd = rs $+$ rt
000000	100 010	sub	sub *rd rs rt*	rd = rs $-$ rt
000000	100 011	subu	subu *rd rs rt*	rd = rs $-$ rt
000000	100 100	and	and *rd rs rt*	rd = rs $\wedge$ rt
000000	100 101	or	or *rd rs rt*	rd = rs $\vee$ rt
000000	100 110	xor	xor *rd rs rt*	rd = rs $\oplus$ rt
000000	100 111	nor	nor *rd rs rt*	rd = rs $\overline{\vee}$ rt
Test Set Operation				
000000	101 010	slt	slt *rd rs rt*	rd = (rs $<$ rt ? 1 : 0)
000000	101 011	sltu	sltu *rd rs rt*	rd = (rs $<$ rt ? 1 : 0)
Jumps, System Call				
000000	001 000	jr	jr *rs*	pc = rs
000000	001 001	jalr	jalr *rd rs*	rd = pc + 4 pc = rs
000000	001 100	sysc	sysc	System Call
Synchronizing Memory Operations				
000000	111 111	cas	cas *rd rs rt*	rd' = m m' = (rd = m ? rt : m) flushes the SB
000000	111 110	mfence	mfence	flushes the SB
TLB Instructions				
000000	111 101	flush	flush	flushes TLB
000000	111 100	invlpg	invlpg *rd rs*	flushes TLB translations for addr. *rd* from ASID *rs*

Table B.4: *R*-Type Instructions of MIPS-86, continued.

Multiplication Instructions				
opcode	fun	Mnemonic	Assembler-Syntax	Effect
011100	000 000	madd	madd *rs rt*	HI,LO = HI,LO + rs · rt
011100	000 001	maddu	maddu *rs rt*	HI,LO = HI,LO + rs · rt
011100	000 010	mul	mul *rd rs rt*	rd = (rs · rt) [31:0]
011100	000 100	msub	msub *rs rt*	HI,LO = HI,LO - rs · rt
011100	000 101	msubu	msubu *rs rt*	HI,LO = HI,LO - rs · rt

Coprocessor Instructions					
opcode	rs	fun	Mnemonic	Assembler-S.	Effect
010000	10000	011 000	eret	eret	Exception Return
010000	00100		movg2s	movg2s *rd rt*	spr[rd] := gpr[rt]
010000	00000		movs2g	movs2g *rd rt*	gpr[rt] := spr[rd]

Table B.5: MIPS-86 Special Purpose Registers.

i	synonym	
0	sr	status register (contains masks to enable/disable maskable interrupts)
1	esr	exception sr
2	eca	exception cause register
3	epc	exception pc (address to return to after interrupt handling)
4	edata	exception data (contains effective address on pfls)
5	pto	page table origin
6	asid	address space identifier
7	mode	mode register $\in \{0^{31}1, 0^{32}\}$

Table B.6: MIPS-86 Interrupt Types and Priority.

interrupt level	shorthand	internal /external	type	maskable	
0	reset	eev	abort	0	reset
1	I/O	eev	repeat	1	devices
2	ill	iev	abort	0	illegal instruction
3	mal	iev	abort	0	misaligned
4	pff	iev	repeat	0	page fault fetch
5	pfls	iev	repeat	0	page fault load/store
6	sysc	iev	continue	0	system call
7	ovf	iev	continue	1	overflow

Appendix C

Quick Reference

This chapter contains non-trivial definitions in page order.

Definition 1.

$$[\, f(e) \mid e \in \varepsilon \wedge P(e) \,] = \varepsilon,$$

$$[\, f(e) \mid e \in (h \circ t) \wedge P(e) \,] = \begin{cases} f(h) \circ [\, f(e) \mid e \in t \wedge P(e) \,] & P(h) \\ [\, f(e) \mid e \in t \wedge P(e) \,] & o.w., \end{cases}$$

Definition 2.

$$\Pi x \in X.Y(x) = \left\{ f : X \to \bigcup_x Y(x) \;\middle|\; \forall x \in X.f(x) \in Y(x) \right\}.$$

Definition 3.

$$A_{PR,i} = A_{IPR,i} \uplus A_{NPR,i}.$$

Definition 4.

$$A_{DEV} = \bigcup_d A_{DEV,d}$$

Definition 5.

