Address Spaces and Virtual Memory

Specification, Implementation, and Correctness

Dissertation

zur Erlangung des Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Mark Hillebrand

mah@cs.uni-sb.de

Saarbrücken, Juni 2005

Tag des Kolloquiums:	13. Juni 2005
Dekan:	Prof. Dr. Jörg Eschmeier
Vorsitzender des Prüfungsausschusses:	Prof. Dr.-Ing. Philipp Slusallek
1. Berichterstatter:	Prof. Dr. Wolfgang J. Paul
2. Berichterstatter:	Prof. Dr. Peter-Michael Seidel
3. Berichterstatter:	Prof. Dr. Kurt Mehlhorn
akademischer Mitarbeiter:	Dr. Sven Beyer

Hiermit erkläre ich, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Saarbrücken, im Juni 2005

"Pretend that you're Hercule Poirot: Examine all clues,
and deduce the truth by order and method."
(Donald Knuth. TEX: The Program; Section 1283)

Danksagung

An dieser Stelle möchte ich allen danken, die zum Gelingen der vorliegenden Arbeit beigetragen haben.

Mein Dank gilt zunächst meinen Eltern, die mich während der gesamten Zeit meiner Ausbildung gefördert haben.

Herrn Prof. Wolfgang Paul danke ich für die wissenschaftliche Unterstützung bei meiner Promotion und die Möglichkeit, meine Arbeit an so interessanter Stelle im Verisoft-Projekt fortsetzen und vertiefen zu können.

Herrn Wolfgang Pihan und Herrn Dr. Jörg Walter danke ich für die großzügige Unterstützung durch IBM, sowie, zusammen mit Frau Dr. Silvia Müller, für die ursprüngliche Idee zu dieser Arbeit.

Meiner Freundin Evelyn Becker danke ich für das aufmerksame und hilfreiche Korrekturlesen dieser für sie (möglicherweise) sinnfreien Arbeit und für ständige Ermutigung und Ermunterung während der vergangenen Jahre.

Danken möchte ich auch meinen (ehemaligen und derzeitigen) Arbeitskollegen, zugleich Freunden, bei der IBM in Böblingen und am Lehrstuhl von Prof. Paul in Saarbrücken. Stellvertretend nenne ich hier meine Zimmerkollegen Markus Braun, Sven Beyer und Thomas In der Rieden.

Nicht zuletzt danke ich meinen Freunden Tom Crecelius und Sebastian Schöning, bei denen ich stets Aufnahme und frisch zubereiteten, heißen Tee fand.

Die vorliegende Arbeit wurde teilweise von der IBM Entwicklung GmbH (Böblingen) und im Rahmen des Verbundvorhabens Verisoft vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IS C38 gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei mir.

Abstract

In modern operating systems tasks operate concurrently on a logical memory. Address spaces control access rights to and the sharing of that memory. They are associated with tasks and manipulated dynamically by memory management operations of the operating system.

For cost reasons, logical memory and address spaces are not implemented directly but simulated. The contents of the logical memory are placed in two different memories, the main and the swap memory. Tasks access their address space by using an architecturally defined address translation mechanism, which is implemented by the memory management unit (MMU) and optimized with a translation look-aside buffer (TLB). This mechanism either redirects a memory access to some main memory location or generates a page fault exception resulting in a call to the page fault handler, a low-level operating system procedure.

This construction is correct iff it is transparent to the tasks, so that they behave as if they would operate directly on the logical memory under control of their address spaces. We call the formalization of this correctness criterion a *virtual memory simulation theorem*.

In our thesis we formulate and prove such a theorem for an abstract multiprocessor. We apply the theorem to a concrete implementation, a VAMP [BJK+03] with a single-level address translation mechanism and an exemplary page fault handler. We show how to extend the architecture and proofs to support TLBs, multi-level translation, and multiprocessing.

Kurzzusammenfassung

In modernen Betriebssystemen operieren Programme nebenläufig auf einem logischen Speicher. Der Zugriff auf diesen Speicher und seine gemeinsame Nutzung wird durch Adressräume geregelt. Diese sind den Programmen zugeordnet und können durch Speicherverwaltungsoperationen des Betriebssystems dynamisch manipuliert werden.

Logischer Speicher und Adressräume werden aus Kostengründen nicht direkt implementiert sondern simuliert. Hierbei verteilen sich die Inhalte des logischen Speichers auf zwei verschiedene Speicher, den Haupt- und den Auslagerungsspeicher. Zugriff auf ihren Adressraum wird den Programmen nur unter Nutzung eines durch die Rechnerarchitektur definierten Adressübersetzungsmechanismus' gewährt, der durch die Memory Management Unit (MMU) und den Translation Look-Aside Buffer (TLB) implementiert wird. Dieser Mechanismus lenkt einen Zugriff entweder auf eine Hauptspeicheradresse um, oder er erzeugt einen Seitenfehler, der den Aufruf der Seitenfehlerbehandlung, eines hardware-nahen Betriebssystemteils, einleitet.

Diese Konstruktion ist korrekt, wenn sie für die Programme transparent ist, das heißt, wenn diese sich mit ihr so verhalten, als griffen sie direkt auf den logischen Speicher unter Kontrolle ihrer Adressräume zu. Die Formalisierung dieser Korrektheitsaussage heißt *Simulationssatz für virtuellen Speicher*.

In der vorliegenden Arbeit formulieren und beweisen wir einen derartigen Satz für ein abstraktes Mehrprozessorsystem. Wir wenden ihn auf eine konkrete Implementierung an, den VAMP [BJK+03] mit einem einstufigen Adressübersetzungsmechanismus und einer exemplarischen Seitenfehlerbehandlung. Wir zeigen, wie Rechnerarchitektur und Korrektheitsbeweise für die Unterstützung von TLBs, mehrstufiger Übersetzung und Mehrprozessorbetrieb erweitert werden können.

Extended Abstract

The overall goal of this thesis is to formalize memory and address space models provided by operating systems today and to justify their implementations based on demand paging. This work is divided into two parts.

In the first part, we propose a formal, abstract approach to the problem. We develop a model for multitasking with sharing called the relocated memory machine (RMM). In this model, tasks execute concurrently with arbitrary, fair scheduling. Their computation is based on an underlying virtual processor with which they can perform local computation steps and operations on a shared abstract memory. The *logical memory*, one of its components, serves as a principal data storage. It can be accessed through regular memory operations. These are, however, *relocated*: the *virtual address* supplied by a task is mapped to a *logical address* before the operation is performed. Relocation thus allows to introduce sharing; two virtual addresses (of two tasks) are shared if their logical addresses are equal. Although relocation already provides simple memory protection, a more fine-grained access control over shared memory regions is desirable. Thus, regular memory operations are also subject to a *rights check*: a memory operation may only be performed (and is then called *legal*) if it is member of a rights set associated with the virtual address and the task. Collectively, all relocations and rights sets of a certain task are known as its *address space*. Address spaces form separate components of the shared abstract memory and can be configured dynamically by complex memory operations also called *(operating) system calls*. To the best of our knowledge, all common address space models and operations can be specified within this model.

For cost reasons, the RMM is not implemented directly but simulated. The virtual memory machine (VMM) places the contents of the abstract shared memory in two memories, the main and the swap memory. Its processors run in two different modes. In *user mode*, a memory operation is subject to *address translation*: it is either redirected to some main memory location or generates a page fault exception, which makes the processor enter *supervisor mode* and execute the page fault handler. In the former case, we call the memory operation *attached*, in the latter *detached*. While certainly a detached memory operation may be illegal, the typical reason for detachment is that the requested data is stored in swap memory. Hence, the page fault handler loads it to the main memory, updates the address translation accordingly, and returns to user mode where the operation is retried. This strategy is called *demand paging* [Den70]; it works efficiently if the *working sets* (locations accessed in a certain time interval) of the user programs fit into the main memory.

For the proof of correctness, we proceed in three steps. First, we define a projection function that maps VMM to RMM configurations; it determines the encoding of the logical memory and address spaces. Second, we derive four *access conditions*; if these hold for an attached and legal memory operation, it is performed equivalently in the VMM and the RMM under the projection function. This result is called *step lemma*. Third, we derive the *runtime conditions* that regulate when to attach operations and when not to detach them. For instance: any legal operation, when requested, must eventually be attached; any attached operation must be legal and satisfy the access conditions; an illegal operation must not be attached; translations must not be changed while being used. Since we argue on multiprocessor correctness, parallel aspects have to be taken into account. Finally, we show the *virtual memory simulation theorem*: every sequentially-consistent computation of the VMM yields, by projection, a computation of the RMM. There is no proof for a similar theorem in the literature.

In the second part we use the VAMP, a DLX-like processor, as a reference architecture. The VAMP without address translation has been specified, implemented, and proven correct in the theorem prover PVS [OSR92] in a previous project at our chair [BJK+03].

We extend its architecture and implementation with an address translation mechanism and memory managements units. The implementation is proven to fulfill its specification; mechanical verification of the presented system has been completed [DHP05]. We sketch the implementation of multi-level translation schemes and how to cache translations consistently using a translation look-aside buffer (TLB).

Furthermore, we develop a multiprocessor VAMP. First, we introduce a two-phased instruction set architecture (with instruction fetch and instruction execution as separate computation steps) and a memory operation architecture. Second, for the implementation, we take the unmodified VAMP processor core. For updating translations consistently, a distributed asynchronous communication mechanism is necessary. We add a barrier mechanism that halts processors running user programs and, as a side effect, clears all entries of every TLB in the machine. Similar yet more elaborate constructions are used in real multiprocessors (e.g. [IBM00]). Viewed in isolation, the barrier mechanism is devoid of meaning. It fully reveals its purpose only in the third step, the correctness proof. The proof consists of two parts. In the first part, we show that the VAMP processor core is correct with respect to a decoupled instruction semantics that is parameterized over a series of memory responses. In the second part, we show that the processors are correctly coupled with the memory, which we assume to be sequentially-consistent for untranslated accesses. In particular, we show that translations are correct if the barrier mechanism is properly used and that pre-fetches retain sequential instruction semantics. With respect to the original VAMP correctness proof, the order of the final two proof steps had to be reversed; we reason on (decoupled) interrupt correctness first and then on prefetching / coupling the processor cores with the memory. This change is unavoidable for a multiprocessor correctness proof. By specialization, of course, the new proof applies to the single-processor VAMP as well.

Finally, we apply the virtual memory simulation theorem to the single-processor VAMP running an exemplary page fault handler. The handler is embedded in a minimal operating system with multitasking but without sharing. Correctness is shown by application of the VMM formalism; an implementation-specific lemma for the page fault handler entails the runtime conditions. As there is no parallelism at all in the system, the step lemma almost suffices to show overall functional correctness. For liveness we show that no more than two page faults are generated by any instruction (one for fetch and the other for load / store).

Zusammenfassung

Das Ziel dieser Arbeit liegt in der Formalisierung von Speicher- und Adressraummodellen, die Betriebssysteme heutzutage anbieten, sowie in der Rechtfertigung ihrer Implementierung, die auf dem Prinzip des *Seitenwechsels nach Bedarf* (Demand Paging) beruht. Die vorliegende Arbeit unterteilt sich in zwei Teile.

Im ersten Teil stellen wir einen formalen und abstrakten Ansatz vor, um das Thema zu behandeln. Wir entwickeln ein Modell für Mehrprogrammbetrieb mit gemeinsamer Speichernutzung, das wir *Relocated Memory Machine* (RMM) nennen. In diesem Modell werden Programme nebenläufig mit beliebigem aber fairem Scheduler ausgeführt. Ihre Berechnungen basieren auf einem virtuellen Prozessor, der lokale Berechnungsschritte und Operationen auf einem geteilten, abstrakten Speicher ausführen kann. Der *logische Speicher*, eine der Komponenten des abstrakten Speichers, fungiert als Haupt-Datenspeicher. Auf ihn kann über herkömmliche Speicheroperationen zugegriffen werden. Allerdings werden diese *reloziert*, das heißt, vor der Ausführung der Operation wird die von einem Programm spezifizierte *virtuelle Adresse* auf eine *logische Adresse* abgebildet. Relozierung ermöglicht somit eine gemeinsame Speichernutzung; zwei virtuelle Adressen (von zwei Programmen) werden gemeinsam genutzt, wenn sie auf die gleiche logische Adresse abgebildet werden. Hierdurch ergibt sich bereits die Möglichkeit eines einfachen Speicherschutzes, dennoch ist eine feingranulare Zugriffskontrolle über geteilte Speicherbereiche wünschenswert. Darum sind herkömmliche Speicheroperationen einer Prüfung der Zugriffsrechte unterworfen: Eine Speicheroperation kann nur dann ausgeführt werden, wenn sie Element einer der Adresse und dem Programm zugeordneten Zugriffsrechtemenge ist, und heißt in diesem Fall *erlaubt*. Alle Relozierungen und Rechtemengen eines Programms zusammen bezeichnen wir als dessen *Adressraum*. Adressräume sind vom logischen Speicher abgetrennte Komponenten des abstrakten Speichers. Sie können zur Laufzeit über komplexe Speicheroperationen manipuliert werden. Diese Operationen entsprechen *(Betriebs-) Systemaufrufen* zur Speicherverwaltung. Unseres Wissens nach ist dieses Modell ausreichend für üblicherweise benutzte Adressraummodelle und die hierauf definierten Operationen.

Die RMM wird aus Kostengründen nicht direkt implementiert sondern simuliert. Die Virtual-Memory-Machine (VMM) legt hierfür den abstrakten Speicher in zwei eigenen Speichern ab, dem Haupt- und dem Auslagerungsspeicher. Ihre Prozessoren unterstützen zwei verschiedene Betriebssmodi. Im *Benutzermodus* ist jede Speicheroperation einer *Adressübersetzung* unterworfen: Die Operation wird entweder auf eine Hauptspeicheradresse umgeleitet oder sie löst einen Seitenfehler aus, der den Prozessor in den Aufsehermodus versetzt und die Ausführung der Seitenfehlerbehandlung startet. Im ersten Fall nennen wir die Operation *verbunden*, im zweiten Fall *getrennt*. Natürlich kann eine getrennte Speicheroperation verboten sein, der normale Grund für die Trennung ist jedoch, dass die angefragten Daten sich im Auslagerungsspeicher befinden. Die Seitenfehlerbehandlung lädt in diesem Fall die angefragten Daten in den Hauptspeicher, aktualisiert die Adressübersetzung entsprechend, und der Prozessor kehrt in den Benutzermodus zurück, wo er den Zugriff wiederholt. Diese Strategie nennt man *Seitenwechsel nach Bedarf* [Den70]; ihre Effizienz hängt von der (zeitlichen und räumlichen) Lokalität von Datenzugriffen der Benutzerprogramme ab.

Den Korrektheitsbeweis führen wir in drei Schritten. Erstens definieren wir eine Projektionsfunktion, die VMM- auf RMM-Konfigurationen abbildet; diese Funktion bestimmt die Kodierung von logischem Speicher und Adressräumen. Zweitens leiten

wir vier *Zugriffsbedingungen* ab; wenn diese für eine verbundene und erlaubte Speicheroperation gelten, führt die VMM sie äquivalent zur RMM aus (unter Anwendung der Projektionsfunktion). Dieses Ergebnis nennen wir *Schrittlemma*. Drittens leiten wir *Laufzeitbedingungen* ab, die regeln, wann Operation verbunden werden, und wann sie nicht getrennt werden dürfen. Beispielsweise fordern wir, dass jede erlaubte Operation nach ihrer Anfrage irgendwann verbunden wird, dass jede verbundene Operation erlaubt sein muss und die Zugriffsbedingungen erfüllt, dass keine verbotene Operation verbunden ist, und dass Übersetzungen während ihrer Benutzung nicht aktualisiert werden. Da sich der Beweis auf ein Mehrprozessorsystem bezieht, sind Aspekte der Parallelität in Betracht zu ziehen. Schließlich können wir den Simulationssatz für virtuellen Speicher zeigen: Jede sequentiell-konsistente Berechnung der VMM ergibt durch geeignete Projektion eine Berechnung der RMM. In der Forschungsliteratur findet sich kein Beweis für ein derartiges Theorem.

Im zweiten Teil benutzen wir den VAMP, einen DLX-artigen Prozessor, als Referenzarchitektur. In einem vorangegangen Projekt [BJK+03], wurde der VAMP ohne Adressübersetzung spezifiziert, implementiert und im Theorembeweiser PVS [OSR92] als korrekt bewiesen.

Wir erweitern Architektur und Implementierung mit einem Adressübersetzungsmechanismus und Memory Management Units. Es wird gezeigt, dass die Implementierung ihre Spezifikation erfüllt; eine computer-gestützte Verifikation des vorgestellten Systems ist fertiggestellt [DHP05]. Wir skizzieren die Implementierung eines mehrstufigen Übersetzungsmechanismus', und wie sich Übersetzungen konsistent mit einem Translation-Look-Aside Buffer (TLB) puffern lassen.

Weiterhin entwicklen wir ein Mehrprozessor-VAMP-System. Zunächst stellen wir eine 2-Phasen-Befehlsarchitektur (mit Laden und Ausführen als getrennten Berechnungsschritten) und eine Speicheroperationsarchitektur vor. Wir verwenden für die Implementierung den unveränderten VAMP Prozessorkern. Um Übersetzungen in einem Mehrprozessorsystem konsistent zu verändern, ist ein verteilter, asynchroner Kommunikationsmechanismus zwingend erforderlich. Wir führen ihn in Form eines Schrankenmechanismus' ein, der Programme im Benutzermodus anhält und, als Nebeneffekt, die Einträge sämtlicher TLBs im System löscht. Ähnliche, wenn auch verfeinerte, Konstruktionen finden sich in echten Mehrprozessorsystemen (z.B. [IBM00]). Für sich betrachtet, scheint der Schrankenmechanismus ohne Zweck, doch er enthüllt sich schließlich im Korrektheitsbeweis. Dieser besteht aus zwei Teilen. Im ersten Teil zeigen wir, dass der VAMP Prozessorkern korrekt in Bezug auf eine entkoppelte Befehlssemantik arbeitet, die über eine Folge von Speicherantworten parametrisiert ist. Im zweiten Teil zeigen wir, dass die Prozessoren korrekt mit dem Speicher gekoppelt werden, den wir als sequentiell-konsistent für nicht-übersetzte Anfragen annehmen. Hierbei ist unter anderem zu zeigen, dass Adressen unter Nutzung des Schrankenmechanismus' korrekt übersetzt werden, und dass Pre-Fetches die sequentielle Befehlssemantik erhalten. Im Unterschied zum ursprünglichen VAMP Korrektsheitsbeweis muss für unseren Korrektsheitsbeweis die Reihenfolge der letzten beiden Beweisschritte vertauscht werden: Wir untersuchen zunächst die (entkoppelte) Korrektheit von Unterbrechungen, und dann die Korrektheit von Pre-Fetching und der Ankopplung der Prozessorkerne an den Speicher. Die Änderung der Reihenfolge ist unvermeidlich für einen Korrektheitsbeweis über ein Mehrprozessorsystem. Natürlich lässt sich der neue Beweis durch Spezialisierung auf den Einprozessor-Fall anwenden.

Zuletzt wenden wir den Simulationssatz für virtuellen Speicher auf den Einprozessor-VAMP mit einer exemplarischen Seitenfehlerbehandlung an. Diese ist in ein

minimales Betriebssystem eingebettet, das Multi-Tasking ohne Sharing unterstützt. Die Korrektheit wird durch eine Anwendung des VMM-Formalismus' bewiesen; ein implementierungs-abhängiges Lemma über die Seitenfehlerbehandlung zeigt die Gültigkeit der Laufzeitbedingungen. Da das System keinen Parallelismus nutzt, ist das Schrittlemma nahezu ausreichend für die gesamte funktionale Korrektheit. Die Lebendigkeit betreffend zeigen wir, dass keine Instruktion mehr als zwei Seitenfehler auslösen kann – einen für das Laden der Instruktion sowie einen weiteren für einen Lese- oder Schreibzugriff.

Contents

Table of contents

List of Figures

Chapter 1

Introduction

Computer systems grow quickly in size nowadays, not only exponentially in the size of their building blocks (for example the number of transistors on a processor [Moo65]), but also, with the advent of GRID computing [FK99], in the number of components comprising a single distributed system.

With the pervasive use of computers in critical and sensitive environments (think of automotive or aeronautical engineering, nuclear power plants, pacemakers, bank accounts, electronic payment systems), errors potentially have disastrous consequences for body and purse. It has been known for decades that the absence of errors may only be proven. Simulation and testing, however useful they are for debugging and evaluation of designs, are "hopelessly inadequate" for this purpose [Dij72].

Although isolated parts of computer systems have been examined, modeled in varying levels of detail, and proven correct, complete computer systems comprising hard- and software have not (with the exception of [Boy89]). Therefore, results of formal verification must always be treated carefully. Absence of bugs is only proven relative to the chosen model. Even slight incompatibilities in the correctness models of components to be combined can allow fatal errors to creep in. With the number of components such opportunities proliferate.

The only solution to this problem is *systems*, *pervasive*, or *persistent verification* [BHMY89, Moo03, Ver03], the use of formal, computer-aided verification throughout all layers of abstraction of a computer system. Pervasive verification thus embraces a system from its transistors to communicating, concurrently running programs. For all layers considered and for every transition between layers, human errors are excluded, full coverage is achieved, and the results are based on a well-known small set of assumptions.

The isolation of verification results mentioned above applies in particular for hardware design and programs. It may be said that the hardware-software gap (again, with the exception of [Boy89]) is yet unabridged by formal verification methods. In terms of implementation, operating systems connect hardware and software worlds. They provide complex execution environments for programs. Every operating system supporting virtual memory is based on address translation mechanisms developed in the early 1960s [KHPS61]. Since their inception, these mechanism have been employed to provide increasingly complex and flexible system / programming

models (for example supporting shared memory segments, memory-mapped I/O, interprocess communication, external pagers, recursive virtual machines, (para-) virtualization [BCD72, BH70, RR81, YTR+87, IBM05, HP01, BDF+03]) but also for efficient implementations thereof (for example demand, asynchronous, and pre-paging, copy-on-write, zero-copy, shared libraries [Den70, BBMT72, FR86, Chu96, BCD72]). Overall, virtual memory techniques have helped solving problems in "storage allocation, protection of information, sharing and reuse of objects, and linking of program components" [Den96]. This is true even for (small) systems that do not inherently support swapping, such as the L4 microkernel [Lie95].

Apart from good implementation and documentation practice (see, for example, [Gor04a]), though, it seems that no attempts have been made to capture overall correctness or even correctness criteria of the "VM", the virtual memory engine, of general-purpose operating systems. In this thesis we try to close this gap mathematically and consider a virtual memory system from gates to tasks.

Outline

The remainder of this thesis is organized in seven chapters.

- In Chapter 2 we give a general introduction into the formalization of computer architectures. In particular, we show the fundamental difference between single-processor and multiprocessor specification. We advocate a multiprocessor specification style that clearly separates processor-local computation steps and shared memory operations. Sequential consistency [Lam79] is introduced; a sequentially consistent memory guarantees that every trace of parallel memory operations is executed according to a certain, not a priori known sequential order.

- In Chapter 3 we develop the formal definition of the relocated memory machine (RMM), a multitasking multiprocessor with memory relocation and protection. The central component of the RMM is the *logical memory*, which can be accessed by task under control of *address spaces*. The RMM can serve as a basis for various aspects of software specification—including that of operating system memory management and task management. On the other hand, the RMM is also the natural specification machine for a real system implementing virtual memory with hardware and software support.

- In Chapter 4 we present the virtual memory machine (VMM). It simulates the RMM's logical memory using a combination of RAM and hard disk storage. As only RAM is directly accessible, the system has to guarantee that data is moved to the RAM whenever it is needed by a program. This is called *demand paging*. We identify the required invariants on a fine-grained level, carefully separating static and dynamic aspects of correctness. Then, we show overall correctness of the VMM against the RMM; in principle, demand paging and related techniques must be transparent to the user tasks.

- In Chapters 5 to 7 we use the VAMP, a DLX-like processor, as a reference architecture. In Chapter 5 we extend it with a single-level address translation mechanism. We show how to implement the mechanism using memory management units (MMU) and how to optimize it using translation look-aside buffers (TLBs). Multi-level address translation, which allows for a flexible memory organization

(and a bunch of new implementation tricks), is presented briefly. The proof of the original VAMP is sketched and adapted at the necessary locations.

- In Chapter 6 we employ the VAMP processor cores in a multiprocessor. A barrier mechanism, consisting only of two trees of gates, is implemented. It allows the halting of processors in user mode for consistent updates of translations. A new proof architecture is developed for the correctness proof. With it, fully decoupled correctness of each processor core is shown, then the coupling of the processors with the memory is proven correct. In the latter step, we deal with the problem of establishing sequential consistency in the presence of prefetching.
- In Chapter 7 we present a page fault handler for the single-processor VAMP. Its correctness is shown by application of the generic VMM correctness theorem. We show how to develop the simple page fault handler into a competitive one.
- Chapter 8 concludes with a summary and a discussion of future work.

Related work is discussed at the end of each chapter. Large parts of this thesis work were financed by and done at IBM Entwicklung GmbH (Böblingen, Germany). Their generous support is gratefully appreciated.

Chapter

2

Basics

Contents

A single-processor machine is traditionally modeled by specifying a set of processor configurations, a set of memory configurations, and a transition function. The machine operates by successively applying the transition function to the current processor and memory configuration. On a more fine-grained level, an instruction-set architecture consisting of individual transition functions called *instructions* specifies the computation steps that the machine can perform. The overall transition function comprises the behavior of all individual instructions; the next configuration is obtained by applying the *current instruction* (determined by the processor configuration) to the machine configuration. The instruction-set architecture allows classification of instructions; the one we are interested in most is that of compute and memory instructions: Compute instructions solely depend on and affect the processor configuration. Memory instructions on the other hand also depend on or affect the memory configuration.

This transition function approach must be modified for a multiprocessor, a machine with many processors, e.g. operating on a shared memory. Here the processors and

the (shared) memory are modeled as individual components that run in parallel. The connection between the processor and the memory is established via an interface; the processor requests memory operations, the shared memory acknowledges them.

Accordingly, memory instructions for a processor are defined by the data they *send* over the interface at request time and how they process the *received* data from the memory at acknowledgment time. Dually, we define for the memory how it processes data from the processor and what data it sends back to the processor. Like the processor with its instruction-set architecture, the memory has a memory operation architecture. Corresponding to instructions, the memory operation architecture consists of memory operations that map processor inputs and memory configurations to processor outputs and updated memory configurations. We still have to define how the memory handles parallel operations. This leads to the definition of sequential consistency.

This chapter proceeds as follows. We review single-processor machine specification in Section 2.1. Thereafter, we treat multiprocessor specification in three sections. In Section 2.2 we formalize interfaces and their handshake. In Section 2.3, we present a generic (multi-) processor instruction-set architecture and its counterpart, the memory operation architecture. Finally, Section 2.4 deals with parallelism and shared memory consistency.

2.1 Single-Processor Machine

A single-processor machine can be thought of as consisting of two components, a *processor* and a *memory*. In each computation step the processor configuration is updated according to the *current instruction word*, a value at a special location in memory. The set of all these values and their effects on the computer is called *instruction-set architecture* (ISA). Instructions that depend on or affect the memory configuration are called *memory instructions*.

Formally, let P denote the set of processor configurations and M denote the set of memory configurations. The machine has configurations of the set $C := P \times M$. For the pair $c = (p,m) \in C$ we use the notation $p(c) := p$ and $m(c) := c$ to denote the first and second component. Function spaces are denoted by $[C \to C]$, membership of an object f in a function space is denoted by $f \in [C \to C]$ or $f : [C \to C]$.

An instruction $i : [C \to C]$ maps a configuration $c \in C$ to its next configuration $c' \in C$ via $c' = i(c)$. The instruction-set architecture $Isa \subseteq [C \to C]$ is the set of instructions that the machine can perform. Every configuration $c \in C$ determines a unique instruction i by the so-called decode function $dec : [P \to Isa]$. Using the decode function, we define the next-configuration function $\delta : [C \to C]$ as

$$\delta(c) := dec(p(c))(c) \; . \tag{2.1}$$

Instructions can be classified according to the way they interact with memory configurations.

Instructions $i \in I$ that neither depend nor affect the memory configuration are called *compute instructions*. Formally, an instruction i is a compute instruction iff it can be replaced by a function $i_p : [P \to P]$ that computes the new processor configuration from the old processor configuration: i_p must satisfy $i(p,m) = (i_p(p),m)$ for all configurations $(p,m) \in C$.

On the other hand, we have *memory instructions* violating the above property. Memory instructions with the potential to change the memory configuration are called

writing memory instructions. Memory instructions whose result possibly depends on the memory configuration are called *reading memory instructions*. These two conditions are not exclusive; memory instructions may be writing and reading at the same time.

Formally, for writing memory instructions we have a configuration $c \in C$ such that $m(i(c)) \neq m(c)$. For reading memory instructions, we have a processor configuration $p \in P$ and two memory configurations $m_1, m_2 \in M$ such that $p(i(p,m_1)) \neq p(i(p,m_2))$.

2.2 Interfaces

An interface describes the connection between modules. Its definition is based on the wires used between modules. The set of all observed wire signals is called the set of *interface observations*. An observation of an interface over time, or formally a function mapping points in time to interface observations, is called an *interface trace* or simply a trace. Not all traces may occur for a given interface; often conditions restrict the set of acceptable traces. These conditions we call *handshake conditions*.

In this section, we define a class of interfaces with simple handshake conditions that we will use throughout this thesis. To support easy analysis, our interfaces have three important characteristics: they connect only two modules, they are *one-way* (or *unidirectional*) and *sequential*. *One-way* means that always the same module is requesting for some action to be performed while the other module performs the action. The former is called the *sender*, the latter the *receiver*. *Sequential* means that requests are strictly separated and do not overlap.

For the interface employed between a processor and the shared memory, the action of requesting and acknowledging can be sketched as follows:

1. The processor raises a request line *req* and sets the desired memory operation *mop* and the desired input data *din*. The processor is required to keep the input data stable and the request line raised until the completion of the request.

2. After some time, the memory responds by raising an acknowledgment line *ack* and returns the output data *dout*. Request and acknowledgment may coincide.

Figure 2.1 depicts a sample timing for a request at an interface. We formalize this figure in the following sections.

2.2.1 Interface Observation and Traces

With *Iobs* we denote the set of *interface observations*. An element $iobs \in Iobs$ is a 5-tuple

$$iobs = (req, mop, din, ack, dout) \in \mathbb{B} \times Mop \times Din \times \mathbb{B} \times Dout\ .$$

The boolean variables $req, ack \in \mathbb{B}$ denote request and acknowledgment signals. The variable $mop \in Mop$ denotes a memory operation identifier from a finite set of memory operation identifiers Mop. The variables $din \in Din$ and $dout \in Dout$ denote the input and the output data of the memory operations; *Din* and *Dout* are uninterpreted sets of data inputs and outputs.

A sequence of interface observations is called an *interface trace*. Usually, a trace is indexed over the natural numbers modeling discrete time. For a multiprocessor trace,

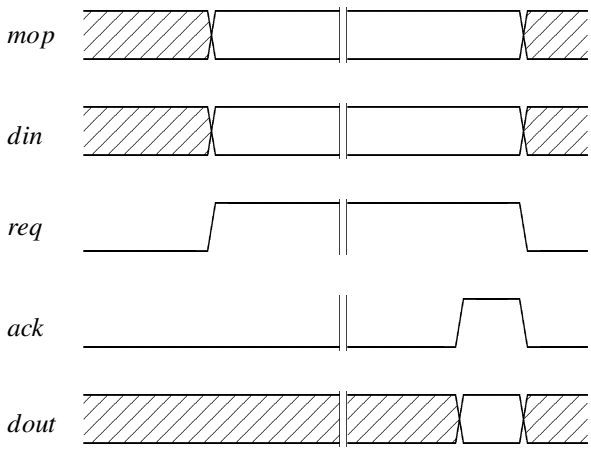

Figure 2.1 Timing Diagram of an Interface Request

we index over time and processor numbers $i \in \{1,\ldots,n\}$. The function space of all multiprocessor traces is denoted by Trc and defined as

$$Trc = [\mathbb{N} \times \{1,\ldots,n\} \to Iobs] \,.$$

2.2.2 Handshake Conditions

We restrict the traces to a subset we call valid traces. Let $trc \in Trc$ be a trace. Let $t \in \mathbb{N}$ denote a time and let $i \in \{1,\ldots,n\}$ denote a processor index. Let $e := trc(t,i) \in Iobs$ denote the associated interface observation. Three so-called *handshake conditions* must be satisfied:

- Request inputs must be stable. If the request flag is observed valid for the event e but the acknowledgment flag is observed invalid, then the memory operation, the data input, and the request flag must not change for the next observation on that processor. If we abbreviate $e' := trc(t+1,i)$ this condition can be written as

$$req(e) \wedge \neg ack(e) \Rightarrow req(e') \wedge (din(e) = din(e')) \wedge (mop(e) = mop(e')) \,. \quad (2.2)$$

- There must be no over-acknowledgments. If the request flag is observed invalid for the event e then the acknowledgment flag must also be observed invalid. Hence,

$$\neg req(e) \Rightarrow \neg ack(e) \,. \quad (2.3)$$

- Each request must be acknowledged. If the request flag is observed valid, there must be a later time $u \geq t$ with an acknowledgment for that processor:

$$req(e) \Rightarrow \exists u \geq t : ack(trc(u,i)) \quad (2.4)$$

Building on these three handshake conditions, we can decompose an interface trace into a uniquely defined series of requests.

A request starts with a rising edge at the request line or when an acknowledgment had been given and the request line remains active. We call the latter condition a *burst request*; it allows requests to follow each other "back-to-back" without a pause. Formally, we let the predicate $reqstart(i,t)$ denote the start of a request for processor i at time t. We define:

$$\begin{aligned} reqstart(i,t) = req(trc(t,i)) \\ \wedge (t = 0 \vee \neg req(trc(t-1,i)) \vee ack(trc(t-1,i))) \end{aligned} \tag{2.5}$$

A request ends the first time an acknowledgment signal has been given. To cover the whole duration of a request, we define the predicate $isreq(t,t',i)$ to hold if a request at processor i occurs between the times t and t'. It is defined as follows:

$$\begin{aligned} isreq(t,t',i) = reqstart(i,t) \\ \wedge t' \geq t \\ \wedge ack(trc(t',i)) \\ \wedge \forall t \leq \tilde{t} < t' : \neg ack(trc(\tilde{t},i)) \end{aligned} \tag{2.6}$$

We have the following simple lemmas characterizing the structure of valid traces:

◀ Lemma 2.1 *For all request starts for processor i at time t there is a minimal time $t' \geq t$ when an acknowledgment is given. Thus, t and t' denote the period of a request at processor i. With $\exists!$ denoting unique existential quantification we have*

$$\forall i,t : reqstart(i,t) \Rightarrow \exists! t' : isreq(t,t',i)\ . \tag{2.7}$$

Given i and t we denote the time t' also by $end(t,i)$.

◀ Lemma 2.2 *Let i denote a processor and $\tilde{t}$ denote a time with an active request signal for i. Then, there are uniquely defined times t and t' with $t \leq \tilde{t} \leq t'$ such that from t to t' we have a complete request for processor i:*

$$\forall i,\tilde{t} : req(i,\tilde{t}) \Rightarrow \exists! (t,t') : isreq(t,t',i) \tag{2.8}$$

In this context, we denote the time t by $strt(t,i)$ and the time t' by $end(t,i)$. The following equations hold:

$$strt(\tilde{t},i) = \max\{0\} \cup \{t \leq \tilde{t} \mid ack(trc(t-1,i)) \vee \neg req(trc(t-1,i))\} \tag{2.9}$$

$$end(\tilde{t},i) = \min\{t' \geq \tilde{t} \mid ack(trc(t',i))\} \tag{2.10}$$

◀ Lemma 2.3 *For any request at processor i from times t to t', the request inputs are stable for the complete duration of the request, so,*

$$\begin{aligned} \forall i,t,t' : isreq(t,t',i) \Rightarrow \forall t \leq \tilde{t} \leq t' : \\ (req(i,\tilde{t}) = req(i,t)) \wedge (mop(i,\tilde{t}) = mop(i,t)) \wedge (din(i,\tilde{t}) = din(i,t))\ . \end{aligned} \tag{2.11}$$

2.3 Instruction Set Architecture

In this section we describe how to define an instruction-set architecture for multiprocessors. In single-processors, computation was modeled as a simultaneous update on the processor and the memory configuration. This close link cannot be preserved in a multiprocessor; all memory operations are executed over the memory interface. We need a separate semantics for the processor as well as for the memory. The former is called the *(processor) instruction-set architecture* (ISA) and the latter is called the *memory operation architecture*. We describe them both in the following two sections. In the third section we point out the subtle differences of single-processor ISAs to new ISAs. In the fourth section, we present a simple RISC architecture as an example.

2.3.1 Processor Instruction Set Architecture

Let Mop denote the set of memory operation identifiers, let Din denote the set of data inputs and let $Dout$ denote the set of data outputs. To model the access to the memory through the interface we need three components for each memory instruction: a memory operation identifier $mop \in Mop$, a send function snd and a receive function rcv. The send function $snd : [P \to Din]$ maps a processor configuration to a data input. The receive function $rcv : [P \times Dout \to P]$ maps a processor configuration and a data output to a next processor configuration. On the other hand, compute instructions are explicitly modeled as operations $cmp : [P \to P]$ on processor configurations.

An instruction i for a multiprocessor architecture is a 5-tuple

$$i = (m, mop, snd, rcv, cmp) \; . \tag{2.12}$$

The flag $m \in \mathbb{B}$ distinguishes memory from non-memory / compute instructions; the memory operation identifier mop, the send function snd and the receive function rcv are applicable to memory instructions; the compute function $cmp : [P \to P]$ is only applicable to compute instructions.

An instruction-set architecture Isa is a subset of the following cross product

$$\mathbb{B} \times Mop \times Snd \times Rcv \times Cmp \; . \tag{2.13}$$

As before, we define a decode function for the processor, dec_p, which decodes the current instruction for a given processor configuration:

$$dec_p : [P \to Isa] \tag{2.14}$$

Processor computation is defined using a non-deterministic automaton with a state set $Pi = P \times Iobs$ representing the processor coupled with its memory interface. For ease of presentation, the transition relation of this automaton is built from two parts: the wiring predicate $w \subseteq Pi$, which must hold for both the current configuration and a successor configuration, and the processor transition relation $\delta \subseteq Pi \times Pi$.

For the wiring predicate we require that the request line req of the interface is raised iff the current instruction is a memory instruction. Additionally, for memory instructions, the data-in bus must hold the value to be sent, and the memory operation identifier bus must hold the operation to be requested. Let $u \in Pi$. Then $w(u)$ holds iff

the following two equations are satisfied:

$$req(iobs(u)) \Leftrightarrow m(dec_p(p(u))) \tag{2.15}$$

$$m(dec_p(p(u))) \Rightarrow \quad din(iobs(u)) = snd(dec_p(p(u)))(p(u)) \wedge mop(iobs(u)) = mop(dec_p(p(u))) \tag{2.16}$$

The processor transition relation δ reflects the update of the processor configuration. Non-memory instructions update the processor configuration after one computation step. Memory instructions update the processor configuration when the memory acknowledges the memory operation. Let $u, v \in Pi$. Then, $\delta(u,v)$ holds iff the following equation is satisfied:

$$p(v) = \begin{cases} cmp(dec_p(p(u)))(p(u)) & \text{if } \neg m(dec_p(p(u))) \\ rcv(dec_p(p(u)))(p(u), dout(iobs(u))) & \text{if } m(dec_p(p(u))) \wedge ack(iobs(u)) \\ p(u) & \text{otherwise} \end{cases} \tag{2.17}$$

We have the following lemma:

◀ Lemma 2.4

A sequence $(pi_0, pi_1, \ldots)$ with $pi_i \in Pi$ satisfying the wiring predicate and the processor transition relation also satisfies the handshake condition for input stableness (Equation 2.2):

$$(\forall i : w(pi_i) \wedge \delta(pi_i, pi_{i+1})) \Rightarrow \forall j, e = iobs(pi_j), e' = iobs(pi_{j+1}) : \\ req(e) \wedge \neg ack(e) \Rightarrow req(e') \wedge (din(e) = din(e')) \wedge (mop(e) = mop(e')) \tag{2.18}$$

2.3.2 Memory Operation Architecture

In this section we describe, how the memory processes its inputs and computes an updated memory configuration and a data output. We merely define sequential semantics here; parallel behavior will be discussed later.

Let M denote a set of (shared) memory configurations. A *memory operation* is a function op that operates on a data input and a shared memory configuration and produces an updated shared memory configuration and a data output. The function space of all memory operations Op is denoted by

$$Op = [Din \times M \to M \times Dout]\,. \tag{2.19}$$

A shared memory implements a subset of the memory operation that we call *memory operation architecture*. The implemented memory operations are identified by a set of *memory operation identifiers* Mop. A *decode function* dec_m is used to map memory operation identifiers to memory operations:

$$dec_m \in [Mop \to Op] = [Mop \to [Din \times M \to M \times Dout]] \tag{2.20}$$

For a single-processor, the sequential semantics of a memory update is as follows: Suppose the memory's configuration is $m \in M$. The response to a memory operation mop with data input din is a data output $dout = dout(dec_m(mop)(din, m))$. The memory updates its configuration to $m' = m(dec_m(mop)(din, m))$.

2.3.3 Comparison to Single-Processor ISAs

To compare multiprocessor ISAs with single-processor ISAs we sequentially interpret multiprocessor ISAs: we concatenate the processor send, the memory operation, and the processor receive operation into a single function δ_{seq}. This function is then compared with the single-processor ISA.

We define $\delta_{seq} : [P \times Mem \to P \times Mem]$ for a configuration $(p,m) \in P \times Mem$ as follows. For a non-memory instruction we take the result of the compute function:

$$\neg m(dec_p(p)) \Rightarrow \delta_{seq}(p,m) = (cmp(dec_p(p))(p),m) \tag{2.21}$$

For a memory instruction, let $din = snd(dec_p(p))(p)$ and $mop = mop(dec_p(p))(p)$ denote the data input and the memory operation identifier. Then, we define the pair $(m',dout)$ of the updated memory configuration and the data output by the memory operation semantics. Additionally, we compute the updated processor configuration p' by the receive function. This already gives us the result of the function δ_{seq}. We have:

$$(m',dout) = dec_m(mop)(din,m) \tag{2.22}$$

$$p' = rcv(dec_p(p))(dout,p) \tag{2.23}$$

$$\delta_{seq}(p,m) = (p',m') \tag{2.24}$$

Definition 2.1 ▶ *Sequential Equivalence*

A multiprocessor ISA is called sequentially equivalent to a single-processor ISA modeled by δ *iff*

$$\delta = \delta_{seq} \, . \tag{2.25}$$

To construct a sequentially equivalent multiprocessor ISA from a single-processor ISA, there is a formally trivial solution if memory operation identifiers, data inputs, data outputs, and memory operation semantics can be chosen freely. By setting the data input and data output sets to whole processor configurations, $Din = Dout = P$, the memory operation identifiers to the empty set, $Mop = \emptyset$, and the memory operation semantics to the single-processor transition function, $dec_m = \delta$, the memory copies the behavior of the single-processor. In each step, the processor then only has to pass its configuration to the memory and receives it back updated on acknowledgment. Of course, this approach is unsatisfactory since experience tells us the processor should be the complex and the memory the simpler part. Taking a typical single-processor ISA, however, we cannot do much better than this. In such an ISA, instructions are fetched from memory. The instruction word is the input to the processor's decode function and determines the instruction to be performed. In a RISC-like ISA, this action may require an additional load or an additional store operation. In our multiprocessor specification we allow at most one memory operation per processor computation step. Thus, instruction fetch and most of the execution (everything that can update the memory) can only be emulated by moving it into the memory operation semantics as described in the previous paragraph.

The obvious solution for this problem is to rework the single-processor ISA to separate instruction fetch from instruction execution. This requires to add an instruction register to the processor configuration that would otherwise have been hidden or invisible. Furthermore, the transition function δ has to be decomposed into an instruction fetch part $\delta_{if} : [P \times Mem \to P]$ that only modifies the instruction register and an execute part $\delta_{ex} : [P \times Mem \to P \times Mem]$ such that, overall,

$$\delta(c) = \delta_{ex}(\delta_{if}(p,m),m) \, . \tag{2.26}$$

In a multiprocessor, this new instruction set can unfortunately behave differently than the old instruction set. This is because the memory guarantees consistency, as it will be defined below, only over memory operations but not over instructions. Consistency over the parts δ_{if} and δ_{ex}, however, does *not* imply consistency over whole instructions given by δ. However, as will become clear, such violations of consistency may only occur for interprocessor code modification, which has to be ruled out for other reasons in practical multiprocessor implementations.

2.3.4 Exemplary RISC ISA

We outline an exemplary RISC ISA in the multiprocessor style. Consider having a memory addressed by and storing natural numbers:

$$M = [\mathbb{N} \to \mathbb{N}] \tag{2.27}$$

There are only two so-called *elementary* memory operations: we can *read* a number from the memory and we can *write* a number to the memory. We identify these operations by the set $Mop = \{r, w\}$. A data input $(a, v) \in Din = \mathbb{N} \times \mathbb{N}$ consists of two numbers, an address a and a value v; a data output $v \in Dout = \mathbb{N}$ is just a single value v. The memory operation semantics is defined by

$$dec_m(r)((a,v),m) = (m, m(a)) \text{ and} \tag{2.28}$$
$$dec_m(w)((a,v),m) = (m \text{ with } [m(a) := v], 0)\ . \tag{2.29}$$

Processor configurations contain a program counter pc, an instruction word iw and a register file configuration $r \in R$. For control, we need an additional boolean flag f that indicates, whether we are in the fetch or in the execute phase of an instruction (cf. Section 2.3.3). So, we have $P = \mathbb{N} \times \mathbb{N} \times R \times \mathbb{B}$. We distinguish four instructions: fetch, load, store, and compute. Of these, all but the last are memory instructions. Fetch and load issue a memory read, stores issue a memory write.

The fetch address is taken from the program counter. The data input is stored in the instruction word and the fetch flag is reset. For load, there is an address computation function $addr : [P \to \mathbb{N}]$ computing an address from the processor configuration. The data returned by memory is processed with another function to yield a new processor configuration with an active fetch flag. Store works similarly; it needs an address computation function and a data input generation function that computes the data to be written.

Note, that by changing $\mathbb{N}$ to $\mathbb{Z}/k\mathbb{Z}$, the factor ring modulo $k > 0$, we obtain an ISA that actually can be implemented.

2.4 Consistency

We already specified the behavior of the processor and the sequential semantics of memory operations by the processor instruction-set architecture and the memory operation architecture. In this section we define the behavior of a multiprocessor based on these definitions. The model according to which the memory responds to memory operations and what effect they have on its internal configuration is called the *memory consistency model*.

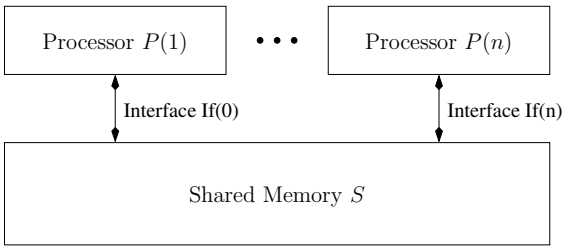

Figure 2.2 A Generic Multiprocessor

Formally, a multiprocessor configuration c consists of n processor configurations, n interface observations, and a single memory configuration:

$$C = P^n \times Iobs^n \times M \tag{2.30}$$

Figure 2.2 depicts this setup.

A multiprocessor computation is simply a trace of multiprocessor configurations. Consistency mainly concerns interface observations, i.e. a multiprocessor interface trace trc of the type $trc : [\mathbb{N} \times \{1,\ldots,n\} \to Iobs]$. The domain of such a trace consists of all pairs $(t,i) \in \mathbb{N} \times \{1,\ldots,n\}$ where $t \in \mathbb{N}$ indicates the time and $i \in \{1,\ldots,n\}$ at which the associated interface observation $trc(t,i)$ was made. Only a subset of all possible traces trc is admitted by the consistency model. For each admitted trace, the consistency model uniquely specifies (by the sequential memory operation semantics) the data outputs and memory updates of each memory interface operation.

In the sequential consistency model we require that an ordering / a sequence of the interface operations exists such that memory operations are consistent according to that order.

2.4.1 Sequential Consistency

Let $trc \in Trc$ be a valid trace with respect to the handshake conditions. Indices to interface observations with active acknowledgment are called *interface events*. Hence, interface events pick out the end times of the requests of a trace. The set of all interface events for a trace trc is denoted by $E(trc)$ and defined as follows:

$$E(trc) := \{(t,i) \in \mathbb{N} \times \{1,\ldots,n\} \mid ack(trc(t,i))\} \tag{2.31}$$

Mappings from natural numbers to interface events are called *event sequences*. The function space of all event sequences is denoted by

$$Seq = [\mathbb{N} \to E(trc)]\ . \tag{2.32}$$

Sequences are used to define the consistency of memory operations: a *sequentially consistent sequence* defines in which order the memory executes the memory operations requested by all processors. Let $seq \in Seq$ be an event sequence and abbreviate $seq(s) = (t,i)$ and $seq(s_j) = (t_j,i_j)$ for natural numbers $j \in \mathbb{N}$. Unless noted otherwise, the formulae below implicitly universally quantify over these elements. To be sequentially consistent, seq has to satisfy three properties:

- It must be bijective with respect to the events of its trace:

$$(t,i) \in E(trc) \Rightarrow \exists! s : seq(s) = (t,i) \tag{2.33}$$

- It must be *globally-ordered*, that means for a given event in the sequence, no event that starts later in the trace may occur in the sequence before it. To define this condition, we make use of the function $strt(t,i)$ defined in Lemma 2.2 that computes the starting time of an ongoing request. Consider two sequence numbers $s_1 < s_2$. Then the start of the event $seq(s_1)$ must precede or equal the time of the event $seq(s_2)$. So:

$$s_1 < s_2 \Rightarrow strt(seq(s_1)) \leq t_2 \tag{2.34}$$

  As a sequence is an ordering of (timed) interface events, the global order condition formalizes the notion, that no event "from the future" is placed in the sequence before an event "from the present". The following formula is equivalent (by contraposition):

$$strt(seq(s_1)) > t_2 \Rightarrow s_1 \geq s_2 \tag{2.35}$$

  For $s_1 = s_2$ the implication assumption does not hold, and hence we may equivalently write

$$strt(seq(s_1)) > t_2 \Rightarrow s_1 > s_2 \ . \tag{2.36}$$

  It can be easily seen that by global order the sequence orders the events of each processor in ascending order. So, as a specialization of global order we have the following property: have two sequence numbers s_1 and s_2 associated with events $e_1 = (t_1, i_1) = seq(s_1)$ and $e_2 = (t_2, i_2) = seq(s_2)$ of the same processor $i = i_1 = i_2$. If $s_1 < s_2$ then $t_1 < t_2$ must hold.

$$s_1 < s_2 \wedge (i_1 = i_2) \Rightarrow t_1 < t_2 \tag{2.37}$$

- Finally, the sequence must be consistent, so it must conform to the semantics of memory operations. We postulate that there must be a sequence $(m_0, m_1, \ldots)$ of memory configurations such that for $e = trc(seq(s))$ the configuration m_{s+1} of the memory is the result of the application of the memory operation $mop(e)$ with inputs $din(e)$ on configuration m_s:

$$(m_{s+1}, dout(e)) = dec_m(mop(e))(m_s, din(e)) \tag{2.38}$$

  We remark that the sequence $(m_0, m_1, \ldots)$ is determined by the initial configuration $m_0 \in M$ and, hence, existential quantification over a sequence is somewhat artificial.

A sequentially consistent memory guarantees the existence of such a sequence for any trace:

◀ Definition 2.2
Sequential Consistency

A memory is called sequentially consistent if for all valid traces $trc \in Trc$ an event sequence seq and a configuration sequence $(m_0, m_1, \ldots)$ exist that are sequentially consistent.

Figure 2.3 shows how a memory operation sequence is associated with interface events.

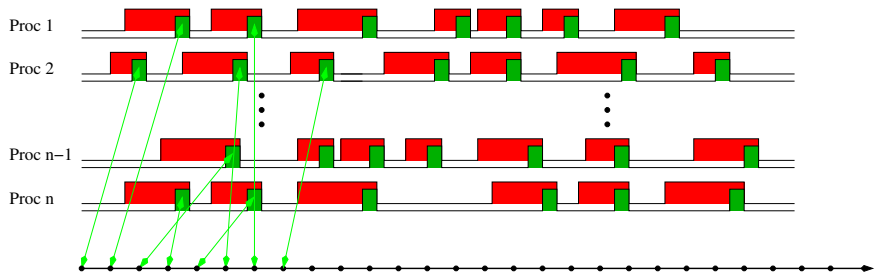

Figure 2.3 Start of a Memory Operation Sequence *seq*. Only request and acknowledgment lines are shown for each processor and slightly overlap. All requests that have been sequenced so far satisfy global order: to precede another request in the sequence, the request must have been started before the other was acknowledged.

2.4.2 Variants

We have presented a version of sequential consistency that is defined using multiprocessor traces. We think it is a natural definition and its restrictions can be readily explained in terms of the underlying interface. Also, it is directly applicable in the verification of a sequentially consistent memory where traces of the described form are bound to occur.

There are of course alternative ways to define sequentially consistent multiprocessor computations and there is a strong relation to the definition of concurrency. We consider briefly some variants involving processor configurations $p_{i,j} \in P$ (for each processor index $i \in \{1,\ldots,n\}$ and numbers $j \in \mathbb{N}$) and memory configurations m_k (for numbers $k \in \mathbb{N}$). The meaning attached to the indices j and k differs:

- For *timed sequential consistency*, the version that we presented, configuration $p_{i,j}$ denotes the configurations of processor i at time j, and the memory configurations m_k represent the contents of the memory before execution of the k-th memory operation. Characteristically, each processor (an automaton with the memory's acknowledgment and data output as input) executes wait states until memory requests are acknowledged. Multiprocessor traces trc and sequences of interface events *seq* are used to link processor and memory configurations.

- Abstracting from the traces, the processors can be modeled without wait states for memory operations. In this case, $p_{i,j}$ indicates the configuration of processor i before the execution of the j-th instruction. The interpretation of m_k does not change. For each $k \in \mathbb{N}$, the memory operation sequence only needs to indicate the processor that is next to perform an operation on the memory.

  Furthermore, the memory operation sequence must be *fair*, guaranteeing that each processor eventually may perform its next request. Before, this assumption was part of the handshake conditions of the interface trace.

- If we are not interested in the internal computations of each processor we can abstract even further. Then, $p_{i,j}$ is the configuration of processor i just before the execution of its j-th memory instruction. Thereby, a transition from $p_{i,j}$ to $p_{i,j+1}$ represents the execution of a memory instruction and all compute instructions

until the next memory instruction. Formally, this only makes sense if the number of subsequent compute instructions is bounded. Such transitions are called *big steps*.

Again, the sequence must be fair.

- Finally, for *globally-scheduled computation* we have a schedule $sched : [\mathbb{N} \to \{1,\ldots,n\}$ that indicates the order in which memory *and* compute instructions are executed. Configurations $p_{i,j}$ and m_j indicate the configurations of the processor i and the memory after execution of the j-th (global) step. Let δ be the single-processor transition function. Define

$$\delta_{mp}(l,(p_1,\ldots,p_n),m) := ((p'_1,\ldots,p'_n),m') \tag{2.39}$$

where p_l and m are updated according to δ and all other components stay unchanged:

$$(p'_l,m') = \delta(p_l,m) \quad \text{and} \quad \forall i \neq l : p'_i = p_i \tag{2.40}$$

Then, the computation according to schedule *sched* is defined inductively by

$$(p_{1,j+1},\ldots,p_{n,j+1},m_{j+1}) = \delta_{mp}(sched(j),p_{1,j},\ldots,p_{n,j},m_j)\ . \tag{2.41}$$

We call the steps j for processors $i \neq sched(j)$ (so $p_{i,j+1} = p_{i,j}$)) *idle steps*.

The schedule *sched* must be fair.

Notably, all of these variants are equivalent with respect to their operations on shared memory; the result of a computation placed in shared memory is invariant of the scheduling of compute (local) operations. Starting out with a computation in any model (with respect to a schedule or an interface trace with an event sequence), this may be proven by constructing the parameters of any other model (with the same initial configuration and, again, a schedule or an interface trace with an event sequence) and comparing the results and updates of the memory operations.

2.5 Related Work

The basics of instruction set design may be found in common computer architecture text books such as [HP96, MP00]. Likewise, protocols and interfaces, more complicated than the one presented, are described there. These are based on similar handshake conditions. Formalization of handshake conditions, for example in the computation tree logic (CTL), are already present in early model checking literature [BCDM86]. Multiprocessor instruction set architectures seem to be a novel concept. Together with their memory operation architecture, they will allow for a relatively modular definition of processors running in system or in user mode.

Memory consistency models are a wide and active topic of research. The original definition of sequential consistency is due to Leslie Lamport [Lam79]. Our version of sequential consistency—with timing made explicit through interface traces—resembles atomic consistency / linearizability [Mis86, HW90]. Predicates on event sequences are also used to describe more complex memory models, e.g. that of the Intel Itanium [YGLS03].

Sequential consistency is the strictest memory consistency model, most easily understood by the programmer but slowest to implement in hardware. Therefore, since

the mid-1980s, a wide range of relaxations were examined allowing for powerful optimizations in real-word shared memory systems (e.g. processor consistency [Goo89], weak consistency [DSB86], data-race-free-0 consistency [AH90], and release consistency [GLL+90]). In all these models, there is no total order of memory operations but several, fine-grained ordering constraints for the memory operations. The resulting models are non-intuitive for the programmer and programs which are correct on one memory need not be so on another [ABJ+93]. By making use of special operations of stronger consistency (called 'acquires', 'releases', or 'fences'), still, these models behave as a sequentially consistent memory [GMG91, AG95]. The fewer special operations inserted, the better the potential performance of the parallel program. This optimization problem is considered in the widely cited paper of Shasha and Snir [SS88]. While Lamport proposes to refine the correctness proofs for an algorithm for a specific memory model in performance-critical areas [Lam97], a carefully crafted compiler infrastructure can help in mapping programs for a high-level language's memory model to a target architecture's memory model [MLP04]. Only recent research work claims to have developed a common formalism for all memory models [SN04].

Chapter

3

The Relocated Memory Machine

Contents

In this chapter, we develop the formal definition of the relocated memory machine (RMM), a multitasking multiprocessor with memory relocation and protection. This machine can serve as a basis for various aspects of software specification—including that of operating system memory management and task management. On the other hand, the RMM is also the natural specification machine for a real system implementing virtual memory with hardware and software support.

We now informally explain the different notions of multitasking, sharing, relocation, and protection.

Multitasking permits to execute many programs (more than the number of processors) simultaneously on a multiprocessor by making them take turns in execution. For a real hardware even with only one processor, multitasking allows to increase the processor utilization. While in a single-tasking environment, high-latency, synchronous operations, such as blocking I/O, force the processor to wait for the completion of the operation, in a multitasking environment another task can continue execution instead [KHPS61]

Furthermore, *multitasking with sharing* is a parallel programming model. Here, many tasks operate on the shared regions of a so-called *logical memory* and thus establish communication. *Relocation* is a means to formally introduce sharing for tasks. With relocation, elementary memory operations (the reading and writing of single cells) change their semantics. A task cannot directly address the logical memory; instead, it specifies a *virtual address* that is transformed to a *logical address* and then

used to access the memory. This translation is performed via a dynamically configurable *address translation function*. Two virtual addresses (of two tasks) are shared, if they are relocated to the same logical address. Intra-task sharing of addresses, although possible, is in practice rarely used.

To keep non-sharing tasks from interfering with each other, either malevolently or erroneously, the memory is *protected*: a task may only perform an operation on a cell if it has the corresponding right for it. Rights are maintained for all virtual addresses of all tasks using a dynamically configurable *rights function*. Protection allows fine-grained access control over shared memory regions, e.g. to create execute-only *shared libraries*. Under memory protection, the scheduling of tasks that do not share addresses is arbitrary: it does not influence the computation result of each individual task. This property is called *tamper-free execution*.

Together, translation and rights function of some task are known as its *address space*. Naturally, for security reasons a task may only have limited, indirect access to it. For this purpose, the operating system (OS) provides a set of memory management operations, e.g. memory allocation or shared memory allocation.

For cost-effectiveness, relocated memory machines are implemented with what we call *virtual memory machines* (VMMs) that use a combination of RAM and hard disk storage to simulate the larger memory of the RMM. Such implementations date back to the 1960s; Denning describes and evaluates several variants in [Den70]. We will investigate the correctness of this construction in detail in Chapter 4.

In this chapter, we proceed as follows. In Section 3.1 we define the set of shared memory configurations. We will see that each configuration consists of several components including logical memory, translation function, and rights function. In Section 3.2 we define regular memory operations that are used to access the logical memory. Section 3.3 concludes with the presentation of task management and memory management operations in the framework of the RMM definition. These are examples of *system calls* as provided by real operating systems.

We do not further characterize processors other than by their memory operations. For the moment, this is sufficient; Chapter 2 justifies this course of action. Eventually, of course, the processor ISA must be specified as it forms an integral part of the task model by defining the tasks' internal configurations and computational steps.

3.1 Structure of a Storage Configuration

Let Tid denote the set of task identifiers, Va and La denote the set of virtual and logical addresses, $Data$ denote the set of data values for a memory cell, P_r denote the set of RMM processor configurations, and finally Mop_r denote the set of memory operations. An RMM memory configuration $m_r \in M_r$ is a 6-tuple $m_r = (mem, atid, ctid, sar, tr, r)$. The components are

- the logical memory configuration $mem : [La \to Data]$ that maps logical addresses to data values,
- the active task identifier function $atid : [Tid \to \mathbb{B}]$ that maps task identifiers to booleans and thus identifies active, runnable tasks,
- the current task identifier function $ctid : [\{1, \ldots, n\} \to Tid]$ that maps processor indices to task identifiers and indicates the tasks that are running on the processors,

- the save area function $sar : [Tid \to P_r]$ that maps task identifiers to processor configurations,
- the translation function $tr : [Tid \times Va \to La]$ that maps task identifiers and virtual addresses to logical addresses, and
- the rights function $r : [Tid \times Va \to 2^{Mop_r}]$ that maps task identifiers and virtual addresses to sets of memory operations (with 2^M denoting the power set of M).

We comment on and elaborate this definition from the perspective of an individual task.

Tasks are identified by numbers $tid \in Tid$. An active task tid (with $atid(tid) = 1$) can be thought of as an execution unit of the RMM. We call it *sleeping* iff there is no processor index $i \in \{1,\ldots,n\}$ with $ctid(i) = tid$. Otherwise, it is called *running* and we demand that i is uniquely defined. So, no task may run on several processors at once and $ctid$ must be injective, for all $i \neq j$ we demand $ctid(i) \neq ctid(j)$. Furthermore, no processor may run an inactive task tid with $atid(tid) = 0$, we require $atid(ctid(i)) = 1$ for all i.

The internal configuration of task tid consists of the configuration of an RMM processor that it uses for computation. For a running task, this configuration is to be found in its processor $j = ctid^{-1}(tid)$. For a sleeping task, it is stored in the save area at location tid.

The external configuration of tid consists of its address space and of the contents of the logical memory that are accessible to it.

The address space of tid is given by the translations $tr(tid,va)$ and the rights $r(tid,va)$ of all virtual addresses $va \in Va$. The exact meaning of these will be formalized when regular memory operations are defined in the next section. Informally, task tid specifies virtual addresses va that translate to logical addresses $la = tr(tid,va)$. These are then used to access mem. However, performing memory operations $mop \in Mop_r$ on addresses va is only possible if mop is contained in the associated rights set $r(tid,va)$. Otherwise, memory operations return constant results and never update mem.

Hence, there are two possible reasons that task tid cannot "access" a certain memory cell $mem(la)$ of the logical memory: (i) la may have no inverse image va with $tr(tid,va) = la$ through the translation function or (ii) all addresses va with $tr(tid,va) = la$ have no associated rights, $r(tid,va) = \emptyset$.

This redundancy may be a nuisance in formally reasoning on the RMM. It is convenient to set the translation of addresses va with $r(tid,va) = \emptyset$ to a special value $0 = tr(tid,va)$ such that no task has any rights on logical address 0. This way, the associated memory contents $mem(0)$ remain constant for the computation of the machine.

For use in Chapter 4 we abbreviate the types of the components of an RMM configuration by

$$Mem = [La \to Data]\,, \tag{3.1}$$
$$Atid = [Tid \to \{0,1\}]\,, \tag{3.2}$$
$$Ctid = [\{1,\ldots,n\} \to Tid]\,, \tag{3.3}$$
$$Sar = [Tid \to P_r]\,, \tag{3.4}$$
$$R = [Tid \times Va \to 2^{Mop_r}]\text{, and} \tag{3.5}$$
$$Tr = [Tid \times Va \to La]\,. \tag{3.6}$$

3.2 Regular Memory Operations

Let Din_r and $Dout_r$ denote the data inputs and data outputs of the RMM memory interface and, again, let Mop_r denote its memory operation identifiers. In this section, we characterize parts of the memory operation semantics represented by the memory operation decode function

$$dec_r : [Mop_r \rightarrow [Din_r \times M_r \rightarrow M_r \times Dout_r]] \; . \tag{3.7}$$

We define memory operations, which we call *regular*, that read out / modify single cells of the logical memory, and are subject to address translation and to the rights function. Accordingly, we say that regular memory operations are *elementary*, *relocated*, and *safe*.

Let $mop_r \in Mop_r$ denote a memory operation identifier and $op_r = dec_r(mop_r)$ denote its associated memory operation. For op_r to be *regular*, we require the existence of three uniquely defined functions

$$\begin{aligned} decin &: [Din_r \rightarrow Va \times Edin_r \times \{1,\ldots,n\}] \; , \\ emop_r &: [Edin_r \times Data \rightarrow Data \times Edout_r] \text{ , and} \\ encout &: [\mathbb{B} \times Edout_r \rightarrow Dout_r] \end{aligned}$$

that capture the behavior of op_r in the way defined below where $Edin_r$ is the set of elementary data inputs and $Edout_r$ is the set of elementary data outputs. In most computer architectures, the sets $Edin_r$ and $Edout_r$ are equal to the set of data values of a cell $Data$. However, for certain memory operation, such as an conditional update,[1] $Edin_r$ needs to be larger than $Data$. Though we know of no example where $Edout_r \neq Data$, we have introduced $Edout_r$ for reasons of symmetry.

Function *decin* designates the virtual address to be accessed, the (elementary) data input to the operation on the cell, and the processor identifier that requests the operation; function $emop_r$ describes the update on the cell and the resulting (elementary) data output if the required right is present; function *encout* defines data passed back to the processor depending on whether the operation was successful or forbidden.

In detail, consider an RMM memory configuration $(mem, atid, ctid, sar, tr, r) \in M_r$ and a data input $din_r \in Din_r$. With the function *decin* we decompose din_r into the triple $(va, edin_r, i_r) = decin(din_r)$. Given these variables, we let $ctid_r = ctid(i_r)$ denote the current task identifier of processor i_r, the logical address $la_r = tr(ctid_r, va_r)$, the translated address va for task $ctid_r$, and the boolean flag $excp_r = 1 \Leftrightarrow mop_r \notin r(ctid_r, va_r)$ be true iff mop_r is not in the set of rights for address va_r and task $ctid_r$.

We distinguish two cases according to $excp_r$. If no exception is indicated, $excp_r = 0$, the memory cell la_r is updated by the elementary memory operation function *emop*, which was decoded by *decin*. Let $d_r = mem(la_r)$ denote the old data at the memory cell. The new cell's data $d'_r \in Data$ and the elementary data output $edout_r \in Edout_r$ are computed by $(d'_r, edout_r) = emop_r(din_r, d_r)$. The function mem' is equal to the function *mem* updated at address la_r with value d'_r. Otherwise, for $excp_r = 1$, the logical memory $mem' = mem$ is not updated and the returned data $edout_r = 0$ is set to some fixed value denoted 0.

[1] For example, the compare and swap instruction `CS` of the S/390 architecture [IBM00] requires $Edin_r = Data \times Data$; it compares the first input operand with the current value of a memory cell and replaces it with the second input operand on equality.

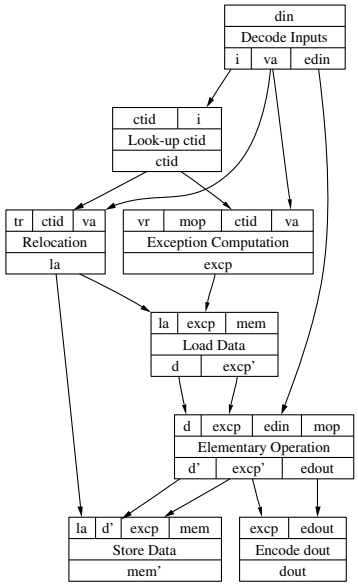

Figure 3.1 RMM Regular Memory Operation Semantics.

In any case, we use the function *encout* that has to be injective to encode the exception signal and the elementary data output into the interface's data output type *Dout* by setting $dout_r = encout(excp_r, edout_r)$.

Now, for any RMM memory configuration $(mem, atid, ctid, sar, tr, r)$, data input din_r, and auxiliary variables as given above, we define the operation op_r by

$$op_r(din_r, (mem, atid, ctid, sar, tr, r)) = (dout_r, (mem', atid, ctid, sar, tr, r)) \ . \quad (3.8)$$

In particular, all components of the RMM memory configuration other than the logical memory configuration *mem* stay unchanged.

Figure 3.1 illustrates the definition of regular memory operations. Each box corresponds to the application of a function with inputs and outputs shown at the top and bottom of the box. The equality of an input operand of one function to the output operand of another function is indicated by an arrow (or, in this diagram, by equal labels of the input and output field). Input operands that are not the target of such an arrow are components of the memory configuration or of the memory operation.

3.3 Complex Memory Operations

In this section we discuss some operating system concepts, namely task and memory management, in the framework of the RMM definition. What we describe here as "memory operation" is usually called an *(operating) system call*. In real systems these are performed using software interrupts, so-called *traps*. With a concrete instruction set architecture for the processor P_r of the RMM it is possible to fully define the system call interface of an operating system. Together with a description of the format of executables (or task images, cf. below) this constitutes the *application binary interface* (ABI) of the OS. However, this task is beyond the scope of this thesis.

3.3.1 Task Management

Task management deals with ways to create and terminate tasks. Let us first level out terminology. We use *task* as the common term for an executing program. A *process* is an address space and a set of *threads* sharing that address space. In our model, an *address space* (r,tr) consists of the (virtual) rights function r and the translation function tr associated with a task. We may identify a process p with a subset of the task identifiers $p \subseteq Tid$ where each member $t \in p$ is called thread. As a context condition for the threads of a process, we then require that they have the same address space, or, formally, for all t_1, $t_2 \in p$ and all virtual addresses $va \in Va$ we must have equality of rights and of translation,

$$tr(t_1,va) = tr(t_2,va) \wedge r(t_1,va) = r(t_2,va)\ . \tag{3.9}$$

This completes the static view on threads and processes. More elaborate models may further divide tasks into groups or arrange them into hierarchies. Let us now consider the dynamic view on tasks, threads, and processes, namely their creation and termination.

Any task creation operation establishes some active task identifier tid' that was inactive before the execution of that operation. We distinguish

- *task loads* that take an encoded initial task configuration, the *task image*, to initialize tid' and
- *task forks* that use the current configuration of an active task as a template for initializing tid'.

Task termination deactivates a task identifier tid. We distinguish regular exit operations and aborts / abnormal exits triggered implicitly, e.g. through rights violations.

3.3.2 Memory Management

Granularity

Rights and translation functions of the RMM are often *granular*. This means, that the rights and translations of certain addresses cannot be chosen independently of each other. In particular, most models designate a parameter called page size $pag \in \mathbb{N}$ according to which the address sets Va, $La \subseteq \mathbb{N}$ are divided into blocks of length pag that are aligned at addresses being multiples of pag. The addresses of each block have the same rights and their translations retain their relative distance:

$$\begin{aligned}\forall i : \mathbb{N}, x_1, x_2 < pag :\ & r(tid, i\cdot pag + x_1) = r(tid, i\cdot pag + x_2) \wedge \\ & tr(tid, i\cdot pag + x_2) - tr(tid, i\cdot pag + x_1) = x_2 - x_1\end{aligned} \tag{3.10}$$

Low granularities result in memory-efficient implementations for the operating system (as translations and rights functions are less complex) while on the other hand the memory consumption of the user tasks may increase (since the size of a memory request must be rounded up to the next multiple of pag).

Rights

Modern machines distinguish two or three elementary memory operations for each virtual memory cell. These are *read*, *write*, and an optional *fetch* operation. The fetch

memory operation is used to read instruction words meant for execution; the read and the write memory operations are used for data access. Without a specific fetch memory operation, instruction fetches are indistinguishable from reads.

Sometimes, not all combinations of rights are allowed. A two-rights machine typically does not allow for an exclusive write-right to an address. So, possible rights set for each memory cell are $\emptyset$, $\{r\}$, and $\{r,w\}$.

In a three-rights machine, it often makes sense to have fetch and write right mutually exclusive. In this case, a task cannot modify its own code or execute "data". This guards against unintentional or malicious code modification, e.g. as in buffer-overflow attacks against insecure programs.

Operations

Memory management operations are concerned with acquirement and release of access rights. A single operation may apply to several tasks and several addresses simultaneously. The former depends on the task management policy (task, threads, or processes) while the latter may be dictated by granularity or additional parameters. To simplify the discussion, we do not consider how tasks and addresses are selected for memory management operations but restrict ourselves to operations concerning a single address of a single task.

Apart from the initialization of a task, we consider three phases in the "lifetime" of an address:

- Without rights to an address, it can be *allocated*, creating a non-empty set of rights for it, a translation, and, optionally, initial content. Sometimes, the choice of the address is left to the allocation operation (cf. regular `malloc`).

  For the translation, we distinguish whether it is used by another active task or not. In the first case we speak of *private allocation*, otherwise we speak of *shared allocation*.

  For shared allocation, the allocated memory region keeps its contents. For private allocation, the memory region will be initialized; this prevents a task from having non-deterministic and potentially sensitive input. Typical initializations are zero-filling the allocated memory, mapping the contents of a file, or copying (not sharing) another task's region.

- When rights to an address do already exist, a task may want to upgrade or downgrade rights for this address without releasing all rights completely. This is a special operation. Consider, for example, a three-rights machine with writes and fetches being mutually exclusive for security reasons. Then, a compiler that compiles code to memory and then starts to execute it (e.g. a just-in-time compiler [DS84]), needs to change the rights for the generated code from writable to executable.

  For addresses with rights, translation changes are possible, as well. However, we believe that in the life cycle of an allocated address such an operation does not make any sense.

- Taking away all rights from an address is called the *freeing* or *deallocation* of the address. After the deallocation of private allocations it may be reused for other allocations. This is not automatically the case for shared allocations.

Let us note, finally, that sharing of addresses may be expressed in a structured way by choosing the set of logical addresses *La* appropriately. For example, *La* typically

includes the crossproduct $Tid \times Va$ of the task identifiers and the virtual addresses, allowing all active tasks a full private, unshared address space. Shared addresses may be arranged into segments, sets of the form $\{seg\} \times Va \subseteq La$ where $seg \notin Tid$ is called segment identifier. Segments may be associated with a length len_{seg}; in this case, addresses (seg, o) with $o \geq len_{seg}$ have no logical rights.

3.3.3 Task Switching and Scheduling

Task switching and scheduling is concerned with updates of the function $ctid$ such that it still satisfies its constraints. Scheduling at least, we cannot fully model in the RMM since we need more data structures and additionally non-deterministic input like a timer interrupt.

However, we may demand here that task switching must be consistent with the processor configurations. For a task that goes to sleep in a certain computation step, its configuration must be taken from the processor and stored in the save area. To formulate this generally, in addition to the RMM memory configuration $m_r \in M_r$ and its successor $m_r' \in M_r$ we must also consider n processor configurations $(p_1, \ldots, p_n)$ and their successors $(p_1', \ldots, p_n')$. We require task switches to leave the active tasks, the translations, the rights, the logical memory unchanged. Furthermore, for any active task tid let us abbreviate

$$X_{tid} = \begin{cases} p_{ctid^{-1}(tid)} & \text{if } tid \text{ running in } m_r\,, \\ sar(tid) & \text{otherwise,} \end{cases} \tag{3.11}$$

and, symmetrically, X_{tid}' for m_r' and p_i'. Then, we require $X_{tid} = X_{tid}'$, which means that processor configurations do not change during a task switch.

By these constraints, an RMM with one processor can simulate an arbitrary RMM.

3.4 Related Work

Our (incomplete) RMM model is targeted to be the most concrete model of an operating system specification and hence the most abstract model of its implementation. Classically, operating systems ('multi-programming systems') are described and implemented in terms of segments, rather than addresses or pages (cf., for example, the early paper of Dennis [Den65] and the description of the Multics system [BCD72]). As Silberschatz et al. note [SGG00], users prefer to see memory as "a collection of variable sized segments with no necessary ordering". However, we deliberately chose the address- / page-orientated logical memory as a more concrete representation of memory than segments for a number of reasons: (i) since users must live with page granularity in real operating system implementation it is not good to hide this information in a (slightly) higher-level abstraction such as segments; (ii) there are system-calls in real operating systems that are more easily expressible in terms of pages than segments (for example, page locking or protection changes); (iii) the same holds for the tricky procedures of shared library handling in relation to task loading or dynamic library loading in contemporary operating systems [Lev00]; (iv) not all computer architecture have built-in segmentation support and, in fact, hardware designers state that support for superpages (power-of-two multiples of pages) is easier to implement than support for segments [TH94].

Obviously, for most operating systems, there are informal descriptions of its interface. For example, the widely-adopted portable operating system interface (POSIX), an IEEE standard [IEE01a, IEE01b, IEE01c], describes a Unix system operation interface. However, formal specification are rare. The most ambitious work on mathematical kernel specification (a kernel is the integral part of an OS) that we know of is that of Bevier and Smith [BS93a, BS93b, BS94a, BS94b]. They have specified large parts of the kernel configuration and kernel calls of the Mach microkernel [ABB+86] in the logic of the Boyer-Moore theorem prover [BM88] and checked its consistency and type correctness [BS94a]. Their research was targeted at providing a formal specification accompanying and making more precise the kernel's informal specification [Loe91]. As such, the specification does not provide an explicit logical memory model[2] or a user computation model and, hence, no application binary interface.

Furthermore, the specification is close to the kernel's implementation. Since difference instances of the Mach microkernel may concurrently handle different clients' kernel calls, Bevier and Smith have only formulated liveness of kernel calls, stating the updates on the kernel configuration based on the observations of the kernel configuration that are made during the execution of that kernel call handler. Safety properties would additionally limit the effects of kernel call handlers on parts of the kernel configuration they are not meant to update. Ideally, liveness and safety properties would lead to a non-interference result for kernel call handlers establishing that the updates of a kernel call handler are still valid in its final configuration.

Apparently, however, Mach kernel call handlers interfere with each other, so such a result cannot be established [BS94a]. This restricts the use of the specification to validating liveness properties of the kernel implementation and makes it unsuitable for reasoning on the execution and correctness of user programs. Though the authors state that the specification of kernel calls with pre- and post-condition is "inadequate", we still think that kernels / operating systems should offer system calls with a sequentially-consistent semantics even for a multi-threaded implementation handling those calls. For such implementations a (certainly non-trivial) proof would have to be conducted that despite the parallelism the illusion of sequential consistency is maintained.[3]

No more progress on Bevier's and Smith's efforts has been reported after 1994.

Finally, let us briefly discuss two possible RMM extensions. For device modeling it is necessary to extend the RMM with an input / output mechanism and interrupt delivery. A synchronous interprocess communication (IPC) mechanism that transmits data only if and when the receiver is willing to take it (rendezvous protocol) can be easily defined in the RMM framework. Such a mechanism can also be used to deliver interrupt by associating an 'external process' with any device that generates IPC messages for interrupts. This idea goes back to Brinch Hansen's original work on kernels (nuclei) [BH70] but is also employed in modern microkernels [Lie95]. For truly asynchronous interruptions, the RMM processor model must be extended in a way similar to interrupt mechanisms of actual processors [MP00]; if the operating system wants to retain control over interrupt delivery, such an extension could, for example, be modeled after the UNIX signal handling mechanism [IEE01b].

Note that the introduction of devices may easily lead to intricate consistency issues with regards to sharing information that they hold (think of a hard disk and file systems). Single address space system (e.g. [KHPS61, BCD72]) solely identify infor-

[2] Possibly because the underlying logic is not expressive enough.

[3] Even in systems based on weak memory models, complex memory operations should be performed with sequential consistency memory operation semantics. Mixed consistency is considered in [SN04].

mations via logical addresses and do not suffer from such issues.

Another, more fundamental extension to the RMM concept are recursive virtual machines [BH75, Gol73, LW73]. With these, an RMM may create new RMMs and these, in turn, create more RMMs, which allows for the recursive construction of hierarchies of RMMs. The execution environments / initial configurations of newly-created RMMs are always identical, hence, by definition, the RMMs do not share memory across the hierarchy and may be assumed to have distinct logical memories. The root RMM is called the 'virtual machine monitor' (VMM). Main frames such as the IBM zServers [IBM00, IBM05] implement recursive virtual machines up to a limited hierarchy depth to allow (multiple) instances of (different) operating systems to run on the same hardware.

Chapter

4

The Virtual Memory Machine

Contents

A direct implementation of the RMM is expensive due to large memory needs. Therefore, in this chapter we present the virtual memory machine (VMM), which is an affordable yet efficient implementation of the RMM.

In the virtual memory machine two memories are used to simulate the RMM: the *main memory* is implemented with a small but fast RAM; the *swap memory* is implemented with a large but slow hard disk. Though they have the same capacity as a single large memory, their cost is usually much lower. By the principle of data locality [Den67] the performance is adequate: often-used data can be placed in the main memory for fast access, while rarely-used data is placed in the swap memory.

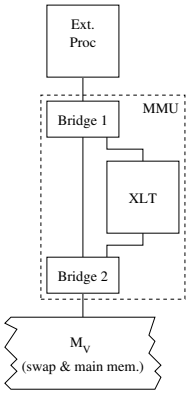

Figure 4.1 Overview of the Virtual Memory Machine

This cost-performance trade-off depends heavily on both hardware and software implementation details. For example, since we have a multitasking environment, the data locality is influenced by the locality of the memory operations of each individual task as well as the task switch / task scheduling policy. If there are many tasks working on disjoint portions of the memory, more task switches imply less locality. We will not further discuss these issues.

From the programmer's point of view, data access works exactly the same as in the RMM, i.e. the VMM is a *transparent* implementation of the RMM. The memory references are made with virtual addresses. With the help of a *translation hardware* the memory references are either redirected to some main memory location or an exception is generated. Typically, this exception indicates, that the desired data is not present in the main memory but at the moment being stored in the swap memory. The processor is extended to handle this exception: it executes an exception handler called *supervisor* to deal with the situation (e.g. to load the data from the swap memory to some main memory location) before the access is retried. The execution of this routine remains unnoticed by the programmer.

We proceed to formally define the interface and the configuration of the VMM formally. Thereafter, we examine the individual modules of the VMM: we describe the extended processor and two auxiliary modules called bridges, then we continue with the shared memory and its operations, the afore-mentioned translator and the semantics of the supervisor.

Along the way, we show properties of the system. These properties help us to establish the simulation theorem: every observed multiprocessor trace of virtual memory operations of the VMM yields a sequence of events and a sequence of RMM memory configurations that are consistent to the RMM semantics.

4.1 The Interfaces and Configuration

Figure 4.1 shows an overview of the environment of a single processor of the virtual memory machine. As we mentioned each processor is surrounded by three additional modules. The most important is the translator (XLT), a hardware address translation mechanism. It takes as input a memory operation identifier and a virtual address (collectively often subsumed simply as memory operation) and outputs a pair of a boolean

flag and a main memory address. The boolean flag determines whether the virtual address can be accessed. If it can be accessed, we call the memory operation *attached*; the access will then take place in the main memory at the specified address. Otherwise, the memory operation is *detached* and a translation exception is signaled. The extended processor will save its current configuration and call the supervisor, an exception service routine to handle translation exceptions. The supervisor typically moves data to and from the swap memory and attaches the memory operation, such that a repeated translation request does *not* cause an exception and provides access to the virtual address at some main memory address. When that is done, it returns from the exception, restoring the processor configuration and makes the processor repeat the offending virtual memory operation. Note, that handling these exception is the only responsibility of the supervisor: it does not manage tasks and it does not manage memory. It must "only" make the simulation of the RMM work.

The remaining two modules, Bridge 1 and Bridge 2, control this interplay: bridge 1 decomposes a request from the processor into a request to the translator followed by a request to bridge 2, if no exception was generated; bridge 2 is used to multiplex requests of the translator and bridge 1 for the memory.

To start our formal definition of the VMM, we give names to the modules and busses shown in Figure 4.1. There are five different modules in the VMM. These are the (extended) processor, the translator, bridge 1, bridge 2 and the memory. We denote the (internal) sets of configurations for these modules by P, Xlt, $B1$, $B2$ and M_v.

Except for the memory all modules are instantiated n times in the machine. A VMM configuration $vmm = (p, xlt, b1, b2, m) \in Vmm = P^n \times Xlt^n \times B1^n \times B2^n \times M_v$ is a five-tuple with $p(i)$ denoting the processor configuration, $xlt(i)$ denoting the translator configuration, $b1(i)$ denoting the bridge 1 configuration, and $b2(i)$ denoting the bridge 2 configuration of a certain processor $i \in \{1, \ldots, n\}$.

The busses between the modules are labeled with names If_{x1x2}, where $x1$ is the requesting module and $x2$ is the acknowledging module. For legibility, we write the concatenation $x1x2$ in lower-case letters. The associated sets of interfaces observations for an interface If_{x1x2} are denoted by $Iobs_{x1x2}$, the interface traces are denoted by trc_{x1x2}:

$$Iobs_{x1x2} = \mathbb{B} \times Mop_{x1x2} \times Din_{x1x2} \times \mathbb{B} \times Dout_{x1x2} \tag{4.1}$$

$$trc_{x1x2} : [\mathbb{N} \times \{1, \ldots, n\} \to Iobs_{x1x2}] \tag{4.2}$$

As usual, each interface observation $(req, mop, din, ack, dout) \in Iobs_{x1x2}$ has five components: the boolean request flag $req \in \mathbb{B}$, the memory operation $mop \in Mop_{x1x2}$, the data input $din \in Din_{x1x2}$, the boolean acknowledgment flag $ack \in \mathbb{B}$ and the data output $dout \in Dout_{x1x2}$.

The VMM has five interfaces If_{x1x2} for

$$x1x2 \in \{pb1, b1xlt, xltb2, b1b2, b2m\}\,. \tag{4.3}$$

Hence, the set of VMM configurations *including interface observations*, Vmm^+, is given by:

$$\begin{aligned} Vmm^+ = \; & M_v \times {Iobs_{b2m}}^n \times B2^n \times \\ & {Iobs_{xltb2}}^n \times Xlt^n \times \\ & {Iobs_{b1b2}}^n \times B1^n \times \\ & {Iobs_{pb1}}^n \times P^n \end{aligned} \tag{4.4}$$

We will examine the different sets of interface observations more closely. The interface at the processor in the VMM extends the RMM memory operation interface by the supervisor memory operations. To distinguish both kinds of memory operations, each memory operation has a memory operation type flag $ty \in \{p, sv\}$. A regular processor memory operation is indicated by the type flag $ty = p$ and a supervisor memory operation is indicated by the type flag $ty = sv$. All in all, the set of interface observations for the processor-bridge-1 interface is defined as follows:

$$Iobs_{pb1} = \mathbb{B} \times Mop_{pb1} \times Din_{pb1} \times \mathbb{B} \times Dout_{pb1} \tag{4.5}$$

$$\begin{aligned} &= && \mathbb{B} \\ &\quad \times && ((\{p\} \times Mop_r) \cup (\{sv\} \times Mop_{sv})) \\ &\quad \times && (Din_r \cup Din_{sv}) \\ &\quad \times && \mathbb{B} \\ &\quad \times && ((\mathbb{B} \times Dout_r) \cup Dout_{sv}) \end{aligned} \tag{4.6}$$

When a request for a regular processor memory operation arrives at bridge 1, the bridge requests the translator to compute the translation of the virtual memory address and the specific memory operation. On acknowledgment, the translator returns a pair of an exception flag and a main memory address. Translator requests are modeled by the bridge-1-translator interface observations, $Iobs_{b1xlt}$. Since there is only one operation performed by the translator for bridge 1, there is no set of memory operation identifiers. The data input Din_{b1xlt} consists of a processor identifier and a virtual address. The data output $Dout_{b1xlt}$ is a pair. Thus, an element of the bridge-1 translator interface observations is a 7-tuple $(req, pid, va, mop, ack, excp, ma) \in Iobs_{b1xlt}$ and the interface observations are structured as follows:

$$Iobs_{b1xlt} = \mathbb{B} \times (\{1, \ldots, n\} \times Va \times Mop_r) \times \mathbb{B} \times (\mathbb{B} \times Ma) \tag{4.7}$$

If the memory operation is attached (i.e. the exception flag returned by the translator is zero), the bridge makes a request to bridge-2 (which will forward the request to the shared memory). With such a request the memory operation identifier supplied by the processor is passed on to bridge 2. The data input consists of the main memory address returned by the translator and the elementary data input for the operation. The data output is the same as for the RMM operation. The set of interface observation $Iobs_{b1b2}$ is defined as follows, supporting both translated elementary processor and supervisor memory operations:

$$Iobs_{b1b2} = \mathbb{B} \times Mop_{b1b2} \times Din_{b1b2} \times \mathbb{B} \times Dout_{b1b2} \tag{4.8}$$

$$\begin{aligned} &= && \mathbb{B} \\ &\quad \times && ((\{p\} \times Mop_r) \cup (\{sv\} \times Mop_{sv})) \\ &\quad \times && ((Ma \times Edin) \cup Din_{sv}) \\ &\quad \times && \mathbb{B} \\ &\quad \times && (Dout_r \cup Dout_{sv}) \end{aligned} \tag{4.9}$$

When the translation has led to an exception, bridge 1 just signals that back to the processor. The processor collects this exception flag and calls the exception service routine.

The translator accesses the memory with its own memory operations, identified by the set Mop_{xltb2}. This set will be defined later in detail. Note, however, that the translator may only read the memory and is not allowed to update the memory.

As has been indicated before, the shared memory must support processor, translator, and supervisor memory operations. Basically, we define this interface to unify

the individual interfaces. To distinguish processor and supervisor memory operations the memory operation contains a type flag $ty \in \{xlt, p, sv\}$. We have added brackets in the following definition, to show how the different fields are grouped into memory operation identifiers, data input, and data output:

$$Iobs_{b2m} = \mathbb{B} \times Mop_{b2m} \times Din_{b2m} \times \mathbb{B} \times Dout_{b2m} \tag{4.10}$$

$$\begin{aligned} = \quad & \mathbb{B} \\ \times \quad & ((\{xlt\} \times Mop_{xltb2}) \cup Mop_{b1b2}) \\ \times \quad & (Din_{xltb2} \cup Din_{b1b2}) \\ \times \quad & \mathbb{B} \\ \times \quad & (Dout_{xltb2} \cup Dout_{b1b2}) \end{aligned} \tag{4.11}$$

4.2 Extended Processor

This section describes the processor used in the VMM. It is an extension of the regular RMM processor with a simple and abstract exception handling mechanism. With this mechanism, we can handle the translation exceptions generated by the translator module.[1]

An extended processor configuration consists of two RMM processor configurations, named p and $savep$, and a boolean flag named $mode$. The second RMM processor configuration is used to save the RMM processor configuration on receiving an exception. The mode flag indicates whether the processor is in regular / RMM mode (processing RMM instructions), or in supervisor mode (handling an exception). Let P_r denote the RMM processor configurations. Thus, an element $p \in P$ from the set of VMM extended processor configurations is a triple

$$p = (p, savep, mode) \in P = P_r \times P_r \times \mathbb{B}\,. \tag{4.12}$$

For the extended processor we have two instruction set architectures, Isa_r and Isa_{sv}.

As in Chapter 3, the RMM ISA is used uninterpreted. It is defined using RMM processor configurations, memory operation identifiers, data inputs, and data outputs. So, Isa_r has the following type:

$$Snd_r : [P_r \to Din_r] \tag{4.13}$$

$$Rcv_r : [P_r \times Dout_r \to P_r] \tag{4.14}$$

$$Cmp_r : [P_r \to P_r] \tag{4.15}$$

$$Isa_r \subseteq \mathbb{B} \times Mop_r \times Snd_r \times Rcv_r \times Cmp_r \tag{4.16}$$

The supervisor ISA is new but we leave it also almost uninterpreted. It operates on the *extended* processor configurations and has its own set of memory operation identi-

[1] Note that we currently do not talk about any other type of exception. In particular, external exceptions / device interrupts that are to be handled by RMM processors need to be first incorporated in the RMM machine model. This may either happen through a truly asynchronous interrupt mechanism or through interrupt delivery through (synchronous) inter-process communication [BH70, LBB+91]. The VMM layer would have to be extended accordingly. However, already at the VMM layer but possibly below it, exceptions have to map to real hardware interrupts. These are asynchronous by definition.

fiers, data inputs, and data outputs. The supervisor ISA has the following signature:

$$Snd_{sv} : [P \to Din_r] \tag{4.17}$$

$$Rcv_{sv} : [P \times Dout_r \to P] \tag{4.18}$$

$$Cmp_{sv} : [P \to P] \tag{4.19}$$

$$Isa_{sv} \subseteq \mathbb{B} \times Mop_{sv} \times Snd_{sv} \times Rcv_{sv} \times Cmp_{sv} \tag{4.20}$$

The processor has two modes of operation, as indicated by the mode flag. For $mode = 0$ it operates using the supervisor ISA, for $mode = 1$ it operates using the RMM ISA. The mode flag is cleared, when an RMM memory operation has been acknowledged with a set exception flag. In this case, the configuration p is saved in the *savep* field. The mode flag is set again by a special supervisor instruction, which is called `rfe` (return from exception). For simplicity, we require, that the `rfe` instruction is a compute instruction and that no other instruction is capable of setting the *mode* flag. Also, we assume that on issuing `rfe` the component *savep* holds the configuration that was originally saved on entering the calling of the supervisor; this way, a return from exception will result in a repetition of the instruction.

We define the transition relation for the extended processor formally. Consider u, $v \in P \times Iobs_p$. We define the conditions that qualify u and v as a valid transition (for v being a possible successor state of u).