$$A_{IPR} = \bigcup_i A_{IPR,i}.$$

Definition 6.

$$ACC_i = A_{NPR,i} \cup A_{IPR} \cup A_{DEV} \cup A_M.$$

Definition 7.

$$BA = A_{IPR} \cup A_{DEV} \cup A_M.$$

Definition 8.

$$A_{IPI}(x) \subseteq A_{IPR} \cup A_{DEV}.$$

Definition 9.

$$c.m \in Val(\mathcal{A})$$

Definition 10.

$$c.wb(i) \in PVal(BA)^*.$$

Definition 11.

$$dc(A) = \bigcup \left\{ A_{DEV,d} \;\middle|\; A_{DEV,d} \pitchfork A \right\}.$$

Definition 12.

$$idc(A) = A \cup dc(A).$$

Definition 13.

$$closed(A) \equiv A = idc(A).$$

Definition 14.

$$\nu \circledast w(a) = \begin{cases} \delta_{DEV,d}(\nu|_{A_{DEV,d}}, w|_{A_{DEV,d}})(a) & a \in dc(A) \cap A_{DEV,d} \\ w(a) & a \in A \setminus dc(A) \\ \nu(a) & o.w. \end{cases}$$

Definition 15.

$$fms_{iM}(c) = c.m \circledcirc c.wb(i).$$

Definition 16.

$$push(l,e) = l \circ e.$$

Definition 17.

$$pop(l,e) = \begin{cases} \varepsilon & l = \varepsilon \\ tl(l) & o.w. \end{cases}$$

Definition 18.

$$noop(l,e) = l.$$

Definition 19.

$$hit(A,\varepsilon) = \bot,$$
$$hit(A, wb \circ \nu) = \begin{cases} \nu & Dom(\nu) \pitchfork A \\ hit(A,wb) & o.w. \end{cases}$$

Definition 20.

$$hit(A,wb) \equiv hit(A,wb) \neq \bot.$$

Definition 21.

$$phit(A,wb) \equiv hit(A,wb) \neq \bot \wedge A \not\subseteq Dom(hit(A,wb)).$$

Definition 22.

$$\Delta_P(core, fetch, wb, x)$$
$$= \begin{cases} wb = \varepsilon & fence(core,x,fetch) \\ \neg phit(Dom(R(core,x,fetch).wba), wb) & o.w. \end{cases}$$

Definition 23.

$$\Delta_{IPI}(V, wbp) \equiv \bigwedge_{j \in V} wbp(j) = \varepsilon.$$

Definition 24.

$$\Delta_{WB}(wb) \equiv wb \neq \varepsilon.$$

Definition 25.

$$wbp =_i wbp' \equiv wbp(i) = wbp'(i).$$

Definition 26.

$$SC_M(c,x) = SC_{u_M(c,x)M}(c).$$

Definition 27.

$$C_M(c,x) = \begin{cases} A_{PR,i} & x \in \Sigma_{P,i} \\ A_{SC,i} & x \in \Sigma_{WB,i}, \end{cases}$$

Definition 28.

$$\Phi_M(c,x) = \begin{cases} \Phi_P(core_M(c,x),x) & x \in \Sigma_{P,i} \\ 1 & o.w., \end{cases}$$

$$\Upsilon_M(c,x) = \begin{cases} \Upsilon_P(core_M(c,x),x,fetch_M(c,x)) & x \in \Sigma_{P,i} \\ 1 & o.w., \end{cases}$$

Definition 29.

$$LL_{i\downarrow}(c) = 1,$$

Definition 30.

$$LL_{i\uparrow}(c) = \neg SC_{iM}(c).$$

Definition 31.

$$LL_M(c,x) \equiv LL_{u_M(c,x)\ M}(c).$$

Definition 32.

$$fms_M(c,x) = \begin{cases} fms_{iM}(c) & x \in \Sigma_{P,i} \\ \emptyset & o.w. \end{cases}$$

Definition 33.

$$v_M(c,x) = \begin{cases} c.m\big|_{R_M(c,x).bpa} \cup fms_M(c,x)\big|_{R_M(c,x).wba} & LL_M(c,x) \\ c.m\big|_{R_M(c,x)} & \neg LL_M(c,x). \end{cases}$$

Definition 34.

$$PW_M(c,x) = \begin{cases} PW(core_M(c,x),x,fetch_M(c,x),v_M(c,x)) & x \in \Sigma_{P,i} \\ (\emptyset,\emptyset) & o.w. \end{cases}$$

Definition 35.

$$BW_M(c,x) = PW_M(c,x).wba.$$

Definition 36.

$$Op_{iM}(c,x) = \begin{cases} push & x \in \Sigma_{P,i} \wedge BW_M(c,x) \neq \emptyset \\ pop & x \in \Sigma_{WB,i} \\ noop & o.w. \end{cases}$$

Definition 37.

$$W_M(c,x) = \begin{cases} PW_M(c,x).bpa & x \in \Sigma_{P,i} \wedge LL_M(c,x) \\ PW_M(c,x).bpa \cup PW_M(c,x).wba & x \in \Sigma_{P,i} \wedge \neg LL_M(c,x) \\ hd(c.wb(i)) & x \in \Sigma_{WB,i} \wedge LL_M(c,x) \\ \emptyset & x \in \Sigma_{WB,i} \wedge \neg LL_M(c,x). \end{cases}$$

Definition 38.

$$c \blacksquare_M x.m = c.m \circledast W_M(c,x).$$
$$c \blacksquare_M x.wb(i) = Op_{iM}(c,x)(c.wb(i), BW_M(c,x)).$$

Definition 39.

$$victims_M(c,x) = \{\, i \mid A_{PR,i} \hat{\cap} Dom(W_M(c,x)) \wedge x \notin \Sigma_{P,i} \cup \Sigma_{WB,i} \,\}$$

Definition 40.

$$\Delta_M(c,x) = \begin{cases} \Delta_P(core_M(c,x), fetch_M(c,x), c.wb(i), x) & x \in \Sigma_{P,i} \\ \Delta_{WB}(c.wb(i)) & x \in \Sigma_{WB,i}. \end{cases}$$

Definition 41.

$$I_M(c,x) = \Phi_M(c,x) \wedge \Upsilon_M(c,x).$$

Definition 42.

$$\Gamma_M(c,x) = I_M(c,x) \wedge \Delta_M(c,x).$$

Definition 43.

$$\Delta_{IPIM}(c,x) = \Delta_{IPI}(victims_M(c,x), c.wb).$$

Definition 44.

$$\Lambda_M(c,x) \equiv \forall i.x \notin \Sigma_{WB,i} \vee \Delta_M(c,x).$$

Definition 45.

$$c_M[s]^0 = c^0,$$
$$c_M[s]^{t+1} = c_M[s]^t \blacksquare_M s(t).$$

Definition 46.

$$\Gamma_M(s) \equiv \forall t.\Gamma_M[s](t).$$

Definition 47.

$$\Gamma'_M(s) \equiv \forall t' \leq t.\Gamma_M[s](t').$$

Definition 48.

$$\Gamma\Phi'_M(s) \equiv \Gamma'^{-1}_M(s) \wedge \Phi_M[s](t).$$

Definition 49.

$$\Delta'_{IPIM}(s) \equiv \forall t' \leq t.\Delta_{IPIM}[s](t).$$

Definition 50.

$$mv[t \leftrightarrow t+1](l) = \begin{cases} l & l \notin \{\, t, t+1 \,\} \\ t+1 & l = t \\ t & l = t+1 \end{cases}$$

Definition 51.

$$mv\varepsilon(l) = l,\ mv\Omega O(l) = mvO(mv\Omega(l)).$$

Definition 52.

$$\Omega^{-1} = \Omega.$$

Definition 53.

$$\varepsilon^{-1} = \varepsilon,\ (\Omega O)^{-1} = O^{-1}\Omega^{-1}.$$

Definition 54.

$$in_M(c,x) = C_M(c,x) \cup F_M(c,x) \cup R_M(c,x).$$

Definition 55.

$$WS_M(c,x) = Dom(W_M(c,x)).$$

Definition 56.

$$devin_M(c,x) = dc(WS_M(c,x)).$$

Definition 57.

$$bufS(x,A,wb,wb') \equiv \begin{cases} hd(wb) = hd(wb') & x \in \Sigma_{WB,i} \\ \exists w.\ wb = w \circ wb' \wedge \neg hit(A,w) & x \in \Sigma_{P,i}. \end{cases}$$

Definition 58.