To access the interface, we define the request, the memory operation, and the data inputs. The request flag must be set, when we are in supervisor mode and have a supervisor memory operation or if we are in RMM mode and have an RMM memory operation. The memory operation identifier and data input must be set accordingly. This condition must hold for both the states u and v, so we must have for $w \in \{u, v\}$:

$$req(w) := \begin{cases} m(dec_r(w)) & \text{if } mode(w) \\ m(dec_{sv}(w)) & \text{otherwise} \end{cases} \tag{4.21}$$

$$mop(w) := \begin{cases} (p, mop(dec_r(w))) & \text{if } mode(w) \\ (sv, mop(dec_{sv}(w))) & \text{otherwise} \end{cases} \tag{4.22}$$

$$din(w) := \begin{cases} (p, snd(dec_r(w))(w)) & \text{if } mode(w) \\ (sv, snd(dec_{sv}(w))(w)) & \text{otherwise} \end{cases} \tag{4.23}$$

For the processor configuration update, we compute two possible next configurations. The one called $newp_r(u)$ shall be used, if the processor is in RMM mode and makes an exception-free computation step. The other called $newp(u)$ shall be used, if the processor is in supervisor mode and makes a computation step. Accordingly, the following definitions use the RMM and the supervisor ISA:

$$newp_r(u) = \begin{cases} cmp(dec_r(p_r(u)))(p_r(p(u))) & \text{if } \neg m(dec_p(p_r(u))) \\ rcv(dec_r(p_r(u)))(p_r(p(u)), dout(u)) & \text{if } m(dec_p(p_r(u))) \end{cases} \tag{4.24}$$

$$newp(u) = \begin{cases} cmp(dec_{sv}(u))(p(u)) & \text{if } \neg m(dec_{sv}(u)) \\ rcv(dec_{sv}(u))(p(u), dout(u)) & \text{if } m(dec_{sv}(u)) \end{cases} \tag{4.25}$$

An update of the processor configuration takes place iff the processor executes a compute instruction or if an acknowledgment was received while executing a memory operation. If there is an acknowledgment in RMM mode, we distinguish two cases: If the

exception flag from memory is zero, we speak of an *update* situation. Otherwise, the situation is called a *save* situation. In a save situation we must perform a switch to the supervisor mode.

Since we can detect whether the processor is executing a compute instruction by looking at the request flag of the interface (see Equation 4.21 above), the conditions under which update takes place are fully described by the following equation:

$$update(u) := \neg req(u) \vee ack(u) \wedge (\neg mode(u) \vee \neg excp(u)) \tag{4.26}$$

A save situation only occurs in RMM mode with a set acknowledgment and a set exception flag:

$$save(u) := mode(u) \wedge ack(u) \wedge excp(u) \tag{4.27}$$

Now we can describe the overall equation to compute the successor processor configuration. In case of updates, we update either the whole processor configuration using $newp(u)$ or we only update the RMM processor part using $newp_r(u)$. In a save situation, the mode has to be reset, the processor configuration has to be saved and the new processor configuration is computed using a supervisor initialization function $init_{sv} : [P_r \rightarrow P_r]$. Otherwise, we have a memory operation but no acknowledgment yet. In this case we just keep the old configuration. The following equation captures the desired behaviour:

$$p(v) = \begin{cases} newp(u) & \text{if } \neg mode(u) \wedge update(u) \\ (newp_r(u), savep(p(u)), 1) & \text{if } mode(u) \wedge update(u) \\ (init_{sv}(p_r(p(u)))), p_r(p(u)), 0) & \text{if } save(u) \\ p(u) & \text{otherwise} \end{cases} \tag{4.28}$$

4.3 The Bridges

4.3.1 Bridge 1

Bridge 1 receives memory operation requests from the RMM processor as well as from the supervisor. Supervisor memory operation requests and their results are passed on transparently by the bridge.

For processor memory operation requests, bridge 1 will do some extra work. It will determine whether the requested memory operation is *attached* or not. An attached memory operation can be directly performed on some main memory cell whose address is given by the result of a translator request. Execution of a detached memory operation is not permitted: either, the (RMM) rights forbid to access the address with the desired operation, or access to the cell is forbidden by implementation reasons of the VMM, e.g. the cell is located in swap memory. In any of these two cases, an exception is passed back to the processor which results in the calling of the supervisor. We further describe the steps that bridge 1 takes after receiving a processor request: Bridge 1 starts by requesting the translator to translate the virtual address of the processor memory operation. On acknowledgment, there are two alternatives how to go on:

- The memory operation may be attached. Then, bridge 1 will execute the operation (p, mop_r) on the returned main memory address. Eventually, the operation

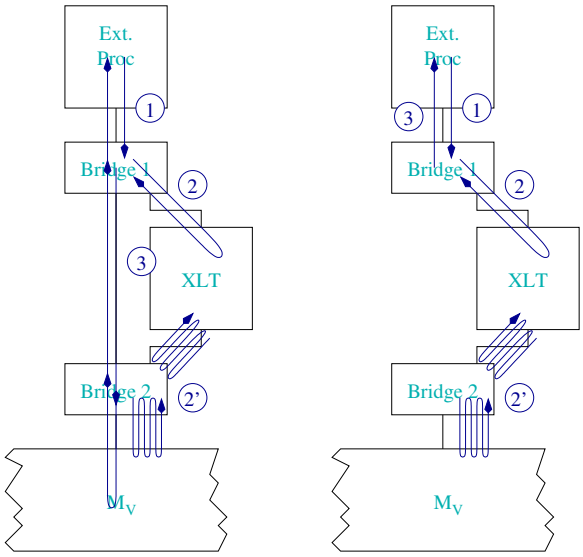

Figure 4.2 Bridge 1 Sequencing a Processor Request

gets an acknowledgment and data output, which are then forwarded to the processor.

This case is displayed in the left diagram of Figure 4.2: step ① is the start of the request from the processor to bridge 1. The translator request is posed and received back in step ②. To fulfill the translator request, the translator usually makes several references to the memory with the help of bridge 2. These requests are shown as a couple of arrows around the marker ②'. In this case, the translator signaled back an attached operation, so bridge 1 poses a request to the shared memory via bridge 2 and passes back the result to the processor. This step is labeled with ③.

- In the other case, the memory operation is not attached. This can mean two things: either the memory operation is not attached because it is illegal (i.e. it is not allowed by the RMM / logical rights), or it is not attached because of some supervisor decision (i.e. the data is swapped out or read-only shared). Bridge 1 then passes back an arbitrary data output with a set exception flag. As we have already seen in Section 4.2, this exception flag is picked up by the processor that calls the exception service routine.

  This case is displayed in the right diagram of Figure 4.2: step ① is the start of the request from the processor to bridge 1. The translator request is posed and received back in step ②. To fulfill the translator request, the translator usually makes several references to the memory with the help of bridge 2. These requests are shown as a couple of arrows around the marker ②'. In this case, the translator signaled back a non-attached operation, so bridge 1 acknowledges with a set exception flag to the processor in step ③.

This procedure is realized in a state automaton with three states. Bridge 1 is in state *Idle*, if it waits for a processor request. The *Idle* state is also used to transparently pass on supervisor requests.

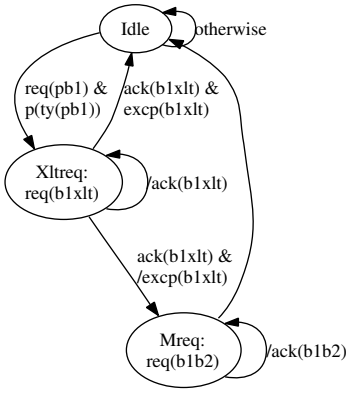

Figure 4.3 Bridge 1 State Automaton. Nodes correspond to states, active signals in each state are listed after the node name. Edges correspond to state transitions and are labeled with the transition condition. The operators '&' and '/' are used for logical conjunction and negation. Here, functional application denotes the selection of components of a bus. The predicate $p(ty(pb1)$ is defined as $ty(pb1) = p$.

After having received a processor request it moves to the state $Xltreq$ in which it requests the translator. Finally, bridge 1 is in stage $Mreq$ while requesting the memory over bridge 2.

In addition to the state the bridge is in, the bridge also needs to store the main memory address returned by the last translator request. This is done with a variable $ma \in Ma$.

Thus, a bridge 1 configuration consists of an automaton state

$$state \in \{Idle, Xltreq, Mreq\} \tag{4.29}$$

and the variable ma. The set of all configurations $Conf_{b1}$ for bridge 1 is equal to the following cross product:

$$Conf_{b1} = \{Idle, Xltreq, Mreq\} \times Ma \tag{4.30}$$

The state is updated as shown in the state automaton drawn in Figure 4.3. The diagram also defines the activation of the request signals to the translator and bridge 2, and the activation of the acknowledgment signal to the processor (at the edge from $Mreq$ to $Idle$). In state $Idle$, we must additionally activate $req(b1b2)$ in case of a supervisor memory operation request; so apart from the signals given in the control automaton we must also satisfy the equation $state = Idle \wedge req(pb1) \wedge sv(ty(pb1)) \Rightarrow req(b1b2)$ (where $x(ty(pb1)$ is defined as the predicate $ty(pb1) = x$ for $x \in \{p, xlt, sv\}$).

We define the update of the internal variables and the setting of memory operations to the translator, bridge 2 and the supervisor. Consider interface observations $x1x2 \in Iobs_{x1x2}$ for the interfaces $x1x2 \in \{pb1, b1xlt, b1b2\}$ which were all made in the same cycle t for processor i:

$$\begin{aligned} pb1 &= trc_{pb1}(t,i) \in Iobs_{pb1} \\ b1xlt &= trc_{b1xlt}(t,i) \in Iobs_{b1xlt} \\ b1b2 &= trc_{b1b2}(t,i) \in Iobs_{b1b2} \end{aligned}$$

Let $c \in Conf_{b1}$ denote the bridge 1 configuration and let $c' \in Conf_{b1}$ denote the successor configuration. The variable *ma* is updated on acknowledgment of the translator. Then, it is taken directly from the translator's data output. So, we have:

$$ma(c') = \begin{cases} ma(dout(b1xlt)) & \text{if } ack(b1xlt) \\ ma(c) & \text{otherwise} \end{cases} \tag{4.31}$$

From the data input of the processor, we define the elementary data input, the processor identifier, and the virtual address of the memory operation:

$$(edin, va, i) = args(din(pb1)) \tag{4.32}$$

These parameters stay constant for the time of a processor request, since the processor-bridge-1 interface is stable by the first handshake condition. We pass the processor index, the virtual address, and the memory operation identifier to the translator:

$$din(b1xlt) = (i, va, mop(pb1)) \tag{4.33}$$

To bridge 2, we copy the memory operation identifier. We transparently pass the data input from the processor whenever we are in the $Idle$ state. Otherwise, we pass down the *ma* variable and the elementary data input *edin*:

$$mop(b1b2) = mop(pb1) \tag{4.34}$$

$$din(b1b2) = \begin{cases} din(pb1) & \text{if } state = Idle \\ (ma(c), edin) & \text{otherwise} \end{cases} \tag{4.35}$$

The data output and the acknowledgment from bridge 2 are also used as data output and acknowledgment for bridge 1. Observe, that the acknowledgment signal from bridge 2 *always* acknowledges a bridge 1 operation.

$$dout(pb1) = dout(b1b2) \tag{4.36}$$

$$ack(pb1) = ack(b1b2) \vee ack(b1xlt) \wedge excp(b1xlt) \tag{4.37}$$

4.3.2 Bridge 2

Bridge 2 operates as a simple multiplexor for bridge 1 and the translator. The logic is easy if we can exploit that at most one of the modules passes a request to the shared memory.

Therefore, we require that the translator generates no stray translator-bridge-2 requests. If there is no request at the bridge-1-translator interface then the request signal of the translator-bridge-2 interface must not be active:

$$\forall i \in \{1, \ldots, n\}, t \in \mathbb{N} : \neg req(trc_{b1xlt}(t, i)) \Rightarrow \neg req(trc_{xltb2}(t, i)) \tag{4.38}$$

Let $x1x2 \in \{b1b2, xltb2, b2m\}$. Consider interface observations of the interfaces $x1x2 \in Iobs_{x1x2}$ which were simultaneously made in the same cycle $t \in \mathbb{N}$ and for the same processor $i \in \{1, \ldots, n\}$:

$$b1b2 \in Iobs_{b1b2}$$
$$xltb2 \in Iobs_{xltb2}$$
$$b2m \in Iobs_{b2m}$$

Bridge 2 will forward a request to the shared memory, if any of the input request signals is active. The memory operation $mop(b2m)$ and the data input $din(b2m)$ forwarded to the shared memory depend on the requesting module. We forward a translation request, if the translator request signal is active; otherwise, we forward a bridge 1 request, if the bridge 1 request signal is active. The priority we chose here is arbitrary, since by bridge 1 control and Equation 4.38, activation of the two request signals is mutually exclusive.

In addition to forwarding a memory operation identifier, the type flag of the memory operation identifier to the shared memory $ty(mop(b2m))$ must be set appropriately. For translator requests the type flag is forced to the value xlt, for bridge 1 requests, the type flag is just copied.

So, we have the following formulae:

$$req(b2m) = req(b1b2) \vee req(xltb2)$$

$$mop(b2m) = \begin{cases} (xlt, mop(xltb2)) & \text{if } req(xltb2) \\ mop(b1b2) & \text{otherwise} \end{cases}$$

$$din(b2m) = \begin{cases} din(xltb2) & \text{if } req(xltb2) \\ din(b1b2) & \text{otherwise} \end{cases}$$

The data output of the shared memory may basically be forwarded all the time to the translator and the bridge 2 module provided we do not over-acknowledge. Formally, however, since we have $Dout_{b1b2} \cap Dout_{xltb2} = \emptyset$, we need embedding functions $embed_{x1x2} : [Dout_{b2m} \to Dout_{x1x2}]$ for $x1x2 \in \{b1b2, xltb2\}$ that are the identity on $Dout_{x1x2}$ and arbitrary elsewhere.

The equations for the acknowledgment and the data output are the following:

$$ack(b1b2) := req(b1b2) \wedge ack(b2m) \tag{4.39}$$

$$ack(xltb2) := req(xltb2) \wedge ack(b2m) \tag{4.40}$$

$$dout(b1b2) := embed_{b1b2}(dout(b2m)) \tag{4.41}$$

$$dout(xltb2) := embed_{xltb2}(dout(b2m)) \tag{4.42}$$

4.4 The Memory

In this section we examine the memory organization in the VMM. VMM's memory consists of two memories: the main memory and the swap memory. In practical implementations, the swap memory is usually much larger than the main memory albeit slower (or even accessed with I/O). The main memory is fast but smaller. These two memories shall simulate the memory of the RMM. To achieve this goal, we need several data structures to manage the swap and the main memory. These data structures reside in the main memory of the VMM. With their help we construct a projection (an abstraction) from VMM memory configurations to RMM memory configurations that is crucial for the simulation theorem. In the next sub section this setup is formally defined.

After having established the structure of the VMM memory configuration, we define the associated memory operations in the second sub section. Of these, the most interesting are the VMM processor memory operations. They are intended to simulate the RMM processor memory operations. Technically, there are two variants of memory

operations: the *local variant* only performs elementary memory operations, assuming that address translation and exception handling have already been done by the translator module; the *global variant* subsumes translator and elementary memory operation semantics. The running system guarantees that both variants lead to the same memory operation semantics.

In the third sub section, we examine the conditions under which processor memory operations in the VMM produce equivalent results as in the RMM. These four so-called *access conditions* have direct parallels in memory management routines of operating systems. The notion of equivalent update for one computation step is formalized and proven in the fourth sub section; the result is called the *step lemma*.

4.4.1 Structure of a Memory Configuration

A memory configuration M_v of the VMM consists of two components: the main memory *mm* and the swap memory *sm*. Both have a similar structure, the main memory maps main memory addresses $ma \in Ma$ to data while the swap memory maps swap memory addresses $sa \in Sa$ to data. The structure of a VMM memory configuration is the cross product of the two:

$$mm : [Ma \to Data] \tag{4.43}$$
$$sm : [Sa \to Data] \tag{4.44}$$
$$M_v = Mm \times Sm \tag{4.45}$$

Imposed on this simple structure are additional data structures. A data structure with type X is formalized by a decode function dec_x operating on the VMM memory configuration or, for simplicity, just the VMM main memory configuration. So, the decode function dec_x is of the form $dec_x : [M_v \to X]$ or $dec_x : [Mm \to X]$. In the following we describe the five groups of data structures encoded in the VMM memory configuration.

For task management, there are three data structures directly taken from the RMM: the active task function *atid* specifies which task identifiers are active, the current task identifier function *ctid* indicates, which processor runs which task, and the save area *sar* stores processor configurations of active but sleeping tasks. Accordingly, we have three decode functions named dec_{atid}, dec_{ctid}, and dec_{sar}:

$$dec_{atid} : [Mm \to Atid] \tag{4.46}$$
$$dec_{ctid} : [Mm \to Ctid] \tag{4.47}$$
$$dec_{sar} : [Mm \to Sar] \tag{4.48}$$

For memory management, the translation function *tr* and the virtual rights function *r* are represented as data structures in the VMM. This defines two more decode functions:

$$dec_{tr} : [Mm \to Tr] \tag{4.49}$$
$$dec_{r} : [Mm \to R] \tag{4.50}$$

In addition to those RMM-inherited data structures we need two new data structures concerned with the management of the main memory and the swap memory.

The implementation translation function *itr*, an abstract version of an address translation mechanism (cf. Chapter 5), controls the processor's memory access: it indicates

whether a memory operation is attached and which (main) memory address to use for a memory operation. The implementation translation function will be used to specify the translator. Inputs are a processor identifier $i \in \{1,\ldots,n\}$, a virtual address $va \in Va$ and a memory operation $mop \in Mop_r$. The output is a pair $(excp, ma)$:

$$(excp, ma) := itr(i, va, mop) \in \mathbb{B} \times Ma$$

If the address is attached for the memory operation, the exception flag $excp$ is false. In this case, the main memory address to be accessed is indicated by component ma. The access semantics is defined in detail below; the function space of the implementation translation function is defined as follows:

$$Itr = [\{1,\ldots,n\} \times Va \times Mop_r \to \mathbb{B} \times Ma] \tag{4.51}$$

The implementation translation decode function is denoted by $dec_{itr} : [Mm \to Itr]$.

The logical address location function $laloc$ indicates for each logical address $la \in La$ whether (and where exactly) it is placed in main memory or swap memory. We let the logical address location function map each address $la \in La$ to a three-tuple

$$(inm, ma, sa) := laloc(la) \in \mathbb{B} \times Ma \times Sa\,.$$

If $inm = 1$ then ma indicates the associated main memory address; if $inm = 0$ then sa indicates the associated swap memory address. So, the function space of the logical address location function is defined as

$$Laloc = [La \to \mathbb{B} \times Ma \times Sa]\,. \tag{4.52}$$

We further restrict the domain to let each triple satisfy

$$inm \Rightarrow sa = 0 \text{ and } \neg inm \Rightarrow ma = 0\,. \tag{4.53}$$

Note that we do not require the logical address location function to be injective. Although this would seem like a "natural" property we will later obtain simulation results without using injectivity. Indeed, in modern implementations performance tricks actually violate injectivity. The logical address location decode function is denoted by $dec_{laloc} : [Mm \to Laloc]$.

Finally, there is the memory decode function, dec_{mem}, projecting the VMM memory configuration onto the memory component of an RMM memory configuration. As may be readily understood this function is crucial for the connection between VMM and RMM.

The memory decode function is defined with the use of the logical address location function. For each address $la \in La$ we look up the corresponding location. Let f denote the value of the logical address location function (for a VMM memory configuration $m_v \in M_v$) at location la,

$$f := dec_{laloc}(mm(m_v))(la) \in \mathbb{B} \times Ma \times Sa\,. \tag{4.54}$$

Then, the value of the memory decode function dec_{mem} at the location la is defined as follows:

$$dec_{mem}(m_v)(la) = \begin{cases} mm(m_v)(ma(f)) & \text{if } inm(f) \\ sm(m_v)(sa(f)) & \text{otherwise} \end{cases} \tag{4.55}$$

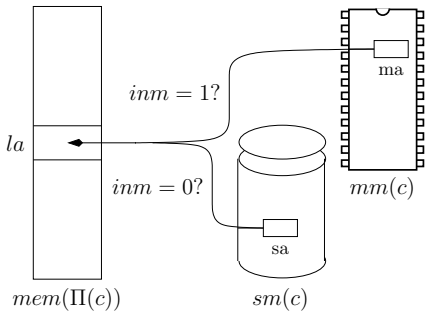

Figure 4.4 Memory projection with $(inm, ma, sa) := dec_{laloc}(mm(m_v))(la)$

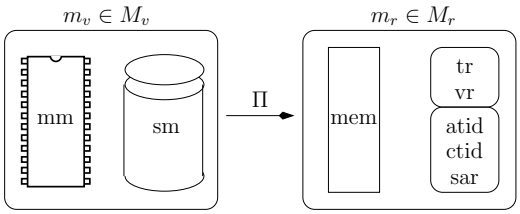

Figure 4.5 Projection Function

Figure 4.4 illustrates the memory decode function.

The functions above are used to construct a projection Π from VMM memory configuration to RMM memory configurations:

$$\Pi : [M_v \to M_r] \tag{4.56}$$

Recall that a RMM memory configuration is a 6-tuple of the memory, the task management functions and the memory management functions:

$$M_r = Mem \times Atid \times Ctid \times Sar \times Tr \times R \tag{4.57}$$

We define Π applied to $m_v \in M_v$ as

$$\begin{aligned} \Pi(m_v) = \quad & (dec_{mem}(m_v), dec_{atid}(mm(m_v)), dec_{ctid}(mm(m_v)), \\ & dec_{sar}(mm(m_v)), dec_{tr}(mm(m_v)), dec_r(mm(m_v))) \ . \end{aligned} \tag{4.58}$$

Figure 4.5 shows the projection function Π.

In the context of the decoding functions, we often need to access the translation of an address $(ctid(i), va)$ and the virtual rights of an address $(ctid(i), va)$ for a given processor identifier $i \in \{1, \ldots, n\}$ and a virtual address $va \in Va$ with respect to a shared memory configuration $m_v \in M_v$. To simplify notation for this case, we define the logical translation function ltr and the logical rights function lr:

$$ltr : [M_v \times \{1, \ldots, n\} \times Va \to La]$$
$$lr : [M_v \times \{1, \ldots, n\} \times Va \to 2^{Mop_r}]$$

We set:

$$ltr(m_v, i, va) := dec_{tr}(mm(m_v))(dec_{ctid}(mm(m_v))(i), va) \tag{4.59}$$
$$lr(m_v, i, va) := dec_r(mm(m_v))(dec_{ctid}(mm(m_v))(i), va) \tag{4.60}$$

4.4.2 Memory Operations

Three kinds of operations must be supported by the shared memory: supervisor memory operation, translator memory operations, and processor memory operations. Processor memory operations come in two variants, depending on whether translation has already been done or not.

The set of memory operation identifiers is identical for both variants. It consists of pairs: the first entry of a memory operation identifier indicates whether the operation is of the supervisor, translator, or processor type, the other entry is a memory operation identifier of the appropriate set. The data outputs for both variants are identical and consist of the union of all possible data outputs. We set:

$$Mop_v = Mop_{b2m} := \{xlt\} \times Mop_{xltb2} \cup \{sv\} \times Mop_{sv} \cup \{p\} \times Mop_r \tag{4.61}$$

$$Dout_v = Dout_{b2m} := Dout_{xltb2} \cup Dout_{sv} \cup Dout_r \tag{4.62}$$

Only the data inputs for both variants differ. For the *global variant*, the data input is a translator, supervisor or RMM data input. For the *local variant*, we assume that address translation has already been performed for RMM elementary memory operations. Therefore, the input consists of an elementary data input $Edin$ and a main memory address Ma. We define the sets of data inputs Din_v and Din_{b2m} accordingly:

$$Din_v := Din_{xlt} \cup Din_{sv} \cup Din_r \tag{4.63}$$

$$Din_{b2m} := Din_{xlt} \cup Din_{sv} \cup (Edin \times Ma) \tag{4.64}$$

Both variants define memory operation semantics by decode functions dec_v and dec_m in the usual manner: the memory operation identifiers are mapped to memory operation functions which map data inputs and a memory configuration to an updated memory configuration and a data output. We have:

$$dec_v : [Mop_v \to [Din_v \times M_v \to M_v \times Dout_v]] \tag{4.65}$$

$$dec_m : [Mop_{b2m} \to [Din_{b2m} \times M_v \to M_v \times Dout_{b2m}]] \tag{4.66}$$

We will now first define the global variant for processor memory operations: Let $mop_r \in Mop_r$ denote a RMM memory operation identifier and consider the VMM memory operation identifier $mop_v = (p, mop_r) \in Mop_v$. We denote the associated global VMM memory operation by op_v:

$$op_v := dec_v(mop_v) \in [Din_v \times M_v \to M_v \times Dout_v] \tag{4.67}$$

The structure of op_v is derived from the structure of the associated RMM memory operation $op_r := dec_r(mop_r) \in [Din_r \times M_r \to M_r \times Dout_r]$. In the RMM, the operation op_r is associated with (unique) functions that decode the input (*decin*), perform an elementary memory operation on a cell (*emop*) and encode the output (*encout*). The left diagram of Figure 4.6 illustrates the interplay of these functions: the decode input box maps the data input via the function *decin* to a processor identifier, a virtual address, and an elementary data input. Relocation provides a logical destination address la, while the exception computation box checks whether the right for the operation mop_r is available in the virtual rights function r. If so, the data is loaded from the memory, the elementary operation *emop* is applied, the updated cell is stored back and the data output to the processor is encoded. For details, refer to Section 3.2.

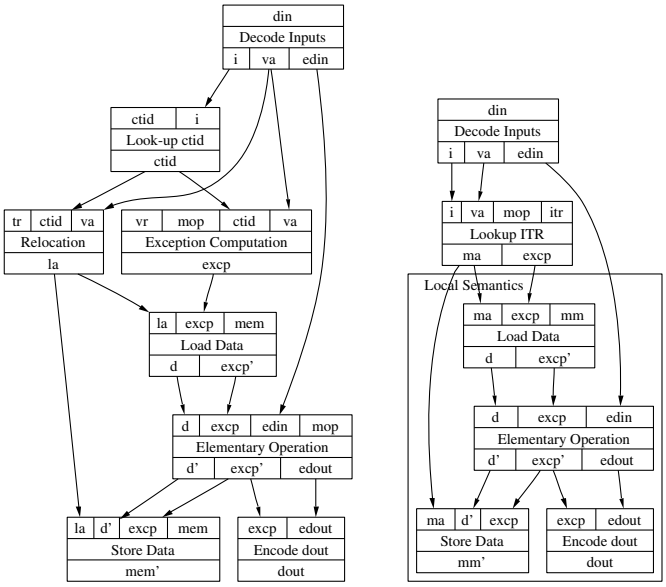

Figure 4.6 RMM and VMM Memory Semantics

The structure of op_v is similar to the structure of op_r. The decode input function, $decin$, maps the input $din_r \in Din_r$ of the operation to three variables $(va, edin_r, i_r) := decin(din_r)$. The virtual address $va \in Va$ specifies the virtual address access location. The elementary data input $edin_v \in Edin_r$ is the input to the operation performed on the memory cell. The processor identifier $i_r \in \{1,\ldots,n\}$ determines the processor identifier which initiates the request.

A lookup of the implementation translation function (to be done in the end by the translator) defines the record $i := dec_{itr}(mm(m_v))(i_r, va, mop) \in \mathbb{B} \times Ma$. The boolean flag $excp_v \in \mathbb{B}$ indicates whether the memory operation can be performed on the main memory. If $\neg excp_v$, the main memory address $ma_r \in Ma$ is used to address the main memory. So, we have:

$$\begin{aligned} excp_v &= excp(i) \\ ma_v &= ma(i) \end{aligned}$$

There are two cases according to the value of $excp_v$.

If no exception is indicated, $excp_v = 0$, the main memory cell ma_v is updated by the elementary memory operation function $emop$ decoded by $decin$. Let $d_v = mm(m_v)(ma_v)$ denote the old data of the main memory cell. The new cell's data $d'_v \in Data$ and the elementary data output $edout_v \in Edout_r$ are computed as

$$(d'_v, edout_v) = emop(edin_v, d_v)\ . \tag{4.68}$$

The new main memory configuration, mm'_v must reflect the update of the operation:

$$mm'_v = mm(m_v) \text{ with } [mm(ma_v) := d'_v] \tag{4.69}$$

However, if an exception is indicated, $excp_v = 1$, the memory must not be updated

and the $edout_v$ is set to a special fixed value denoted 0:

$$\begin{aligned} edout_v &= 0 \\ mm'_v &= mm \end{aligned}$$

In any case, the encode output function *encout* encodes the exception signal and the elementary data output injectively back to the interface's data output type $Dout_v$:

$$dout_r = encout(excp_r, dout_r) \tag{4.70}$$

All other addresses and the swap memory stay unchanged:

$$m'_v = m_v \text{ with } [mm := mm'_v] \tag{4.71}$$

The local variant of the processor memory operations only performs the elementary memory update (without translation and exception checking). With variables from above, we define the local memory operation semantics by

$$dec_m(p, mop_r)(edin_v, ma_v) = (mm'_v, edout_v)\ . \tag{4.72}$$

The right diagram of Figure 4.6 illustrates the VMM memory operation semantics. As can be seen, start and end resemble the diagram for the RMM semantics. RMM's current task identifier lookup, relocation, and exception computation are replaced by a lookup of the implementation translation function. Loading and storing the memory cell operates on the main memory instead of the RMM's memory. The right-hand side box indicates the local memory operation semantics.

The supervisor operation identified by (sv, mop_{sv}) for $mop_{sv} \in Mop_{sv}$ and the translator operation identified by $(xlt, mop_{xlt}) \in Mop_{b2m}$ for $mop_{xlt} \in Mop_{xlt}$ are not defined here.

4.4.3 Access Conditions

In the previous section we have seen that VMM memory operations are, structurally, quite different from RMM memory operations because of the VMM's memory organization. In this section we examine how, despite the structural differences, the execution of a single VMM processor memory operation leads to the same result as the execution of a single RMM memory operation. This property is called the *step lemma*.

Naturally, the step lemma can only be established for *attached* memory operations that do not cause an exception under the implementation translation function. For such memory operations, we define four access conditions which are formalizations of common memory management correctness assumptions.

Our setting is as follows: Consider a memory request to the shared memory for the memory operation $(p, mop) \in Mop_v$ with the data input din decoded as

$$(i, va, edin) = decin(p, mop)(din)\ . \tag{4.73}$$

Let $m_v \in M_v$ be a VMM memory configuration. Let

$$(excp, ma) = dec_{itr}(m_v)(i, va, mop) \tag{4.74}$$

denote the result of the implementation translation lookup.

Let la denote the logical translation of the processor identifier and the virtual address, i.e. $la := ltr(m_v, i, va)$. Now let the triple

$$(inm, ma_f, sa_f) = dec_{laloc}(m_v)(la) \tag{4.75}$$

denote the result of the logical address location lookup.

Assume that the memory operation is attached; i.e. we have $\neg excp$. The four access conditions can be viewed as predicates on the VMM memory configurations, the memory operations, the processor identifiers, and the virtual addresses. The conditions are called the system memory conditions (*sys*), the rights consistency condition (*rc*), the *laloc* consistency condition (*lc*) and the copy-on-write condition (*cow*):

$$sys : [M_v \times Mop_r \times \{1,\ldots,n\} \times Va \to \mathbb{B}] \tag{4.76}$$
$$rc : [M_v \times Mop_r \times \{1,\ldots,n\} \times Va \to \mathbb{B}] \tag{4.77}$$
$$lc : [M_v \times Mop_r \times \{1,\ldots,n\} \times Va \to \mathbb{B}] \tag{4.78}$$
$$cow : [M_v \times Mop_r \times \{1,\ldots,n\} \times Va \to \mathbb{B}] \tag{4.79}$$

The step lemma, a commutativity result for data consistency, holds under the conjunction of these conditions. Denote with $m_r := \Pi(m_v) \in M_r$ the projection of the VMM configuration m_v. We consider the memory operation $mop_v = (p, mop_r)$ and mop_r with data inputs din in both machines. Let m'_v and m'_r be the successor configurations of m_v and m_r and let $dout_v$ and $dout_r$ be the data outputs in the RMM and VMM. The claim is: (i) the successor configurations are equal again by projection, i.e. $m'_r = \Pi(m'_v)$, and (ii) both data outputs are the same, i.e. $dout_v = dout_r$.

System Memory Condition

The system memory is the part of the main memory used to encode the special data structures of the VMM. This means that they do not depend on the values of memory cells outside of the system memory. The concept of system memory helps to keep the data structures and the user memory separate.

Formally, we define a select function *select*, which hides parts of the memory by zeroing them out. A memory region is called the source of some decode function, if the decode function produces the same result for the selection of this region as for the unselected, whole memory.

The function *select* takes a subset of the main memory addresses $set \in 2^{Ma}$ as input and produces a filter function that maps main memory configurations from $[Ma \to Data]$ to modified main memory configurations:

$$select : [2^{Ma} \to [Mm \to Mm]] \tag{4.80}$$

For $mm \in Mm$ and $ma \in Ma$, the select function maps the address ma to zero, if $ma \notin set$ and leaves it unchanged otherwise. We define:

$$select(set)(mm)(ma) = \begin{cases} mm(ma) & \text{if } ma \in set \\ 0 & \text{otherwise} \end{cases} \tag{4.81}$$

Now fix a set $sysmem \subseteq Ma$ called the system memory. If the decode function dec_x for $x \in \{atid, ctid, sar, r, tr, itr, laloc\}$ produces the same result for all pairs $mm \in Mm$ and $select(sysmem)(mm) \in Mm$, then, apparently, dec_x can be computed independently from addresses outside the system memory.

We assume that our decode functions have this property:

$$\forall ma : dec_x(ma) = dec_x(select(sysmem)(ma)) \tag{4.82}$$

All implementation translations must respect the system memory, i.e. they must never point to it. This way it can be guaranteed that user tasks cannot modify the critical data structures for memory and task management. The system memory access condition therefore requires, that the main memory address obtained by the implementation translation for an attached memory operation is not an element of the system memory. With the variables given before (Equations 4.73 to 4.75) we define

$$sys(m_v, mop, i, va) \Leftrightarrow ma \notin sysmem\ . \tag{4.83}$$

Rights Consistency Condition

Since all attached memory operations are actually executed (without exception) by the VMM, we want the attachment of a memory operation to imply the right to perform this memory operation in the RMM. So, the rights consistency access condition requires, that the memory operation is present in the set of logical rights given by the lr function defined in Section 4.4.1, Equation 4.60. We define (again, with reference to Equations 4.73–4.75)

$$rc(m_v, mop, i, va) \Leftrightarrow mop \in lr(m_v, i, va)\ . \tag{4.84}$$

laloc Consistency Condition

The logical address location function models the VMM's knowledge about where in swap or main memory the RMM's logical memory cells are stored. The implementation translation function on the other hand provides the memory access semantics. Up to now, we have not related those two functions, although it is clear, that they must somehow correspond, since otherwise the implementation translation function could point to anywhere in the main memory. It turns out that the main memory address indicated by the implementation translation function must correspond to the location the logical address location function points to. Only this way we can guarantee that memory reads are the same in the VMM as for the RMM and memory updates are correctly reflected in the projection of the VMM memory configuration.

In a straightforward formalization of this criterion, we require that the logical address location function must indicate the access address as being in main memory and at the very location indicated by the implementation translation function. Hence, the $laloc$ consistency access condition is defined using the variables from Equations 4.73 to 4.75 by

$$lc(m_v, mop, i, va) \Leftrightarrow inm \wedge ma_f = ma\ . \tag{4.85}$$

Figure 4.7 illustrates the equality of the logical address location function and the implementation translation function main memory address field. *All* user task memory reads and updates take place through the implementation translation function (the upper arrow), so the logical address location (through the middle arrow) must point to this location as well. Pointing to the swap memory is therefore forbidden, the dashed arrow may not exist in this situation.

Note that this condition effectively *forbids* physical aliases of the same logical address, a property that may be desirable for example in non-uniform memory access (NUMA) machines for distributed caching. In such situations equivalent properties must be derived. We do not further pursue this topic. Regular caching is unaffected since it is below the level of abstraction presented here.

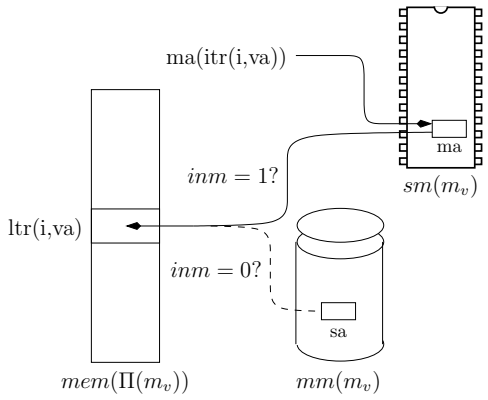

Figure 4.7 The *laloc*-Consistency Condition

Copy-On-Write Condition

By the implementation translation mechanism of the VMM, memory cells can be physically shared, even if they are *logically* different, i.e. the logical translation maps them to distinct addresses. In modern operating systems, this property is used for the benefit of faster task management operations and efficient memory usage.

For example, the semantics of a Unix `fork` operation requires that a task is created from a running task, copying its address space. As long as the two (the old and the new) address spaces are not modified, they may be physically shared. However, when one of the two tasks intends to write to its address space, physical sharing may no longer sustain and data must be copied. This strategy is called copy-on-write (COW) and was first described in [BBMT72].

In this section, we formulate the copy-on-write access condition, a correctness criterion concerning physical sharing of logically different addresses. It states that for attached memory write operations the main memory address must not be shared by a logically different address.

We formalize this description for the VMM. Consider two addresses (tid, va) and (tid', va') for the RMM. Let la and la' denote the (logical) translations of both addresses. Both addresses are called *logically shared*, if their translations map to the same logical address, $la = la'$. We denote this by $(tid, va) \sim_l (tid', va')$. Both addresses are called *physically shared*, if their logical address location results are equal, so $dec_{laloc}(m_v)(la) = dec_{laloc}(m_v)(la')$. We denote this by $(tid, va) \sim_p (tid', va')$. Our interest lies in addresses that are physically but not logically shared. This condition is termed "*pnl*-sharing", we define the predicate *pnl* using the relations $\sim_l$ and $\sim_p$ above by

$$pnl((tid, va), (tid', va')) \Leftrightarrow \\ (tid, va) \sim_p (tid', va') \wedge (tid, va) \not\sim_l (tid', va') \; . \quad (4.86)$$

The copy-on-write invariant requires that if *mop* is an attached write operation then it must not be *pnl*-shared by another address. We define

$$cow(m_v, mop, i, va) \Leftrightarrow (mop \in W \Rightarrow \neg sh_{pnl}(m_v, tid, va)) \quad (4.87)$$

where $W \subseteq Mop_r$ denotes the subset of write memory operation identifiers and the

predicate $sh_{pnl}(m_v, tid, va)$ indicates the presence of a *pnl*-shared address,

$$sh_{pnl}(m_v, tid, va) = \exists tid', va' : pnl((tid, va), (tid', va')) \; . \tag{4.88}$$

Figure 4.8 shows an example of how the Unix system `fork` is implemented using copy-on-write. In this example, however, no copy operation is saved.

4.4.4 The Step Lemma

The projection and the memory operation semantics commute under the access conditions for attached memory operations: applying RMM memory semantics to a projected memory configuration leads to the same result as applying VMM memory semantics and then projecting.

We state and prove the step lemma.

◀ Lemma 4.1

Let $m_v \in M_v$ be a VMM memory configuration. Consider a memory operation with memory inputs

$$(mop, i, va, din) \in Mop_r \times \{1, \ldots, n\} \times Va \times Din \; .$$

Let $(m_v', dout)$ denote the successor VMM memory configuration and the data output, i.e. $(m_v', dout) = dec_m(p, mop)(din, m_v)$. Assume that the memory operation is attached and all access conditions hold.

Then, the following equality holds:

$$dec_r(mop)(din, \Pi(m_v)) = (\Pi(m_v'), dout) \tag{4.89}$$

PROOF

The proof is rather straightforward. Still, we have chosen to present it in all detail to make the reader familiar with the data structures and concepts used to manage the memory in the VMM.

Let $excp_r$ denote the exception flag from the definition of RMM memory operations. We claim that $\neg excp_r$ holds. This is because of the rights consistency condition: we have $mop \in lr(m_v, mop, i, va)$ and therefore $mop \in r(m_r)(va)$. Since presence of rights precludes exceptions, we have $\neg excp_r$.

Let d_r and d_v denote the memory input to the elementary memory operation as read by the RMM and the VMM. We show now that $d_r = d_v$.

Because of the *laloc* consistency condition, we know the value of the *laloc*-function at the location $la = ltr(m_v, i, va) = tr(m_r)(tid, va)$: it must correspond to the main memory address indicated by the implementation translation function. So, we have

$$dec_{laloc}(m_v)(la) = (1, ma, 0) \; . \tag{4.90}$$

We use this knowledge to rewrite the definition of the data input d_v. By definition, d_v equals the data in the main memory cell *ma*; this corresponds to the data of the projected memory configuration at address *la*; this data equals the RMM data input d_r. Hence, the following equations hold:

$$\begin{aligned} d_v &= mm(m_v)(ma) \\ &\overset{lc}{=} dec_{mem}(m_v)(la) \\ &= mem(m_r)(la) \\ &= d_r \end{aligned}$$