$$fin_M(c,x) = \begin{cases} in_M(c,x) & LL_M(c,x) \\ \emptyset & o.w. \end{cases}$$

Definition 59.

$$bufS_M(x,c,c') = bufS(x, fin_M(c,x), c.wb(u_M(c,x)), c'.wb(u_M(c,x))).$$

Definition 60.

$$c =^x_{M,N} c' \equiv c.m =_{C_M(c,x) \cup F_M(c,x)} c'.m \wedge v_M(c,x) = v_N(c',x) \wedge \Delta_M(c,x) \equiv \Delta_N(c',x).$$

$$c =^x_{M,N} c' \equiv c.m =_{C_M(c,x)} c'.m \wedge hd(c.wb(i)) = hd(c'.wb(i)).$$

Definition 61.

$$c =^x_M c' \equiv c =^x_{M,M} c'.$$

Definition 62.

$$out_M(c,x) = idc(WS_M(c,x)).$$

Definition 63.

$$diffu(x,y) \equiv u(x) \neq u(y).$$

Definition 64.

$$diffu[s](t,k) = diffu(s(t),s(k)).$$

Definition 65.

$$int_M(t,k) \equiv u_M(k) \in victims_M(t).$$

Definition 66.

$$ucon_M(t,k) \equiv \neg int_M(t,k) \wedge diffu(t,k).$$

Definition 67.

$$ocon_M(t,k) \equiv out_M(t) \not\cap C_M(k) \wedge o_M(t) \neq o_M(k).$$

Definition 68.

$$out_M[s](I) = \bigcup_{t \in I} out_M[s](t).$$

Definition 69.

$$vws_M[s](t,k) = WS_M[s](t) \setminus out_M[s]((t:k)).$$

Definition 70.

$$vout_M[s](t,k) = dc(WS_M[s](t)) \cup vws_M[s](t,k).$$

Definition 71.

$$\#X(s) = \#\{\, t \mid o(s(t)) = X \,\},$$

Definition 72.

$$\#X[s](t) = \#X(s[0:t-1]).$$

Definition 73.

$$X(s) = \{\, n \mid \exists t.\#X[s](t) = n \,\}.$$

Definition 74.

$$\#X{\approx}n(s) = \varepsilon\,\{\, t \mid o(s(t)) = X \wedge \#X[s](t) = n \,\}.$$

Definition 75.

$$s \equiv_M r \iff \forall X.X(s) = X(r) \wedge \forall t,k.t = \#X{\approx}n(s) \wedge k = \#X{\approx}n(r) \\ \rightarrow s(t) = r(k) \wedge c_M[s]^t =_M^{s(t)} c_M[r]^k.$$

Definition 76.

$$mwrite_M(c,x) \equiv out_M(c,x) \not\subseteq A_{NPR,u_M(c,x)}.$$

Definition 77.

$$issue_M[s]^0(i) = \varepsilon \\ issue_M[s]^{t+1}(i) = Op_{iM}(t)(issue_M[s]^t(i),t).$$

Definition 78.

$$mvO(l) = [mvO(t') | t' \in l].$$

Definition 79.

$$volR_M(c,x) \equiv in_M(c,x) \dot{\cap} A_{DEV} \cup \bigcup_{i \neq u_M(c,x)} A_{IPR,i}.$$

Definition 80.

$$volW_M(c,x) \equiv out_M(c,x) \dot{\cap} A_{DEV} \cup \bigcup_i A_{IPR,i}.$$

Definition 81.

$$vol_M(c,x) = volR_M(c,x) \vee volW_M(c,x).$$

Definition 82.

$$Sh_M(c,x) = \begin{cases} iSh_i\,(core_M(c,x)) \vee vol_M(c,x) \vee \neg SC_M(c,x) & x \in \Sigma_{P,i} \\ 1 & o.w. \end{cases}$$

Definition 83.

$$ShR_M(c,x) = \begin{cases} iShR(core_M(c,x)) \vee volR_M(c,x) \vee \neg SC_M(c,x) & x \in \Sigma_{P,i} \\ 0 & o.w. \end{cases}$$

Definition 84.

$$G_M(c,x) = ShR_M(c,x) \vee mwrite_M(c,x) \wedge Sh_M(c,x).$$

Definition 85.

$$L_M(c,x) = \neg G_M(c,x).$$

Definition 86.

$$WR_M^A(t,t') \equiv out_M(t) \cap in_M(t') \dot{\cap} A.$$

Definition 87.

$$WW_M^A(t,t') \equiv out_M(t) \cap out_M(t') \dot{\cap} A.$$

Definition 88.

$$RW_M^A(t,t') \equiv in_M(t) \cap out_M(t') \dot{\cap} A.$$

Definition 89.

$$X_M(t,t') = X_M^{\mathcal{A}}(t,t').$$

Definition 90.

$$CM_M(t,t') \equiv out_M(t) \dot{\cap} F_M(t').$$

Definition 91.

$$VR_M(t,t') \equiv vout_M(t,t') \dot{\cap} in_M(t').$$

Definition 92.

$$\mathcal{I}_M^t[s](O,k,k') \equiv m_M^k =_{\mathcal{A} \setminus vout_M(t,k)} m_M O^{k'} \wedge \forall i \neq u_M(t).wb_M^k =_i wb_M O^{k'}.$$

Definition 93.

$$\mathcal{J}_M^{f_i}[s](O,k,k') \equiv \forall i.issue_M^k(i) = f_i(mvO^{-1}(issue_M O^{k'}(i))).$$

Definition 94.

$$\frac{t < k \qquad \neg ocon_{\uparrow}(t,k)}{t \rhd k}\ \textsc{NotConcurrent}$$

$$\frac{t = hd(issue_{\uparrow}^k(i)) \qquad s(k) \in \Sigma_{WB,i}}{t \rhd k}\ \textsc{IssueWrite}$$

$$\frac{t < k \qquad s(t) \in \Sigma_{WB,i} \qquad s(k) \in \Sigma_{P,i}}{t \rhd k}\ \textsc{ProcessorFence}$$

Definition 95.

$$\frac{t \rhd k}{t \rhd\!\!\rhd k}$$

$$\frac{t < k \qquad VR_{\uparrow}(t,k) \qquad Sh_{\uparrow}(t) \qquad ShR_{\uparrow}(k)}{t \rhd\!\!\rhd k}\ \textsc{CommExt}$$

Definition 96.

$$\frac{t \Rrightarrow k}{t \blacktriangleright k}$$

$$\frac{\text{INTERRUPTED}\quad t < k \qquad mwrite_{\uparrow}(t) \qquad int_{\uparrow}(k,t)}{t \blacktriangleright k}$$

$$\frac{\text{RMWDISABLE}\quad t < k \qquad ShR_{\uparrow}(t) \qquad mwrite_{\uparrow}(t) \qquad RW_{\uparrow}(t,k)}{t \blacktriangleright k}$$

Definition 97.

$$\Psi_M(c,x) = \Phi_M(c,x) \wedge \Lambda_M(c,x).$$

Definition 98.

$$\Gamma\Psi^t_M(s) \equiv \Gamma^{t-1}_M(s) \wedge \Psi_M[s](t).$$

Definition 99.

$$\begin{aligned} pval[s](t,k) = \Gamma^t_{\uparrow}(s) \\ &\wedge \forall t' \in (t:k).\Gamma_{\uparrow}(t') \vee \Gamma_{\uparrow}[t \to k](t'-1) \\ &\wedge (\Psi_{\uparrow}(k) \vee \Psi_{\uparrow}[t \to k](k-1)). \end{aligned}$$

Definition 100.

$$delay[s](t,k) \equiv pval[s](t,k) \wedge \forall t' \in (t:k].t \not\blacktriangleright [s]t'.$$