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

char s[16] = "empty";

int main( int argc, char **argv ) {
    switch( fork() ) {
        case -1: exit(-1);
        case 0: sprintf( s, "parent" ); break;
        default: sprintf( s, "child" );
    }
    exit(0);
}
```

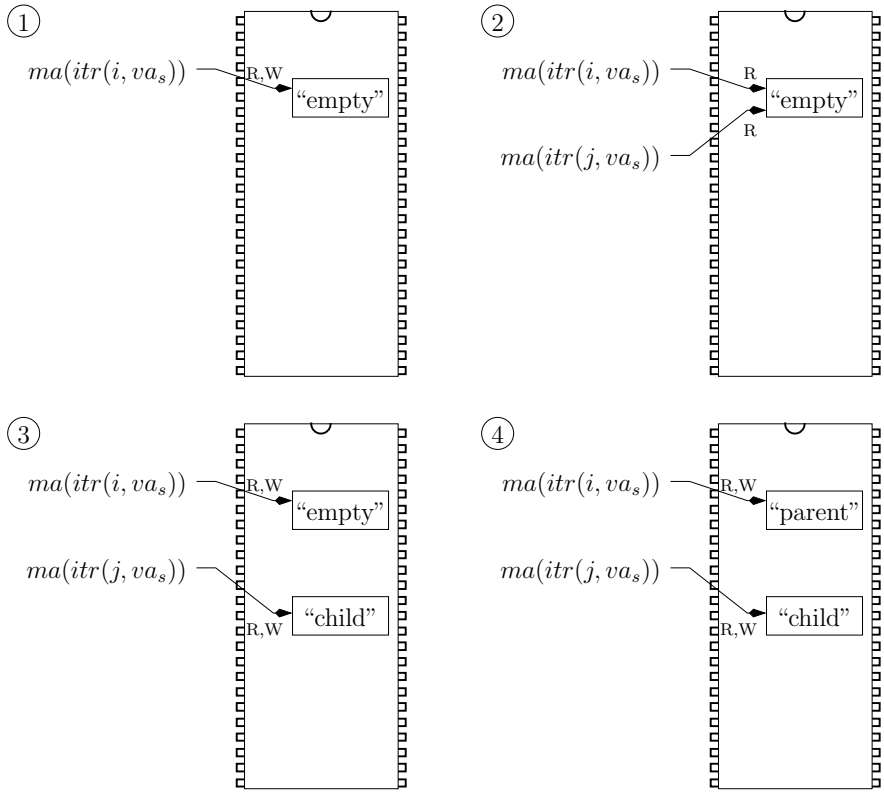

Figure 4.8 A `fork` Implementation with Copy-On-Write. Let va_s denote the virtual address of the strings `s`, assume that the parent gets executed on processor i and the child gets executed on processor j. With itr we denote the implementation translation function as appropriate for the cycle. Step ① is before the `fork`, step ② is after the `fork` before any `sprintf()`, step ③ is after the child's `sprintf()` where COW-resolving is needed, step ④ is after the parent's `sprintf()`.

Since $d_v = d_r$, the result of the elementary memory operation function *emop* is equal as well. Hence, we have $(d', edout) \in Data \times Edout$ with

$$emop(edin, d_v) = (d', edout) = emop(edin, d_r) \tag{4.91}$$

This implies the first part of our proof goal: since we have the same elementary data output and the same exception flag for the RMM and the VMM, the encoded data output, as computed by the *encout*-function, is also equal for the RMM and the VMM.

Both machines update their memory by storing the updated data d' in the appropriate location: the VMM writes d' to the main memory at address *ma*, the RMM writes d' to the memory at address *la*. We must show that the update is equivalent which means that in the memory projection of the VMM configuration m'_v only the address *la* must change. Our first observation is that all data structures in the VMM but dec_{mem} remain unchanged by the update: by convention the data structures only depend on the system memory and by the system memory condition the main memory address *ma* lies outside the system memory, i.e. $ma \notin sysmem$ (cf. Equation 4.83). Therefore the claim is (with m'_r denoting the successor configuration obtained by RMM memory semantics):

$$dec_{mem}(m'_v) = mem(m'_r) \tag{4.92}$$

By extensionality, the claim holds iff it holds for all addresses $la' \in La$:

$$\forall la' \in La : dec_{mem}(m'_v)(la') = mem(m'_r)(la') \tag{4.93}$$

This is broken down into two cases:

- Assume la' is the translation of (tid, va), i.e. $la' = la$. We can show, that the VMM update to the main memory is reflected in the projection of the memory. The reasoning is similar to the proof on the equality of d_v and d_r. The logical address location function value for *la* is known:

  $$\begin{aligned} dec_{laloc}(m'_v)(la) &= dec_{laloc}(m_v)(la) \\ &= (1, ma, 0) \end{aligned}$$

  Therefore, we also know that the value of the memory projection at the address *la* is equal to d'. This is in turn equal to the value of the new RMM memory at location *la*:

  $$\begin{aligned} dec_{mem}(m'_v)(la) &= mm(m'_v)(ma) \\ &= d' \\ &= mem(m'_r)(la) \end{aligned}$$

- Now consider a logical address la' with $la' \neq la$. For such addresses, the RMM memory cells do not change. We have to show, that this is also the case in the VMM.

  For instruction $mop \in Mop_r \setminus W$, no memory change whatsoever is observable and therefore the claim holds:

  $$\begin{aligned} dec_{mem}(m'_v)(la') &= dec_{mem}(m_v)(la') \\ &= mem(m_r)(la') \\ &= mem(m'_r)(la') \end{aligned}$$

Otherwise, for $mop \in W$, we must apply the copy-on-write condition: since la' is different from la both addresses may not be physically shared, i.e. their logical address location must also be different:

$$dec_{laloc}(m'_v)(la') = dec_{laloc}(m_v)(la') \neq dec_{laloc}(m_v)(la) = dec_{laloc}(m'_v)(la) \quad (4.94)$$

Since the main memory is only changed at the single location ma and the swap memory remains unchanged, the logical address location of la' points to an unchanged swap or main memory cell. Therefore the same sequence of equalities from above holds, proving the claim:

$$\begin{aligned} dec_{mem}(m'_v)(la') &= dec_{mem}(m_v)(la') \\ &= mem(m_r)(la') \\ &= mem(m'_r)(la') \end{aligned}$$

This proves the whole claim of the step lemma.

4.5 The Translator

This section is concerned with a semantics definition of the translator module. Informally, it is a hardware module that decodes the implementation translation function at some requested location.

There are three points requiring special attention. First, anyone designing a translator should be relieved from the burden of the parallel environment in which it is going to run. Second, nevertheless, the semantics definition should exhibit nice parallel properties that we can use in the proofs of this chapter. Third, the definition must enable easy integration of a TLB (a translation look-aside buffer, a cache for translations) in the translator module. A translator with a TLB is able to translate a request without performing a memory operation; this requires a special treatment of translation consistency.

Let trc_{x1x2} denote the traces of a VMM computation. Let seq_{b2m} and $\mathcal{M}_v$ denote the event sequence and the memory configuration sequence at the $b2m$-interface, which is, by assumption, sequentially consistent.

Recall that the translator is controlled by the following interface: request inputs are a processor identifier, a virtual memory address, and a memory operation, provided by bridge 1; request outputs are a boolean exception flag and a main memory address.

$$Iobs_{b1xlt} := \mathbb{B} \times (\{1,\ldots,n\} \times Va \times Mop_r) \times \mathbb{B} \times (\mathbb{B} \times Ma) \quad (4.95)$$

Let $r \in \mathbb{N}$ denote the starting time of a translator request of processor $i \in \{1,\ldots,n\}$ and let $u \in \mathbb{N}$ denote the ending time of the same request, so $r \leq u$. This can be expressed with the request predicate defined in Equation 2.6 in Section 2.2 simply as $isreq_{b1xlt}(r,u,i)$. Corresponding to r and u we define sequence numbers s_r and s_u which denote the sequence number of the first and last memory operation request by the translator for the time period between r and u. We assume that throughout the whole translation request, the translator has exclusive access to the memory, which means that between s_r and s_u only translator memory operation events of processor i take place.

On acknowledgment, the translator must return the value of the address translation with respect to the memory configuration at the start of the translation request.

This must be formalized in terms of the memory events taking place between the indices s_r and s_u. Let m_v denote the memory configuration resulting from the memory operation with sequence index $s_r - 1$, i.e. $m_v := \mathcal{M}_v(s_r)$. We call m_v the *translation base memory configuration*. Assume that all operations with sequence indices $u \leq s' \leq r$ have a processor index i and a memory operation type $ty = xlt$. Let $(mop, i, va) := din(trc_{b1xlt}(r,i)) \in Mop_r \times \{1,\ldots,n\} \times Va$ denote the request inputs. Let $(excp, ma)$ denote the implementation translation of (i, va) with respect to the memory configuration m_v:

$$(excp, ma) := dec_{itr}(m_v)(i, va, mop) \tag{4.96}$$

Under these assumptions, the translator must return the result of the implementation translation lookup:

$$(excp, ma) = dout(trc_{b1xlt})(u, i) \tag{4.97}$$

Later, we will introduce additional *software* assumptions that guarantee satisfaction of Equation 4.97 even when the translator has non-exclusive access to the memory.

4.6 The Supervisor

The supervisor is the exception service routine that is called, when the processor tries to perform a non-attached operation. The supervisor eventually terminates (i.e. returns from exception) after it has made appropriate modifications to the VMM memory configuration. Subsequently, the offending instruction will be repeated.

There are two cases to distinguish for a non-attached operation: either, there is a logical right for the operation or there is none.

In the first case, there have been implementation reasons not to attach the operation before even though it is *legal* by logical rights. As we pointed out when discussing access condition several such reasons exists, e.g. data might be in swap memory or it might need protection because of being *pnl*-shared. For legal accesses, the supervisor must attach the operation so that the processor (in RMM mode!) may eventually execute it. The exception and the supervisor's operation go unnoticed by the RMM program.

In the second case, it is impossible to attach the operation. The memory operation is *illegal*: the absence of logical rights for the operation indicates a programming error or ill intent of the user task run by the processor. The reaction should be the same in the RMM and the VMM: in real operating system environments, the task's software exception handler gets called to cope with the situation; the standard response is to terminate the offending task.

We call an exception due to the first reason a *memory management exception*; an exception due to the second reason is called a *violation exception*.

In this section we give a supervisor specification. This specification is given in two parts, each one being associated with a specific property of the parallel VMM.

First, we derive the so-called *attachment invariant*. This invariant states a cooperative goal of all supervisors: to guarantee that the access conditions (cf. Section 4.4.3) always hold for all attached memory operations. The attachment invariant is an implication of four runtime conditions. The step lemma (from Section 4.4.4) and the

attachment invariant are the key parts to data consistency between the RMM and the VMM.

Second, we derive the liveness of the VMM, this was already defined as a critical goal in Section 4.3.1. Liveness is needed when we project the virtual memory operations of the VMM on RMM machines: every legal memory operation requested by the processor must eventually be acknowledged. In the VMM, this corresponds to the exception-free acknowledgment of an RMM memory operation. To show this property we need as the core property of the supervisor that it attaches legal exception operations. This alone is not sufficient. Other supervisors have to heed the supervisor semantics; they must not detach an exception operation (attached by another supervisor) before its repetition.

The memory management strategy can be summarized in several "rules of thumb":

- If a detached legal operation is requested, it must eventually be attached.
- An attached operation must satisfy the access conditions.
- An illegal operation must not be attached.

There are three typical supervisor operations that are concerned with user task data movement and are present in all implementations: *swap-in* is a movement of user task data from swap memory to main memory; *swap-out* is a movement of user task data from main memory to swap memory; *copying* is a movement of user task data from a main memory location to a different main memory location (to resolve *pnl*-sharing). A fourth variant—cell relocation inside the swap memory—is usually not needed.

In any case, user task data may only be moved or copied *but not changed*. This property is called *tamper-freeness* of the supervisor. Formally, for all supervisor memory operations on a memory configuration m_v with successor memory configuration m_v' we must have equal projections:

$$\Pi(m_v') = \Pi(m_v) \tag{4.98}$$

4.6.1 The Attachment Invariant

The processor may perform a memory access whenever the address is marked as being in main memory and has the appropriate right available as indicated by the implementation translation function. Any such processor identifier, virtual address, and memory operation identifier is called *attached*. We usually abbreviate this by saying that a particular memory operation is attached or not. In the latter case we also speak of *detachment*.

In the running system we would like all attached addresses to satisfy the access conditions; we call this property *attachment invariant*. As we saw in Section 4.1, proving the step lemma, we can guarantee that operations with satisfied access conditions are performed equivalently on the RMM and the VMM.

To formalize this: given a VMM memory configuration $m_v \in M_v$ we define the following invariant $inv(m_v)$:

$$\forall mop, i, va : att(m_v, mop, i, va) \Rightarrow ac(m_v, mop, i, va) \tag{4.99}$$

Here, $att(m_v, mop, i, va) := \neg excp(dec_{itr}(m_v)(i, va, mop))$ denotes attachment and the predicate $ac(m_v, mop, i, va)$ denotes the conjunction of all access conditions. We intend to inductively satisfy this invariant.

Let seq_{b2m} denote the event sequence at the $b2m$-interface and let $\mathcal{M}_v$ denote the memory configuration sequence, which exists by the assumption of sequential consistency on the VMM memory. Let $s \in \mathbb{N}$ denote some sequence number. Let $(t,i) \in \mathbb{N} \times \{1,\ldots,n\}$ denote the event that is determined by s through $(t,i) := seq_{b2m}(s)$. Let $e \in Iobs_{b2m}$ denote the associated interface observation, i.e. $e := trc_{b2m}(t,i)$.

We assume that the operation associated with e is a supervisor operation. This is indicated by the type flag of the memory operation of the interface observation that must equal sv, i.e. $ty(mop(e)) = sv$.

Let m_v, $m'_v \in \mathbb{N}$ denote the memory configuration prior to and after the execution of the memory operation. These memory configurations are given by the memory configuration sequence at location s and location $s+1$:

$$m_v := \mathcal{M}_v(s) \tag{4.100}$$
$$m'_v := \mathcal{M}_v(s+1) \tag{4.101}$$

To satisfy the attachment invariant inductively we look for suitable restrictions on the supervisor operations, such that from $inv(m_v)$ we can conclude $inv(m'_v)$.

In the following section, we develop these restrictions calling them *runtime conditions*. After this, we prove the inductive preservation of the attachment invariant.

Operation Restrictions

The supervisor may not execute operations involving changes to the memory management and task management data structures. So, in particular, it may not switch, kill or create tasks. It may not modify save areas of sleeping tasks. It may not change logical translations and rights.

Let m_v, $m'_v \in M_v$ be VMM memory configurations with m'_v being the successor memory configuration to m_v after executing some supervisor memory operation. Then, non-modification of the memory management and task management data structures is easily expressed with the decode functions:

$$dec_{ctid}(m_v) = dec_{ctid}(m'_v) \tag{4.102}$$
$$dec_{atid}(m_v) = dec_{atid}(m'_v) \tag{4.103}$$
$$dec_{sar}(m_v) = dec_{sar}(m'_v) \tag{4.104}$$
$$dec_{tr}(m_v) = dec_{tr}(m'_v) \tag{4.105}$$
$$dec_{r}(m_v) = dec_{r}(m'_v) \tag{4.106}$$

Figure 4.9 illustrates this property. The dashed arrow in the bottom of the figure indicates that the memory management and task management data structures must not change.

Conservative Attachment

According to the supervisor semantics, as defined above, the supervisor attaches the exception operation, specified by the inputs to the supervisor request. The supervisor may do more than this: for example, supervisors typically move data to and from the main memory and the swap memory in big blocks to optimize I/O operations. Thus, a supervisor may want to update the implementation translation and the logical address location not only for the exception address but for other addresses as well.

Special attention is needed for such updates as different processors might try to access these addresses while the supervisor is running. This is not the case for the

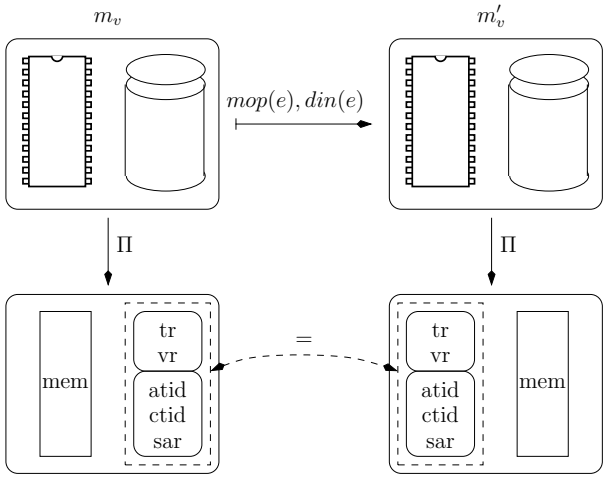

Figure 4.9 Operation Restriction Runtime Condition

exception address: by definition of the VMM, the program which caused the exception is suspended until the supervisor returns and there cannot be another processor trying to access that address because we require the processors to run distinct tasks.

Whenever the supervisor attaches a memory operation, it has to ensure that the access conditions hold. Let m_v and m'_v denote the input and output memory configuration of a supervisor operation, as already defined. Let $(mop, i, va) \in Mop_r \times \{1, \ldots, n\} \times Va$ denote a memory operation, a processor identifier, and an address. Then, we require that "becoming-attached" of an operation in m_v and m'_v establishes the access conditions:

$$\neg att(m_v, mop, i, va)) \wedge att(m'_v, mop, i, va) \Rightarrow ac(m'_v, mop, i, va) \tag{4.107}$$

Moving Restrictions

The last runtime condition is concerned with the moving of addresses while they are attached. Formally, two conditions are considered a move of an address (i, va): first, the memory address pointed to by the implementation translation function may change from configuration m_v to m'_v; second, the logical address location of the associated logical address may change from configuration m_v to m'_v.

We define a *moved*-predicate for the memory configurations m_v and m'_v, the memory operation $mop \in Mop_r$, the processor identifier $i \in \{1, \ldots, n\}$, and the virtual address $va \in Va$. Assume that (mop, i, va) is an attached operation in memory configuration m_v as well as in memory configuration m'_v. Thus, we have:

$$att(m_v, mop, i, va) \wedge att(m'_v, mop, i, va) \tag{4.108}$$

If this is the case, then the *moved*-predicate holds if (i) the memory address indicated by the implementation translation function has changed or (ii) the logical address location has changed. The first condition can be written as

$$dec_{laloc}(m_v)(la) \neq dec_{laloc}(m'_v)(la) \tag{4.109}$$

where $la = ltr(m_v, i, va)$ is the logical translation of (i, va), which is the same in both

configurations. The second condition can be written as

$$ma(dec_{itr}(m_v)(i,va,mop)) \neq ma(dec_{itr}(m'_v)(i,va,mop)) \ . \tag{4.110}$$

So, the whole predicate is defined as follows:

$$moved(m_v,m'_v,mop,i,va) := (4.108) \wedge ((4.109) \vee (4.110)) \tag{4.111}$$

For moved addresses, we have to pay attention on four things:

First, the supervisor has to ensure that the moved address still maps to some main memory address outside the system memory.

$$moved(m_v,m'_v,mop,i,va) \Rightarrow sys(m'_v,mop,i,va) \tag{4.112}$$

Second, the supervisor has to keep the address consistent according to the *laloc* consistency access condition, so that the main memory address indicated by the implementation translation function still points the same main memory location that the logical address location function points to. Note, that by this conditions atomic simultaneous updates of the logical address location function and the implementation translation functions are required. This is not farfetched as one may use clever encodings of the data structures which easily allow for such updates. We require:

$$moved(m_v,m'_v,mop,i,va) \Rightarrow lc(m'_v,mop,i,va) \tag{4.113}$$

The likeliest reason for a supervisor to move an address is due to copy-on-write; supervisors typically need to copy (read-) attached addresses in main memory to resolve *pnl*-sharing. Note, however, that if this takes place for the exception address only, no harm can be done, since no other processor may access it.

Third, if *mop* is a writing instruction, i.e. $mop \in W$, we must require that the new main memory location is not *pnl*-shared by another address. This is exactly the copy-on-write access condition. We require:

$$moved(m_v,m'_v,mop,i,va) \Rightarrow cow(m'_v,mop,i,va) \tag{4.114}$$

Fourth, there is a copy-on-write condition with swapped roles: no write-attached address may exist that is logically different from $ltr(m'_v,i,va)$ but located at the same main memory address. This would be a violation to the COW access condition of the *other* address. We call this condition the reversed copy-on-write condition, *rcow* and define it this way:

$$\begin{aligned} rcow(m_v,mop,i,va) = \ & \forall mop',i',va' : \\ & mop' \in W \wedge \\ & att(m'_v,mop',i,va') \wedge \\ & ltr(m_v,mop,i,va) \neq ltr(m'_v,mop',i',va') \\ & \Rightarrow ma(dec_{itr}(m_v)(i,va,mop)) \neq ma(dec_{itr}(m_v)(i',va',mop') \end{aligned} \tag{4.115}$$

So, for all moved addresses, *rcow* must hold:

$$moved(m_v,m'_v,mop,i,va) \Rightarrow rcow(m'_v,mop,i,va) \tag{4.116}$$

Of course, the easiest thing to satisfy the moving restrictions in a real implementation is by not moving attached addresses at all.

Inductive Preservation of the Attachment Invariant

Remember that our goal is to show the inductive preservation of the invariant $inv(m_v)$ which requires that for all attached operations the access conditions hold:

$$inv(m_v) := \forall mop, i, va : att(m_v, mop, i, va) \Rightarrow ac(m_v, mop, i, va) \tag{4.117}$$

Let m_v and m'_v denote the input and output memory configuration of some supervisor memory operation, as already defined in the introduction. Let $rtc(m_v, m'_v)$ denote the conjunction of the runtime conditions defined above, i.e. the operation restrictions, the conservative attachment, and the moving restrictions. We claim:

$$inv(m_v) \wedge rtc(m_v, m'_v) \Rightarrow inv(m'_v) \tag{4.118}$$

Consider a memory operation mop, a processor identifier $i \in \{1, \ldots, n\}$ and a virtual address $va \in Va$. Assume that (mop, i, va) is attached in m'_v, i.e. $att(m'_v, mop, va)$ holds.

If $\neg att(m_v, mop, i, va)$ the conservative attachment property applies, satisfying the access condition $ac(m'_v, mop, i, va)$.

Otherwise, we have $att(m_v, mop, i, va)$. Now consider the four access conditions for (m'_v, mop, i, va).

1. For unmoved addresses, the system memory condition holds for m'_v because it held for the operation in configuration m_v. For a moved address, the system memory condition holds by the first moving restriction, Equation 4.112.

2. The rights consistency condition holds because of the operation restrictions. If the operation was attached in m_v then it was rights-consistent by assumption. Since memory and task management data structures did not change, it must still be rights-consistent.

3. For the *laloc* consistency condition we distinguish two cases: If (i, va) was moved, i.e. $moved(m_v, m'_v, i, va)$ holds, then the *laloc* consistency condition holds by the second property of the moving restrictions. Otherwise, using the operation restrictions and $lc(m_v, mop, i, va)$ we derive $lc(m'_v, mop, i, va)$.

4. Now we have to show $cow(m'_v, mop, i, va)$ which is defined as

   $$mop \in W \Rightarrow \neg sh_{pnl}(m_v, tid, va)\ . \tag{4.119}$$

   Assume that $mop \in W$. By expanding sh_{pnl} and the sharing relations, our goal reduces to

   $$\neg\exists tid', va' : la \neq la' \wedge dec_{laloc}(m_v)(la) = dec_{laloc}(m_v)(la') \tag{4.120}$$

   where $la = tr(\Pi(m_v))(tid, va)$ and $la' = tr(\Pi(m'_v))(tid', va')$. Assume that (i, va) was not moved (otherwise the claim follows from the third property of the moving restrictions).

   Then, distinguish two cases: if (i', va') was moved, then we get a contradiction by the reverse COW condition of the moving restrictions; otherwise, we get a contradiction by the COW condition of the previous cycle, $cow(m_v, mop, i, va)$.

4.6.2 Liveness

Up to now, we have stated that a memory operation performs equivalently on the RMM and on the VMM when a set of access conditions is met. Then, we examined what it takes to preserve the access conditions for all attached memory operations at once.

Yet, we have not examined *when* memory operations must get attached and when they must not get detached: this issue is closely related with the liveness of the machine.

When a processor tries to perform a non-attached memory operation, the supervisor is called by the exception handling mechanisms; it "receives" as input the exception memory operation (indirectly, by looking at the save processor configuration *savep*). If the exception memory operation is legal, so that it is allowed in the corresponding RMM memory configuration, the supervisor must guarantee that the operation is attached after it returns. The *other* supervisors must guarantee, that they will not detach the operation, before the processor repeats it. Thus, we can guarantee forward progress: any legal memory operation can be performed after at most one supervisor request. This result is required at a quite early point of the VMM machine specification: liveness of the $pb1$-interface is one of the four interface handshake conditions.

In the following three sections we will define the above conditions. After this we will show a liveness property.

Supervisor Semantics

Informally, the supervisor must attach the exception operation. In formalizing this we have to decide *when*—with respect to which memory configuration of our memory configuration sequence—this should happen. We choose the memory configuration after the last executed memory operation of the processor (must be a supervisor memory operation) as a reference point. With respect to it, the exception memory operation and exception memory address have to be attached.

Let seq_{b2m} denote the sequence of memory operations at the $b2m$-interface and let $\mathcal{M}_v$ denote the sequence of memory configurations, as before. The interface traces are denoted by trc_x.

Let $u \in \mathbb{N}$ denote any time for which the supervisor of processor $i \in \{1,\ldots,n\}$ returns from exception, i.e. $mode(p(u,i)) \wedge \neg mode(p(u+1,i))$. Let s denote a sequence number of the bridge-2 sequence such that its associated time t is less than u but maximal. This means, that (t,i) is the last acknowledged memory event of processor i before time u at the bridge-2 to shared memory interface. Since the supervisor has to execute at least a single memory operation in order to attach the exception operation, t denotes the last memory operation made by the supervisor to bridge 2. The maximality requirement can be formulated as follows:

$$\forall \hat{s} > s, \hat{t} : seq_{b2m}(\hat{s}) \neq (\hat{t},i) \vee \hat{t} \geq u \tag{4.121}$$

In memory configuration $\mathcal{M}_v(t+1)$ which is the result of that memory operation the exception operation must be attached. Let *emop* denote the exception memory operation and let *eva* denote the exception virtual address, which can be taken from the saved processor configuration. We demand

$$att(\mathcal{M}_v(t+1), emop, i, eva)\ . \tag{4.122}$$

Supervisor Call Persistence

According to the semantics of the supervisor, it attaches the exception operation. Then, the memory operation is repeated by the processor. After the supervisor acknowledg-

ment and before the repetition of the instruction, however, arbitrary memory operations from other processors may be performed on the memory. For regular processor operations, this does not cause any harm: processor memory operations have no access to the special data structures of the VMM. Other processor's supervisor instructions may, however, undo the effect of the supervisor request (as the supervisor is a routine of the operating system and thus usually trusted, we should add that this should happen only by accident).

Therefore we require supervisors to heed each other, they may not undo the effects of a supervisor routine before instruction repetition. This way, the result of a supervisor call *persists* until the exception operation's repetition. In detail, other supervisors may not detach an operation between supervisor return and repetition of the operation.

Let s_1 and s_2 with $s_1 < s_2$ denote two sequence numbers. These sequence numbers shall be associated with the last supervisor operation and the instruction repetition of a processor i. So, we must have times t_1, $t_2 \in \mathbb{N}$, such that $seq_{b2m}(s_1) = (t_1, i)$ and $seq_{b2m}(s_2) = (t_2, i)$. Of course, since $s_1 < s_2$ we also have $t_1 < t_2$.

The interface observation $trc_{b2m}(t_1, i) \in Iobs_{b2m}$ must be a supervisor type memory operation; the interface observation $trc_{b2m}(t_2, i) \in Iobs_{b2m}$ must *not* be a supervisor type memory operation. If there is no intervening operation of processor i, then the first operation must be the last operation of a supervisor request, while the second operation is the repetition operation. So, for all $\hat{s}$ with $s_1 < \hat{s} < s_2$, the event $seq_{b2m}(\hat{s})$ must not be for processor i. In formula,

$$sv(ty(trc_{b2m}(t_1, i))) \wedge \neg sv(ty(trc_{b2m}(t_2, i))) \wedge \forall s_1 < \hat{s} < s_2, \hat{t} : seq_{b2m}(\hat{s}) \neq (\hat{t}, i) \; . \tag{4.123}$$

As in the previous section, the exception operation, which caused the request to the supervisor, can be determined by looking at the processor configuration when the supervisor was called on processor i. We denote the exception memory operation identifier by $emop \in Mop_r$ and the exception data input by $edin \in Din_r$. Also, we let eva denote the exception virtual address, i.e. $eva := va(args(edin))$.

Let s' denote the sequence number of a supervisor operation of a processor $i' \neq i$ between the other two operations, so

$$s_1 < s' < s_2 \text{ and } seq_{b2m}(s') = (t', i') \text{ and } sv(ty(trc_{b2m}(t', i'))) \; . \tag{4.124}$$

We now define a condition on the memory configuration after the execution of supervisor operation s'. Let $m_v = \mathcal{M}_v(s')$ denote the memory configuration before the operation and $m'_v = \mathcal{M}_v(s' + 1)$ denote the memory configuration after the operation. Then, if the exception operation is attached in configuration m_v it must be attached in configuration m'_v, too. So,

$$att(m_v, emop, i, eva) \Rightarrow att(m'_v, emop, i, eva) \; . \tag{4.125}$$

Translation Persistence

In this section we are concerned with a property of the supervisor which allows to lift the translator semantics given in Section 4.5 to a fully parallelized translator semantics. Particularly, we expect that translator output corresponds to a consistent lookup of the dec_{itr} function. This can only be guaranteed if other processors do not modify the translation while it is being computed. Again, through the system memory convention such modifications can only happen by the use of other supervisor operations. Therefore, a supervisor may not change a translation used by another processor.

This condition is cumbersome to formulate precisely for a general translation procedure; but nevertheless, the right rule of thumb can be derived from it. While one processor is translating, no other processor shall by any supervisor memory operation change the result of any of the (read-only) translator memory operations used to perform the address translation. This must be guaranteed until the last translator operation (if the translator determines detachment of the desired memory operation) or until the associated processor memory operation (if the translator determines attachment).

Let s_1 and s_2 with $s_1 < s_2$ denote two sequence numbers. Assume that the first sequence number is associated with a translator operation and the second sequence number is associated with a (different processor's) supervisor operation. Furthermore, we assume that the translation has not yet ended, so there is no sequence number between s_1 and s_2 that is associated with a non-translation operation of the first processor.

Again, let t_1, $t_2 \in \mathbb{N}$ denote the times of the operations and i_1 and i_2 (for $i_1 \neq i_2$) denote the processors; i.e. we have $seq_{b2m}(s_1) = (t_1, i_1)$ and $seq_{b2m}(s_2) = (t_2, i_2)$. Of course, since $s_1 < s_2$ we also have $t_1 < t_2$. The interface observation $trc_{b2m}(t_1, i_1) \in Iobs_{b2m}$ must be a translator type memory operation; $trc_{b2m}(t_2, i_2) \in Iobs_{b2m}$ must be a supervisor type memory operation.

Since there should be no intervening non-translation operation of processor i_1 between s_1 and s_2, we require that for all $\hat{s}$ with $s_1 < \hat{s} < s_2$, the event $seq_{b2m}(\hat{s})$ must not be associated with processor i_1 or be a translation operation of processor i_1. So, overall

$$\begin{aligned} & xlt(ty(trc_{b2m}(t_1, i_1))) \\ \wedge\ & sv(ty(trc_{b2m}(t_2, i_2))) \\ \wedge\ & \forall s_1 < \hat{s} < s_2, \hat{t} : seq_{b2m}(\hat{s}) \neq (\hat{t}, i_1) \vee xlt(ty(trc_{b2m}(\hat{t}, i_1))) \ . \end{aligned} \tag{4.126}$$

In this case, we require that the translation operation applied on the memory configuration in step s_1 gives the same result *dout* when applied to the memory configuration in step s_2:

$$\begin{aligned} & dout(dec_m(mop(trc_{b2m}(t_1, i)))(din(trc_{b2m}(t_1, i)), \mathcal{M}_v(s_1))) = \\ & \qquad dout(dec_m(mop(trc_{b2m}(t_1, i)))(din(trc_{b2m}(t_1, i)), \mathcal{M}_v(s_2))) \end{aligned} \tag{4.127}$$

Note that it is not necessary to demand this if the translator has already determined detachment of a memory operation; this allows to attach operations "on-the-fly" in practice.

Translation Persistence Lemma. Under translation persistence, a translator returns the value of the implementation translation function looked up at the input arguments based on the first memory configuration it sees. Furthermore, this function lookup does not change for all memory configurations seen by the translator. For an exception-free translator result, it stays the same even for the memory configuration seen by the next processor memory operation at this processor.

To prove this lemma, we exploit that the translator operates deterministically. Its output depends on the starting state and the response of the memory operation requests. We can construct a computation that has only the translator operations of the current translation operation. Such a computation would satisfy the assumptions of the translator semantics, hence we know that in this computation, the translator returns the implementation translation function lookup on acknowledgment. By induction, we can show, that the translator configurations of both computation are identical for each

cycle. As the translator computes its output as a function of its configuration, the output of the original computation must also be equal to the implementation translation function lookup.

Proving Liveness

We use the translator semantics, the supervisor semantics, the supervisor call persistence, and the translation persistence to show that the VMM is live with respect to RMM operations performed by the processor.

We show liveness for legal operations.

$$\forall t : req(trc_{pb1}(t,i)) \wedge p(ty(trc_{pb1}(t,i)) \Rightarrow \\ \exists u > t : ack(trc_{pb1}(u,i)) \wedge \neg excp(trc_{pb1}(u,i)) \quad (4.128)$$

Since the exception flag of the processor is directly connected with the exception flag of the translator output, the address translation is crucial for the liveness of the machine.

We will show that, after one exception has occurred and the supervisor has completed its execution, the translator will not signal an exception on the retry of the translation. As the supervisor attaches the exception operation by its semantics, we need to reason, that the translator *will see* this attachment and hence generate no exception.

The proof sketch is as follows:

1. According to the supervisor semantics, the supervisor must have attached the memory operation after its last memory update.

2. Between the last supervisor operation and the first translation operation, other supervisors may not detach the exception operation: this is due to supervisor call persistence.

3. When the first translation operation gets processed, the exception operation is still attached. The translation persistence lemma guarantees, that the translator returns a result to bridge 1, based on the memory configuration seen at the first translator operation. Hence, we will have no exception.

Let us remark here, that proving liveness for real instruction set architectures, such as the VAMP described in Chapters 5–7, is slightly more complicated since page-faulting memory accesses may require the repetition of more than one processor computation step. This is because, several computation steps (called *phases* for the multiprocessor VAMP) form larger computation steps (called *instructions* for the VAMP, *units of operations* for the IBM S/390 architecture [IBM00]). Chapter 7 treats such a case in more detail.

4.7 Simulation Theorem: VMM implements the RMM

In this section we prove that the virtual memory machine implements the relocated memory machine: our claim is that all processor-bridge-1 traces in the VMM are sequentially consistent. This means that all the bridges, translators, supervisors in combination with the shared VMM memory can be interpreted as a sequentially consistent, shared RMM memory on which the processors operate. The VMM machine structure is transparent to the processors.

To prove this claim, we observe an arbitrary computation of the VMM. We must prove the existence of a sequence over the (parallel) events at the processor-bridge-1 interface of all processors and a sequence of RMM memory configurations such that the sequential consistency properties (Equations 2.33 to 2.38) are satisfied.

From the bridge-2 to shared-memory interface of all processors, which is sequentially consistent by assumption, we already have a sequence seq_{b2m} over the supervisor-type, the processor-type and the translator-type operations. From this sequence we construct the processor-bridge-1 event sequence: we define it as the subsequence of processor-type events from the sequence seq_{b2m}.

For this sequence we can show that it is surjective and globally-ordered.

For the data consistency we additionally need to define a sequence of RMM memory configurations. We do this by projecting the following subsequence of the VMM memory configuration sequence: the first VMM memory configuration is projected for initialization reasons and every VMM memory configuration resulting a processor-type operation is projected.

We claim that the updates on these projected VMM memory configurations are data-consistent with respect to RMM memory operation semantics. This can be shown in two steps, examining the intermediate VMM memory configurations between the two projected configurations. First, the projection of the VMM memory configuration does not change until the second projected configuration because only translator and supervisor memory operations may have been executed. Second, the access conditions hold before the VMM memory operation is performed; application of the step lemma shows that this operation is—by projection—equivalent to the RMM memory operation.

4.7.1 The Claims

We observe an arbitrary VMM computation by tracing all the interfaces $x1x2$ with the trace functions $trc_{x1x2} : [\mathbb{N} \times \{1,\ldots,n\} \to Iobs_{x1x2}]$.

By assumption, the bridge-2 to shared-memory interface is sequentially consistent: we have an event sequence seq_{b2m} and a memory configuration sequence $\mathcal{M}_v : [\mathbb{N} \to M_v]$ for the $b2m$-interface, which satisfies the sequential consistency properties (Equations 2.33 to 2.38).

$$seq_{b2m} : [\mathbb{N} \to E(trc_{b2m})] \tag{4.129}$$

$$\mathcal{M}_v : [\mathbb{N} \to M_v] \tag{4.130}$$

Figure 4.10, depicting two VMM computation steps around the sequence position $s \in \mathbb{N}$, illustrates the definition of the event sequence seq_{b2m}. The memory configurations $\mathcal{M}_v(s-1)$, $\mathcal{M}_v(s)$, and $\mathcal{M}_v(s+1)$ are drawn as boxes exposing the main memory mm and the swap memory sm. For each computation step there is a memory operation mop and a data input din. Since they are passed as inputs to the memory operation from the processor (bridge 2, more specifically), they are drawn as circles inside a downward arrow. Each such arrow is annotated with a pair $(i_k,t_k),ty_k$ indicating the associated event index and the type of the memory operation where $(i_k,t_k) = seq_{b2m}(s-1+k)$ and $ty_k = ty(mop(trc_{b2m}(i_k,t_k)))$ for $k \in \{0,1,2\}$. The (application of the) memory operation itself is indicated by an arrow from left to right labeled by the VMM's memory operation decode function dec_v. Inputs to this (curried) function are the memory operation, the data input, and the input memory configuration; outputs are updated memory

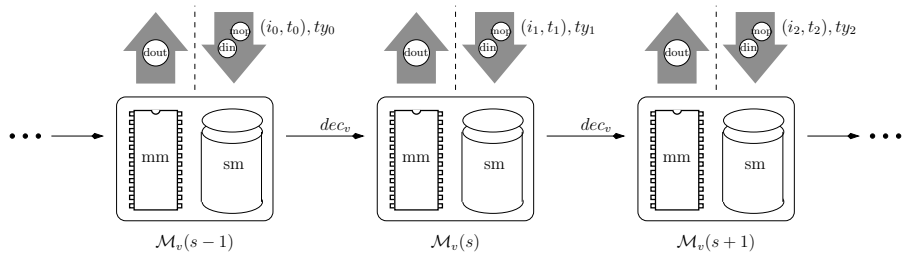

Figure 4.10 Sequentially-Consistent VMM Computation. Let $(i_k,t_k) = seq_{b2m}(s-1+k)$ and $ty_k = ty(mop(trc_{b2m}(i_k,t_k))) \in \{xlt, sv, p\}$ for $k \in \{0,1,2\}$.

configuration and the data output. The latter, meant to be passed back to the processor through the bridge-2-memory interface, is drawn as a circle in an upward arrow.

The first VMM memory configuration $\mathcal{M}_v(0)$ must satisfy the attachment invariant $inv(m_v)$. Thus, we already know that the attachment invariant is preserved for all VMM memory configurations:

$$inv(\mathcal{M}_v(0)) \Rightarrow \forall s : \forall mop, i, va : att(\mathcal{M}_v(s), mop, i, va) \Rightarrow ac(\mathcal{M}_v(s), mop, i, va) \quad (4.131)$$

To define the event sequence $seq_{pb1} : [\mathbb{N} \to E(trc_{b1s})]$ we take the subsequence of bridge-2-memory events that are associated with a processor type operation. Formally, we define an index sequence $idx : [\mathbb{N} \to \mathbb{N}]$ that indexes over all the processor-type observations of the bridge-2-shared-memory event sequence. This sequence must (i) be monotonous, i.e. $\forall s \geq t : idx(s) \geq idx(t)$, and (ii) map bijectively to the events that are associated with p-type operations, i.e. for all times t and processor indices i with $p(ty(trc_{b2m}(t,i)))$ and $ack(trc_{b2m}(t,i))$ there exists a unique sequence number s such that $seq_{b2m}(idx(s)) = (t,i)$. For $s \in \mathbb{N}$ we set

$$seq_{pb1}(s) := seq_{b2m}(idx(s)) \; . \quad (4.132)$$

We must first show that seq_{pb1} is well-defined: for each sequence number $s \in \mathbb{N}$, the bridge-2-memory event $seq_{b2m}(idx(s))$ must be a processor-bridge-1 event, too:

$$\forall s : seq_{b2m}(idx(s)) \in E(trc_{pb1}) \quad (4.133)$$

Then, we show the sequential consistency properties (Equations 2.33 to 2.38). The first two simple-to-prove claims are the following:

- The sequence seq_{pb1} is surjective with respect to the events of the processor-bridge-1 interface:

$$\forall t, i : (t,i) \in E(trc_{pb1}) \Rightarrow \exists s : seq_{b2m}(s) = (t,i) \quad (4.134)$$

- The sequence seq_{pb1} is globally-ordered, so a given event may only be preceded by events that start earlier in the trace:

$$\forall s_1, s_2, t_2, i_2 : seq_{pb1}(s_2) = (t_2, i_2) : s_1 < s_2 \Rightarrow strt(seq_{pb1}(s_1)) \leq t_2 \quad (4.135)$$

The complex part is the data consistency. Here we have to show the existence of a sequence of memory configurations $\mathcal{M}_r : [\mathbb{N} \to M_r]$, which represents the sequential

updates on the RMM memory. We construct this sequence with the help of the index function and the projection function. The first configuration is the projection of the first VMM configuration. The other configurations are all obtained by projecting the VMM configuration resulting from processor-type operations. The indices of these configurations can be obtained by incrementing the result of the index function for various arguments. We set:

$$\mathcal{M}_r(0) := \Pi(\mathcal{M}_v(0)) \tag{4.136}$$

$$\mathcal{M}_r(s+1) := \Pi(\mathcal{M}_v(idx(s)+1)) \tag{4.137}$$

The claim for the data consistency property is as follows: Let $s \in \mathbb{N}$ be a sequence number and let e denote the associated interface observation at the bridge-1-processor interface, i.e. $e := trc_{pb1}(seq_{pb1}(s))$. The memory configuration $\mathcal{M}_r(s+1)$ and the data output $dout(e)$ are the results of the application of the RMM memory operation $op_r := dec_r(mop(e))$ to the memory configuration $\mathcal{M}_r(s)$ and the data input $din(e)$.

$$\forall s \in \mathbb{N} : (\mathcal{M}_r(s+1), dout(e)) = op_r(\mathcal{M}_r(s), din(e)) \tag{4.138}$$

4.7.2 Proof of Data Consistency

In this section, we prove the data consistency of the defined RMM event and memory configuration sequence. We must show that:

$$\forall s \in \mathbb{N} : (\mathcal{M}_r(s+1), dout(e)) = dec_r(mop(e))(\mathcal{M}_r(s), din(e)) \tag{4.139}$$

where e is the associated interface observation, i.e. $e := trc_{pb1}(seq_{pb1}(s))$.

Let $s \in \mathbb{N}$ denote an arbitrary sequence position. Let $s'_1 := idx(s-1)$ and let $s'_2 := idx(s)$. Let $seq_{b2m}(s'_1) = (t_1, i_1)$ and let $seq_{b2m}(s'_2) = (t_2, i_2)$. By the construction of the memory configuration sequence, the RMM memory configurations $\mathcal{M}_r(s)$ and $\mathcal{M}_r(s+1)$ are projections of the VMM memory configurations $\mathcal{M}_v(s'_1+1)$ and $\mathcal{M}_v(s'_2+1)$. For the processor operation at sequence index s'_2 a translation base memory configuration (cf. Section 4.5) must exist, which we index by $s_x < s'_2$. The translation base memory configuration is usually associated with a translator memory operation of processor i_2, i.e. we have $seq(s_x) = (t_x, i_2)$ and $xlt(ty(trc_{b2m}(seq_{b2m}(s_x))))$ for some $t_x < t_2$ where t_2 Note that we know nothing of the relation between t_1 and t_x, we have $t_x < t_1$ or $t_x > t_1$ (or even equality in case the translator has cached the translation and $i_1 = i_2$).

Figure 4.11 illustrates this situation. The top row shows the computation step for the RMM projection: the memory operation $mop(e)$, the data input $din(e)$ (represented by the downward arrow) and the memory configuration $\mathcal{M}_r(s)$ serve as inputs to the memory execution function me_1 (which abbreviates decoding an application of an RMM memory operation). The outputs are the updated memory configuration $\mathcal{M}_r(s+1)$ and the data output $dout(e)$. In the bottom row we show the sequence of VMM memory operations. The memory configuration $\mathcal{M}_v(s_x)$ is drawn for reasons of clarity to the left of $\mathcal{M}_v(s'_1)$. To the right, we have $\mathcal{M}_v(s'_2)$ and its successor configuration. The upward arrows indicate the construction of the memory projection. We project $\mathcal{M}_v(s'_1+1)$ and $\mathcal{M}_v(s'_2+1)$, both being successors of processor memory operations.

The proof of data consistency can be derived from two important results:

1. Between s'_1+1 and s'_2 the projected memory configuration does not change.

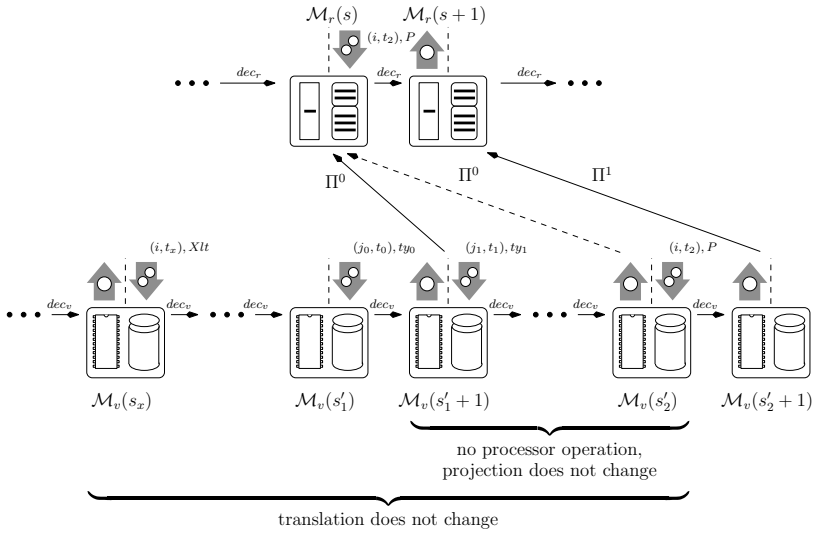

Figure 4.11 Proof Sketch for the Data Consistency Claim

We have no processor operation between s'_1+1 and s'_2 since otherwise this would violate bijectivity of the index function *idx* with respect to *p*-type operations. As translator operations are read-only and supervisor operations must not change the projected memory image (by tamper-freeness, Equation 4.98) the projection does not change:

$$\Pi(\mathcal{M}(s'_2)) = \Pi(\mathcal{M}(s'_1+1)) \tag{4.140}$$

2. Between s_x and s'_2 the translation of the processor operation does not change.

   This follows from the translation persistence lemma, which we apply for the extended case (attachment). If $(excp, ma)$ is the result returned by the translator, we have

$$excp = 0\,, \tag{4.141}$$
$$dec_{itr}(\mathcal{M}_v(s_x))(i_2, va) = (excp, ma)\,, \tag{4.142}$$
$$dec_{itr}(\mathcal{M}_v(s'_2))(i_2, va) = (excp, ma)\,. \tag{4.143}$$

Now we know that the operation is attached in $\mathcal{M}_v(s'_2)$ and the memory address, for which we make the elementary VMM memory operation, corresponds to the address returned by the implementation translation for $\mathcal{M}_v(s'_2)$. Because of the attachment invariant, we know that the access conditions hold, i.e. $ac(\mathcal{M}_v(s'_2, mop, i_2, va))$. Also, since the translation was correct, we know that $\mathcal{M}(s'_2+1) = dec_m(mop_r)(edin, ma) = dec_v(mop_r)(din_r)$. Thus, we can apply the step lemma in the situation indicated by the trapezoid on the right-hand side between both rows in Figure 4.11 and establish data consistency.

4.8 Related Work

Denning, in his 25-year retrospective [Den96], states that virtual memory is one of the "engineering triumphs" of computer science. It has helped solving problems in "storage allocation, protection of information, sharing and reuse of objects, and linking of program components". This is true even for systems that do not swap. Although Denning states that virtual memory should not slow down overall performance by more than ten percent, the original intention of the designers of the Atlas computer, the first system to implement virtual memory at the end of the 1950s, was to increase run-time efficiency (more accurately: resource utilization) by "overlapping" input, output, and computation [KHPS61].

While we concentrated on correctness concerns, early virtual memory research was dominated by runtime analyses (cf. the survey [Smi78]) in particular with respect to page replacement algorithms. The classical study in this area is [Bel66]. In his early overview article [Den70], Denning characterizes the phenomenon known as "thrashing" that may cause constant paging operations for user memory pages if the main memory is over-committed, i.e. the memory size of all running tasks exceeds the size of the main memory. Denning observes that in such a condition computation is immensely slow and it is more worthwhile (in terms of efficiency / execution speed) to suspend the execution of certain tasks altogether; processes make good progress only if their *working set*, the set of recently used pages, is available in main memory [Den67, Den80]. A concrete page replacement policy and its variants are considered in Chapter 7.

It is somewhat hard to track down what features or tricks related to virtual memory management were conceived when. We mentioned already that the earliest reference we found for copy-on-write was [BBMT72]. In addition to using it for task forks, the same mechanism may also be used for interprocess communication or message passing. Such techniques were first used in the Accent operating system [RR81] but, reportedly, also in the german EUMEL operating system at the end of the 1970s (as described in [LBB$^+$91]). Memory-mapped I/O that allows accessing files as part of the memory has been implicitly used in the earliest single address space systems (with logical addresses identifying every information), such as the Atlas system itself [KHPS61]. The same concept was used in Multics; in [BCD72], the authors notice that memory-mapped I/O also simplifies the I/O programming model.

A feature that is currently not covered by the material presented in this chapter is *asynchronous paging*. It is so commonly used that we refrain from giving a special reference for it. In real computer systems, page fault handlers (i.e. the supervisor) do not perform synchronous, blocking access of the swap memory device. Rather, a page fault might be handled in several phases (typically a swap-out, a swap-in, and a completion phase). All but the last phase end with the issuing of an I/O request and all but the first phase start with processing the result of the previous issued I/O request. Execution of the first phase is triggered by a page fault and the page fault is handled completely after the termination of the last phase, for which then the presented invariant on supervisor semantics must be established. The phases must not follow back-to-back and may be interleaved with the execution of a task for which no page fault is currently being handled. We do not cover this sort of paging because we do not have an overall I/O model; if that is provided, the correctness criteria (in particular for liveness) may be easily extended.

Another, less-used feature we did not model is that of external pagers / page fault handlers. The Mach microkernel introduced the external memory manager (XMM)

interface, which allows user processes to perform the actions of the page fault handler [RTY+87, YTR+87]. This was intended to keep the (micro-) kernel lean and let the user implement the paging policy he deems most beneficial for his applications. The correctness criteria for external pagers must be extended with respect to an in-core page fault handling mechanism as the supervisor we presented. External pagers (in fact any pagers running with address translation) must not cause a page fault while they handle a page fault, because that would result in a deadlock. Also, there are some system-critical liveness issues may arise with external pagers; therefore, in addition to the user pagers, a trusted default memory manager, the 'pager of last resort', (that may still be an external pager itself [GD91]) must be present in the system to ensure system integrity. Modern microkernel like L4 [Lie95] have an external pager interface that allows nesting of page faults, i.e. let the page faults caused by an external pager be handled by a different external pager. The benefit of using nested pagers other than for virtualization purposes is unclear and research literature does not present a (non-trivial) system making use of that feature. For virtualization purposes, however, deeply nested pagers with a software interface are not used because of the incurred performance delays; modern, fast virtualization layers (virtual machine monitors) directly use address translation mechanisms provided by the hardware [BDF+03].

At the end of Chapter 7 we discuss more aspects and related work in relation to a concrete architecture and concrete implementation of a page fault handler.

Chapter 5

VAMP with Virtual Memory Support

Contents

In the following chapters we describe how to extend the VAMP (verified architecture microprocessor) to support virtual memory for single- and multiprocessors. We cover both the hardware and software side down to the implementation level. Doing so, we present a complete example of a correct virtual memory implementation which fits into the framework developed in the previous chapter and hence shows its applicability for concrete systems.

Implementation choices of the system presented have been guided by simplicity, correctness concerns, and ease of presentation rather than by runtime and memory efficiency. There is a plethora of optimization options which may only be justified by

extensive benchmarking, an area of research in itself. Standard benchmarks for typical OS workloads (i.e. concurrently running and communicating programs) are not available and operating system writers report on the difficulty to properly check "optimizations" of the virtual memory engine [Gor, Gor04b].

In this chapter we describe a VAMP single-processor architecture with virtual memory support. In Section 5.1 we introduce the VAMP architecture extended with a simple address translation mechanism. We focus on memory operations and the details of the exception handling mechanism. In Section 5.2 we describe the current VAMP implementation and its extension with two memory management units to support address translation. Section 5.3 shows how to prove the implementation correct. We will see that with prefetching we must already guarantee a property similar to translation persistence for multiprocessors. We conclude with extensions and optimizations of the address translation mechanism in Section 5.4.

Parts of this chapter are a reformulation and extension of joint work with Wolfgang Paul [HP03] in the formalism of Chapters 2 to 4.

5.1 Architecture

We provide an introduction to the VAMP architecture. The VAMP is a DLX-like, i.e. a typical RISC load / store architecture (cf. [Pat85, HP96]). It supports fixed-point arithmetic, floating-point arithmetic and jumps and branches with one delay slot. Memory operations have access width 1, 2, 4, or 8 bytes and all addresses must be aligned to their access width. Regular instruction execution may be interrupted by calls to a handler due to internal or external conditions[1] and later resumed or skipped. We will use interrupts to call the page fault handler.

In Section 5.1.1, we describe the register set of the VAMP and its instructions. In Section 5.1.2, we define the memory operations with which instructions are fetched and memory instructions are performed. In Section 5.1.3, we define the exception handling mechanism introducing an alternative semantics for all instructions. Finally, in Section 5.1.4 we point out how an implementation detail (prefetching) restricts the class of programs to be run on the architecture.

5.1.1 Instruction Set Architecture

Registers

We briefly describe the registers of the VAMP. They are arranged in three register files:

- There are 32 general-purpose registers (GPRs), each 32-bits wide. The registers are denoted $R[0]$ through $R[31]$. Register 0 always reads as zero and cannot be written to.[2] The majority of the VAMP instructions operates on general-purpose registers.

- For floating point operations, there are 32 floating-point registers (FPRs). They are denoted $fpr[0]$ through $fpr[31]$. They can be used in singles, encoding an

[1] External interrupts are currently tied to zero for lack of an I/O architecture.

[2] Strictly speaking, register 0 always reads as 0^{32}. For more concise formulation, we identify natural numbers with their bit vector's representation and integers with their two's complement representation as long as the length of the bit vector is clear from context.

IEEE single-precision floating point number, or in even-odd pairs, encoding an IEEE double-precision floating point number.

- There are special-purpose registers for specific tasks such as interrupt handling, floating point modes / flags and also newly-introduced ones for address translation. The old registers are the status register *sr*, the exception status register *esr*, the exception cause register *eca*, the exception PC register *epc*, the exception delayed PC register *edpc*, the exception data register *edata*, the IEEE rounding mode register *rm*, the IEEE flag register *ieeef*, the floating-point condition code register *fcc*. The first six are related to interrupt handling and described in Section 5.1.3. The latter three are related to floating-point instructions and not described here (refer to [MP00, IEE85, EP97] for further documentation).

  To support address translation, we introduce four new special-purpose registers:

  - The page table origin register $pto \in \{0,1\}^{20}$ and the page table length register $ptl \in \{0,1\}^{20}$ designate a special region in main memory called the *page table*; the page table origin register points to the start of the page table, the page table length register encodes its length. The page table is used to define the implementation translation function.
  - The mode register $mode \in \{0,1\}$ consists only of a single bit which is set if the processor runs in *system mode* and cleared if it runs in *user mode*.

    In system mode, the processor requests supervisor operation from the memory, in user mode it requests user mode memory operations. Additionally, in user mode, the context control registers (i.e. the extra registers present in the implementation with respect to the user task computation model) are protected from modification and read-out. Naturally, all the registers mentioned here (including the mode flag) are part of the context control.
  - The exception mode register $emode \in \{0,1\}$ keeps a copy of the mode register during exception handling. It is used to restore the mode that was active when the exception occurred after the exception has been handled; details are given in Section 5.1.3.

  All symbolic names for special-purpose registers are aliases for the registers $spr[i]$ of a special-purpose register file 32 entries, i.e. $0 \leq i < 32$. Some of these are currently not used and behave like register $R[0]$, i.e. they are constantly zero.

  Table 5.1 lists all special-purpose registers with their indices.

The VAMP architecture features delayed branches, which are implemented using the delayed PC mechanism. Hence, there is a delayed program counter register dpc (used each round to fetch instructions) and a next program counter register pc'.

Instructions

The VAMP has (i) fixed-point arithmetic, comparison, bit-wise logical and shift instructions, (ii) floating-point addition, subtraction, division, multiplication, comparison, and conversion instructions, (iii) register-indexed load / store operations, (iv) unconditional and conditional relative or computed control-flow instructions, and (v) several special instruction, for example, related to interrupt handling. Most instructions have two source operands and one destination operand; the register file that these operands are taken from is determined by the instruction. Single-precision floating-point and general-purpose operands have a width of 32 bits, only double-precision floating-point operands have a width of 64 bits.

Index	Alias	Description
0	*sr*	Status Register
1	*esr*	Exception Status Register
2	*eca*	Exception Cause Register
3	*epc*	Exception Program Counter
4	*edpc*	Exception Delayed Program Counter
5	*edata*	Exception Data
6	*rm*	Rounding Mode
7	*ieeef*	IEEE flags
8	*fcc*	Floating Point Condition Code
9	*pto*	Page Table Origin
10	*ptl*	Page Table Length
11	*emode*	Exception Mode
16	*mode*	Mode

Table 5.1 Indices and Aliases of the Special-Purpose Registers. The registers $SPR[i]$ with $i \in \{12,\ldots,15,17,\ldots 31\}$ are currently undefined, i.e. behave like $R[0]$.

We take a closer look at the handling of control-flow changes and program counters in the VAMP and at its load- and store-operations. For the other parts of the instruction set architecture we refer the reader to [HP96, MP00].

Control Flow. The VAMP architecture has one *delay slot*: all control-flow changes (other than being interrupted or returning from an interrupt via the special `rfe` instruction), take effect only after the execution of another (in-line) instruction. The position of this subsequent instruction is called delay slot; delay slots should not be filled with control-flow instructions.[3] Formally, this is modeled by two program counter registers, dpc and pc'. The former is used as an address to fetch instructions, the latter is used as a target register for control-flow instructions. Both are 32 bits wide although the lower two bits need to be zero for instructions to be aligned. In each round the register pc' is copied to the register dpc before modification. If no control flow change occurs, the register pc' is incremented by four. For an absolute control-flow change the target is directly written to pc'; for a relative control-flow the branch offset is added to pc'. To illustrate this, consider the following example of an endless loop:

```
beqz R[0], -4
add R[0], R[0], R[0]
```

The first instruction is a branch which checks for the general-purpose register $R[0]$ to be zero (a condition which is always true) and specifies a jump offset of -4. As was noted, this offset is added to the location of the *next* instruction. Therefore, the effect of the branch instruction is to jump-back to its own location *after executing its delay slot instruction*. The delay slot is filled with a fixed-point add instruction which stores the sum of $R[0]$ and $R[0]$ in $R[0]$, i.e. it does nothing.

Although delay slots are usually 'introduced' only in the implementation to help reducing the fetch latency, a clean specification requires them to be made explicit in the instruction set architecture, as has been done in the VAMP. However, as the number of delay slots often varies with the implementation, it is desirable to have a higher-level

[3] If they are, delay slots could be located at non-in-line instructions.

specification, e.g. an assembler language based on the VAMP ISA, which hides delay slots. The assembler may then fill delay slots as required for the target architecture; the generated code simulates the assembler code (but not step-by-step). Some assemblers for the MIPS architecture follow this approach [KH92a, Lau].

Memory Access. The VAMP is a load-store architecture, hence there are only a few instructions operating on main memory. We have (i) implicit 32-bit fetches which specify the instruction currently to be executed,[4] (ii) 8-bit, 16-bit, and 32-bit general-purpose register loads and stores, and (iii) 32-bit and 64-bit floating-point register loads and stores. While instructions are addressed directly as was just described, the other operations use register-indexed addressing, i.e. they supply an 'effective address' which is the sum of a general-purpose register and an immediate constant encoded in the instruction word. The effective address $a \in \{0,\ldots,2^{32}-1\}$ has to be aligned with respect to the access width $d \in \{1,2,4,8\}$, i.e. d must divide a. Hence, byte operations are always aligned, half-word operations are aligned iff the effective address is even, word and double-word operations are aligned iff the effective address is divisible by four and eight respectively.

The operations can be carried out over a memory interface which supports double-word read operations and 15 write operations (for all combinations of aligned accesses up to double-word size).[5] The memory operation identifiers are encoded in bit strings, we have

$$Mop \subseteq \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^8 . \tag{5.1}$$

Let $(t, mr, mw, mbw) \in Mop$. The translation flag t distinguishes between translated and untranslated operations; the latter are used if the processor runs in system mode and the former are used if the processor runs in user mode. The flag *mr* indicates a read operation, the flag *mw* indicates a write operation. Both are mutually exclusive. For write operations, the bit vector *mbw* indicates which bytes are to be written in a certain double word. Clearly, not all values of *mbw* are valid with respect to the access width and access address. Table 5.2 lists the valid combinations; invalid combinations are also called *misaligned*.

The data input consists of a double word address $addr \in \{0,\ldots,2^{29}-1\}$ and (for write operations) of write data $din \in \mathbb{B}^{64}$. Additionally, for translated operations, the page table origin $pto \in \mathbb{B}^{20}$ and the page table length $ptl \in \mathbb{B}^{20}$ are needed as inputs for the address translation. The data output is an exception flag $excp \in \mathbb{B}$ and a double-word $dout \in \mathbb{B}^{64}$. So, the set of data inputs Din' and data outputs $Dout'$ are defined as follows:[6]

$$(addr, din, pto, ptl) \in Din' = \{0,\ldots,2^{29}-1\} \times \mathbb{B}^{64} \times \mathbb{B}^{20} \times \mathbb{B}^{20} \tag{5.2}$$

$$(excp, dout) \in Dout' = \mathbb{B} \times \mathbb{B}^{64} \tag{5.3}$$

The interface wires just described are listed in Table 5.3.

[4] A memory-decoupled architecture requires splitting up instructions into phases, each requesting up to a single memory operation only. This was already explained in Chapter 2. Here, however, we will delay this step until it becomes unavoidable, i.e. we introduce a memory-decoupled architecture Chapter 6 on the multiprocessor VAMP.

[5] Memory interface is a loose term here; for single-processor non-decoupled architectures the choice of the interface is mostly an implementation detail. For the memory-decoupled architecture, the memory interface is a strict component of the specification and controls the connection between the processor(s) and the memory.

[6] We denote these sets with Din' and $Dout'$ instead of *Din* and *Dout* to avoid a name clash with their similarly named components *din* and *dout*.

d	a mod 8	$mbw[7:0]$
1	0	1
	1	1
	2	1
	3	1
	4	1
	5	1
	6	1
	7	1
2	0	11
	2	11
	4	11
	6	11
4	0	1111
	4	1111
8	0	11111111

Table 5.2 Memory Byte Write Signals for Aligned Memory Operations. Zero bits are not shown. All other combinations of access widths d with rests of a modulo 8 are *misaligned*.

	Name	Description
▷	t	translation flag
	mr	memory read
	mw	memory write
	$mbw[7:0]$	memory byte write
	$addr[28:0]$	double-word address
	$din[63:0]$	data input (wrt. interface operation)
	$pto[19:0]$	page table origin
	$ptl[19:0]$	page table length
◁	ack	acknowledgment (= inverted memory busy)
	$excp$	exception flag
	$dout[63:0]$	data output (wrt. interface operation)

Table 5.3 Processor Interface Observations

5.1.2 Memory Operations

The memory consists of two components: the main memory and the swap memory. The main memory is (up to) $4G$ bytes large and is accessed in double words. The swap memory is (up to) $4G$ pages large, where each page contains $4K$ bytes.

We denote the main memory configuration by $mm : [\{0,\ldots,2^{29}-1\} \to \mathbb{B}^{64}]$ and the swap memory configuration by $sm : [\{0,\ldots,2^{27+9}-1\} \to \mathbb{B}^{64}]$. We introduce a special notation to model how the machine accesses the memory, i.e. with byte addressing and in (aligned) tuples of $d \in \{1,2,4,8\}$ bytes. Let $a \in \{0,\ldots,2^{32}-1\}$ and assume a is divisible by d, i.e. $d \mid a$. Then, square-bracketed $mm_d[a]$ denotes the aligned tuple of d

bytes with base address d. We define inductively:

$$mm_d[a] = \begin{cases} mm(a/8) & \text{if } d = 8 \\ mm_{2\cdot d}[\lfloor a/2 \rfloor \cdot 2][16\cdot d-1:8\cdot d] & \text{if } d < 8 \text{ and } (a \bmod 2\cdot d) = d \\ mm_{2\cdot d}[\lfloor a/2 \rfloor \cdot 2][8\cdot d-1:0] & \text{if } d < 8 \text{ and } (a \bmod 2\cdot d) = 0 \end{cases} \quad (5.4)$$

We see that $mm_d[a] \in \mathbb{B}^{8\cdot d}$. Without further notice, we use for integer arguments or expressions e to $mm(e)$ and $mm_d[e]$ always the rest modulo 2^{29} and 2^{32}.

For $d = 8\cdot 2^k$ and $d \mid a \in \{0,\ldots,2^{32}-1\}$, we define $mm_d[a]$ as the tuple

$$(mm(a/8+2^k-1),\ldots,mm(a/8+1),mm(a/8)) \,. \quad (5.5)$$

Hence, we have $mm_d[a] = (mm_1[a+d-1], mm_1[a+d-2],\ldots,mm_1[a])$.

Now we define the semantics of the read and write operations on the main memory. To this end, we define the implementation translation function, and with its help user and system mode memory operations. Swap memory operations are not defined for lack of an I/O architecture.

Implementation Translation Function

The decode implementation translation function dec_{itr} as introduced in Section 4.4.1 maps a main memory configuration mm, a task identifier tid, a virtual address $va \in \{0,\ldots,2^{29}-1\}$, and a memory write flag mw (identifying a memory operation) to an exception flag $excp$ and a main memory address ma:

$$dec_{itr}(mm,tid,va,mw) := (excp,ma) \quad (5.6)$$

It was already remarked in that section, that dec_{itr} is more than just an architectural definition; rather, it also models parts of the memory management data structures of the operating system. Accordingly, we decompose dec_{itr} into two parts.

The first part is a function of the operating system. Here, it designates for each (active) task a *page table*. Thus, it takes a task identifier $tid \in \mathbb{N}$ and returns a page table origin $pto \in \mathbb{B}^{20}$ and a page table length $ptl \in \mathbb{B}^{20}$. We denote this function by dec_{itr1} and will use it uninterpreted for now; a concrete example will be given in Chapter 7. We will also see there, that the results of dec_{itr1} for the currently running task $ctid \in \mathbb{N}$ are held in the special-purpose register pto and ptl.

The second part called dec_{itr2} models the translation performed by the hardware and will be defined here. It takes a main memory configuration, a page table origin, a page table length, a virtual address, and a memory write flag and returns an exception flag and a main memory address, i.e.

$$dec_{itr2}(mm,pto,ptl,va,mw) = (excp,ma) \,. \quad (5.7)$$

We define $excp$ and ma in terms of the input arguments. As was mentioned, the pto and ptl arguments specify a table in main memory that is called the *page table*. The page table maps indices $x \in \{0,\ldots,\langle ptl\rangle\}$ to page table entries (PTEs) which are 32-bits wide. Let $(px,off) \in \{0,\ldots,2^{20}-1\} \times \{0,\ldots,2^9-1\}$ be the decomposition of the virtual address va into the page index px and the (double word) offset off, i.e. $va = px\cdot 2^9 + off$. Let pte be the page table entry for va, which is defined as

$$pte := mm_4[\langle pto\rangle \cdot 2^{12} + px\cdot 4]) \in \mathbb{B}^{32} \,. \quad (5.8)$$

Clearly, the address of the page table entry is a multiple of four qualifying it as a valid word address. The page table entry has three fields which are interpreted by the hardware:

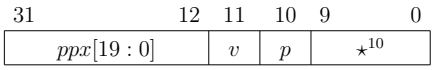

Figure 5.1 Page Table Entry. The ten lower bits are free for software use (cf. Chapter 7).

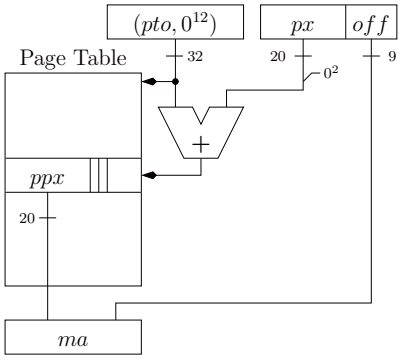

Figure 5.2 Address Translation for the Virtual Address $va = \langle px, off \rangle$

- The physical page index $ppx = pte[31:12]$. Under certain conditions (no exception is caused for the memory access), the physical page index indicates the page in main memory in which the contents of *va* are stored.
- The valid bit $v = pte[11]$. It is set iff the page is valid (i.e. it is currently stored in main memory and available at least for read access).
- The protection bit $p = pte[10]$. It is set if the page can be (currently) written to (provided it is valid additionally).

Figure 5.1 depicts a page table entry.

The exception result of the implementation translation function is set iff any of the following three exception conditions is met: (i) the page index is greater than the page table length,[7] (ii) the page is invalid, (iii) the page is protected and the requested operation is a write. All of these conditions are general called *translation exceptions*, the first condition is also called *length exception*, and the latter two conditions are also called *page table entry exceptions*. Hence, we have

$$excp = (px > \langle ptl \rangle) \vee \neg v \vee (p \wedge mw) \; . \tag{5.9}$$

The memory address is set to zero if there is an exception. Otherwise, it is computed from the physical page index stored and the offset of the input address. We set

$$ma = \begin{cases} 0 & \text{if } excp \; , \\ \langle ppx \rangle \cdot 2^9 + off & \text{otherwise.} \end{cases} \tag{5.10}$$

Figure 5.2 sketches the computation of the memory address *ma*. This completes the definition of the implementation translation function. As in Chapter 4, we say that

[7] So, actually, 'page table length' is a misnomer—though a common one—as its value is one smaller than it should be.

the memory operation (tid, va, mop) is *attached* with respect to the memory configuration mm if the implementation translation does not signal an exception on inputs (mm, tid, va, mop). We abbreviate this fact by $att(mm, tid, va, mop)$.

The address translation mechanism defined here is a standard, table-based (single-level) address translation mechanism. It may be generalized by decomposing the address into more than two fields and performing more table lookups. All but the last field are taken as table indices; each table-entry lookup specifies where to find the next table to look at. The target address is formed by adding up (binary interpretations) of portions of the final table-entry with the last field of the input address (the offset). Multi-level translation allows for a space-efficient and flexible organization of the table space. However, without optimization, it is of course intrinsically slower than single-level translation. More details will be given in the Sections 5.4.1 and 5.4.2.

Semantics

Take any memory operation identifier (t, mr, mw, mbw) and memory operation inputs $(addr, din, pto, ptl)$. Let mm be a main memory configuration and let mm' denote its successor configuration. Let $(excp, dout)$ be the output of the memory operation.

Let $(excp, ma) = dec_{itr2}(mm, pto, ptl, addr, mw)$. If $t \wedge excp$, the result of the output of the memory operation is $(excp, dout) = (1, 0^{64})$ and the memory configuration does not change, i.e. $mm' = mm$.

Otherwise let pa denote the input address for untranslated operations and the translated main memory address otherwise, i.e.

$$pa = \begin{cases} addr & \text{if } t = 0 \\ ma & \text{if } t = 1\ . \end{cases} \tag{5.11}$$

For read operations ($mw = 0$), the data output is the double word at the address pa. We set $(excp, dout) = (0, mm(pa))$.

For write operations, we replace the bytes indicated by the mbw bus with the data input. We define the new value of byte $i \in \{0, \ldots, 7\}$ as follows:

$$mm'(pa)[i \cdot 8 + 7 : i \cdot 8] = \begin{cases} din[i \cdot 8 + 7 : i \cdot 8] & \text{if } mbw[i] \\ mm(pa)[i \cdot 8 + 7 : i \cdot 8] & \text{otherwise} \end{cases} \tag{5.12}$$

All other memory locations remain unchanged; the data output is zero, i.e. $dout = 0^{64}$.

After (exception-free) translation (i.e. if $excp = 0$ and pa is known), the memory operations do not depend on the translation flag t, the page table origin pto and the page table length ptl anymore. Additionally, the exception is always known to be zero, and as such, needs not to be returned by the memory. This simplified interface corresponds to the bridge-2 to memory interface from our generic VMM (cf. Section 4.1). In the VAMP implementation, this interface is known as the *cache interface*. Table 5.4 lists its interface wires.

5.1.3 Exceptions

As we have seen in Chapter 4, our processor must be able to handle translation exceptions. To repeat a page-faulting memory operation it is necessary to recreate the exact user processor configuration before the page fault.

The VAMP exception handling mechanism is much more complex. It supports up to 32 distinct interrupts. These interrupts can be classified according to three sources:

	Name	Description
▷	*mr*	memory read
	mw	memory write
	$mbw[7:0]$	memory byte write
	$addr[28:0]$	double-word address
	$din[63:0]$	data input
◁	*ack*	acknowledgment (= inverted memory busy)
	$dout[63:0]$	data output

Table 5.4 Cache Interface Observations

- *Internal interrupts* are set during instruction execution according to conditions specified by the instruction set architecture. Typical examples of internal interrupts are arithmetical exceptions (e.g. divisions by zero) and traps, which are triggered directly by `trap` instructions and used to implement system calls.

- *External interrupts* are set from external sources and are thus indeterministic inputs to the VAMP at our level of modeling. External interrupts are the reset interrupt and any I/O interrupt generated from devices or timers. As will be explained below, in general, we assume that no external interrupts are generated.

- *Page fault interrupts* are related to the virtual memory mechanism and generated by the memory for non-attached virtual memory operations. We distinguish two different page faults: *page fault on fetch* occurs for a non-attached instruction fetch; *page fault on load/store* occurs for a non-attached load / store operation.

  In contrast to the two preceding groups, translation exceptions are caused externally (with respect to the processor) yet deterministically: they are computed by the decode implementation translation function dec_{itr2} which takes the memory configuration as an input.

Each interrupt is associated with a number called priority. Its intended meaning is that interrupts with a low priority take precedence over and may interrupt handling of interrupts with a higher priority. Generally, if interrupt handlers may interrupt each other, we speak of *nested interrupts*. For the VAMP architecture, interrupt nesting is mostly controlled in software (i.e. the interrupt handler), the VAMP architecture merely provides facilities to ignore certain interrupts under software control. Any such interrupt is called maskable; usually, the interrupt handler for priority i will disable all interrupts with priority $j \geq i$.

Each interrupt is associated with a *resume type* which specifies whether and whence the program returns after the interrupt has been handled. There are three different resume types: *repeat* exceptions will re-execute the instruction where the exception was detected, *continue* exceptions continue the program after the instruction where the exception was detected, *abort* exceptions have serious causes (e.g. reset) and do not return to (or near to) the point of interruption.

Table 5.5 lists all interrupt signals with abbreviations, their priority, their resume type, their maskability, and whether they are considered external or not. With respect to the original VAMP design most interrupts have the same meaning. The page fault interrupts pff and $pfls$ which were previously tied to zero are now caused on translation exception. The illegal interrupt *ill* is extended. In user mode, all instructions that

Interrupt	Symbol	Priority	Resume	Maskable	External
Reset	reset	0	abort	no	yes
Illegal instruction	ill	1	abort	no	no
Misaligned access	mal	2	abort	no	no
Page fault fetch	pff	3	repeat	no	no
Page fault load / store	pfls	4	repeat	no	no
Trap	trap	5	continue	no	no
FXU overflow	ovf	6	continue	yes	no
FPU overflow	fovf	7	continue	yes	no
FPU underflow	funf	8	continue	yes	no
FPU inexact result	finx	9	continue	yes	no
FPU divide by zero	fdbz	10	continue	yes	no
FPU invalid operation	finv	11	continue	yes	no
FPU unimplemented	ufop	12	continue	no	no
External I/O	ex_i	12+i	continue	yes	yes

Table 5.5 VAMP Interrupts

would access interrupt- or translation-related registers are considered illegal (i.e. they are not part of the user's computation model). Hence, it is illegal in user mode to try to execute an `rfe` instruction, a `movs2i` instruction with the source register or a `movi2s` instruction with the destination register being interrupt- or translation-related.

From now on, we are primarily interested in translation exceptions. We assume that user programs generate only page fault interrupts and that system code (i.e. operating system initialization and interrupt handlers) generates no interrupts at all. Not making this restriction would have two far-reaching consequences:

- If programs generate other interrupts than page faults, there must also be a formal interface to handle these interrupts. This is not trivial. For example, modern operating systems allow user programs to provide user exception handlers. These so-called *user signal handlers* require the introduction of an abstract exception call mechanism in the user computation model. In the implementation, such user program exceptions are first received by the operating system in system mode and then delegated to the user signal handler (running in user mode).

- Allowing external interrupts only makes sense with an I/O model. A formal I/O model needs the specification of a syntax and semantics of I/O devices and their operations which are asynchronous to regular processor operation. External interrupts would be used as completion signals (acknowledgments) of such I/O operations.

Both tasks are beyond the scope of this thesis and hence provide a direction of further research. We believe that at least the framework of this thesis would be adequate for extensions in these directions.

Since it is too expensive (and inflexible) to provide a full set of save registers as it was described in Section 4.2 the exception handling mechanism is a combined hardware-software mechanism. The hardware will only save and restore a bare minimum of processor registers while the exception handler is entrusted to save and restore the other registers. Exception handlers which satisfy criteria guaranteeing correct exception handling in combination with the hardware are called *admissible*.

In the following sections we describe the four phases of handling a translation exception: detecting the exception, entering the interrupt handler, executing the interrupt handler, and leaving the interrupt handler. The first and the last step are performed with architectural support while the other steps are realized purely in software.

Interrupt Detection

The VAMP detects an interrupt, if an interrupt event line is active and not masked out. If this happens for interrupt i, we say that the VAMP *sees interrupt i*. Let ca_i denote the event line for interrupt i. The masked cause register *mca* computed by

$$mca[i] = ca[i] \wedge (sr[i] \vee (i < 6 \vee i = 12)) \tag{5.13}$$

for all $i \in \{0,\ldots,31\}$ indicates the visibility of each interrupt i. We remark that not all internal interrupts may be seen by all instructions; in particular instruction page fault and misalignment exclude the appearance of any other internal interrupt.

The signal *jisr* indicates the visibility of any interrupt. It is defined as

$$jisr = \bigvee_i mca[i] \; . \tag{5.14}$$

Since page faults exceptions are non-maskable, both a page fault on fetch and a page fault on load/store make the VAMP enter the interrupt handler in the next cycle.

Entering the Interrupt Handler

Assume that a page fault exception has been caused, i.e. $jisr = 1$ and $ca[3] \vee ca[4] = pff \vee pfls$. Entering the exception handler makes the hardware perform all the following updates simultaneously:

- The hardware jumps to the start of the interrupt service routine (currently hardwired to 0) and switches to system mode. Hence, the VAMP has a single point of entries for all different interrupts. We have:

$$dpc = 0^{32} \tag{5.15}$$
$$pc' = 0^{29}100 \tag{5.16}$$
$$mode = 0 \tag{5.17}$$

- The mode register and the program counters are saved into the exception mode and PC registers to allow later continuation of the program flow. For repeat interrupts, these are the old program counters:

$$edpc = dpc \tag{5.18}$$
$$epc = pc' \tag{5.19}$$
$$emode = mode \tag{5.20}$$

- The hardware clears the status register, to disable all maskable interrupts, and saves its old value into the exception status register for later restoration:

$$sr = 0^{32} \tag{5.21}$$
$$esr = sr \tag{5.22}$$

- Since the VAMP uses only a single interrupt vector, which is the starting point for all interrupt service routines, the hardware must allow the interrupt handler to query for the cause of the exception. Therefore, the hardware saves the masked cause register (which was defined in Equation 5.13) into the exception cause register:

$$eca = mca \tag{5.23}$$

- The exception data register holds input data for the exception service routine. In case for page faults on load/store, the exception data register holds the (effective) address of the faulting memory operation:

$$edata = ea \tag{5.24}$$

Executing the Interrupt Handler

The execution of a typical (i.e. non-aborting) interrupt handler can be decomposed into three parts: a common entry code for all interrupts, an interrupt-specific part and a common exit code.

We describe the common part at the start of each interrupt handler. As was said in the last section, any interrupt makes the VAMP start execution at address 0 in system mode with all maskable interrupts disabled. The interrupt handler pushes the contents of certain special registers (epc, $edpc$, $emode$, eca and esr) on the so-called *interrupt stack*. Then, by a find-first-one computation in software it computes the *interrupt level*, the minimal j such that $eca[j] = 1$. All interrupts with a priority less than j may be re-enabled by storing an appropriate bit mask in the status register sr. It is crucial that any register used in the above computation is saved to a special memory location (e.g. the interrupt stack) before being overwritten. At the end of the interrupt handler all these temporary registers are restored. Now the handler calls the interrupt-specific part for interrupt j. The start address of this routine is stored in the interrupt-vectors table in memory. This completes the first part of the interrupt handler.

The interrupt handler j for a non-aborting interrupt must be terminating and may only change temporary registers. Hence, if more registers are needed they have to be saved and restored separately.

The common end part restores the contents of the temporary and special-purpose registers from the interrupt stack. This is done with all maskable interrupts disabled. Then it issues the `rfe` (return from exception) instruction described in the next section.

Leaving the Interrupt Handler

The interrupt handler is left using the special `rfe` (return from exception) instruction. This instruction restores the program counters, the mode and the status registers from the corresponding exception registers.

$$sr = esr \tag{5.25}$$
$$pc' = epc \tag{5.26}$$
$$mode = emode \tag{5.27}$$
$$sr = esr \tag{5.28}$$

5.1.4 Self-Modification

By writing in their own instruction streams, programs may modify their own code. This is called *self-modification*. For reasons given below in Section 5.2 not all self-modifying programs can be executed correctly on the hardware. Therefore, it is necessary to restrict the programs that modify their own code which may be run on the architecture. Programs to be executed on the VAMP architecture must satisfy the following rule: between any write to and fetch from some memory location either (i) a special synchronization instruction is executed,[8] (ii) a return-from-exception instruction is executed, or (iii) an interrupt is detected. This rule is called the *synced code predicate*. It is slightly extended with respect to the version that [Bey05] presents; this will be of use later (cf. Section 7.3).

We define some auxiliary predicates. Let the predicate $store(i,ma)$ indicate that logical instruction I_i writes into the double word at address $ma \in \{0,\ldots,2^{29}-1\}$. There are two possible cases for $store(i,ma)$ to be true: either the main memory ma is equal to the effective address in untranslated mode or the translated effective address is equal to the main memory address ma (and the translation of the effective address was exception-free, as well as for the delayed program counter). Let the predicate $fetch(j,ma)$ indicate that the fetch of instruction I_j depends on the double word at main memory address ma. Again we have two cases: in the untranslated case, it is true if the main memory address ma equals the delayed program counter dpc; in the translated case, it is true (i) if the main memory address ma equals the address of the page table entry used for the translation of the delayed program counter dpc or (ii) if it equals the translated main memory address. Let $sync(k)$ indicate that a synchronization instruction has been executed in time k, let $rfe(k)$ that an `rfe` instruction has beep executed in time k, and let $jisr(k)$ indicate the detection of an interrupt in time k.

We define now formally the synced code predicate. Consider $i,\ j \in \mathbb{N}$ with $i < j$ and $ma \in \{0,\ldots,2^{29}-1\}$. Assume that $store(i,ma)$ and $fetch(j,ma)$ hold both. Then, we require the existence of a time $k \in \mathbb{N}$ with $i < k < j$ such that $sync(k)$, $rfe(k)$, or $jisr(k)$ holds. Overall, we have

$$\forall i < j, ma : store(i,ma) \wedge fetch(j,ma) \Rightarrow \exists i < k < j : sync(k) \vee rfe(k) \vee jisr(k)\ . \quad (5.29)$$

This criterion is low-level and ignores the structure that is present in the real computations of user tasks running in an operating system environment. It is expected that in such an environment the synced code predicate may be obtained by composing similar yet local properties of the user tasks and the operating system code.

Note, that there are two flavors of self-modification, which are not formally distinguished:

- Code loading and linking (to start programs from the hard disk)
- Self-modifying code (to reduce code size and / or to increase execution speed)

The latter form of self-modification is deprecated and regarded as bad programming style. Guaranteeing the synced code predicate makes such code fragments considerably more expensive and thus discourages the programmer from using such techniques.

[8] A synchronization instruction empties the pipeline of the processor. For the current VAMP implementation, the instruction `movs2i` with source operand $IEEEf$ is synchronizing; it reads out the $IEEEf$ special-purpose register and thus computes the IEEE 754 floating-point exceptions accumulated up to the last preceding instruction.

Code loading and linking, on the other hand, is a necessity to implement in an operating system. The performance and code size impact on inserting a single synchronization instruction after loading a whole code segment is negligible.

5.2 Implementation

Both the VAMP specification and the VAMP implementation are written in the specification language of the theorem prover PVS [OSR92] allowing a mechanical correctness proof to be conducted in PVS. An unverified tool is used to translate the implementation source into the hardware description language Verilog which in turn can be synthesized for a Xilinx FPGA hosted on a PCI board. Hence, the VAMP is both a mechanically-verified and synthesizable architecture, cf. [BJK$^+$03].

The next section gives an overview of the VAMP processor. Then, we present an MMU design, which performs memory operations over the cache interface. In the modified VAMP design, two MMUs are used, one for instruction memory and the other for data memory access. We show how they are integrated into the existing memory functional unit. Finally, we sketch how the changes related to interrupt handling are implemented.

5.2.1 Overview

Figure 5.3 shows an overview of the VAMP top-level datapaths. The VAMP implementations has five pipeline stages: the first and second stage implement instruction fetch (using the delayed PC mechanism) and instruction decode; stages three to five implement a single-scalar Tomasulo scheduler for out-of-order execution. To support precise interrupts, instructions write their results to the reorder buffer from which they will be written back in-order to the register files.

In the following, an *uncompleted instruction* is an instruction which has already been fetched but has not yet finished its execution. We sketch the execution of a single instruction according to Tomasulo's algorithm:

- The instruction is fetched by the memory functional unit and written to the instruction register.
- The instruction word is decoded to determine the destination functional unit, the addresses of the source operands and the addresses of the destination operands. There is a functional unit for load / store instructions, one for fixed-point arithmetical and logical instructions and three for the various floating-point operations. The remaining instruction classes do not visit any functional unit, they are stalled until they can be issued with results directly to the ROB.

  The instruction word is sent to an input slot, called reservation station, of the destination functional unit. Additionally, for each source operand of the instruction we either supply (i) its value if no uncompleted instruction writes to it, or (ii) a unique identifier, called *tag*, of the newest instruction, which has the source operand as a destination and is still uncompleted. This information is stored in the register files and the so-called producer table.

  Additionally, the instruction itself is associated with a new (currently unused) tag which is also supplied to the reservation station. Each destination operand

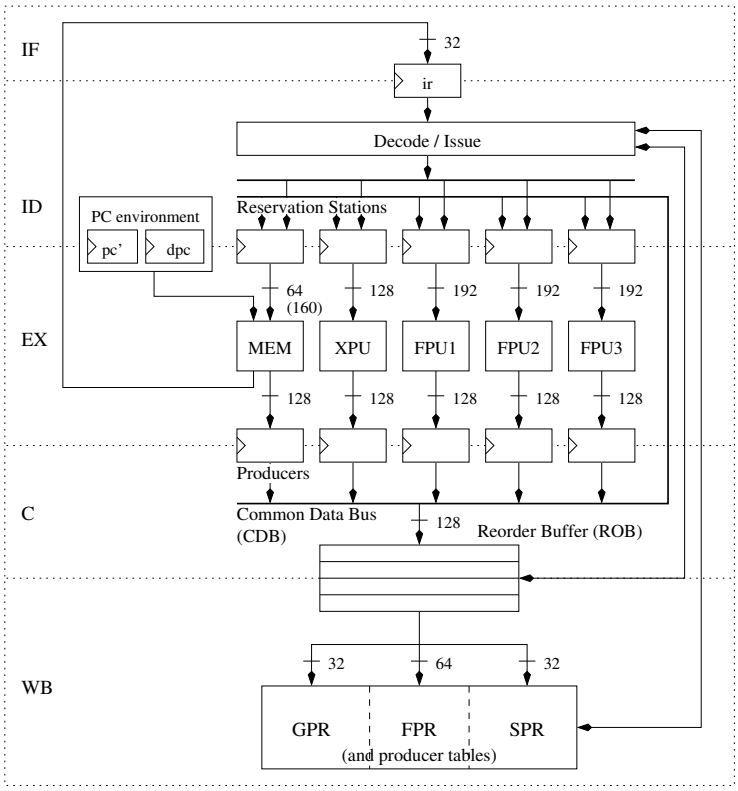

Figure 5.3 Top-Level Datapaths of the VAMP Processor Core. For the virtual memory extensions, the width of the input bus to the memory unit *MEM* (and of its reservation station) changes from 64 to 160 bits.

is marked as invalid (meaning "currently being computed") and has the instruction's tag noted in the producer table.

- To complete source operand values, each reservation station is connected to the *common data bus* (CDB) which broadcasts instruction results and their tags. Whenever a reservation station sees a matching tag on the CDB, it can obtain the value for the associated source operand. When the values of all source operands are available, the instruction can be *dispatched* to the functional unit.
- Each functional unit computes the results, i.e. the values of the destination registers, as a function of the source operands and the instruction word. For the memory functional unit this is not true; its results additionally depend on memory operations requested over the data memory interface.
- The results of each functional unit (including the tag) are buffered in its *producer*. One producer per round writes its contents to the CDB and thereby to the reorder buffer (ROB). The ROB is addressed by tags and its contents are organized as a wrap-around queue with a head pointer indicating the next entry to retire and a tail pointer indicating the next free tag. A full ROB (i.e. equality of tail and head pointer) entails a stall condition for instruction issue.

  Once an instruction completes, its ROB entry is marked as valid.

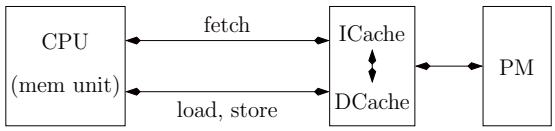

Figure 5.4 Overview of the VAMP and Memory Interfaces without Address Translation

- The ROB writes back the instruction results *in order* to the register files. The oldest entry which is at the head pointer location can be processed iff it is valid, i.e. its results are available. Part of the result of each instruction are the (internal) interrupt flags it generates. If any of these internal flags is active or if an external interrupt is observed, the machine prevents the execution of all newer instruction in the machine (by clearing all reservation stations, producers, newer ROB entries, and resetting functional units) and jumps to the interrupt service routine (by setting the program counters appropriately).

As can be seen from the description above, all memory accesses take place through the memory functional unit located in the execute stage. This unit is connected to an instruction memory port (for fetches) and a data memory port (for loads and stores). These ports implement a consistent interface to the actual main memory. This interface hides the instruction cache and data cache implementation. The arrangement is shown in Figure 5.4.

The fetch port is used to *prefetch* instructions: the instruction word for a logically later instruction is fetched before the previous instructions have been completed. However, if a program modifies its code by writing in its instruction stream, instruction prefetching may return wrong instructions as compared to the specification machine. As it was already pointed out in Section 5.1.4, the solution lies in defining a synchronization instruction which inhibits prefetching (by flushing the pipeline) and which must be used for self-modifying code.

With address translation, MMUs will be placed between the CPU and the instruction and data caches.

5.2.2 MMU Design

In this section we design a memory management unit for the VAMP. We prove the (local) correctness for this unit under the assumption that all inputs remain constant.

We remark that the VAMP with the presented MMUs performs badly: our MMU is slow by design; it adds at least two cycles to the execution of *any* memory operation. This delay is dearly paid for instruction fetches.

To optimize the design, MMUs usually have a cache of recently-used translations (the translation look-aside buffer). However, even with a TLB, the penalty for translated memory operations is typically only reduced but not nullified. Hence, further optimizations of the fetch stage would be needed to allow pipelining of the instruction fetch / the instruction MMU, e.g. by implementing a superscalar and speculative instruction fetch unit. These are not presented here.

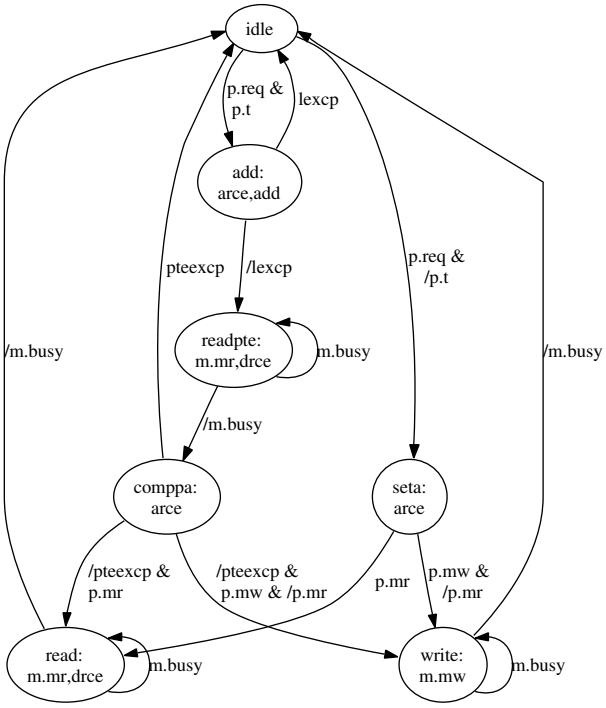

Figure 5.5 Control Automaton for the MMU. Nodes correspond to states, active signals in each state are listed after the node name. Edges correspond to state transitions and are labeled with the transition condition. The operators '&' and '/' are used for logical conjunction and negation. Signals prefixed 'p.' and 'm.' denote signals from the processor and the memory (cache) interface. We define $p.req := p.mr \vee p.mw$. Additionally, we have the Mealy control signal $p.busy := \neg(state' = idle)$ where $state'$ indicates the automaton's next state. Transitions to the *idle* state taken with an active *reset* input are not shown.

Interface

The interface of the MMU to the processor and to the cache has already been given in Section 5.1.1. We prefix the signals between the MMU and the processor with the symbol p and the signals between the cache (i.e. the memory) and the processor with the symbol m.

Control

The control automaton, drawn in Figure 5.5, has seven different states. All four request types (translated versus untranslated, read versus write) start in the *idle* state. When a processor request $p.req = p.mr \vee p.mw$ is observed, the *idle* state is left.

We describe which paths untranslated and translated requests take through the automaton:

- Untranslated requests ($\neg p.t$) make the automaton enter the set-address state *seta*. In this state the address register is set to the processor input address, which is already the physical address of the memory operation. Untranslated reads enter the read state *read*, in which 64-bit from the memory at the address given by the address register are read over the cache interface. Untranslated writes enter the write state *write*, and write the processor input data to the address stored

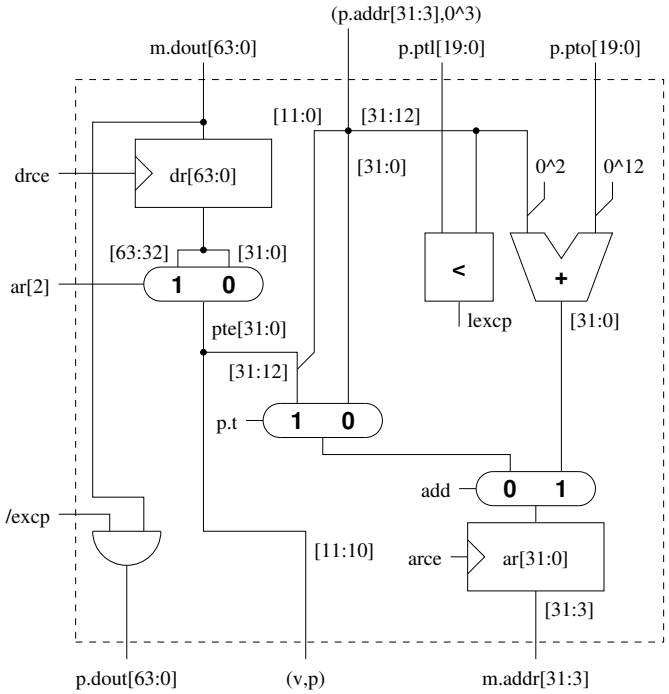

Figure 5.6 Datapaths of the MMU. Let $p.t = t = mode$, ar the address, and dr the data register.

in the address register. The memory byte-write signals are also taken from the processor interface.

- Translated request ($p.t$) make the automaton first enter the *add* state. In this state the address of the page table entry to be accessed is computed and written to the address register. Then, in the read page table entry state $readpte$, a read request from this address is issued to the cache. The result of this read request is stored in the data register. Afterwards, in the state $comppa$ we compute the physical address of the translated request *or* return to the $idle$ state in case of exceptions. The physical address is stored in the address register. Translated reads will then enter the *read* state to perform the actual read operation. Translated writes enter the $write$ state to perform the actual write operation.

The control automaton may be reset asynchronously by activating the special input *reset* which makes it unconditionally return to the $idle$ state. This input will later be activated in interrupt situation (e.g. power-up).

Datapaths

Figure 5.6 shows the datapaths of our MMU (the notation being used is described in Figure 5.7). They contain two registers:

- The upper 30 bits of the address register $ar[31:0]$ hold the address for cache memory operations. We will see that there are two possible addresses for any cache memory operations: either the address of the page table entry needed to perform a translated operations or the physical address of an exception-free translated or an untranslated operation.

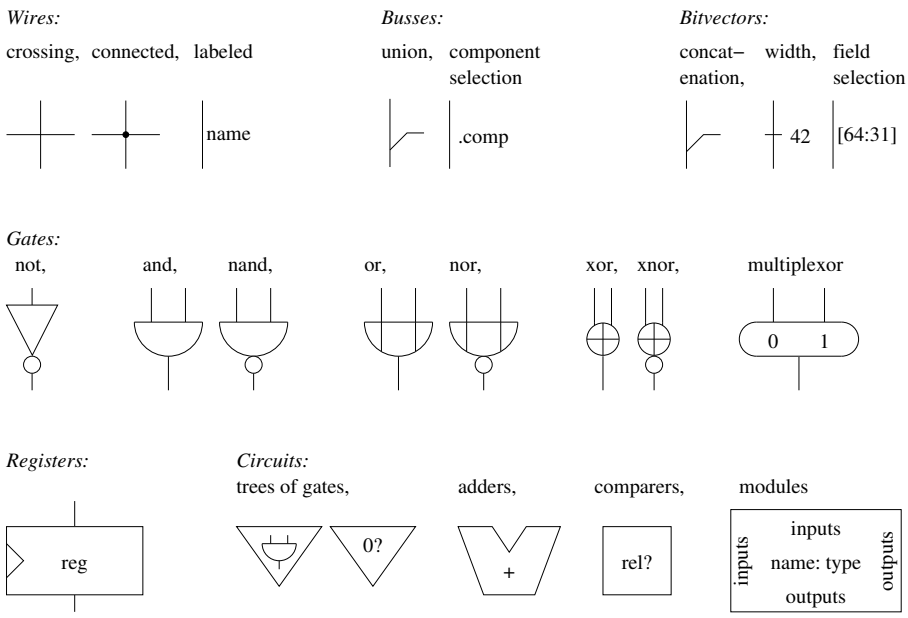

Figure 5.7 Symbols used for Schematics

- The data register $dr[63:0]$ stores the results of page table entry reads.

Apart from multiplexors, the datapaths consist of an adder to compute the page table entry address (computed from the page table origin and the page index) and a comparer to detect page table length exceptions.

Correctness

Consider the two interfaces the MMU is connected to. These interfaces are marginally modified instantiations of our general interfaces presented in Chapter 2: the requests signal of the interface is the disjunction of the read and the write signal; the acknowledgment signal is the negation of the busy signal.

For the local correctness proof of the MMU, we consider the handshake conditions at both interfaces. Some of the handshake conditions are proof assumptions while the others are proof goals:

- The processor interface is considered to be stable, while we must show that it is also live, does not over-acknowledge and fulfills its semantics.

- For the cache interface, on the other hand, we must guarantee, that the inputs provided by the MMU are stable over the time of a request. The cache then guarantees that for each request there is a cache response in finite time and according

to the semantics of the cache memory operations. Also, the cache does not over-acknowledge requests.

Note that our MMU implements the functionality of both bridges and of the translator presented in the Chapter 4. We may assume that the MMU has exclusive access to the memory; as we will see later, the outcome of every cache memory operation performed during an MMU request remains constant for the duration of the request.

We will not show all properties mentioned above. The most interesting properties are liveness and functional correctness. The liveness part follows from the liveness of the cache interface and the structure of the control automaton. As such a property could have been easily proven automatically by a model checker we omit the proof here.

To prove the operation semantics, we prove the correctness of four operations: translated read, translated write, untranslated read, and untranslated write. For the translated operations, we consider the cases with and without exceptions.

Since the proofs differ only in details, we will only show the (more complex) proofs of translated operations and in particular the translated read operation.

Assume there is a request from the processor to the MMU between the times t and t'. We denote this fact by the predicate $isreq_p(t,t')$ (cf. Chapter 2). Furthermore assume that there is not reset from t to t' and the request is for a translated operation, i.e. $p.t^t$ holds (since the symbol t is overloaded here, we introduce the convention that t is used as the translation flag only if prefixed with p; otherwise, it is used as the time index t).

Consider the result $(excp, ma)$ of the (hardware part of the) implementation translation function at the input address for the memory state at time t:

$$(excp, ma) := dec_{itr2}(mm^t, p.pto^t, p.ptl^t, \langle p.addr^t \rangle, p.mw^t) \tag{5.30}$$

We show that (i) if there is a length exception, it is signaled correctly, (ii) otherwise, the PTE is looked-up correctly, (iii) if there is a page table entry exception, it is signaled correctly, (iv) otherwise the translated read or write is performed correctly.

◀ Lemma 5.1

If there is a length exception, then at time t' the MMU acknowledges with an exception:

$$\langle p.addr^t[31:12] \rangle > \langle p.ptl^t \rangle \Rightarrow p.excp^{t'} = excp = 1 \tag{5.31}$$

Additionally, we have $t' = t+1$.

PROOF

Because of input stableness, all inputs in time $t+1$ are the same as in time t. At time $t+1$, the control automaton must be in state *add*, since it was in state *idle* in time t and a translated request had been started. By assumption, we have a length exception for the input address and therefore $excp = 1$. Therefore we also have

$$\begin{aligned} \langle p.addr^t[31:12] \rangle > \langle p.ptl^t \rangle &\Leftrightarrow \langle p.addr^{t+1}[31:12] \rangle > \langle p.ptl^{t+1} \rangle \\ &\Leftrightarrow lexcp^{t+1} \\ &\Rightarrow p.excp^{t+1} \ . \end{aligned}$$

Hence, we know that we take the transition $add \rightarrow idle$ and signal 'not busy' in the same cycle. Therefore

$$t' = t+1 \ . \tag{5.32}$$

This proves the claim.

For the following lemmas, we always assume that there is no length exception, i.e. $\langle p.addr^t[31:12]\rangle \leq \langle p.ptl^t\rangle$. We let

$$ptea := \langle p.pto^t\rangle \cdot 2^{12} + \langle p.addr^t[31:12]\rangle \cdot 4 \bmod 2^{32} \tag{5.33}$$

abbreviate the address of the page table entry.

Lemma 5.2 ► *If there is no length exception, then there exists a time $t_2 > t_1$ such that at time t_2+1, the $pte[31:0]$ bus holds the specified page table entry*

$$pte[31:0]^{t_2+1} = mm_4^t[ptea] \tag{5.34}$$

if the memory does not change its contents, i.e. $mm^{t_2}[ptea] = mm^t[ptea]$. Additionally, the control automaton is in state comppa state at time t_2+1.

PROOF If there is no length exception we enter the state *readpte* in the cycle $t_1 = t+2$. Therefore this cycle is also the starting cycle of a cache read operation:

$$reqstart_c(t_1) \wedge m.mr^{t_1} \tag{5.35}$$

By the cache liveness assumption, there exists a time $t_2 \geq t_1$ in which the cache responds. After acknowledgment we leave the *readpte* state and enter the *comppa* in cycle t_2+1:

$$state^{t_2+1} = comppa \tag{5.36}$$

In this cycle the address register ar (still) holds the encoding of the page table entry address $ptea$ and the data register dr holds the double-word $mm_8^{t_2}[\lfloor ptea/2\rfloor \cdot 2]$. Technically, these results can be proven by finite induction over all cycles between t_1 and t_2. We obtain:

$$\begin{aligned}
\langle ar^{t_2+1}\rangle &= \langle ar^{t_2}\rangle \\
&= \langle ar^{t_1}\rangle \\
&= \langle p.addr^{t+1}[28:17], 0^2\rangle + \langle p.pto^{t+1}[19:0], 0^{12}\rangle \bmod 2^{32} \\
&= \langle p.addr^{t}[28:17], 0^2\rangle + \langle p.pto^{t}[19:0], 0^{12}\rangle \bmod 2^{32} \\
&= ptea \\
dr^{t_2+1} &= mm_8^{t_2}[\lfloor ptea/2\rfloor \cdot 2]
\end{aligned}$$

Therefore

$$pte^{t_2+1} = \begin{cases} dr^{t_2+1}[63:32] & \text{if } ar^{t_2+1}[2] \\ dr^{t_2+1}[31:0] & \text{otherwise} \end{cases} \tag{5.37}$$

$$= \begin{cases} dr^{t_2+1}[63:32] & \text{if } ptea \bmod 8 = 4 \\ dr^{t_2+1}[31:0] & \text{if } ptea \bmod 8 = 0 \end{cases} \tag{5.38}$$

$$= mm_4^t[ptea] \tag{5.39}$$

which proves the claim.

Corollary 5.3 ► *Length and page table entry exceptions are correctly signaled by the MMU.*

Let *ma* abbreviate the address of the access, i.e.

$$ma = \langle mm_4^t[ptea][31:12]\rangle \cdot 2^9 + \langle p.addr^t[8:0]\rangle \ . \tag{5.40}$$

The MMU fulfills the operation semantics for translated read. ◀ Theorem 5.4

PROOF

Consider a translated read request; i.e. additionally to the previously described scenario, we also have $p.mr^t$.

We know by Corollary 5.3 that for $excp = 1$ at the time of acknowledgment, we return $p.excp^{t'}$ and $p.dout^{t'} = 0^{64}$. Also, no write operation gets issued to the cache between cycles t and t'. This corresponds to the memory operation semantics for translated reads in case of an exception.

Assume now that $excp = 0$. By Lemma 5.2, there exists a time t_2 such that the automaton is in state *comppa* in time $t_2 + 1$ and the $pte[31:0]$ bus holds the desired page table entry, i.e.