Definition 101 (Invariants)**.**

NearlySame *The configurations at $k+1$ and k are nearly the same*

$$\mathcal{I}^t_{\uparrow}([t \to k], k+1, k).$$

IssueSame *Except for the operation made in step t*

$$f_i(l) = Op_{i\uparrow}(t)(l,t),$$

the sequences of issued writes are nearly the same at $k+1$ and k

$$\mathcal{J}^{f_i}_{\uparrow}([t \to k], k+1, k).$$

NothingIssued *No new write buffer entries are issued by the unit making step t*

$$\forall t' \in issue_{\uparrow}[t \to k]^k(u_{\uparrow}(t)).t' < t.$$

Valid *Steps before k are still valid resp. WB-feasible in both schedules*

$$\Gamma\Psi^k_{\uparrow}(s) \wedge \Gamma\Psi^{k-1}_{\uparrow}(s[t \to k]).$$

The validity of step k is also the same

$$\Gamma_{\uparrow}(k) = \Gamma_{\uparrow}[t \to k](k-1).$$

SameState *If step t is a write and step k is valid*

$$mwrite_{\uparrow}(t) \wedge \Gamma_{\uparrow}(k),$$

step t can be moved to k and sees the same values from memory

$$m_{\uparrow}[t \to k]^k =_{in_{\uparrow}(t)} m^t_{\uparrow},$$

and the buffers in step t subsume those in step k after the reordering

$$bufS_{\uparrow}(s(t), c^t_{\uparrow}, c_{\uparrow}[t \to k]^k).$$

Definition 102.

$$\blacktriangleright\!\!\blacktriangleright = \blacktriangleright^* \circ \triangleright.$$

Definition 103.

$$\blacktriangleright\!\!\blacktriangleright = \blacktriangleright^* \circ \blacktriangleright\!\!\blacktriangleright.$$

Definition 104.

$$s \in \mathrm{ABS}_k \equiv \forall t < k.c_{\downarrow}[s]^t =^{s(t)}_{\downarrow,\uparrow} c_{\uparrow}[s]^t.$$

Definition 105.

$$c_{\downarrow} \doteq c \equiv \forall i.c_{\downarrow}.m =_{A_{PR,i}} c.m \wedge c_{\downarrow}.wb =_i c.wb.$$

Definition 106.

$$dirty_M[s](t,i) \equiv SC_{iM}[s](t) \wedge \exists t' \in issue_M[s]^t(i).Sh_M[s](t').$$

Definition 107.

$$clean_M[s](t) \equiv \forall i.\neg dirty_M[s](t,i).$$

Definition 108.

$$ord_M[s](t) \equiv \forall i.dirty_M[s](t,i) \wedge G_M[s](t) \to s(t) \in \Sigma_{WB,i}.$$

Definition 109.

$$s \in \mathrm{ORD}_k \equiv \forall t < k.ord_{\downarrow}(t).$$

Definition 110.

$$sord_M[s](t) \equiv \forall i.dirty_M[s](t,i) \to \neg ShR_M[s](t).$$

Definition 111.

$$s \in \mathrm{ORD}^-_k \equiv s \in \mathrm{ORD}_{k-1} \wedge sord_{\downarrow}[s](k-1).$$

Definition 112.

$$utd_i(k,A) \equiv \forall j \neq i.SC_{j\uparrow}(k) \to \neg hit(A, wb^k_{\uparrow}(j)).$$

Definition 113.

$$sync_i(k,A) \equiv m^k_{\uparrow} =_A \begin{cases} m^k_{\downarrow} & \neg SC_{i\uparrow}(k) \\ pfms_{i\downarrow}(k) & SC_{i\uparrow}(k). \end{cases}$$

Definition 114.

$$minv_i(k,A) \equiv utd_i(k,A) \to sync_i(k,A).$$

Definition 115.

$$\#BW_i[s](0) = 0,$$
$$\#BW_i[s](n+1) = \#BW_i[s](n) + \begin{cases} 1 & BW_{\downarrow}[s](\#P, i \approx n(s)) \neq \emptyset \\ 0 & o.w. \end{cases}$$

Definition 116.

$$s \in bal \iff \forall i. \{ \#BW_i[s](n) \mid n \in P, i(s) \} = WB, i(s).$$

Definition 117.

$$p_{\downarrow}(t) \equiv L_{\downarrow}(t) \wedge wb^t_{\downarrow}(u_{\downarrow}(t)) = \varepsilon \wedge BW_{\downarrow}(t) = \emptyset.$$

Definition 118.

$$O_0 = \varepsilon.$$

$$g_t = \min \left\{ g \geq t \;\middle|\; G_{\downarrow}O_t(g) \vee p_{\downarrow}O_t(g) \right\}.$$

$SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i}$**:**

$$k_t = \min \left\{ k \geq t \;\middle|\; sO_t(k) \in \Sigma_{P,i} \wedge k \leq hd(issue_{\downarrow}O_t^{g_t}(i)) \vee k = g_t \right\}.$$

$\neg(SC_{\downarrow}O_t(g_t) \wedge sO_t(g_t) \in \Sigma_{WB,i})$**:**

$$k_t = \min \left\{ k \geq t \;\middle|\; \neg ucon_{\downarrow}O_t(g_t, k) \right\}.$$

$$O_{t+1} = O_t[t \leftarrow k_t].$$

Definition 119. *We say that schedule s has* a local tail from t to k *and write $\mathcal{L}[s](t,k)$ when all of the following hold.*

1. *The schedule is t-ordered, valid until k, and IPI-valid until $t-1$*

$$\Gamma^k_{\downarrow}(s) \wedge \Delta^{t-1}_{IPI\downarrow}(s) \wedge s \in \mathrm{ORD}_t,$$

2. *and all steps from t and before k are local*

$$\forall t' \in [t:k). L_{\downarrow}[s](t').$$

Definition 120. *We say that schedule s has* a local tail with an independent end from t to k *and write $\mathcal{L}^I[s](t,k)$ when all of the following hold.*

1. *The schedule has a local tail from t to k*

$$\mathcal{L}[s](t,k),$$

2. *the configuration at t is clean*

$$clean_{\downarrow}[s](t),$$

3. *step k is not a write buffer step in strong memory mode*

$$\nexists i.SC_{\downarrow}(k) \wedge s(k) \in \Sigma_{WB,i},$$

4. *and all steps from t and before k are unit-concurrent with k*

$$\forall t' \in [t:k).ucon_{\downarrow}[s](k,t').$$

Definition 121.

$$\mathcal{E}(t,k,k') \equiv c_{\uparrow}[t \leftarrow k]^t =_{\uparrow}^{s[t\leftarrow k](t)} c_{\uparrow}[k' \leftarrow k]^{k'} \wedge \forall t' \in [k':k).ucon_{\uparrow}[k' \leftarrow k](k',t'+1).$$

Definition 122. *We say that* the condition applies at *l and write*

$$ca(l)$$

when all of the following are true.

1. *The configurations at $l+1$ when k is moved to k' and at l when k is moved to k' and then k' is moved to k are nearly the same*

$$\mathcal{I}_{\uparrow}^{k'}[k' \leftarrow k]([k' \to k], l+1, l),$$

2. *the schedule is semi-valid until $l+1$*

$$\Gamma\Phi_{\uparrow}^{l+1}(s[k' \leftarrow k]),$$

3. *if step $l+1$ is step $k'+1$, it is buffering a write, and if it is after $k'+1$, it has a buffered write from $k'+1$*

$$\begin{cases} BW_{\uparrow}[k' \leftarrow k](l+1) \neq \emptyset & k' = l \\ k'+1 \in issue_{\uparrow}[k' \leftarrow k]^{l+1}(i) & k' < l. \end{cases}$$

4. *step $l+1$ is a processor step in strong memory mode*

$$s[k' \leftarrow k](l+1) \in \Sigma_{P,i} \wedge SC_{\uparrow}[k' \leftarrow k](l+1).$$

Definition 123. *We say that schedule s* has a local tail from *t* to *k* that ends with with an issued write buffer step *and write $\mathcal{L}^W(t,k)$ when all of the following hold.*

1. *The schedule has a local tail from t to k*

$$\mathcal{L}[s](t,k),$$

2. *step k is a write buffer step in strong memory mode*

$$s(k) \in \Sigma_{WB,i} \wedge SC_{\downarrow}(k)$$

3. *the write committed at k was already buffered before t*

$$hd(issue_{\downarrow}^{k}(i)) < t,$$

4. only unit i can have dirty buffers at t

$$dirty_{\downarrow}(t,j) \rightarrow i = j.$$

Definition 124. *We say that schedule s* has a local tail from t to k with a global end *if it has an independent end or ends with an issued write buffer step, i.e., there is* $G \in \{ I, W \}$ *such that*

$$\mathcal{L}^{G}(t,k).$$

Definition 125.

$$s \in HW \iff s \in bal \wedge \forall t'.\Gamma_{\downarrow}[s](t') \wedge \Delta_{IPI\downarrow}[s](t').$$

Definition 126.

$$\mathcal{R}[s](t) \equiv \forall i.dirty_{\downarrow}O_t(t,i) \rightarrow sO_t(g_t) \in \Sigma_{WB,i} \wedge issue_{\downarrow}O_t^t(i) = hd(issue_{\downarrow}O_t^{g_t}(i)).$$