$$state^{t_2+1} = comppa \text{ and } pte^{t_2+1}[31:0] = mm_4^t[ptea] \ . \tag{5.41}$$

Since we have $p.mr^{t_2+1} = p.mr^t = 1$ by input stableness, we enter the state *read* in time $t_2 + 2$ starting a cache read request:

$$(state^{t_2+2} = read) \wedge reqstart_c(t_2+2) \wedge m.mr^{t_2+2} \tag{5.42}$$

The address of the request is stored in the address register, which is clocked in cycle $t_2 + 1$ and remains unchanged for the whole duration of the request. The address register contains the concatenation of the *ppx* field of the page table entry and the double-word offset of the input address:

$$\begin{aligned}
\langle m.addr^{t_2+2}\rangle &= \langle ar^{t_2+2}[31:3]\rangle && (5.43)\\
&= \langle pte^{t_2+1}[31:12], p.addr^{t_2+1}[8:0]\rangle && (5.44)\\
&= \langle pte^{t_2+1}[31:12], p.addr^{t}[8:0]\rangle && (5.45)\\
&= \langle mm_4^t[ptea][31:12], p.addr^{t}[8:0]\rangle && (5.46)\\
&= ma && (5.47)
\end{aligned}$$

This request finishes with an acknowledgment from the cache. This must be the cycle t', since by definition of the Mealy signal *p.busy* an acknowledgment is given by the MMU to the processor. We obtain:

$$(p.excp^{t'}, p.dout^{t'}) = (0, mm_8^{t'}[ma]) \tag{5.48}$$

as required by the translated read operation semantics.

The MMU fulfills the operation semantics for translated write. ◀ Theorem 5.5

PROOF

This proof goes along the same lines as the proof of Theorem 5.4. Instead of entering the *read* state, of course, an exception-free, translated write enters the *write* state and performs the write over the cache interface on the translated address.

5.2.3 Instruction Fetch

We describe the instruction fetch environment of the VAMP as given in [Bey05] and extend it to support address translation and integration of the instruction MMU.

Interface

The data inputs to the instruction memory (IM) access environment are a 32-bit program counter $if.pc[31:0]$, the mode, page table origin, and page table length special-purpose registers. Furthermore, the environment receives two control inputs, the $fetch$ signal requesting for instruction fetches and the *rollback* signal which indicates a "restart" of the processor due to interrupt or mis-speculation conditions. Outputs are: a busy signal $if.mbusy$, the instruction misalignment flag *imal*, the page fault on fetch flag pff, and the instruction register input $if.ir'[31:0]$.

Under a number of stableness conditions on the inputs (including the memory cells inspected by the MMU), the IM access environment returns

- $(imal, pff, if.ir'[31:0]) = (1, 0, 0^{32})$ for misaligned fetches,
- $(imal, pff, if.ir'[31:0]) = (0, 1, 0^{32})$ for page-faulting translated fetches, and
- the fetched instruction word and cleared flags otherwise.

Control

The most delicate part of the control of the IM access environment is the definition of the $fetch$ signal. Fetches may only start if (i) the *mode*, the *pto* and the *ptl* registers are stable, and (ii) all previous synchronization instructions have terminated.

We define the $fetch$ signal as the conjunction of these conditions, i.e.

$$fetch = fetch_1 \wedge fetch_2 \; . \tag{5.49}$$

By Tomasulo's algorithm, the first condition is met if the valid bits of the mentioned registers are true and in the decode stage there is no instruction writing to any of them. The only instruction directly writing to these registers are special moves and the return-from-exception instruction. We set

$$fetch_1 = mode.v \wedge pto.v \wedge ptl.v \wedge (\neg s_1.full \vee \neg(s_1.id.movi2s \vee s_1.id.rfe)) \tag{5.50}$$

where s_1 prefixes the signals for stage 1 (instruction decode and issue). In particular $s_1.full$ indicates whether the stage is full and $s_1.id.movi2s$ and $s_1.id.rfe$ are decode signals indicating the presence of an `movi2s` and `rfe` instruction. For simplicity, we have made the condition a bit stronger than needed.

The second condition is met if stage 1 does not contain a synchronization instruction. We simply set

$$fetch_2 = \neg(s_1.full \wedge s_1.id.sync) \tag{5.51}$$

where $s_1.id.sync$ is the decode signal for a synchronization instruction (i.e. a `movs2i` instruction reading out the *ieeef* register).[9]

Once activated, the fetch signal will remain active until the fetch request is acknowledged. Note that in case of interrupt conditions, the fetch signal stays or gets activated immediately because such conditions (i) validate all register and (ii) flush all stages, so, in particular establish $\neg s_1.full$.

Note that by the definition of $fetch$, we now have three instruction classes, that make the machine flush its pipeline: (i) return from exception instructions, (ii) writes to the *pto*, *ptl*, or *mode* register, and, (iii) as before, special reads of the *ieeef* register. Furthermore, the effect of synchronization is stronger than before, since fetches may not be started as long as the machine is not completely flushed.

[9] Currently, the second condition provides a means for user programs to change their own page table (and use them, too). This mechanism might be replaced if TLBs are introduced which would need a special system call to support TLB purging for such user mode programs.

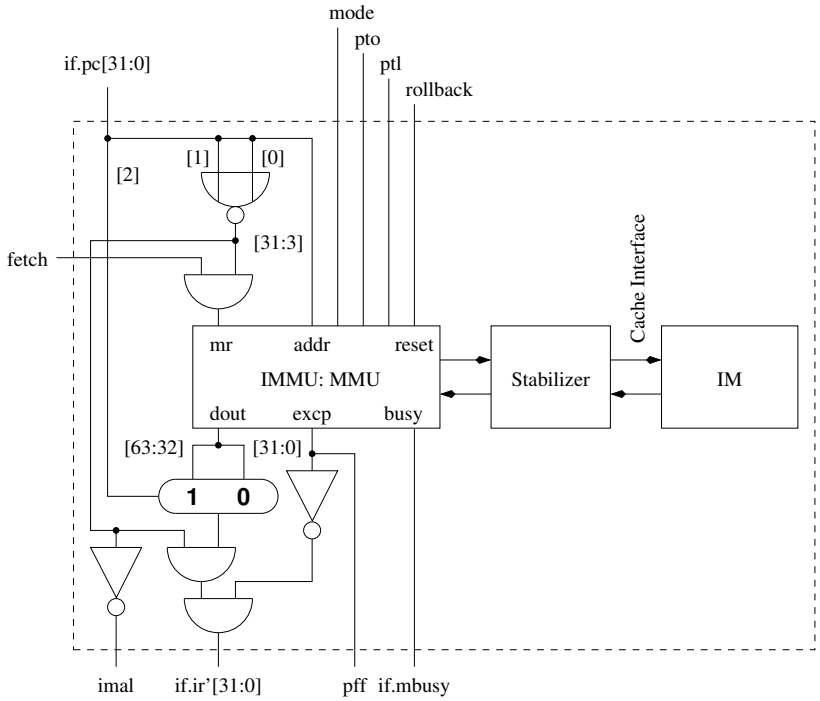

Figure 5.8 VAMP Instruction Memory Access Environment. The interface between the IMMU and the stabilizer has the same signature than the cache interface but does not obey its handshake conditions (in particular it disobeys stability). The original control does not have the MMU, the address translation inputs, and the exception output pff was tied to zero.

Datapaths

Figure 5.8 shows an overview of the IM access environment. Its main components are the instruction MMU and a stabilizer circuit. A request is posed to the IMMU, when the program counter is aligned and the $fetch$ signal is active. The MMU address input is constructed from the 30 leftmost bits of the program counter, additional inputs are taken directly from the inputs to the IM access environment. Any request may be ended prematurely by activation of the *rollback* signal which is therefore connected to the MMU reset input. Due to such resets, the MMU cannot guarantee stable inputs to the instruction memory environment. Therefore, all requests are fed to a stabilizer circuit which is described below.

On acknowledgment, the MMU returns a double word which contains the instruction word to be read: it is the high word if $if.pc[2] = 1$ and the low word otherwise. The multiplexor operates accordingly. On misalignment or MMU exception, the instruction word is zeroed out.

Figure 5.9 shows the implementation of the interface stabilizer circuit. It has one control and one data register:

- The register *stalled* is set when a read starts and cleared when it is acknowledged. If the acknowledgment is given in the same cycle as the read, this register will never be set. Additionally, the register is cleared on reset.

  Hence, the register *stalled* is active for any multi-cycle request to the instruction

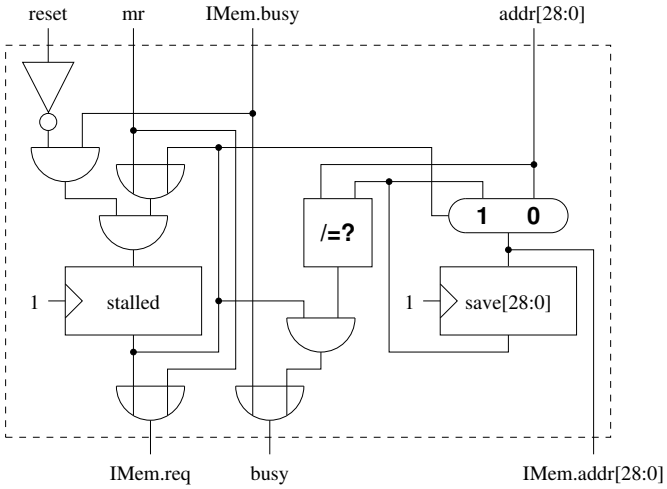

Figure 5.9 VAMP Instruction Memory Input Stabilizer

memory.

- The register $save[28:0]$ holds the address of a request, until it has been acknowledged. Its value is unchanged as long as the *stalled* register indicates an ongoing request. Otherwise, it receives its value from the input address $addr[28:0]$.

We pose a request to the instruction memory, if the *stalled* register indicates an ongoing request or the *mr* input indicates a request start. The address is taken either from the *save* register for an ongoing request or from the input address bus for a new request.

We signal back busy to the instruction MMU, if the instruction memory is busy or if we have a multi-cycle request where the input address $addr[28:0]$ does *not* match the saved address $save[28:0]$. The data output from the instruction memory $im.dout[63:0]$ is directly connected to the MMU's data input (we prefix the signals of the IMMU's processor inputs with *immu*):

$$immu.dout[63:0] = im.dout[63:0] \tag{5.52}$$

Note that, as reported in [BJK+03], by its construction (the instruction cache address is taken from a multiplexor output), the stabilizer circuit may produce *glitches*, so, in a real implementation it is not guaranteed that the address bus remains stable for the duration of several cycles. Absence of glitches is not proven for the VAMP implementation, because the RAM used in the construction of the processor are *synchronous*, i.e. only sample addresses at the beginning of clock cycles. For asynchronous RAM chips, it would be sufficient (yet introduce a cycle's delay) to modify the design such that it takes address bus inputs directly from registers.

5.2.4 Data Memory Accesses

The data memory is accessed via the so-called Data MMU (DMMU) for load and store instructions. Load and store instructions have a specific access width $d \in \{1,2,4,8\}$ in bytes. The target address (which is called effective address) is computed as the

sum of a source register and an immediate constant. If the effective address is not aligned with respect to the access width d, no interaction with the data memory takes place and a data misalignment (*dmal*) exception is caused. Aligned stores are executed only if preceding instructions have already terminated, a property which we call *write-preciseness*. To obtain it we need an additional input from the reorder buffer, the head pointer, which is compared with the instruction's tag. Only on equality, i.e. if the store is the next instruction to complete, a write operation is requested over the data port.

The execution of a load instruction can be interrupted by an external reset or a machine rollback. In this case, however, the data memory access environment must still guarantee that the data port is accessed with stable inputs. We remark that due to write-preciseness, stores may be interrupted only by an external reset.

Note that the design of the data memory access environment is based on the original VAMP's memory unit [Bey05]. In contrast to that design, for ease of presentation, we have chosen to introduce a control automaton for that environment.

Interface

We describe the interface to the DM access environment. It consists of three parts, an interface to the reservation station, an interface to the producer, and a miscellaneous part.

The reservation station supplies the input instructions to the DM access environment in program order.[10] Its signals are prefixed with the symbol *rs*. For interface handshake two control signals are used. The input signal $rs.valid$ indicates that the reservation station holds a valid instruction. The output signal $rs.stallout$ indicates that the DM access environment cannot take a new instruction and any new input instruction must wait. Each input has the following data fields: an instruction tag $rs.tag$, an instruction word $rs.f$ and input operands $rs.op_0$ to $rs.op_5$. While the original design used only three out of six possible 32-bit input operands we now use all of them: Operand 0 stores the source register (here: the base address), operand 1 stores the mode register, operand 2 and 3 store the data to write, operand 4 stores the page table origin register, and operand 5 stores the page table length register.

The producer interface is used to store the results of the DM access environments. Its signals are prefixed with the symbol p. Again, we have two control signals. With the output signal $p.valid$, the DM access environment signals that it has an instruction ready. The input signals $p.stallin$ indicates that the producer cannot take a new result. Additionally, there are the following data busses: a tag $p.tag$, 64 bits of data $p.data$ for memory reads, a cause register $p.ca$ which has bits set for the generated interrupts (only $p.ca[2] = mal$ and $p.ca[4] = pfls$ may be set), and 32 bits of exception data $p.edata$ (which contains the effective address of the memory operation in case of an exception). Refer to [Kro01] for a detailed discussion on how the cause bits and the exception data are processed in the reorder buffer environment and interrupts are detected during write-back.

The three miscellaneous inputs are not subject to any handshake conditions and may change their value in any cycle. The external reset signal *ext_reset* is active in case of an external reset, i.e. on power-up; it initiates the clearing of all caches. The *clear* signal is activated in case of a rollback situation, i.e. if an older instruction generates an interrupt. Last, to support in-order stores, the DM access environment needs to know the currently completing instruction. Since the reorder buffer is organized as a

[10] In the current VAMP implementation this is trivially guaranteed since the memory functional unit only has a single reservation station.

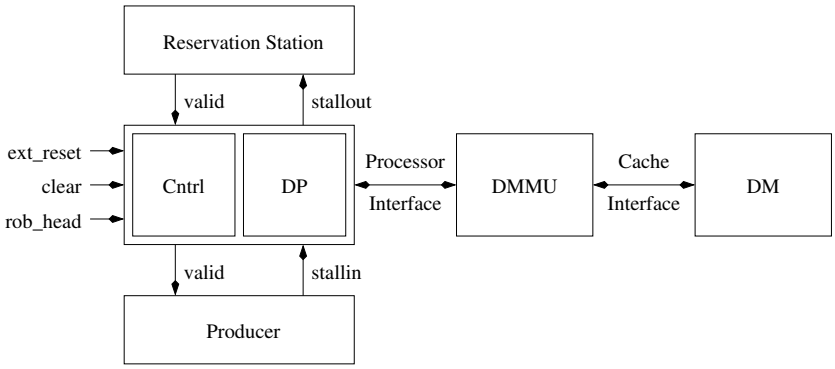

Figure 5.10 VAMP Data Memory Access

(wrap-around) queue, the tag of the currently completing instruction is the head of the reorder buffer denoted *rob_head*.[11] As soon as the ROB head is equal to the tag of a store instruction inside the DM access environment, the write operation is allowed to start.

Figure 5.10 shows an overview of the data memory access environment. We will refer to signals of the processor interface of the data MMU by prefixing them with *dmmu* (so, we replace '*p*.' by '*dmmu*.' with respect to Figures 5.5 and 5.6).

Control

Data memory accesses are currently (sub-optimally) controlled by an automaton. At most one load / store instruction is processed in the DM access environment at any given time. Hence, when it is not idle, the DM access environment raises its stall-out line, signaling to its reservation station that it cannot receive more instructions:

$$rs.stallout = \neg idle \tag{5.53}$$

The control automaton drawn in Figure 5.11 has six different states:

- The *idle* state is active if no instruction is inside the DM access environment. Additionally, the *idle* state is the initialization state and hence entered in case of clear / reset.

  If the reservation station input is valid and there is no clear, we enter one of three successor states: if there is a memory misalignment, we enter the *ready* state (fast exception termination), otherwise we enter the *read* state if we have a load operation (indicated by the signal i_l computed by decoding the instruction word input) or the *wait4tag* state if we have a store operation (indicated by the signal i_s).

- In the *ready* state, the actual operation of the instruction (a load, a store or just the detection of misalignment) has already been completed. We must wait for the producer to signal $\neg stallin$. If during that period a *clear* is observed (which means that the machine has detected an interrupt situation in the reorder-buffer), we enter the *idle* state.

[11] Previously (cf. the PVS source for the VAMP [VAM03]), the naming of the head and tail pointers was contrary to the naming conventions for queues. We fix that and use the head pointer to denote the oldest element in the ROB and the tail pointer to denote the newest element in the ROB. Thus, elements are taken away from the head and added at the tail, as for regular queues.

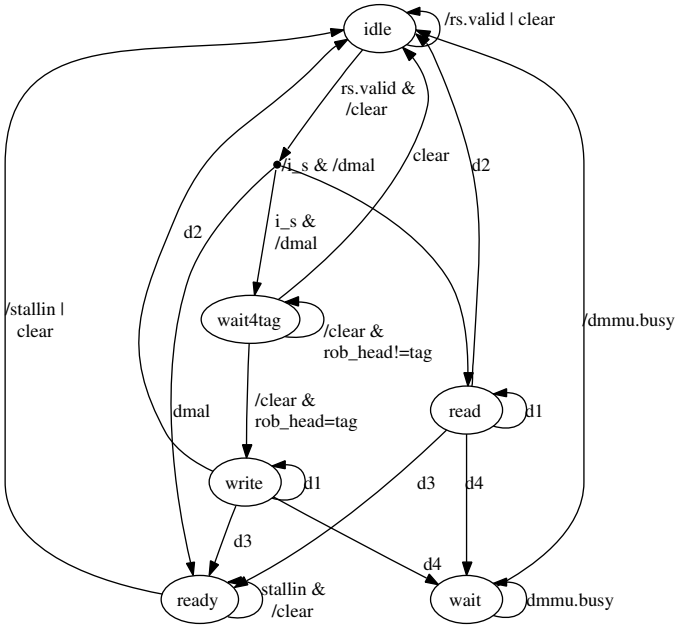

Figure 5.11 Automaton Controlling the Access to the Data Memory. The operators '|', '=' and '!=' are used for logical disjunction, equality, and inequality. For clarity, transition to the *idle* state due to an active *clear* signal are not shown. We abbreviate the following conditions: memory operation not acknowledged and not aborted ($d1 = \neg clear \wedge dmmu.busy$), memory operation aborted or completed ($d2 = (clear \vee \neg stallin) \wedge \neg dmmu.busy$), memory operation finished but producer stalled ($d3 = \neg clear \wedge \neg dmmu.busy \wedge stallin$) and memory operation aborted before acknowledgment ($d4 = clear \wedge dmmu.busy$).

- In the *read* state, the load is issued to the data port. If there is a *clear* signal while the load is not finished, we enter the *wait* state and stay in it until the data memory acknowledges the request. Otherwise, we enter the *ready* state described above.
- In case of stores, the *wait4tag* state is entered and not left until comparison the *rob_head* input with the tag indicates that the store instruction is the next to be written-back. Then, the *write* state is entered. If a *clear* signal is observed before or at the same time, the *idle* state is entered.
- In the *write* state, the actual write request to the data port is performed. If there is a *clear* signal before the request was finished, the *wait* state is entered. Otherwise the *ready* state is entered at the end of the request.
- The *wait* state is not left until the data port signals $\neg dmmu.busy$.

The data memory is accessed only in the *read*, the *write* or the *wait* state. We define the read and the write control signal of the data memory port as follows:

$$dmmu.mr = read \vee wait \wedge i\_l \tag{5.54}$$

$$dmmu.mw = write \vee wait \wedge i\_s \tag{5.55}$$

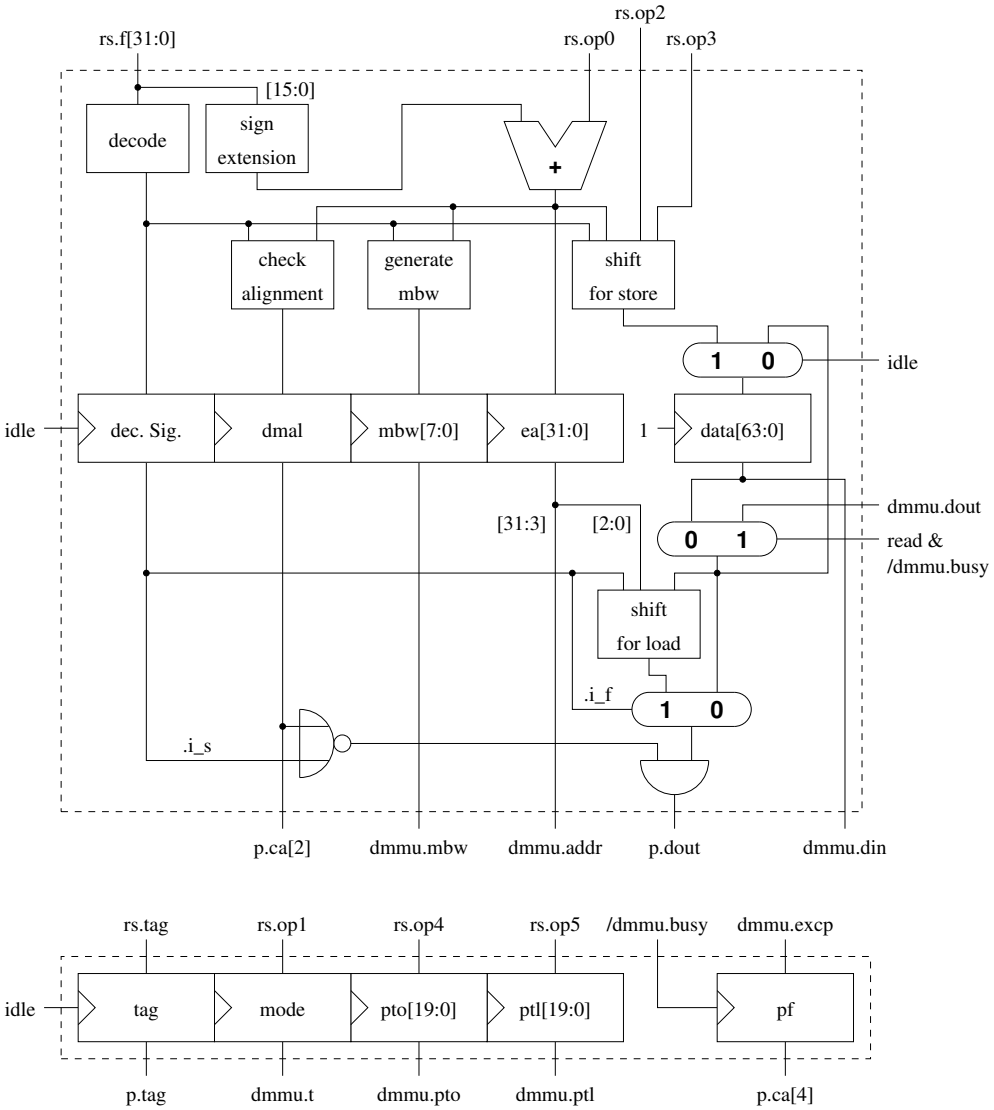

Figure 5.12 Datapaths of the VAMP Data Memory Control. The Data MMU is not shown.

Datapaths

Figure 5.12 shows an overview of the datapaths of the VAMP data memory control.

We start by computing the decode signals related to load / store instructions and sign-extending the instruction word's 16-bit immediate constant $rs.f[31:0]$. Table 5.6 shows the decode signals. These signals encode the operand width (8-bit, 16-bit, 32-bit or 64-bit), whether the operation should sign-extend the operand or not and whether the operation is a load or a store.

The immediate constant and the register source operand are fed into a 32-bit adder to compute the effective address of the instruction which is clocked into the effective address register *ea*.

Using these three intermediate results, we can (i) check whether the memory in-

Name	Description
i_l	Indicates a load instruction
i_s	Indicates a store instruction
i_f	Indicates 64 bit, i.e. double precision floating-point operand
i_u	Indicates unsigned integer operations
i_s	Indicates signed integer operations
i_b	Indicates byte operation
i_h	Indicates half-word operation
i_w	Indicates word operation

Table 5.6 Decode Signals Related to Load / Store Instructions

struction is aligned, (ii) compute which bytes to write in the target double-word, and (iii) shift the input data to the location of the double-word where it should be written to. The specification of these operations is simple. For example a memory operation is aligned iff its access width $d \in \{1,2,4,8\}$ divides the effective address. Implementation of this circuit is straightforward and shown elsewhere (cf. [MP00]).

All the results as well as the tag, the mode, the page table length, and the page table origin inputs are clocked into a register stage, if the control automaton is in the *idle* state and a new instruction might arrive.

The data register *data* is clocked always; it is used to save the store operand as well as the result of a load. Hence, in case we are in the *idle* state, we store the shifted input data into the data register. If we are in state *read* and the data port acknowledges the load operation with $\neg dmmu.busy$, we store the data output of the memory (the load result) into the register. Otherwise, the register does not change its value.

The page fault register pf is clocked on acknowledgment of the data MMU and receives the MMU's exception result.

In principle, the register stage also provides the outputs to the producer. For the *data* register, however, some glue logic is needed: for misalignments and store instructions the data output bus $p.dout$ is forced to zero. If we have a 64-bit operation (indicated by the decode signal i_f), the data register is used unmodified. Otherwise, a shift for load circuit performs the necessary result shifting and embedding based on the effective address.

Unlike the instruction MMU, the data MMU is never reset asynchronously for interrupt / rollback conditions but only on power-up. As we have seen, the data memory access environment keeps the inputs to the MMU stable by which the MMU can guarantee legal access to the data memory. Although completing an ongoing cache request is somewhat faster than completing an ongoing MMU request, such an optimization would yield no global speed-up and is therefore not presented here.

This completes the description of the data memory access environment.

5.2.5 Interrupt-Related Changes

Several interrupt-related changes in the instruction set architecture must be implemented:

- Exception detection must be extended. The illegal exception *ill* is not only caused for illegal opcodes. In user mode, executing the `rfe` instruction and

reading out or writing to special-purpose registers other than those for floating point modes and flags causes such an exception. Therefore, the mode register is now an input to the instruction decode environment and the computation of the flag *ill* must be changed accordingly. Since instructions are only fetched when the *mode* register is stable, it is also stable for instruction decode; the validness of the *mode* register needs not to be checked.

- On interrupt conditions $jisr = 1$ the mode register must be saved to the exception mode register and be cleared. This requires a few extra gates in the special-purpose register file environment.

- The trickiest interrupt-related implementation change has to do with the extended semantics of the `rfe` instruction which now has an additional special-purpose register destination operands, the *mode* register. In principle, the Tomasulo implementation allows for four destination operands: destination operand 1 may be any regular register, destination operand 2 is only used for the duplicated single precision or the second half of a double precision floating pointer operand, and destination operands 3 and 4 are hardwired to the exception cause and the exception data result.

  Similar to the implementation of the floating-point register file, the special-purpose register file is split into two halves, the low half containing the SPRs 0 to 15 and the high half containing SPRs 16 to 31. Destination operand 2 is now also used for special-purpose register destinations with addresses between 16 and 31 (currently only the *mode* register). Hence, the `rfe` instruction is implemented with the status register *sr* as destination operand 1 and the mode register as destination operand 2. This requires a number of small changes in the instruction decode and special-purpose register file environment. Since the `rfe` and the `movi2s` instructions are issued with results to the ROB and do not visit any functional unit, still only FPR registers are ever transmitted on the destination operand 2 bus of the CDB; no changes to the snooping hardware of the reservation stations are required compared to the design in [Bey05].

We will not describe or reason about the above changes any further; they are fully treated in [Dal05].

5.3 Correctness

We give an overview of the VAMP correctness proof and show how to adapt it for the VAMP with virtual memory support. Since we must change the proof structure on a high level to reason about the correctness of a multiprocessor VAMP in Chapter 6, we will treat only the most important additions here. We refer the reader to [Bey05] which gives an in-depth mathematical treatment of the original VAMP correctness proof and to [Dal05], which details and implements the extensions to the formal correctness proof sketched here.

Section 5.3.1 presents an overview of the original VAMP's proof structure. In Section 5.3.2 we sketch the necessary adaptations and extensions.

5.3.1 Overview of the Proof Structure

In the VAMP correctness proof we first show that the implementation is correct for phases of non-interrupted computation. To this end, we derive an auxiliary specification which ignores interrupt conditions. Then, we show that the Tomasulo core correctly processes instructions with respect to that specification if no interrupt is caused. Under the self-modification criterion, it may then be shown that the instruction fetch mechanism loads the specified instructions. Finally, we show that the processor correctly processes interrupted instructions and thus the results may be extended to arbitrary computations.

The following sections explain the above steps in more detail.

Auxiliary Specification

An important proof tool is an auxiliary specification of the VAMP. In this specification, all interrupt conditions—internal and external—are ignored. Formally, the specification collects (masked) interrupt causes during the execution of an instruction. If an interrupt cause is detected, a jump to the interrupt service routine takes place; this involves saving (crucial parts of) the processor configuration, entering system mode, and forcing the program counters to the start of the interrupt service routine (ISR).

For the auxiliary specification, the interrupt causes are always treated as inactive. It is worth noting that both specifications force `nop` instructions into the instruction stream for failed (say page-faulting) instruction fetches. The `nop` instruction is neither a store instruction nor a control-flow instruction. Hence, on fetch failure the auxiliary specification continues straight-line fetch with no additional effect on the memory and the registers.[12]

Tomasulo Core

The instruction fetch mechanism feeds instructions into the Tomasulo core. This interface is controlled by a simple protocol: the instruction fetch mechanism signals to the core when a new instruction is available while the core signals back to the instruction fetch mechanism whether it is currently willing to accept a new instruction. Assuming the instruction fetch mechanism to be live, the Tomasulo core thus receives an input instruction sequence $(J^0, J^1, \ldots)$. From this sequence and an initial configuration, one can derive a sequence of updates to all registers but the instruction register, conforming to the auxiliary specification. Each register update is specified by a list of source registers addresses, a list of destination register addresses and a list of destination values. The destination values are meant to be computed from the source registers values.[13]

The Tomasulo implementation provides data consistency for these updates: essentially, at the beginning of a cycle t, the regular registers (i.e. GPR, FPR, and SPR) in the register file have the same content as the specified registers have *before the execution of the currently retiring instruction*. The program counters are already computed in the issue stage of the Tomasulo core; their values are equal to the specified program counters of the *currently issuing instruction*. Formally, *scheduling functions* determine which instruction is currently issuing or retiring: they are used to tag groups of registers (such as reservation stations or ROB entries) with instruction indices over time. A group of registers k at time t tagged with index i indicates that this group of registers

[12] Since the program counter wraps around even some non-page-faulting code might be reached.

[13] Technically, the results may also depend on additional inputs, say the state of a functional unit. This trick is used to model memory access inside the memory functional unit; decoupling memory access we provide a cleaner solution to this.

holds intermediate or final results of the instruction i. This condition is abbreviated as $sI(k,t) = i$.

Furthermore, the Tomasulo core also provides liveness, i.e. it guarantees that it eventually processes any of its input instructions. Also, two additional properties which are of crucial importance for precise interrupts are provided. First, the core is rollback-capable, so on activation of a *rollback* signal line, it will enter a well-defined machine state in the next cycle clearing all non-terminated instructions from its pipeline. Such a state is called *flushed*. Second, the Tomasulo core is *write-precise*, that means write operations are only issued if all preceding instructions have terminated and caused no interrupt.

Instruction Fetch and the Double-Ported Memory Interface

In the previous section we stated that the Tomasulo core operates on an instruction sequence $(J^0, J^1, \ldots)$ provided by the instruction fetch mechanism. The claim for the instruction fetch mechanism is that as long as there is no interrupt, these instructions are those to be fetched by the auxiliary specification.

The instruction fetch mechanism depends on the program counters which are located in the issue stage of the Tomasulo core. By the core correctness, their values are consistent with the issuing instruction. Since the architecture has a delay slot, it is simple to compute from the program counters the location of the next instruction to be fetched. Therefore instruction fetch could be pipelined reducing the fetch latency.

Technically, however, this constitutes prefetching. To make this work, we required code synchronization (cf. Section 5.1.4): before fetching an instruction modified earlier programs have to execute a special synchronization instruction. This instruction makes the machine drain its pipeline thus preventing prefetching in critical cases.

Interrupt Handling

The three preceding results are all applicable to time intervals $[t : t']$ which start in a 'flushed' machine state in time t (all stages empty) and continue execution until time t' without encountering any interrupt. Next, these results are extended by considering several such intervals separated by exactly one interrupt. If we have a maximal interval $[t : t']$, in time $t' + 1$ an interrupt is detected. In this step, the instruction in the write-back stage is retired, yet this action is outside the afore-mentioned claim of the Tomasulo correctness properties. We reason therefore separately, that the machine indeed conforms to the specification *with interrupts* and is at the start of the interrupt service routine in cycle $t' + 2$ with all required register updates. Moreover, the state in time $t' + 2$ qualifies as the starting state of another interval of uninterrupted execution if it again satisfies the self-modification criterion.

This final step yields overall correctness of the VAMP implementation against its specification.

5.3.2 Adaptation of the Proof

Naturally, the proof of correctness must be adapted at all places where the implementation has been changed. For the interrupt-related changes, one has to show the split special-purpose register file works as before, more illegal exceptions are caused in user mode, the `rfe` instruction writes to its second destination operand, and on interrupts *mode* is saved to *emode* and cleared. For the data memory access environment, it needs

to be shown that the inputs to the MMU are stable. We have seen that this is the case for the inputs supplied from the processor to the MMU. Furthermore, as the memory contents cannot change *while* the data MMU accesses it (a write takes effect only the cycle after the data MMU request has completed), the stableness assumptions for the memory are also met; the MMU assumptions are satisfied and it guarantees to carry out the requested operation.

The trickiest change concerns instruction fetch and the instruction MMU. First, we need to show that the inputs for the instruction MMU are stable. Second, we need to show that due to the synced code property and its implementation the memory cells inspected by the MMU do not change. Only then it is possible to apply MMU correctness to show the correctness of instruction fetch.

We show a series of lemmas which guarantee stable inputs to the instruction MMU and characterize synchronization instructions.

◀ Lemma 5.6

If instruction J_i sets the mode bit, changes the pto register, or changes the ptl register then for all $j \leq i$ the instruction J_j is terminated before the translation of the fetch for J_{i+1} starts.

PROOF

Let t_1 denote the time at which instruction J_i was fetched and not discarded, i.e. the issue stage did not signal a stall out condition. At time $t_2 = t_1 + 1$, instruction J_i is in the instruction decode / issue stage, i.e. $sI(issue, t_2) = i$. Because of the assumption of the lemma, J_i must either be an `rfe` or a `movi2s` instruction. Since

$$s_1.full \wedge (s_1.id.movi2s \vee s_1.id.rfe) \Rightarrow \neg fetch \tag{5.56}$$

the fetch of instruction J_{i+1} does not start as long as J_i remains in the issue stage. Hence, if J_i is issued at time $t_3 \geq t_2$, for all times $t_2 \leq t' \leq t_3$ we have $\neg fetch^{t'}$.

Now, after issue, the destination registers of J_i have been marked as invalid in the producer table. They are set again earliest at time $t_4 \geq t_3$ of retirement of J_i (so $sI(wb, t_4 + 1) = i + 1$). Because of in-order termination (monotonicity of $sI(wb, \cdot)$), at that time all instruction J_j for $j < i$ must also have terminated. Since

$$\neg mode.v \vee \neg pto.v \vee \neg ptl.v \Rightarrow \neg fetch \tag{5.57}$$

the fetch of instruction J_{i+1} cannot start between times t_3 and t_4. This proves the claim.

◀ Lemma 5.7

If instruction J_i is a synchronization or an `rfe` instruction then for all $j \leq i$ the instruction J_j is terminated before the translation of the fetch for J_{i+1} starts.

PROOF

The proof of this lemma is very similar to the proof of Lemma 5.6, the case for the `rfe` instruction was in fact already covered there. Synchronization instructions will not leave the issue stage until all preceding instruction have terminated. Using $s_1.full \wedge s_1.id.sync \Rightarrow \neg fetch$ one can show the claim.

◀ Lemma 5.8

If instruction I_i is an instruction causing an interrupt then for all $j \leq i$ the instruction I_j is terminated before the translation of the fetch for I_{i+1} starts.

PROOF

Let I_i cause an interrupt at its retirement time t, i.e. $jisr(t) \wedge sI(wb, t) = i$. Then, by in-order retirement / preciseness of the interrupt mechanism, all preceding instructions have already terminated. As we have seen in Section 5.2.3, the instruction MMU is reset, so ongoing fetches are interrupted as well. At time $t + 1$ the fetch for instruction I_{i+1} starts. This proves the claim.

With the above lemmas and the synced code property, we may show that for any uninterrupted fetch request between times t and t', the inputs to the instruction MMU and the memory cells inspected by the instruction MMU do not change. Hence, local MMU correctness may be applied and the instruction fetch meets its specification.

5.4 Extensions

5.4.1 Multi-Level Translation

Virtual memory architectures usually do not translate from virtual to physical addresses (or exceptions) using one level of tables but rather more. This is a concession to operating system writers: multi-level translations are more space-efficient and allow for a more flexible organization of the table space.

Of course, multi-level translation is slower to implement, therefore there is even greater need than before for optimizations, e.g. through caching of translations with a translation look-aside buffer (TLB).

In general, for each architecture there is a maximum number of table-lookups for each translation. At each level, parts of the address are used as an index for some table-lookup.

Let $l \in \mathbb{N}$ denote the number of levels. Let f denote a size factor function, which maps zero and every level to a power of two:

$$f : \{0,\ldots,l\} \to \{2^k \mid k \geq 1\} \tag{5.58}$$

The factor function f induces the size function n, which is defined as

$$n(i) := \Pi_{0 \leq j < i}\, f(j) \tag{5.59}$$

for all $i \in \{0,\ldots,l+1\}$.

The number $n(l+1)$ denotes the maximum number of addresses, we define

$$Va := \{0,\ldots,n(l+1)-1\}\,. \tag{5.60}$$

Each address $va \in Va$ can be written as a sum of products of $n(i)$

$$va = \sum_{0 \leq i \leq l} a_i \cdot n(i) \tag{5.61}$$

where the coefficients a_i are uniquely determined under the range condition

$$a_i \in \{0,\ldots,f(i)-1\}\,. \tag{5.62}$$

Since the factors $f(i)$ are all powers of two, the coefficients can be directly taken from the binary representation of the address va. So, if we have bit vectors b_i with lengths $\log_2 f(i)$ for all $i \in \{0,\ldots,l\}$ then $\langle b_l,\ldots,b_1,b_0\rangle = va$ iff $a_i = \langle b_i\rangle$ for all i.

The result of an l-level translation depends on up to l individual memory lookups in l tables, indexed l through 1. They are looked up in descending order; the table of level i is indexed by the coefficient a_i. So, in the first iteration, the coefficient a_l is used as an index into the table l; in the second iteration, the coefficient a_{l-1} is used as an index into the table $(l-1)$ and so on. Typically, the address translation may stop in any iteration due to (i) the detection of an exception or (ii) the computation of the translated address

$f(0) := 2^{12}$	number of relative bytes
$f(1) := 2^{10}$	number of relative pages
$f(2) := 2^{10}$	number of relative segments
$n(0) := 1$	size of a byte
$n(1) := 2^{12}$	size of a page
$n(2) := 2^{22}$	size of a segment
$n(3) := 2^{32}$	size of an address space

Table 5.7 Parameters of the Exemplary $(l = 2)$-level Translation

and the certainty that no exception occurs. These two cases are called early exception and early translation. This corresponds to the translation function having a locally large granularity. In fact, early responses are one of the reasons to use multi-level translation: they reduce the number of tables to visit, which makes them faster again, and they reduce the total number of tables to store for a single address space. Hence, a sparsely occupied address space requires less space to store than the corresponding address space for single-level translations.

The Intel 32-bit architecture IA32 supports both types of early responses allowing for an efficient encoding of certain translation data. *Large pages* allow to map $4M$ of addresses with a single segment table entry; *segmentation* allows to control the exception flag of $4M$ of addresses also with a single segment table entry.

We remark that multi-level translation in principle allows to do paging for and physical sharing of page tables which reduces the (resident) memory consumption of tasks especially in connection with shared memory.

We give an example of a simple two-level translation mechanism. Table 5.7 shows the parameter we choose. An aligned interval of $2^{22} = 4194304 = 4M$ addresses is called a *segment*, an aligned interval of $2^{12} = 4096 = 4K$ addresses is (still) called a page. The translation uses two types of tables, the first table is called the segment table and the second table is called the page table. Both types of tables always have a fixed length of 2^{10} entries à 32 bits, hence the size of each table is $4K$ or 4096 bytes.[14] For each task, a single, top-level segment table of that size is needed. Additionally, for each segment which contains an address attached for some memory operation a page table is needed.

The format of the entries is the same for both tables. Each table entry has three fields interpreted by the hardware. The upper 20 bits store a physical page index; for segment table entries, it is the page index of the next table, the page table; for page table entries, it is the page index of the translated address. Bits 11 and 10 are the valid and the protection bit.

Consider a main memory configuration mm, a task identifier tid, a virtual address va, and a memory write flag mw. We write the virtual address uniquely as the sum of products

$$va = sx \cdot 2^{22} + px \cdot 2^{12} + bx \tag{5.63}$$

where $bx \in \{0,\ldots,4095\}$ and px, $sx \in \{0,\ldots,1023\}$.

The implementation translation function maps the input arguments to the pair of an exception flag $excp$ and a main memory address ma:

$$dec_{itr}(mm,tid,va,mw) := (excp,ma) \tag{5.64}$$

[14] The mechanism is slightly atypical since individual page tables do not have an associated length. However, some hardware designers argue, that this may be a good thing to do [TH94].

Again, this function is composed of two parts, a software part dec_{itr1} and a hardware part dec_{itr2}.

The former function maps a task identifier to the segment table origin, the pointer to the first table to start translation with:

$$dec_{itr1}(mm, tid) = sto \tag{5.65}$$

Substituting the page table origin, the segment table origin is stored in a special-purpose register by the architecture. Because the translation tables have fixed length, a segment table length register *stl* is *not* necessary.

The latter function models the multi-level lookup. It is a function of the main memory, the segment table origin, the address, and the memory operation:

$$dec_{itr2}(mm, sto, va, mw) = (excp, ma) \tag{5.66}$$

The final result of this function depends, as may be expected, on two intermediate values from memory corresponding to the two table lookups.

The segment table entry is the word at the address computed as the segment table origin plus four times the segment index:

$$ste := mm_4[sto \cdot 4096 + sx \cdot 4] \in \mathbb{B}^{32} \tag{5.67}$$

We denote the upper 20 bits of the segment table entry, the page table origin, by $ste.pto$ and the next two bits, the segment valid bit and the segment protection bit, with $ste.v$ and $ste.p$.

The page table entry is the word at the address computed as the page table origin plus four times the page index:

$$pte := mm_4[pto \cdot 4096 + px \cdot 4] \in \mathbb{B}^{32} \tag{5.68}$$

The format of the page table entry is as before, it consists of the three fields which are interpreted by the hardware, the physical page index $ppx = pte[31:12]$, the page valid bit $pte.v = pte[11]$, and the protection bit $pte.p = pte[10]$.

An exception is signaled, if the segment valid bit or the page valid bit is not set or the operation is write and the segment protection bit or the page protection bit is set:

$$excp = \neg ste.v \vee \neg pte.v \vee mw \wedge (ste.p \vee pte.p) \tag{5.69}$$

The translated memory address is, as for one-level translation computed from the physical page index and the byte index:

$$ma = \begin{cases} ppx \cdot 2^{12} + bx & \text{if } \neg excp \\ 0 & \text{otherwise} \end{cases} \tag{5.70}$$

Now, from the definition of translation functions, it is clear that the result of the segment table lookup depends only on the memory operation and the segment index of the virtual address. The result of the page table lookup (which must be performed in the hardware only if $(\neg mw \vee \neg ste.p) \wedge ste.v$) depends on the memory operation and the page index of the virtual address.

Figure 5.13 shows a schematic representation of the defined multi-level lookup.

Designing a memory management unit which implements the translations is a straightforward task. The MMU has to perform at most $l+1$ memory accesses, all but the last of them read-only. The segment table origin for the example translation would of course be stored in a special-purpose register.

This concludes our introduction to multi-level address translation and its hardware support.

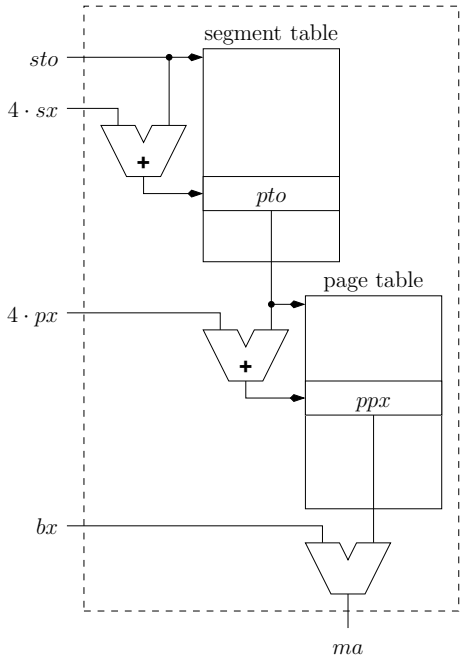

Figure 5.13 Exemplary Multi-Level Lookup

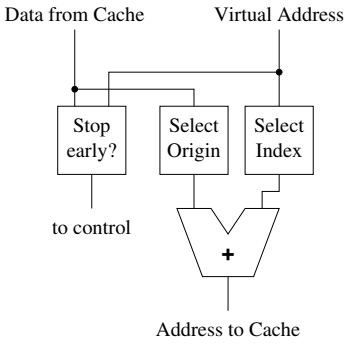

Figure 5.14 Abstract Datapaths of an MMU for Multi-Level Lookup. Cache output and address are buffered in a data register *dr* and an address register *ar* which are not drawn here.

5.4.2 Translation Look-Aside Buffers

Introducing multi-level translation greatly increases the need for optimization. There are several alternatives for optimization:

- A virtually-addressed cache allows direct addressing of the memory cache with virtual addresses. This way, the cache can directly perform virtual memory operations. This approach, though very fast, comes with several draw-backs: (i) for write-back purposes, either an address translation has to be performed, or the cache must store the physical address for each entry, increasing the entry size of each cache entry, (ii) memory sharing or address aliasing lead to consistency

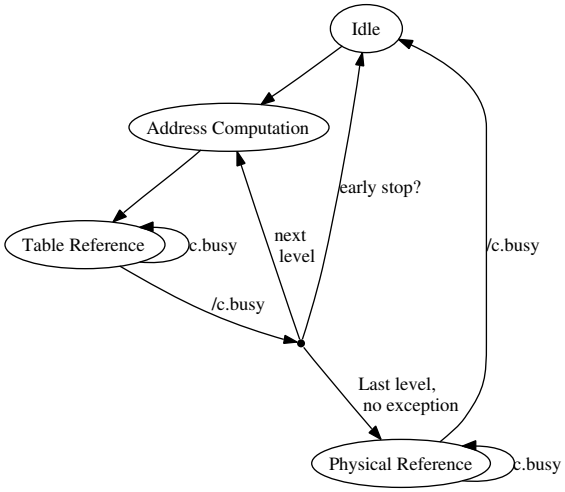

Figure 5.15 Abstract Control of an MMU for Multi-Level Lookup

issues which cannot easily be resolved.

- The most common optimization approach for address translation is by buffering address translations using a translation look-aside buffer (TLB). With an address and a memory operation input, the TLB returns the translation without further accessing the memory. In the following we sketch two TLB variants.
  - A TLB may be accessed by *virtual addresses*. Typically, there are consistency issues related to updates of the translation functions. The exact nature of these depend also on the fact, whether the TLB allows to cache just one page table or many page tables, i.e. over address space switches.
  - For single-level translation, TLBs accessed by the *address of the page table entry* provide an efficient solution for speeding up address translation. Their elegance stems from the fact, that they can be formally interpreted as a cache. Consistency can be guaranteed by implementing a cache protocol running between four caches: the instruction cache, the data cache, the instruction TLB, and the data TLB. Only one of them, the data cache, supports explicit write operations. Hence, the protocol might easily be an update-based one with the TLBs snooping the write operations on the data cache. The TLB may be filled through an interface to a secondary cache or to regular memory.

    The latency of the address computation for the page table entry address can be hidden in the CPU core; with no impact on cycle time, the CPU could provide the PTE for all translated operations.

    Depending on the concrete implementation, it might still be necessary for address space switches to clear the TLB.

With all these optimizations, still a memory management unit is necessary. However, the MMU must perform translations less often: for virtually addressed caches, it is only used for cache misses (and possibly also for write-back), for the TLBs it is used for TLB misses. TLBs are usually controlled by a slightly modified MMU. Virtually-addressed caches, on the other hand, control the MMU.

We present the virtually-addressed TLB approach and implementation in more detail, since it is suited for single-level as well as multi-level translation.

Buffers

The main component of a TLB is of course the buffer itself which we specify in this section. Buffers are also used in a central place in cache design and in units for control-flow prediction (branch target buffers and branch predictors [LS84]).

Let $f : \{0,1\}^n \to \{0,1\}^m$ be a switching function. The characteristic function (or relation) of f is the function $\chi_f : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ where

$$\chi_f(a,b) = 1 \Leftrightarrow f(a) = b\ . \tag{5.71}$$

The characteristic function may also be interpreted as the subset of pairs of inputs and corresponding outputs of f:

$$\chi_f \subseteq \{0,1\}^n \times \{0,1\}^m \tag{5.72}$$

A *buffer* for the function f maintains (or rather *should* maintain) a subset of the characteristic relation as a buffer for f. Hence, we call $buf \subseteq \{0,1\}^n \times \{0,1\}^m$ consistent for f iff $buf \subseteq \chi_f$.

We abbreviate the current buffer content by $buf \subseteq \{0,1\}^n \times \{0,1\}^m$. Buffers support three operations which are defined in terms of inputs, outputs, the current buffer contents $buf \subseteq \{0,1\}^n \times \{0,1\}^m$, and the next-state buffer contents $buf' \subseteq \{0,1\}^n \times \{0,1\}^m$.

- *Read operations* supply an input $a \in \{0,1\}^n$; the buffer returns a hit flag $hit \in \mathbb{B}$ and a result $b \in \mathbb{B}^m$ such that $hit = 1 \Leftrightarrow \exists b : (a,b) \in buf$ and $hit \Rightarrow (a,b) \in buf$ (where b is uniquely determined). The buffer contents do not change for reads, so $buf' = buf$.

- *Write operations* supply an input / output pair (a,b) of f, i.e. $f(a) = b$. The write operations possibly adds the pair to the buffer, at the same time, since buffers are typically much smaller than the function they encode, possibly forgetting other entries:

  $$buf' \subseteq buf \cup \{(a,b)\} \tag{5.73}$$

  This somewhat strange specification is sufficient to reason about the correctness of buffering yet insufficient to reason about performance.

  Of course, any regular buffer implementation forgets entries in a systematic way. For example, a fully-associative buffer implementation with k entries can store k arbitrary pairs $(a,b) \in \chi_f$. If the buffer is full, writing requires entry replacement which is usually done with a pseudo-random or least-recently-used strategy. Not fully-associative buffers have additional restrictions. They partition the set $\{0,1\}^n$ into subsets; for each subset S a small number l of pairs from $S \times \{0,1\}^m$ may be stored in the buffer. This number l is called the number of ways (or compartments).

- *Clear / purge operations* delete entries from the buffer. Dynamic updates of the function f may render a consistent buffer inconsistent and violate $buf \subseteq \chi_f$. Therefore, along with updates of the function f buffer entries need to be cleared, but, for performance, as few entries as possible should be deleted. Depending

on whether entries are purged in hardware or in software we speak of automatic versus manual consistency.

Resets (the purging of all entries) should be avoided. Also, purging single entries only may not suffice; as an example consider the multi-level translation, where the update of a single memory cell (say a segment table entry) can modify the translations for many addresses (say all pages in that segment).

A generic purge operation comes with an input predicate $p : [\mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}]$ and satisfies for the updated buffer

$$\forall (a,b) : p(a,b) \Rightarrow (a,b) \notin buf' \ . \tag{5.74}$$

Not arbitrary sequences of operations may be supported by a specific buffer implementation. A common constraint is that write operations must be preceded by a read attempt for the same input.[15] We do not present a buffer design here. Since buffers are the core part of caches they are pretty well described in common textbooks [HP96, MP00].

Using a Buffer as a TLB

A buffer may be employed as a translation look-aside buffer by letting it cache the (slightly transformed) implementation translation function dec_{itr}. As we have seen, dec_{itr} consists of a software and a hardware part. The TLB will only cache the hardware part, i.e. dec_{itr2}. Furthermore, we will temporarily consider the main memory configuration (the page tables) fixed, so the translation result depends only on the hardware registers (the translation registers of the processor), the address and the memory operation.

For the single-level translation, we have the hardware registers page table origin pto and page table length ptl, both of 20 bits length. Since the translation is granular on pages, virtual input and physical output addressed may be shortened to page indices. Hence, the buffer function f_1 for single-level translation has the signature

$$f_1 : [\mathbb{B}^{20} \times \mathbb{B}^{20} \times \mathbb{B}^{20} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^{20}] \ . \tag{5.75}$$

The exemplary multi-level translation had only one hardware register, the segment table origin $sto \in \mathbb{B}^{20}$. Again, the translation is granular on pages. This allows to shorten the input and output addresses to page indices. Hence, the buffer function f_2 for the multi-level translation has the signature

$$f_2 : [\mathbb{B}^{20} \times \mathbb{B}^{20} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}^{20}] \ . \tag{5.76}$$

Assuming the hardware registers to be *constant*, the buffer entries shrink by 40 bits for f_1 and 20 bits for f_2. This is a worthwhile optimization, however, it is closely tied to consistency issues. For example, both functions "change" whenever tables in the main memory change. As noted, the buffer has to reflect those changes by having the affected entries *purged* before they are used again. Also, when not storing hardware registers, updates to those registers (e.g. the table origin) might make all buffer entries inconsistent in a single instant. Purges may either be issued automatically by the hardware or by purge instructions provided by the architecture. The latter solution is

[15] In a set-associative buffer with least-recently used replacement, the read operation will compute and store internally either the index of the hit way or the index of the least-recently used way in the set. The subsequent write operations will replace this entry. Thus, it may be guaranteed, that the buffer does not contain two entries $(a,b) \neq (a,b')$.

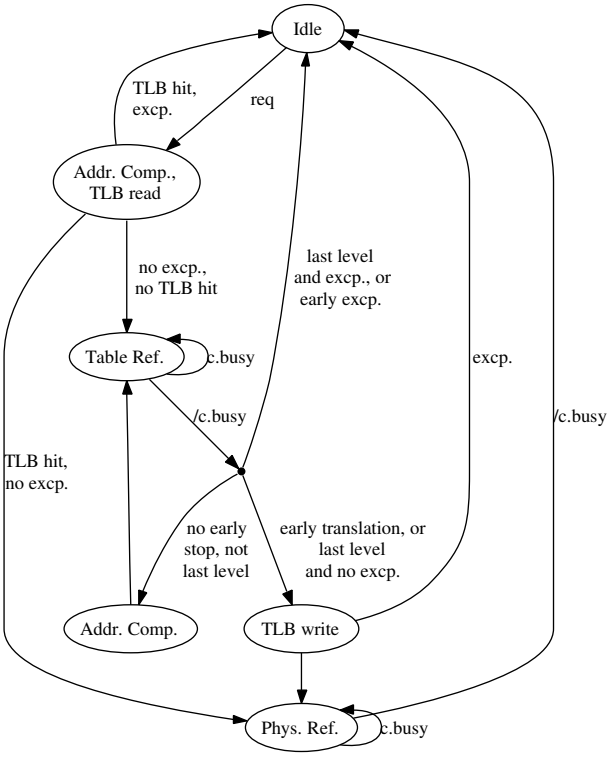

Figure 5.16 Abstract Control of an MMU with TLB for Multi-Level Lookup. For a known level number, the loop in this automaton can be unrolled.

somewhat cumbersome while the former is possibly expensive by decreasing the buffer hit ratio.

Since there are only a few TLB entries, virtual addresses without attachment are typically not stored in the TLB: there is no need to optimize for translations that subsequently invoke the page fault handler.

Building an MMU that controls the buffer is easy. Additional datapaths have to be provided to make use of the TLB result for address and exception computation in case of a hit. We will show a concrete implementation for the one-level translation in the next section. Figure 5.16 shows an overview of an abstract, extended control automaton for an MMU with multi-level lookup (not necessarily the one specified in Section 5.4.1) and TLB integration. In the first level of the translation, when no table reference has yet been made, the TLB is checked for a hit. If there is a hit, the automaton enters the *Idle* state in case of an exception or the *Phys. Ref.* state in case of no exception.[16] Otherwise, the translation continues "slowly" with table references and address computation being executed in a loop. If there is no exception, the translated entry is written into the TLB and the physical memory operation is performed.

[16] An alternative strategy for the exception case is to continue with regular translation. This way, the TLB is lazily and automatically updated with translations that have been attached recently.

	Name	Description
▷	$t.req$	request
	$t.r$	read command flag
	$t.w$	write command flag
	$t.p$	purge command flag
	$t.i.vpx$	virtual page index
	$t.i.mw$	memory write flag
	$t.i.v$	valid flag
	$t.i.p$	protection flag
	$t.i.ppx[31:0]$	physical page index
◁	$t.o.hit$	hit signal
	$t.o.excp$	exception flag
	$t.o.ppx$	physical page index

Table 5.8 TLB Interface

An Example Implementation

In this section we present a simple TLB design and the required MMU extension for the one-level translation mechanism introduced earlier in this chapter.

TLB and Its Interface. The TLB stores a subset of the valid page table entries which lie within the current page table. Hence, each TLB entry is a triple $e = (vpx, ppx, p)$ where vpx is the virtual page index associated with the entry, ppx is the physical page index of the translation and p is the protection flag of the translation.

Table 5.8 shows the interface to the TLB. The TLB implements the three operations read, write, and purge (reset). They are activated by (mutually exclusive) command flags and a request flag. All operations take one cycle to complete.

MMU Datapaths. Figure 5.17 shows the extended datapaths of the MMU designed in Section 5.2.2 to support TLB accesses. Signals from or to the processor are prefixed with p, signals from or to the cache are prefixed with c, and signals from or to the TLB are prefixed with t. Still, this unit is not optimized and wastes a cycle at the beginning of the access. The only changes with respect to the original design are the addition of another multiplexor between the address registers and the adder. This multiplexor allows to clock in the translated address as constructed from the physical page index returned by the TLB and the byte offset of the input address.

MMU Control. In the control automaton (cf. Figure 5.18) there are no additional states. However, there are more transitions and changed control signals. In the *add* state, the TLB is read-out by setting the TLB request t_req and the TLB read flag t_r. If there is a TLB hit (even in case of an exception signaled by the TLB), the TLB result is clocked into the address register. However, if there is either a length exception or a TLB hit and a TLB exception, the automaton returns to the *idle* state, finishing the request with an exception result. The two new important edges go from the *add* state directly into the *read* or *write* state, according to the requested memory operation. They are taken, if the TLB signals a hit and no exception. Otherwise, the regular translation path over *readpte* and *comppa* is taken. In the *comppa* state, a TLB entry is written by setting the TLB request flag t_req and the TLB write flag t_w.

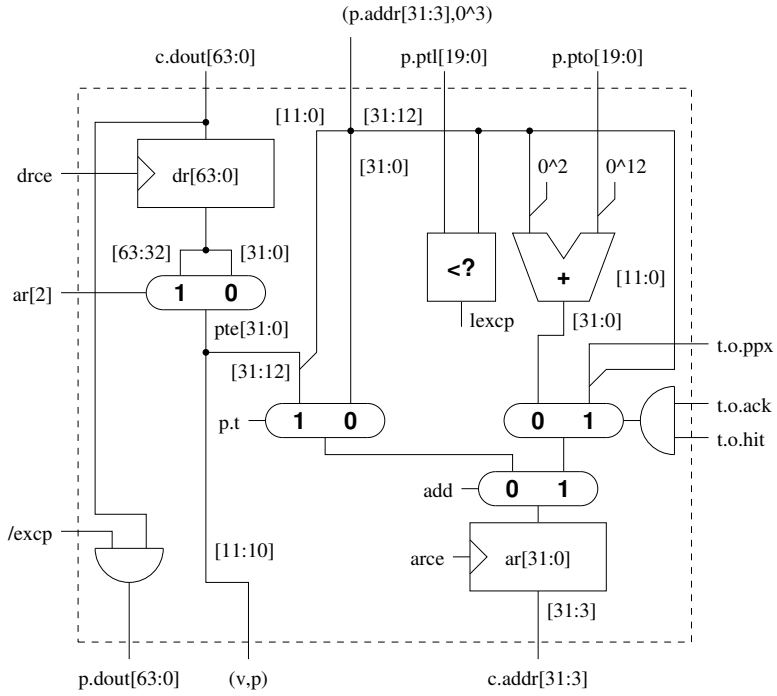

Figure 5.17 Datapaths of an MMU with TLB Integration

Correctness. We purge the TLB on a raising mode flank, i.e.

$$t.p := mode' \wedge \neg mode \; . \tag{5.77}$$

This brute-force approach enables us to show consistency of the TLBs very easily, provided that translation data is not accessible in user mode: the key observation again is that writes in user mode cannot violate consistency by the system memory convention and that system mode instructions are separated from user mode instructions by a synchronization barrier.

To allow for user-mode modifiable translation data (which is not necessary from an operating system's point of view), special trap instructions may be defined to trigger TLB flushes.

5.5 Related Work

The verified architecture micro-processor (VAMP) project has been started in 2000 at Saarland university [VAM03]. The design of the processor was based on the text book [MP00], verification has been based on the paper and pencil correctness in that book and on the scheduling function approach [KMP00]. The formal work in the theorem prover PVS [OSR92] has been started by Kröning, who, among other things, implemented and verified the Tomasulo scheduler [Kro01]. The floating point adder was implemented and proven correct by Berg [Ber01], the remaining floating point units including the rounder and an iterative divider were verified and proven correct by Jacobi [Jac02]. Beyer has implemented and verified a cache system with split instruction

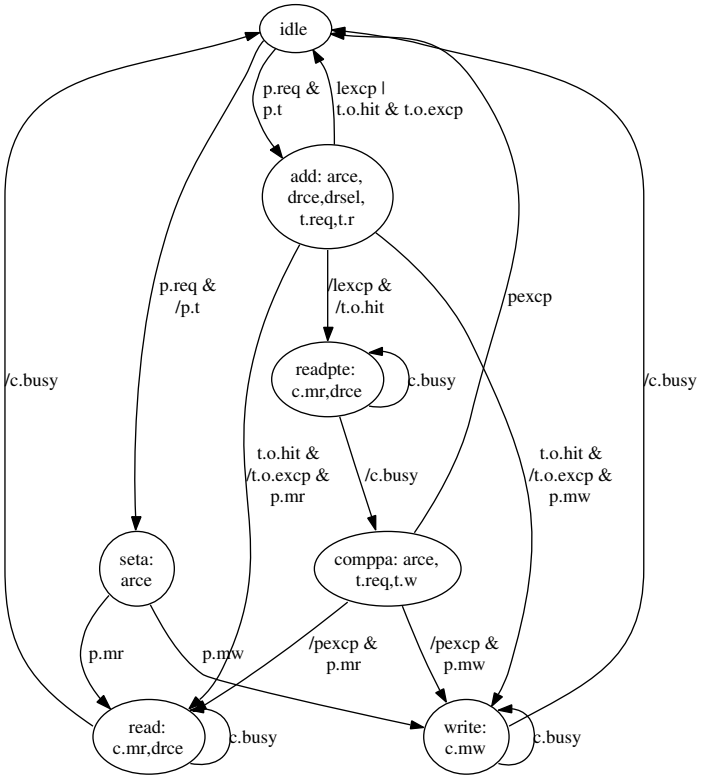

Figure 5.18 Control of an MMU with TLB Integration. The TLB is assumed to respond in the same cycle that a request is started.

and data cache as well as combined the previous proof efforts in a single correctness proof [Bey05]. The VAMP is synthesizable, a separately developed parser translates a subset of the PVS specification language used for hardware description into Verilog [Lei02]. All these results were presented in [BJK$^+$03].

Dalinger [Dal05] has extended the VAMP's implementation in the way described in this chapter, synthesized it, and extended the formal correctness proof in PVS. To our knowledge, Dalinger's proof is the first overall formal correctness proof for a processor with an address translation mechanism. In addition to the original VAMP correctness statement, the correctness of the interrupt mechanisms for external interrupts was also shown.

There is also other work in hardware verification; in fact, hardware verification is probably the most active and successful field in formal verification. See [BJK$^+$03, Bey05] for a comparison of the VAMP processor correctness proofs to related work.

High-level processor designs with memory management units and translation look-aside buffers are given in [HP96]. No correctness arguments is given nor a mathematical correctness proof is conducted for these designs.

A wide range of address translation mechanisms have been employed in computer architecture. For an overview of six relatively modern architectures consider [JM98a] and the quantitative evaluation in [JM98b]. The implications of address translations on operating system design are far-reaching. In addition to the basic format of translation data, its latency, and the generation of page fault interrupts, it is also important to which

extent the address translation mechanism enables sharing of (super-) pages (within a process, across processes) possibly under different rights, making economic use of TLB entries. Therefore, the authors of [JM98a] opt for a standardization of address translation mechanisms since the hardware abstraction layers employed in operating system design are too coarse to make effective use of the specific features of an address translation mechanism.

Three fundamentally different ways of address translation may be distinguished.

First, there are (multi-level) address translation mechanisms like the ones we have characterized and presented in this chapter. Common examples employing such mechanisms are the Intel x86 architecture (or IA-32) [Int04]. IBM S/370 mainframes and its descendants [IBM00] use multi-level lookup mechanism for dynamic address translation (two levels for models with 24 or 31 bit addresses; more levels for the current machines with 64 bit addresses). Several 'address spaces', basically identified by a pointer to the first translation table, may be attached simultaneously (the address to be used for a memory access is determined by the instruction). Additionally, multi-level lookup is also used for address space number (ASN) translation, an architectured mechanism for user processes to attach address spaces.

Second, *inverted page tables* are a different, table-based address translation mechanism. The classical variant of this (IBM System/38, or HP Spectrum, cf. [Tan01]) operates as an array of virtual page indices indexed by physical page indices. To translate a virtual page index, the entry has to be found (by linear search or, in hardware, by an associative lookup) in the inverted page table. More elaborate schemes use hashing to speed up translation; entries of a hashed inverted page table contain a virtual page index, the physical page index, and a pointer (to the next entry in a collision chain). For translation, the virtual page index is hashed into an inverted page table index; starting from the entry at that index, the collision chain is traversed until a matching virtual page index is found. The size of inverted page tables scales with the size of the *physical* memory, so they are much more space-efficient for systems with large virtual address spaces. However (at least in the classical design), each physical page index may only be associated with one virtual page index, prevent the use of address sharing. Modern variants (e.g. Hewlett-Packard PA-RISC [Lee89], IBM PowerPC [AIM95]) mitigate this limitation.

Third, besides hardware-walked page tables, the most wide-spread fundamentally different solution is that of a software-managed TLB. With this approach, used first in MIPS architectures [Kan88, KH92a], a software interrupt is caused on a TLB miss. For a legal access, the interrupt handler is meant to place an entry in the TLB for the page faulting access using special system mode instructions. To avoid unnecessary TLB flushing, the entries are tagged with an "address space" identifier, so TLB entries of different programs may coexist in the TLB. The main advantages of software-managed TLBs are low implementation cost and flexibility. Their main disadvantage is in speed. Even for optimized TLB miss handlers consisting of a handful instructions, translation is slower than with hardware walked page tables. Furthermore, TLB misses require flushing the pipeline, as any other interrupt; in modern processors this may well mean the squashing of dozens of instructions currently in flight in the processor. Finally, the execution of the TLB miss handler may also have a negative effect on the instruction cache hit rate. Jaleel and Jacob propose to in-line the interrupt handling for software-managed TLBs without flushing the processor [JJ01]; a correctness proof for this approach is not given and probably far from trivial. Jacob and Mudge propose a user-programmable state machine for page-table walking to combine the advantages of software-managed TLBs and hardware-walked page tables [JM98b].

Modern architecture may even provide several of the above translation mechanisms. The Intel Itanium architecture (IA-64) [Int02] supports software-managed TLB and two hardware-walked page table formats, a hashed page table ('long-format virtual hashed page tables (VHPT)') and linear page table (misleadingly called 'short-format VHPT'). (A basic comparison of the two latter mechanisms and their impact on kernel performance is presented in [CWH03])

Because of the inherent latency increase of address translation, look-aside buffers have been used and described even for the earliest designs with address translation [Lee69, Den65]. In his early, extensive overview paper on caches, Smith describe regular caches as well as translation look-aside buffers [Smi82]. He notes the commonly used optimization that by having the number of lines in a regular cache being equal to the page size (or granularity), the line address of a cache line used for a lookup is invariant under translation and the TLB and the cache may be accessed in parallel with the TLB. This is also reported in [HP96]; however, the approach limits the size of the regular cache.

Besides using TLBs, there have been other attempts to tackle the latency problem of address translation. Virtually-addressed caches are among the earliest such tries [IC78] and have also been treated in [Smi82]. As has already been pointed out, virtually-addressed caches have complex consistency issues in operating systems that allow sharing of data [WB92, CD97a, CD97b].

Chapter 6

Multiprocessor VAMP

Contents

In this chapter we present a multiprocessor VAMP with virtual memory support. We focus on the changes of the specification / architecture and the correctness proof.

For the architecture, to prepare for the definition of parallel computation, we decouple processor and memory computation, as we already suggested in Chapter 2. We break down computation into phases which come in two varieties: *compute* phases only operate on the processor configuration while *memory* phases additionally carry out one memory operation over a strictly defined memory interface. These phase types correspond to *local* and *communication* steps of concurrent computations. A concurrent or sequentially consistent computation interleaves the phases performed by the individual processors (fairly). An instruction is modeled by two subsequent phases, a fetch phase and an execute phase. The latter is skipped, if a reset was detected in the fetch phase.

We have seen in Chapter 4 that for multiprocessors the parallelism of address translation, translation updates, and regular computation requires special treatment: translations must not be updated while being used. We add architectural support for this, the so-called system barrier. By a software convention, the system barrier must be used for translation updates; while it is active, no user mode phases are executed.

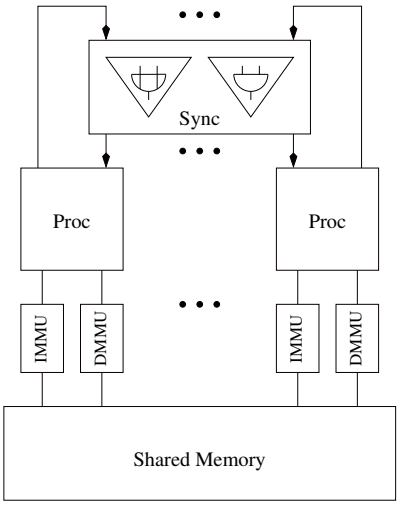

Figure 6.1 Datapaths of a Multiprocessor with System Barrier Mechanism

Additionally, to justify the prefetching implementation of the processor cores, we need to extend the synced code property. Now, code may be modified by the processor running it as well as by other processors. Such code modifications may only be performed if they are properly synchronized, i.e. if the fetch of modified code is preceded by the execution of a synchronization instruction on the fetching processor.

Major adaptations are necessary for the correctness proof. First, we prove the processor cores correct against a decoupled specification by showing that local computation of the processor is correct against a sequence of inputs provided as a parameter. Second, we prove that coupling processors and memory is correct against the concurrent specification. Recall, that the processors of the specification are fundamentally different from the implementation processor in having a single memory port only and not using prefetching. To justify prefetching, we must apply the extended code modification criterion described above.

The implementation itself is based on the processor cores presented in Chapter 5 (with minimal modifications). Without going into cache design we assume that a sequentially consistent (parallel) cache is given to start with. This cache provides two ports for each processor for the instruction and data memory operations. The memory management units are connected to those ports. Additional hardware is required for the implementation of a synchronization mechanism by means of a system barrier consisting of just a few registers and gates. Figure 6.1 shows an overview of the extended hardware.

6.1 Architecture

6.1.1 A Memory-Decoupled Architecture

In this section we remodel the VAMP architecture into a two-phased instruction set architecture and a memory operation architecture. The former defines the computation of the processor while the latter defines responses and updates of the memory.

Register	Update time
$fetch$	set after execute or JISR, cleared after fetch
pff	after fetch
IR	after fetch
DPC	after execute or JISR
PC'	after execute or JISR
$GPR[i]$	after execute, but $GPR[0]$ always equal to 0^{32}
$FPR[i]$	after execute
$SPR[i]$	for $i \in \{6,7,8\}$ (FP-related): updated after execute
	otherwise (interrupt-related): only updated on JISR

Table 6.1 Overview of the VAMP registers and Their Update Time. Use of *fetch* and *execute* implies absence of interrupts, i.e. $\neg JISR$. The individual registers of the register files GPR, FPR and SPR are indexed with a variable $i \in \{0,\ldots,31\}$. The registers $flag$ and IR are new visible registers of the specification.

Phases

A *phase* is a single step of computation of the processor. We distinguish two fundamentally different phase types. *Compute* phases describe (local) updates of the processor configuration, *memory* phases describe processor updates involving a memory operation. For the VAMP, memory operations are aligned double-word reads and aligned double-word writes (with separate write-enables for each byte in the double-word). Naturally, only writes entail an update of the memory configuration.

Furthermore, we distinguish fetch phases and execute phases. Under the absence of resets both types alternate; a fetch phase and the subsequent execute phase are called an *instruction*. Since the computation starts with a fetch phase indexed with zero, fetch phases are even-numbered and execute phases are odd-numbered. This means that all interrupts except reset are handled in the execute phase of an instruction. Still, no interrupt is lost because of "being ignored" in the fetch phase: instruction misalignment and page fault are passed on silently to the execute phase (which requires an additional flag in the configuration), external interrupts must, by convention, be kept up until serviced, and other interrupts are not generated during fetch.[1]

The configuration of the original VAMP architecture is extended with three new components. The flag $fetch$ determines, whether the processor is currently in a fetch or an execute phase. The flag pff indicates, whether a page fault occurred in the last fetch phase; without address translation, it is defined to be zero. The instruction register IR passes the results of a (successful) fetch phase to the execute phase; it was an invisible (derived) component before.

Table 6.1 lists all the registers of a processor configuration. All but the $fetch$ and pff flags have a width of 32 bits. We denote the set of all processor configurations by P. Additionally—in anticipation of the definition of computation—the table lists the situations in which the registers are updated. Three such situations are distinguished: fetch without JISR, execute without JISR, and JISR (because of reset or during execute). Most importantly, during regular computation, fetch reads the DPC and updates the IR and pff registers while execute reads the IR and pff and updates the DPC

[1] Alternatively, interrupt misalignment or page fault might directly enter another fetch phase which saves the additional flag in the configuration. However, these phase shifts that disturb the even-odd analogy have to be taken into account for the correctness proof.

register (among other things).

Compute (or local) phases operate on them via functions $cmp : [P \to P]$. For memory phases, let Mop denote the set of memory operation identifiers, let Din' denote the set of data inputs and let $Dout'$ denote the set of data outputs (more on these later).[2] Two components are needed to model each memory phase: a send function and a receive function. The send function $snd : [P \to Mop \times Din']$ maps a processor configuration to a memory operation identifier and a data input. The receive function $rcv : [P \times Dout' \to P]$ maps a processor configuration and a data output to another processor configuration. To distinguish both types of phases, a flag $m \in \mathbb{B}$ is introduced which is true for memory phases and false otherwise.

Hence, phases are modeled by 4-tuples

$$ph = (m, snd, rcv, cmp) \in \mathbb{B} \times Snd \times Rcv \times Cmp\ . \tag{6.1}$$

The decode function dec takes a processor configuration p and maps it into a phase ph. For a compute phase $\neg m(ph)$ the processor updates its state to $cmp(ph)(p)$. For a memory phase, the processor provides the input $snd(ph)(p) \in Din'$ to the memory and, on receiving the return value $dout \in Dout'$, it updates its state to $rcv(ph)(p, dout)$. We let $Compute(cmp)$ and $Memory(snd, rcv)$ abbreviate the phases $(0, snd', rcv', cmp)$ and $(1, snd, rcv, cmp')$ where snd', rcv' and cmp' are chosen arbitrarily.

If $fetch(p) = 1$ we speak of a *fetch phase*, if $fetch(p) = 0$ we speak of an *execute phase*. To give a feeling for these definitions, we sketch how the phases related to fetch, load, and store are defined.

There are two different fetch phases. If the program counter DPC is misaligned ($DPC[1:0] \neq 0^2$), no actual instruction fetch is performed, the instruction word is forced to zero, and a jump to the interrupt service routine takes place. This compute phase is denoted by $fetch\_mal$. Otherwise, a real fetch is performed by issuing a memory read operation and processing the memory results. We denote this phase simply by $fetch\_al$. It is a memory phase; on reset it jumps to the interrupt service routine. Hence, if $fetch(p)$ then $dec(p) = fetch\_mal$ if $DPC(p)[1:0] \neq 0^2$ and $dec(p) = fetch\_al$ otherwise.

For load and store, the modeling of the phases is similar, that means for each access width there is one compute phase modeling misaligned access and one memory phase modeling an actual access to the memory. Byte operations, however, are always aligned and may be modeled with a single phase.

The jump to the interrupt service routine is modeled as a separate, parameterized function $jisr(mca) : [P \to P]$ to allow it to be reused in the individual definition of the phases. It takes the masked causes as input; $mca[i] = 1$ signals the occurrence of interrupt i. If the fetch flag is active and there is no reset ($mca[0] = 0$) or all other masked causes are also zero, the $jisr(mca)$ function behaves as the identity, deferring treatment of all interrupts but reset to the execute phase. Otherwise, it returns a processor configuration where the appropriate special-purpose registers have been updated and the program counters have been forced to the start of the interrupt service routine (which should handle the interrupt i with $mca[i] = 1$ and i minimal, although this is a software convention).

Memory Operations

The VAMP is a load-store architecture and as such has a very limited number of phases operating on the main memory. The phases are (i) a 32-bit fetch, (ii) 8-bit, 16-bit, and

[2] We use primed versions of Din' and $Dout'$ to avoid name clashes similar than in Chapter 5.

32-bit GPR loads and stores, (iii) 32-bit and 64-bit FPR loads and stores. All come in two variants, translated or not translated. The memory operations identifiers are encoded in bit strings, we have

$$Mop \subseteq \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B}^8 \,. \tag{6.2}$$

Let $(t, mr, mw, mbw) \in Mop$. The translation flag t distinguished translated from untranslated operations; The flag mr indicates a read operation, the flag mw indicates a write operation. Both are mutually exclusive. For write operations, the bit vector mbw indicates which bytes to write in a certain double word. Not all combinations of mbw are valid; we have a single bit set for byte operations, two consecutive bits sets for half-word operations, either the upper or the lower half set for word operations and all bits set for double-word operations. Invalid combinations are also called misaligned but will never be requested by the specified machine (instead, as has been sketched, a compute phase will be executed for misaligned operations).

The data input consists of a double word address $addr \in \{0, \ldots, 2^{29} - 1\}$ and (for write operations) of write data $din \in \mathbb{B}^{64}$. Additionally, for translated operations, the page table origin $pto \in \mathbb{B}^{20}$ and the page table length $ptl \in \mathbb{B}^{20}$ are needed as inputs for the address translation. The data output is an exception flag $excp$ needed for translated operations and a double-word $dout \in \mathbb{B}^{64}$.

$$(addr, din, pto, ptl) \in Din' = \{0, \ldots, 2^{29} - 1\} \times \mathbb{B}^{64} \times \mathbb{B}^{20} \times \mathbb{B}^{20} \tag{6.3}$$

$$(excp, dout) \in Dout' = \mathbb{B} \times \mathbb{B}^{64} \tag{6.4}$$

Let $mm : [\{0, \ldots, 2^{29} - 1\} \to \mathbb{B}^{64}]$ denote a (main) memory configuration. Formally, a memory operation maps data inputs and a memory configuration to an updated memory configuration and data outputs. By the memory decode function dec_m each memory operation identifier $mop \in Mop$ is mapped to a memory operation, i.e. it has the type

$$dec_m : [Mop \to [Din' \times Mm \to Mm \times Dout']] \,. \tag{6.5}$$

We now instantiate this scheme for the VAMP. Consider a memory operation identifier (t, mr, mw, mbw) and memory operation inputs $(addr, din)$. Let mm denote the current memory configuration and mm' the updated memory configuration.

We first compute the pair $(excp, pa)$ which is equal to zero and the original address for untranslated operations and equal to the result of the (hardware) implementation translation function dec_{itr2} (cf. Section 5.1.2) otherwise:

$$(excp, pa) = \begin{cases} (0, ma) & \text{if } \neg t \\ dec_{itr2}(mm, pto, ptl, ma, mw) & \text{if } t \end{cases} \tag{6.6}$$

If an exception is signaled, $excp = 1$, we set $mm' = mm$ and additionally return $dout = 0^{64}$.

Otherwise, for read operations ($mw = 0$), the data output is the double word at the address pa. We set $dout = mm(pa)$. Additionally, the memory configuration is not updated, i.e. $mm' = mm$. For write operations, we replace the bytes indicated by the mbw bus with the data input. We define the new value of byte $i \in \{0, \ldots, 7\}$ as follows:

$$mm'(pa)[i \cdot 8 + 7 : i \cdot 8] = \begin{cases} din[i \cdot 8 + 7 : i \cdot 8] & \text{if } mbw[i] \\ mm(pa)[i \cdot 8 + 7 : i \cdot 8] & \text{otherwise} \end{cases} \tag{6.7}$$

The data output is zero, i.e. $dout = 0^{64}$.

Computation

In this section we introduce five functions modeling computations of the architecture at different levels:

- The function δ is the top-level computation function, operating on pairs of processor and memory configuration. It is composed of a function δ_f modeling fetch phases and a function δ_x modeling execute phases.
- The function η decouples processor computation from the memory computation by taking a stream of memory outputs as an additional input.
- The function η_u is identical to η but disregards interrupt conditions.
- The function η_c decouples the execute phase (i.e. what is later to be executed by the Tomasulo core) from the instruction fetch mechanism by taking a stream of instructions (and instruction page faults) as an additional input.
- The function $\eta_{c,u}$ is identical to η_c but disregards interrupt conditions.

The function δ is the top-level computation function. It operates on the processor and memory configuration synchronously / simultaneously; therefore, we call it also *coupled* computation function. A computation step consists of processing a phase. Correspondingly, we define it in two parts, one for fetch and the other for execute phases:[3]

$$\delta(p,mm) = \begin{cases} \delta_f(p,mm) & \text{if } fetch(p) \\ \delta_x(p,mm) & \text{otherwise} \end{cases} \tag{6.8}$$

Fetch and execute phases always alternate if there is no reset. Under this assumption, we have $\neg fetch(\delta_f(p,mm))$ and $fetch(\delta_x(p,mm))$ from which we obtain

$$\delta^{2\cdot i}(p,mm) = (\delta_x \circ \delta_f)^i(p,mm)\ . \tag{6.9}$$

We will not define δ_f here; depending on the lower bits of the fetch program counter, either a `nop` is forced into the instruction stream or a real fetch operation is performed. Additionally, the page on fetch flag pff is updated.

For the definition of δ_x we assume there is a decode function $dec_p : [P \to Ph]$ mapping a processor configuration into a computation phase. The decode function only depends on the instruction register and the general-purpose registers (to distinguish misaligned and aligned phases for load / store instructions). The result of $\delta_x(p,mm)$ depends on the phase type of $dec_p(p)$. Assume that $dec_p(p) = Compute(cmp)$. Then, the memory configuration remains unchanged and the update of the processor configuration is computed by cmp, so in this case

$$\delta_x(p,mm) = (cmp(p),mm)\ . \tag{6.10}$$

Otherwise we have $dec_p(p) = Memory(snd,rcv)$. Let $(mop,din) = snd(p) \in Mop \times Din'$. The next memory configuration and the output of the memory are determined by

[3] Note that we use $mm \in Mm = [\{0,\ldots,2^{29}-1\} \to \mathbb{B}^{64}]$ in this chapter consistently to denote the main memory configuration of the VAMP architecture. This is in continuation of Chapter 5 but breaks with the more general use of $mem \in Mem$ for generic memory configurations from Chapter 2.

the memory operation $dec_m(mop)$ where $dec_m : [Mop \to [Din' \times Mm \to Mm \times Dout']]$ is the decode function for the memory. We have

$$(mm', dout) = dec_m(mop)(din, mm) \; . \tag{6.11}$$

Finally, the next processor configuration is computed by the receive function which results in

$$\delta_x(p, mm) = (rcv(p, dout), mm') \; . \tag{6.12}$$

To compute many steps, one may iteratively apply the function δ to an initial configuration (p, mm). We abbreviate an i-fold application by $\delta^i(p, mm)$. This completes the definition of (coupled) computation with interrupts.

Now we additionally decouple the processor computation from the memory computation which was the original intention in developing phased computation. The function η models the computation of a single step based on a (possibly ignored) data output *dout* of the memory. It produces a new processor configuration and possibly a memory operation and data input to the memory.

$$\eta : [Dout' \times P \to P \times (Mop \times Din')_\perp] \tag{6.13}$$

Here, $T_\perp := T \uplus \{\perp\}$ denotes the optional type, i.e. the disjunct union of T with the symbol $\perp$ ("bottom"). We define

$$\eta(p, dout) = \begin{cases} \eta_f(p, dout) & \text{if } fetch(p) \; , \\ \eta_x(p, dout) & \text{otherwise.} \end{cases} \tag{6.14}$$

Formally, the input *dout* is used later on to denote the result of the memory operation (mop, din) issued in *the same cycle*. The definition of η_f and η_x is similar to their coupled counterparts. The result of $\eta_x(p, mm)$ depends on the phase type of $dec_p(p)$. Assume that $dec_p(p) = Compute(cmp)$. Then, no memory operation is issued and the update of the processor configuration is computed by *cmp*, in this case

$$\eta_x(p, dout) = (cmp(p), \perp) \; . \tag{6.15}$$

Otherwise we have $dec_p(p) = Memory(snd, rcv)$. The next processor configuration is computed by the receive function and the outputs are computed by the send function. We define

$$\eta_x(p, dout) = (rcv(p, dout), snd(p)) \; . \tag{6.16}$$

The function η_u is identical to η but disregards interrupt conditions. It is defined with the auxiliary functions $\eta_{u,f}$ and $\eta_{u,x}$ for fetch and execute phases. The former will, even on reset, always enter an execute phase. It never sets the instruction page fault flag. The latter function is similar to η_x but does not heed the masked cause flag *mca*. This can be modeled by an auxiliary decode function dec'_p, which always regards all masked causes $mca[i]$ as zero.

The decoupled core computation function η_c is derived from η by only performing the execute phases. Results of fetch phases namely the instruction register *IR* and the page fault on fetch flag pff are provided as additional input to the function. These

inputs override the corresponding entries of the processor configuration; additionally, the fetch flag is forced to zero (although η_x does not depend on it). We define η_c as

$$\eta_c(IR, pff, dout, p) = \eta_x(p \text{ with } [IR(p) = IR, pff(p) = pff, fetch(p) = 0], dout) . \quad (6.17)$$

The corresponding variant ignoring interrupts $\eta_{u,c}$ is defined similarly but in terms of $\eta_{u,x}$. For this function, the instruction page fault flag is ignored, still we will supply it as an input for reasons of symmetry.

For the different coupled and decoupled computation functions relatively boring equivalence lemmas of the following form hold.

Lemma 6.1 ► *Consider (p,m) with $dec_p(p) = Memory(snd, rcv)$. Let*

$$(mop, din) = snd(p) , \quad (6.18)$$
$$(m', dout) = dec_m(mop)(din, m) , \text{ and} \quad (6.19)$$
$$p' = rcv(p, dout) . \quad (6.20)$$

Thus, $(p', m') = \delta(p,m)$. Then, we also have

$$\eta(dout, p) = (p', (mop, din)) . \quad (6.21)$$

These lemmas are proven directly by the definition of the computation functions.

6.1.2 Concurrency

We review and extend definitions from Chapter 2. A multiprocessor configuration is denoted by $c_{mp} = (p_1, \ldots, p_n, mm) \in C_{mp} = P^n \times Mm$; it consists of n processor configurations p_i and a shared memory configuration mm. The function δ_{mp} executes a single step for a processor $i \in \{1, \ldots, n\}$ specified as an additional parameter. Naturally, the definition of δ_{mp} is based on δ. Let

$$\delta_{mp}(i, p_1, \ldots, p_n, mm) = (p'_1, \ldots, p'_n, mm') . \quad (6.22)$$

Then, p_i and mm are updated according to δ, i.e. $(p'_i, mm') = \delta(p_i, mm)$, all other processor configurations stay unchanged, i.e. $p'_j = p_j$ for $j \neq i$.

Computations are parameterized over a schedule $sched : [\mathbb{N} \to \{1, \ldots, n\}]$ and an initial configuration $(p_{1,0}, \ldots, p_{n,0}, mm_0)$. The schedule specifies which processor executes in which step; it has to be fair, each processor index has to appear infinitely often. The inductive definition of concurrent computation is given by

$$(p_{1,x+1}, \ldots, p_{n,x+1}, mem_{x+1}) = \delta_{mp}(sched(x), p_{1,x}, \ldots, p_{n,x}, mem_x) \quad (6.23)$$

with $(p_{1,x}, \ldots, p_{n,x}, mem_x)$ denoting the configuration before execution of the $(x+1)$-th step of computation.

Reasoning about processor-memory coupling, it is convenient to abstract from the scheduling of local (compute) operations of the processors. We define big step functions, which compute the result of a memory phase and a maximum number of compute phases in a single application. There are two variants of big step functions, for coupled and decoupled computation. Let $(p, mm) \in P \times Mm$ with p encoding a memory phase,

$dec(p) = Memory(\ldots)$. The (coupled) big step function $\Delta : [P \times Mm \to P \times Mm]$ is defined by

$$\Delta(p, mm) = \delta^i(p, mm) = (p', mm') \tag{6.24}$$

where $i \in \mathbb{N}^+$ is minimal such that p', again, encodes a memory phase, $dec(p') = Memory(\ldots)$. This function is well-defined, provided the processor cannot get stuck in an infinite sequence of compute operations, a condition that holds for the VAMP ISA (in fact, we have $i \in \{1,2\}$ for the VAMP). A big step computation (p_i, mem_i) is defined by inductive application of Δ on an initial configuration (p_0, mm_0) where we assume p_0 to encode a memory phase. We let j_x indicate the index of the (memory) phase to be executed at the beginning of big step x. Hence, $j_0 = 0$ and $j_{x+1} = j_x + i$ where $i \geq 1$ is the number of applications of δ in big step x as in Equation 6.24.

For decoupled computation, we denote the big step function by $\Gamma : [Dout' \times P \to P \times Mop \times Din']$. Since Γ is always applied on processor configurations p encoding a memory phase, the memory operation and data input pair need not to be optional anymore as it was for $\eta : [Dout' \times P \to P \times (Mop \times Din')_\perp]$. Let $(dout, p) \in Dout' \times P$ with p encoding a memory phase, $dec(p) = Memory(\ldots)$. We define $\Gamma(dout, p) = (p', mop, din)$ as follows. First,

$$(mop, din) = (mop(dec(p)), snd(dec(p))(p)) \,. \tag{6.25}$$

Second, let $\tilde{p} = rcv(dec(p))(p, dout)$ and $\widetilde{mm} \in Mm$ be arbitrary. Then,

$$(p', mm') = \delta^i(\tilde{p}, \widetilde{mm}) \tag{6.26}$$

for $i \in \mathbb{N}$ minimal, possibly zero, such that $dec(p') = Memory(\ldots)$. For the VAMP instruction set architecture, we have $i \in \{0,1\}$. Phase indices for a computation with respect to Γ are defined by $j_0 = 0$ and $j_{x+1} = j_x + i + 1$ with i as in Equation 6.26.

For concurrent big step computation, we use Δ instead of δ as a step function. For such computations, we rename the schedule parameter *sched* to the sequence parameter $seq_S : [\mathbb{N} \to \{1, \ldots, n\}]$. This draws the connection to the sequential consistency definition; seq_S specifies the order of memory operations. So, for seq_S and an initial configuration $(p_{1,0}, \ldots, p_{n,0}, mem_0)$ we define the big step concurrent computation inductively by

$$(p_{1,x+1}, \ldots, p_{n,x+1}, mem_{x+1}) = \Delta_{mp}(sched(x), p_{1,x}, \ldots, p_{n,x}, mem_x) \tag{6.27}$$

where $(p_{1,x}, \ldots, p_{n,x}, mem_x)$ denotes the configuration before execution of the $(x+1)$-th big step and

$$\Delta_{mp}(i, p_1, \ldots, p_n, mm) = (p'_1, \ldots, p'_n, mm') \tag{6.28}$$

with $(p'_i, mm') = \Delta(p_i, mm)$ and $p'_j = p_j$ for $j \neq i$. For a concurrent big step computation we use variables $j_{i,x}$ to denote the phase indices to be executed at the beginning of big step x of processor i. Initially $j_{i,0} = 0$ for all i; after big step x we update $j_{seq_S(x),x+1}$.

Similar, but less meaningful, definitions can be made for the concurrent computations with respect to η and Γ. We do not use them in such a context.

The big and small step variants of concurrency can be used interchangeably if only memory configurations are observed.

6.1.3 System Barrier

Translated memory operations are implemented with a MMU (and a TLB for further speed-up) which breaks them down into a series of cache read operations trailed by a cache read or write operation if no page fault is caused. Thus, they are defined as atomic operations but not implemented as such. For this to work address translations must not be updated while being used; this property is called *translation persistence* (cf. Section 4.6.2).

To establish translation persistence, we introduce special architectural support: *system barriers* allow to temporarily suspend all user mode and translation activity. In contrast to interrupts, in particular IPIs (inter-processor interrupts), system barriers do not change the control flow but merely halt it on user mode processors.

The system barrier protocol which runs partly in hardware and partly in software can be divided into three phases:

- Entering the barrier lets ongoing translations complete and keeps new translations from starting.
- In the barrier, critical updates may be performed by more than one system-mode processor.
- Leaving the barrier continues execution.

With TLBs, the clearing of TLBs (i.e. of the updated entries) is necessary before leaving the barrier.

Two new special-purpose registers (SPRs) are introduced to support barrier operations.

- The (local) barrier request register brq_i for processor i is written whenever processor i wants to enter a barrier. The global system barrier request signal is computed as the disjunction of all those local request signals:

$$sysrq = \bigvee_i brq_i \tag{6.29}$$

- The system flushed register $sysflushed$ may be read out by all processors and indicates whether the whole system is in a (user mode) flushed state, i.e. has completed the entering of the barrier. It may not be written. A single processor is considered flushed if it does not currently compute any instruction (for the VAMP: the ROB is empty) or if it is in system mode. We would like to define the system flushed register as the conjunction of these disjunctions for the individual processors,

$$sysflushed = \bigwedge_i (flushed_i \vee \neg mode_i)\ . \tag{6.30}$$

  However, on the specification level, the $flushed_i$ signals are not visible. We therefore characterize the $sysflushed$ register by other means. Consider a computation with any schedule and an interval throughout which $sysrq = 1$ is satisfied.

  For any such computation, we require that writing the barrier request register brq_i and reading the system flushed register $sysflushed$ on a processor i is separated

by the execution of a synchronization instruction. This is due to an implementation constraint.

If the register *sysflushed* is active in any step of the interval it stays active until the end of the interval. During that time, no user mode computation steps are executed. The exact time of the activation of the *sysflushed* register is unknown, however, activation of *sysrq* eventually entails activation of *sysflushed*. Hence, no indefinite interval with *sysrq* set but *sysflushed* not being activated may exist.

If there is no system barrier request, the system flushed register reads as zero.

Any attempt to access *brq* and *sysflushed* in user mode causes an illegal instruction exception; the barrier mechanism would *deadlock* in user mode. This completes the description of the new SPRs.

An exemplary procedure to enter the barrier on a processor in system mode requires six instructions. In the first three instructions, we set the barrier request register and issue a synchronization instruction to make sure that we do not read out the system flushed register before the write has completed:

```
R1 := 1
brq := R1
SYNC
```

Then we poll the system flushed register in a loop to wait for its activation:

```
loop:
    R1 := sysflushed
    PC' := (R1 = 0 ? loop : PC' + 4)
    NOP
```

After execution of this loop, the system is in a *flushed* state: (i) Processors in user mode do neither fetch nor execute instructions, (ii) processors in system mode execute normally, and (iii) processors entering user mode are immediately stalled. Several system-mode processors may be in the critical section simultaneously; they are expected to cooperate purely using software. The memory which is sequentially consistent may be used to implement mutual exclusion in various ways.

To leave the barrier, a processor just needs to clear its barrier request register:

```
brq := R0
```

Barriers must be used to protect accesses to translation data (e.g. page tables). As user mode programs cannot use the barrier mechanism, they may directly change such data.[4] We now formulate a software condition called the *translation persistence condition*. The exact form of this condition varies with the consistency mechanism employed by the TLBs (manual or automatic). The single-processor TLB design presented in the last chapter was cleared on a raising mode flank. This is now insufficient; additionally, we flush *all* TLBs on an active *sysflushed* signal.

Consider any computation according to some schedule $sched : \mathbb{N} \to \{1,\ldots,n\}$. Similar to Section 5.1.4, let the predicates $store(s,ma)$ indicate that processor $sched(s)$ writes to memory cell *ma* in step s of the computation. Let the set $T(s)$ denote the set of memory addresses inspected for the translation of a memory operation in step s. The translation persistence condition states that between a store to an address *ma* and

[4] However, the operating system might allow them to do so via system calls.

its use in translation the translating processor must perform a system mode step or the system must be flushed:

$$\forall ma, s_1 < s_3 : store(s_1, ma) \wedge ma \in T(s_3) \Rightarrow$$
$$\exists s_1 < s_2 < s_3 : \neg mode(s_2) \wedge (sched(s_2) = sched(s_3)) \vee sysflushed(s_2) \quad (6.31)$$

From the above formula one can make an important observation: a translation that is used by just one processor may be updated by that processor's interrupt handler without initiating a system barrier.

6.1.4 Code Modification

Like in the single-processor machine the processors here use prefetching to reduce the fetch latency. On architecture level, prefetching requires the code to be synced: after storing into a cell, a synchronization instruction must be issued before fetching from the same cell. For address translation, this was extended in straightforward way to take into account the multiple memory locations which influence instruction fetch.

This uniprocessor criterion can be formulated as a local criterion of the individual processors in our system. However, this only helps to deal with local self-modification as opposed to inter-processor code-modification where other processors write into the instruction stream. For the latter problem, there is no elegant solution; we can do no better than to assume that a processor issues a synchronization instruction before executing code modified by any processor. This reveals the care with which such modifications must be performed: as sequential consistency does not a priori restrict the order in which memory operations are executed, programs have to ensure by additional means (e.g. through semaphores [Dij68], monitors [Hoa74], or generally mutual exclusion [Dij65, Lam74]), that inter-processor code-modification is properly synchronized. For the same reason, special hardware support for inter-processor code modification building on a sequentially consistent cache layer seems infeasible and not worth the effort; if the cache layer provided page locking, the situation would be different.

We define a synced code property similar to that of the machine without virtual memory support. It is important to keep in mind that even the modification of the translation data of a translated fetch constitutes self-modification. However, such updates are already guarded by the translation persistence condition. Therefore, it is sufficient to require that the physical address used to fetch—the translated or the untranslated delayed program counter—is guarded appropriately: this may be done locally be the fetching processor executing a `sync` instruction, an `rfe` instruction, or entering an interrupt handler, or globally, for user code modification, by means of the barrier mechanism. The barrier mechanism may thus also be used for inter-processor code modification of user programs *without* their cooperation. Since the barrier only suspends user activity, this is of no use for system code modification.

Formally, let the predicates $fetch(s, ma)$ indicate that processor $sched(s)$ fetches from the memory cell *ma* in step s of the computation. Let the predicates $sync(s)$, $rfe(s)$ and $jisr(s)$ indicate the execution of a `sync` instruction, an `rfe` instruction, or the detection of an interrupt for processor $sched(s)$ in step s. Also, abbreviate $lsync(s,i) := (sync(s) \vee rfe(s) \vee jisr(s)) \wedge sched(s) = i$. Then, as was noted above, we require that between stores and fetches from the same memory location *ma* a local

Index	Alias	Description
12	*brq*	Barrier request
13	*sysflushed*	System flushed (read-only)

Table 6.2 Indices and Aliases of the New Special-Purpose Registers. Registers $SPR[i]$ with $i \in \{14, 15, 17, \ldots 31\}$ are currently undefined, i.e. behave like $R[0]$ (cf. Table 5.1 in Chapter 5).

synchronization is performed or, for user code modification, the system was flushed:

$$\forall ma, s_1 < s_3 : store(s_1, ma) \wedge fetch(s_3, ma) \Rightarrow \\ \exists s_1 < s_2 < s_3 : lsync(s_2, sched(s_3)) \vee mode(s_3) \wedge sysflushed(s_2) \quad (6.32)$$

For $sched(s_1) = sched(s_3)$, this corresponds to the uniprocessor synced code property and guards against (local) self-modification. Otherwise, it guards against inter-processor code modification; its implications are far-reaching. Since we require the processor whose code has been modified by another processor to execute a synchronization instruction *and* the ordering of memory operations is not a priori known, additional cooperation / synchronization is required between both processors. For example, semaphores can be used to ensure that code modification and code execution of certain regions exclude each other. We remark however, that in operating systems, the capability to modify other tasks' (or even the own task's) code is restricted;[5] some code modifications may only take place, if a task is not running.

6.2 Implementation

Not much is to be said on the implementation of the multiprocessor. The overall datapaths were shown in Figure 6.1. We assume that we already have a sequentially consistent shared memory implementation with $2 \cdot n$ ports.

The processors (with MMUs) can be taken almost unmodified from Chapter 5. For the system barrier mechanism, each processor implements a new special-purpose register *brq* with index 12. It is updated using the `movi2s` instruction like all the other special-purpose registers. The output of this register is fed into the barrier mechanism hardware. Special-purpose register index 13 is reserved for the global system flushed signal *sysflushed*. The processor does not have a corresponding special-purpose register. The signal is taken directly from the barrier mechanism hardware. The system flushed signal may only be read out, writes via `movi2s` have no effect. The register is considered always valid and will, if active, stall user mode fetches. Table 6.2 lists the two new special-purpose registers.

The barrier mechanism hardware itself consists of an or-tree computing the system barrier request signal *sysrq* as in Equation 6.29 and an and-tree computing the system flushed signal *sysflushed* as in Equation 6.30.

The reader may recall that the system barrier is initiated by the requesting processor setting its barrier request signal *brq*. Thereafter, it reads out the system flushed register repeatedly, until it becomes active. Since the system flushed register is always marked as valid in the Tomasulo scheduler, the processor may read it with an instruction I_j before an earlier instruction I_i with $i < j$ has completed its write to the barrier

[5] This is of course security-relevant with regard to malicious code injection.

request register. If an earlier system flush completes between the time that I_i reads out $sysflushed$ and the time I_j sets brq, the process may falsely assume that the system is already flushed (again). Therefore it is necessary to separate these two instructions by a synchronization instruction. This requirement was already noted in Section 6.1.3.

6.3 Correctness

6.3.1 Tomasulo Core with Memory Interface

We develop the specification of a decoupled Tomasulo core which features precise interrupts and writes.

We proceed as follows. We define *scheduling functions* which are used to reason about the implementation by associating register contents with instruction indices. Then, we extend scheduling functions to memory requests which are carried out by the core over a *memory port*. Finally, we present the correctness criteria of the core.

Scheduling Functions

The implementation of the Tomasulo core consists of an issue stage, reservation stations, functional units, producers, the common data bus, the reorder buffer, the register files, and the producer tables (cf. Figure 5.3 in Chapter 5). Its correctness is formulated in terms of scheduling functions [Kro01]. A group of registers k tagged with an index i at time t indicates that this group of registers holds intermediate or final results of instruction i. We write

$$sI(k,t) = i\,. \tag{6.33}$$

The two scheduling functions relating to the core's top and bottom pipeline stages are of special interest: the issue scheduling function indicates which instruction will next start execution and the write-back scheduling function indicates which instructions were already fully processed. Hence, the (open) interval of indices formed by both functions indicates the instructions currently being processed by the core.

On reset, the write-back scheduling function is set to zero and then incremented whenever an instruction is written back. The other scheduling functions which refer to earlier pipeline stages usually run ahead of the write-back scheduling function. On reset or interrupts, they are synchronized with the write-back scheduling function modeling the effects of a core flush with no instruction then being processed by the core. Otherwise, they are updated according to the flow of instructions through the pipeline. If no interrupt is detected at time t, we set $sI(k,t+1) = sI(k',t)$ if stage k is updated from stage k' at time t. This is well-defined since conflicting updates for a register group k never occur. For the isolated Tomasulo core the issue scheduling function has no predecessor; if no interrupt occurs, it is incremented when an instruction is issued, $sI(issue,t+1) = sI(issue,t)+1$. For details, see [Kro01].

Table 6.3 provides an overview of all scheduling functions / all groups of registers in the Tomasulo core. There are scheduling functions for the issue stage, all reservation stations, the inputs to and outputs from each functional unit (for instruction dispatch and the producers), for the common data bus, for each entry tag of the ROB, and for the write-back stage (which holds the register files).

Function	Denotes the instruction...
$sI(issue,t)$	... in the issue stage
$sI(rs_{i,j},t)$	... in the reservation station j of functional unit j
$sI(dispatch_i,t)$	... being input to the functional unit i
$sI(fuout_i,t)$	... in the producer of / being output by the functional unit i
$sI(cdb,t)$	... broadcast on the CDB
$sI(rob_{tag},t)$	... in the ROB entry tag
$sI(wb,t)$	... at the ROB head / writing back to the register files

Table 6.3 Overview of the scheduling functions

Memory Interface

The core performs memory operations (issued on behalf of load / store instructions) over a so-called (data) memory port. This is represented by a trace *port* of memory interface observations. Each observation $port(t)$ consists of control, data input, and data output components. The trace must meet certain handshake conditions (interrupt stableness, mutual exclusiveness of the read and write signal, request liveness) which allow to divide it into memory requests.

A new scheduling function $sI(port,\cdot)$ is introduced to reason about the consistency of memory operations. In addition to several rather technical properties (the scheduling function must be stable over a request, memory phases are bijectively associated with memory operations), we require the core to perform memory operations *in order* and with *precise writes*. The mathematical formulation of all these properties is complicated since we have to take into account interrupts which initiate a rollback and may occur asynchronously to memory operations.

First, we formulate the in-order requirement to memory operations. Basically, this can be reduced to monotonicity of the port scheduling function. However, because in case of interrupts / rollback conditions, the core starts over execution from the instruction in the write-back stage (or that after it), we need to relax monotonicity to intervals of no interrupts. Let $t < t'$ denote two times of ongoing requests, i.e. $port(t).mr \vee port(t).mw$ and $port(t').mr \vee port(t').mw$. We require $sI(port,t) \leq sI(port,t')$ or the existence of a time $t \leq \tilde{t} < t'$ for which an interrupt was detected, i.e. $JISR^{\tilde{t}}$.

Second, we formulate the preciseness of writes. The core may perform a write only if it belongs to the next instruction to retire; thereby no preceding instruction may cause an interrupt anymore. Hence, if at time t we have a write request ($port(t).mw = 1$), we require $sI(port,t) = sI(wb,t) + 1$ and the absence of interrupts throughout the whole request.

The remaining properties are best introduced in the next section.

Consistency

To formulate consistency properties, we define a special, decoupled core computation function $\eta_{c,s}(t_s)$ which takes an additional input $t_s \in \mathbb{N}$ called the split time. All instructions which were written back until the split time are meant to be executed *with* interrupt handling, all later instructions are executed *without* interrupt handling.

In the following, we use the notation X_I^t to denote the contents of register X (visible or hidden with respect to the specification) in cycle t in a computation of a processor. We use the notation X_S^i to denote the value of a component X in step i of a computation

with respect to a certain computation function S (for example, $S = \eta_{c,s}(t_s)$ that we are about to define, or, ultimately for decoupled processor consistency, $S = \eta$).

To define $\eta_{c,s}(t_s)$, we first need to identify the data inputs that the core received until t_s. We have to define three input sequences: the instruction registers $IR(t_s,i)$, the instruction page faults $pff(t_s,i)$, and the data outputs of the memory $dout(t_s,i)$. Basically, each item is associated with a register value or memory response of the implementation by some scheduling function. However, all but the write-back scheduling functions are generally not strictly monotonic and even not injective; they give the same result for different times, if instructions were falsely speculated (here: an interrupt occurred) and have to be retried. The computation of the core at time t_s is based on the most current inputs up to that time. Inputs received after this time may be chosen arbitrarily as they cannot have an influence on the computation (yet).

Instruction registers and page faults are defined with the issue scheduling function:

$$IR(t_s,i) = \begin{cases} IR_I^t & \text{if } t = \max\{t' \le t_s \mid sI(issue,t') = i\} \text{ is defined,} \\ \text{arbitrary} & \text{otherwise.} \end{cases} \tag{6.34}$$

$$pff(t_s,i) = \begin{cases} pff_I^t & \text{if } t = \max\{t' \le t_s \mid sI(issue,t') = i\} \text{ is defined,} \\ \text{arbitrary} & \text{otherwise.} \end{cases} \tag{6.35}$$

Memory outputs are defined with the port scheduling function:

$$dout(t_s,i) = \begin{cases} port(t).dout & \text{if } t = \max\{t' \le t_s \mid \neg port(t').mbusy \wedge sI(port,t') = i\} \\ & \quad \text{is defined,} \\ \text{arbitrary} & \text{otherwise.} \end{cases} \tag{6.36}$$

Now we define $\eta_{c,s}(t_s)$ using the decoupled core computation functions. The split time t_s induces as split instruction $i_s = sI(wb,t_s)$ via the write-back scheduling function. As noted, instructions up to i_s are executed with interrupts, later instructions without. We define $\eta_{c,s}(t_s)$ recursively as follows:

$$\eta_{c,s}(t_s)(i+1,p) = \begin{cases} p & \text{if } i = 0\,, \\ \eta_c(IR(t_s,i), pff(t_s,i), dout(t_s,i), \eta_{c,s}(t_s)(i,p)) & \text{if } i \le sI(wb,t_s)\,, \\ \eta_{c,u}(IR(t_s,i), pff(t_s,i), dout(t_s,i), \eta_{c,s}(t_s)(i,p)) & \text{if } i > sI(wb,t_s)\,. \end{cases} \tag{6.37}$$

We use $\eta_{c,s}$ mainly to express consistency properties for the (regular) registers, the program counters, and the memory operations of the Tomasulo core.

- At time t any regular register is equal to its specified counterpart for the currently retiring instruction. Therefore,

  $$R_I^t = R_S^{sI(wb,t)} \tag{6.38}$$

  holds with respect to $S = \eta_{c,s}(t)$ or, equivalently, with respect to $S = \eta_c$ parameterized over the same input sequences.

- At time t the program counters are equal to their specified counterparts for the currently issuing instruction (assuming that no interrupt occurs from now on).

Therefore the equations

$$DPC_I^t = DPC_S^{sI(issue,t)} \text{ and } PCP_I^t = PCP_S^{sI(issue,t)} \tag{6.39}$$

hold with respect to $S = \eta_{c,s}(t)$. (In the formula above, we have abbreviated $PCP \equiv PC'$ to avoid using double superscripts)

- Consistency of the operations requested through the memory port is slightly more complicated to handle because for stableness reasons (read) memory operations are carried out to completion even if an interrupt occurs. Formally, interrupts alter certain suffixes of a computation $\eta_{c,s}(t)$; consistency of an ongoing memory operation at time t thus needs to be specified with respect to the start time t_0 of the memory request. Hence, the equations

$$port.mop^t = mop_S^{sI(port,t)} \text{ and } port.din^t = din_S^{sI(port,t)} \tag{6.40}$$

for the memory operation and the data input must hold with respect to $S = \eta_{c,s}(t_0)$ (note the parameter t_0). Without interrupts being observed from time t_0 to time t, the function $S = \eta_{c,s}(t)$ may equivalently be taken for specification.

6.3.2 The Memory-Decoupled Processor

In this section we show that the decoupled Tomasulo core is, by connecting it with the fetch mechanism, correct with respect to the memory-decoupled computation function η.

Fetch Scheduling Function

We define a scheduling function $sI(fetch,t)$ to reason about instruction fetch. This scheduling function is initialized to zero and incremented when the instruction register *IR* is updated. On interrupts, the fetch scheduling function takes the (next) value of the write-back scheduling function. Hence, it contains the index of the instruction (not the phase!) which is currently being fetched. We define it by

$$sI(fetch,t+1) = \begin{cases} 0 & \text{if } t = 0 \\ sI(wb,t)+1 & \text{if } JISR^{t+1} \\ sI(fetch,t)+1 & \text{if } \neg JISR^{t+1} \wedge t > 0 \wedge ue_{fetch} \\ sI(fetch,t) & \text{if } \neg JISR^{t+1} \wedge t > 0 \wedge \neg ue_{fetch} \end{cases} \tag{6.41}$$

where ue_{fetch} denotes the update enable signal for the fetch stage; it depends on the actions of the issue stage. If the issue stage is filled without an issue taking place, the fetch stage must not be clocked, since it would overwrite the not-yet issued contents of the instruction register. Hence, if the issue stage is stalled, the VAMP will perform the same fetch operation again. The scheduling function of the issue stage (defined according to Section 6.3.1) closely interacts with the fetch scheduling function. It may be shown that on issue (indicated by ue_{issue}), the former copies its value from the latter and remains unchanged otherwise:

$$sI(issue,t+1) = \begin{cases} sI(fetch,t) & \text{if } ue_{issue}\,, \\ sI(issue,t) & \text{otherwise.} \end{cases} \tag{6.42}$$

Furthermore, by the above property and the definition of the respective update enable signals, one may also derive that the scheduling function of the fetch stage is equal to the scheduling function of the issue stage plus a corrective term of 1 if the decode / issue stage $ID = 1$ is full. We have

$$sI(issue,t) = sI(fetch,t) + S1.full\ . \tag{6.43}$$

Identical properties are given in [Kro01, Bey05].

Consistency

We have already seen that the Tomasulo core guarantees

$$DPC_S^{sI(issue,t)} = DPC_I^t \text{ and } PCP_S^{sI(issue,t)} = PCP_I^t \tag{6.44}$$

for the program counters with respect to the function $S = \eta_{c,s}(t)$. The instruction fetch mechanism uses these to compute the so-called fetch program counter $fetchPC$, which is defined as

$$fetchPC_I^t = \begin{cases} PCP_I^t & \text{if } S1.full, \\ DPC_I^t & \text{otherwise.} \end{cases} \tag{6.45}$$

As justified in [Kro01], an implementation of the above equation is a forwarding circuit for the program counters. To repeat the argument, we use Equation 6.44 and obtain

$$fetchPC_I^t = \begin{cases} PCP_S^{sI(issue,t)} & \text{if } S1.full, \\ DPC_S^{sI(issue,t)} & \text{otherwise.} \end{cases} \tag{6.46}$$

From the specification of the delay program counter, we have $PCP_S^t = DPC_S^{t+1}$. Thus, both cases of the above equation resolve to

$$fetchPC_I^t = DPC_S^{sI(fetch,t)}\ . \tag{6.47}$$

As with decoupled core consistency, we introduce a special computation function $\eta_s(t_s)$ based on a split time t_s. Fetch and execute phases alternate for this function.

We construct the additional inputs to this function similarly to $\eta_{c,s}$. However, only a single sequence of inputs, the memory responses $dout(t_s,i)$ have to be supplied. For odd phases, the data output comes from the data cache (and goes to the core); we set

$$dout(t_s, 2\cdot i+1) = dm(t).dout \tag{6.48}$$

if $t = \max\{t' \leq t_s \mid \neg dm(t').mbusy \wedge sI(port,t') = i\}$ is defined. For even phases the data output comes from the instruction cache and is the result of a fetch; we set

$$dout(t_s, 2\cdot i) = im(t).dout \tag{6.49}$$

if $t = \max\{t' \leq t_s \mid \neg im(t').mbusy \wedge sI(fetch,t') = i\}$ is defined. All other results are arbitrary.

With this input sequence, the function η_s is defined as follows:

$$\eta_s(t_s)(j+1,p) = \begin{cases} \eta_u(dout(t_s,j), \eta_s(t_s)(j,p)) & \text{if } 2\cdot sI(wb,t_s) < j\ , \\ \eta(dout(t_s,j), \eta_s(t_s)(j,p)) & \text{if } 0 < j \leq 2\cdot sI(wb,t_s)\ , \\ p & \text{otherwise.} \end{cases} \tag{6.50}$$

By easy-to-prove properties of the function $\eta_s(t_s)$, not all registers may change in every phase. The fetch flag toggles in each phase and is thus zero in even-numbered configurations and one in odd-numbered configurations. The instruction register may only change after fetch phases and stays unchanged in the successor configuration of any odd-numbered configuration. All other registers may only change after execution phases and stay unchanged in the successor configuration of any even-numbered configuration. Formally for all natural numbers i we have

$$R_S^{2\cdot i+1} = R_S^{2\cdot i} \text{ for any regular register } R\,, \tag{6.51}$$
$$DPC_S^{2\cdot i+1} = DPC_S^{2\cdot i}\,, \tag{6.52}$$
$$PCP_S^{2\cdot i+1} = PCP_S^{2\cdot i}\,, \text{ and} \tag{6.53}$$
$$IR_S^{2\cdot i+2} = IR_S^{2\cdot i+1}\,. \tag{6.54}$$

The functions $\eta_s(t_s)$ and $\eta_{c,s}(t_s)$ are closely related. It may be shown that

$$\eta_s(t_s)(2\cdot i, p) = \eta_{c,s}(t_s)(i,p)\,. \tag{6.55}$$

Hence, the core consistency properties are valid also for $\eta_s(t_s)$. For $S = \eta_s(t_s)$ we have

$$R_I^t = R_S^{2\cdot sI(wb,t)+1} = R_S^{2\cdot sI(wb,t)}, \tag{6.56}$$
$$DPC_I^t = DPC_S^{2\cdot sI(issue,t)+1} = DPC^{2\cdot sI(issue,t)}, \text{ and} \tag{6.57}$$
$$PCP_I^t = PCP_S^{2\cdot sI(issue,t)+1} = PCP^{2\cdot sI(issue,t)} \tag{6.58}$$

by the Tomasulo core correctness criteria (Equations 6.38 and 6.39).

Consistency of the instruction register is derived from fetch correctness. As explained, the fetch scheduling function holds the index of the instruction which is being fetched. Hence, the instruction actually stored in the instruction register *IR* lags behind by one. Therefore, we have

$$IR_I^t = IR_S^{2\cdot sI(fetch,t)-2} = IR_S^{2\cdot sI(fetch,t)-1} \tag{6.59}$$

under the assumption that at least one instruction has been fetched, $sI(fetch,t) \geq 1$.

Until now we have only considered data consistency with respect to the function $\eta_s(t_s)$ parameterized over a split point $t_s \in \mathbb{N}$. Conceptually, for data consistency with respect to the function η we let parameter t_s tend towards infinity.

For clarification of the approach we first examine the traces of memory operation at the instruction and data memory port. We see that at both interfaces certain operations are only *speculated* and do not contribute to a computation with respect to η. The falsely speculated memory operations fall into two classes: fetch operations which are not used due to stall-out conditions of the Tomasulo core and memory operations whose results are not used because an interrupt-condition is detected in the write-back stage. By write preciseness, no falsely speculated memory operation may be a write. This will become important in the next section.

We define a predicate $fspec : [\{I,D\} \times \mathbb{N} \to \mathbb{B}]$ on the memory operations which identifies falsely speculated memory operations.

We set $fspec(I,t_2)$ for a fetch request performed over time interval $[t_0 : t_2]$ iff (i) the Tomasulo core stalls in the same cycle, (ii) an interrupt was detected during the request, or (iii) an interrupt for a logically preceding instruction was detected *after* the request.

Hence,

$$\begin{aligned} fspec(I,t_2) = \quad & S1.stallout^{t_2} \\ & \vee \exists\, t_1 \in [t_0 : t_2] : JISR^{t_1} \\ & \vee \exists\, t_3 > t_2 : JISR^{t_3} \wedge sI(wb,t_3) < sI(fetch,t_2)\ . \end{aligned} \tag{6.60}$$

Similarly, we set $fspec(D,t_2)$ iff (i) an interrupt was detected during the request, or (ii) an interrupt for a logically preceding instruction was detected *after* the request. Hence,

$$\begin{aligned} fspec(D,t_2) = \quad & \exists\, t_1 \in [t_0 : t_2] : JISR^{t_1} \\ & \vee \exists\, t_3 > t_2 : JISR^{t_3} \wedge sI(wb,t_3) < sI(port,t_2)\ . \end{aligned} \tag{6.61}$$

Because of write-preciseness, write operations are never falsely speculated, $fspec(D,t)$ implies $\neg dm.mw^t$.

Now, to construct the inputs for the decoupled computation with respect to η we will for every scheduling index and every port only consider the last / most recent operation. As usual, the inputs for odd phases are fetch results, the inputs for even phases are results of loads or stores. We set

$$\begin{aligned} & dout(2\cdot i) = im(t).dout \\ & \quad \text{for } t = \max\{t' \mid \neg im(t').mbusy \wedge \neg fspec(I,t') \wedge sI(I,t') = i\} \text{ and} \end{aligned} \tag{6.62}$$

$$\begin{aligned} & dout(2\cdot i+1) = dm(t).dout \\ & \quad \text{for } t = \max\{t' \mid \neg dm(t').mbusy \wedge \neg fspec(D,t') \wedge sI(D,t') = i\} \end{aligned} \tag{6.63}$$

if the right-hand sides are defined. Otherwise we set the results to arbitrary values.[6] We remark, that by liveness of the processor, the maxima above are constructed over *finite* sets with the cardinality being bounded by a hardware-dependent constant.

6.3.3 Coupling Processors and Memory

We have already established decoupled correctness for the processor cores. In this section we show that the processors are correctly coupled with the memory under the translation persistence and synchronization property. In order to do this, we have to argue about the different memory interfaces in the machine.

First, we treat a simpler case in detail, assuming no address translation is used / only system mode operations take place. Second, we sketch how to extend this case for the full architecture and the system barrier.

Restatement of the Code Modification Criterion

Consider a concurrent computation with respect to the big step function Δ, a sequence $seq_S : [\mathbb{N} \to \{1,\ldots,n\}]$ and an initial configuration $(p_{1,0},\ldots,p_{n,0},mm_0)$. The *augmented sequence* $seq_S^+ : [\mathbb{N} \to \{1,\ldots,n\} \times \mathbb{B} \times \mathbb{B} \times Mop \times Din' \times Dout']$ redundantly specifies for each step the data transmitted between the processor and the memory and two boolean flags, bit 2 of the delayed PC $dpc2$ (which may be used to identify the instruction word in a double word loaded from main memory) and the flag *sisr*, indicating that an interrupt was detected during that processor's last big step. For $x \in \mathbb{N}$

[6] The maxima may equivalently be constructed over non-speculated memory operations.

and $i = seq_S(x)$, we define

$$seq_S^+(x) = (i, dpc2, jisr(p'), mop, din, dout) \tag{6.64}$$

where p' is the processor configuration for processor i in its last small step before reaching configuration $p_{i,x}$ and

$$dpc2 = dpc(p_{i,x})[2] \wedge (j_{i,x} \text{ even}) , \tag{6.65}$$
$$mop = mop(dec(p_{i,x})) , \tag{6.66}$$
$$din = snd(dec(p_{i,x}))(p_{i,x}), \text{ and} \tag{6.67}$$
$$dout = dout(dec_m(mop)(din, mem_x)) . \tag{6.68}$$

For the simplified version of a processor without translation, the translation persistence criterion is not needed anymore and the barrier mechanism case from the code modification criterion (Equation 6.32) can be dropped. So, for the correct functioning of the prefetch mechanism, we require that stores to a certain memory address *ma* are separated from fetches of the same address *ma* by a local synchronization of the fetching processor. Recall that a local synchronization condition is the execution of a `sync` or an `rfe` instruction or the detection of an interrupt. This condition is, for our processor implementation, equivalent to the *fetch* of a `sync`, an `rfe`, or the first instruction of the ISR.

The code modification criterion is thus expressible in terms of the additional fields of seq_S^+. Let $s_1 < s_3$ and $ma \in \{0, \ldots, 2^{29}-1\}$. Let $e_1 = seq_S^+(s_1)$ and $e_3 = seq_S^+(s_3)$. To access the record components of e_i, $mop(e_i)$, and $din(e_i)$ we use functional notation (cf. Equations 6.2 and 6.3 for the definition of the components); additionally, we abbreviate the address component of the data input by $a(din(e_i)) := addr(din(e_i))$. Operation $seq_S^+(s_1)$ writes to *ma* iff $mw(mop(e_1))$ and $a(din(e_1)) = ma$. Operation $seq_S^+(s_3)$ fetches from *ma* iff $j_{i(e_3),s_3}$ is even and $a(din(e_3)) = ma$. Both conditions form the antecedent of the implication of the code modification criterion; if they hold we require the existence of $s_1 < s_2 < s_3$ witnessing a local synchronization of processor $i(e_3)$. Let $e_2 = seq_S^+(s_2)$ and have $i(e_2) = i(e_3)$. Then, the phase index $j_{i(e_2),s_2}$ must be even and $sisr(e_2) = 1$ or I_2 a `sync` or an `rfe` instruction for

$$I_2 = \begin{cases} dout(e_2)[63:31] & \text{if } dpc2(e_2) , \\ dout(e_2)[31:0]) & \text{otherwise.} \end{cases} \tag{6.69}$$

Processor Interface

Each processor is connected via two ports to the memory, the instruction and the data port. We identify a port in the multiprocessor by a pair $(pty, i) \in Port = \{I, D\} \times \{1, \ldots, n\}$ where $pty \in \{I, D\}$ denotes its type (instruction or data) and $i \in \{1, \ldots, n\}$ its processor index. Without translation, an interface observation of the processor interface is a five-tuple $iobs_p = (req, mop, (a, din), ack, dout) \in Iobs_p$ consisting of

- a request signal $req \in bool$,
- a memory operation $mop = (mr, mw, mbw) \in Mop$ consisting of read $mr \in \mathbb{B}$, write $mw \in \mathbb{B}$, and byte write $mbw \in \mathbb{B}^8$ flags,
- a memory operation's data input, a pair (a, din) of an address $a \in \{0, \ldots, 2^{29}-1\}$ and a double word $din \in \mathbb{B}^{64}$ to be written,

- an acknowledgment signal $ack \in \mathbb{B}$, and
- a data output $dout \in \mathbb{B}^{64}$ (we do not have an exception flag without address translation).

An *augmented* processor interface observation is an eight-tuple

$$(j, dpc2, sisr, req, mop, (a, din), ack, dout) \in Iobs_p^+ = \mathbb{N} \times \mathbb{B} \times \mathbb{B} \times Iobs_p \quad (6.70)$$

with the following additional components:

- The phase index j of the instruction that the interface operation has been performed for.
- Bit 2 of the delayed program counter $dpc2$ for fetches (zero for non-fetches).
- The flag *sisr* indicating the fetch of the first instruction of the ISR after detecting an interrupt condition.

These extensions are in parallel to those of the augmented memory operation sequence seq_S^+, the additional components can be easily provided by a (locally correct) processor.

As usual, we consider traces of interface observations. In particular, an augmented processor interface trace $trc_p^+ : [\mathbb{N} \times Port \to Iobs_p^+]$ is a sequence of augmented interface observations over time and ports. Interface traces need to conform to the regular handshake conditions. Trace indices $e = (t, (pty, i))$ for which the interface observation $trc_p^+(e)$ has a valid acknowledgment are called *event indices* or *events*. For a given event index e with $(j, dpc2, sisr, 1, mop, (a, din), 1, dout) = trc_p^+(e)$, the tuple $(j, dpc2, sisr, mop, (a, din), dout)$ is called the *event data*, *full event*, or *event content* of e.

Potential Implementation Sequences

We define the potential implementation sequences that combine information derived from the processor interface trace and the memory operation sequence. They represent computations that are consistent to the memory operation semantics, correct with respect to the local processor correctness, and do *not* exhibit falsely speculated / rolled-back operations. As we will see, we can derive an initial potential implementation sequence from a multiprocessor computation with locally correct processor cores simply by filtering out the falsely speculated operations. Then, we show that we can construct a concurrent multiprocessor computation from this sequence by inductively transforming it and applying the code modification criterion in its restated form. It is important, that potential implementation sequences capture enough information that such a proof may be conducted.

Consider a function

$$\begin{aligned} seq_I : [\mathbb{N} \to\ & \{1, \ldots, n\} \times \mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{B} \times \\ & Mop \times (\{0, \ldots, 2^{29} - 1\} \times \mathbb{B}^{64}) \times \mathbb{B}^{64}] \,. \end{aligned} \quad (6.71)$$

We define under which conditions we call seq_I a potential implementation sequence. For this purpose, for $s_k \in \mathbb{N}$ abbreviate the results of looking up $seq_I(s_k)$ by

$$seq_I(s_k) = (i_k, j_k, ss_k, dpc2_k, sisr_k, mop_k, (a_k, din_k), dout_k) \,. \quad (6.72)$$

The tuple $seq_I(s_1)$ characterizes the memory operation performed in step $s_1 \in \mathbb{N}$ of the computation. It consists of

- the processor index $i_1 \in \{1,\ldots,n\}$,
- the phase index $j_1 \in \mathbb{N}$,
- the synchronization / store flag $ss_1 \in \mathbb{B}$ that is true for j_1 even (a fetch) if the operation is synced (all preceding memory operations of processor i_1 already being completed) and for j_1 odd if the operation is a store,
- bit 2 of the delayed program counter $dpc2_1$ for fetches,
- the flag $sisr_1$ indicating the execution of the first instruction of the ISR,
- the memory operation mop_1, the data input (a_1, din_1), and the data output $dout_1$.

We call the sequence seq_I a *potential implementation sequence* for an initial configuration $(p_{1,0},\ldots,p_{n,0},mm_0)$ iff (i) its phase indices satisfy certain ordering properties, (ii) the data outputs are consistent with respect to the memory operation semantics, and (iii) phase indices, memory operation inputs, and the flags are consistent with respect to decoupled processor consistency. We define all of these properties in detail.

Phase Ordering Properties. Given two ordered sequences indices $s_1 < s_2$ for the same processor $i_1 = i_2$, the phase ordering properties restrict the relative order of the associated phase indices j_1 and j_2. We have four ordering properties:

- Accesses to the instruction port are in-order for each processor:

$$s_1 < s_2 \wedge (i_1 = i_2) \wedge (j_1 \text{ even}) \wedge (j_2 \text{ even}) \Rightarrow j_1 < j_2 \tag{6.73}$$

- Accesses to the data port are in-order for each processor:

$$s_1 < s_2 \wedge (i_1 = i_2) \wedge (j_1 \text{ odd}) \wedge (j_2 \text{ odd}) \Rightarrow j_1 < j_2 \tag{6.74}$$

- After a load / store only fetches of greater phase index may follow:

$$s_1 < s_2 \wedge (i_1 = i_2) \wedge (j_1 \text{ odd}) \wedge (j_2 \text{ even}) \Rightarrow j_1 < j_2 \tag{6.75}$$

- A synchronization condition, indicated by the flag ss_1 for a fetch operation s_1, separates the phases of its processor:

$$(i_1 = i_2) \wedge ss_1 \wedge (j_1 \text{ even}) \Rightarrow (s_1 < s_2 \Leftrightarrow j_1 < j_2) \tag{6.76}$$

By the phase ordering properties, every pair (i_1, j_1) from the sequence is unique. Note that prefetches ($s_1 < s_2$ with j_1 even, j_2 odd, and $j_1 > j_2$ are not forbidden by the above conditions. This, however, is the only possible mismatch between sequence and local processor order.

Memory Operation Consistency. The memory operations and data inputs of the elements of seq_I induce a memory configuration sequence mem_s and a data output sequence $dout'_s$ via

$$(mem_{s_1+1}, dout'_{s_1}) = dec_m(mop_1)(mem_{s_1}, (a_1, din_1)) \tag{6.77}$$

given the memory's initial configuration mm_0. We require that the data outputs defined thereby are equal to the data outputs specified in seq_I, so $dout'_{s_1} = dout_1$ for all $s_1 \in \mathbb{N}$.

Decoupled Processor Consistency. Finally, we require that the sequence represents the computations of locally correct processors. While decoupled correctness was originally defined in terms of η, it is more conveniently expressed here with the decoupled big step function Γ.

Fix any processor index $i \in \{1,\ldots,n\}$. Let the function $v_i : [\mathbb{N} \to \mathbb{N}]$ enumerate the events of processor i in ascending order of phase indices. Consider the big step computation with initial configuration $p_0^\Gamma = p_{i,0}$ and data outputs $dout_x^\Gamma = dout(seq_I(v_i(x)))$. Let j_x^Γ denote the phase indices and $(mop_x^\Gamma, dout_x^\Gamma)$ denote the memory operation and data inputs in that computation. We demand the following consistency properties:

- For all $x \in \mathbb{N}$ and $s_1 = v_i(x)$ we require the phase index, the memory operation, and the data input components of $seq_I(s_1)$ to equal those of the decoupled computation,

$$(j_1, mop_1, dout_1) = (j_x^\Gamma, mop_x^\Gamma, dout_x^\Gamma)\ . \tag{6.78}$$

- For fetch phases (with j_x^Γ even), we also require that the $dpc2_1$ flag holds bit 2 of the delayed program counter and that the $sisr_1$ flag indicates the detection of an interrupt during the previous big step of processor i. Both flags correspond to those in augmented specification sequences seq_S^+. Let p' denote the last small step processor configuration before p_x^Γ. Then,

$$j_x^\Gamma \text{ even} \Rightarrow (dpc2_1 = dpc(p_x^\Gamma)[2]) \wedge (sisr_1 = jisr(p'))\ . \tag{6.79}$$

- The ss_1 flag is a shorthand identifying a local synchronization condition for fetches and memory writes otherwise. Its definition for fetches is based on the flags $dpc2_1$ and $sisr_1$ that are required to be consistent to the decoupled big step computation. We abbreviate $I_1 = (dpc2_1 ? dout_1[63:31] : dout_1[31:0])$ and set

$$ss_1 = \begin{cases} sisr_1 \vee I_1 \in \{\texttt{rfe}, \texttt{sync}\} & \text{if } j_1 \text{ even,} \\ mw(mop_1) & \text{otherwise.} \end{cases} \tag{6.80}$$

- For prefetches, the processor must guarantee, that the program counter is determined by the initial processor configuration and the memory operation output observed previously *with respect to the sequence position of the prefetch*.

  In other words, prefetch locations must not depend on outstanding load / store operations and must therefore not be speculated. Formally, let $s_1 < s_2$ with $i_1 = i_2$ and have $j_1 > j_2$ with j_1 even and corresponding big step indices x_1 and x_2, so $v_i(x_1) = s_1$ and $v_i(x_2) = s_2$. By definition of v_i, we also have $x_1 > x_2$. Then, we require that

$$dpc(p_{x_1}^\Gamma) \text{ is independent of } (dout_{x_2}^\Gamma, dout_{x_2+1}^\Gamma, \ldots, dout_{x_1-1}^\Gamma)\ . \tag{6.81}$$

  Clearly, this condition can only be satisfied, if falsely speculated operations (in particular fetches) are filtered out from potential implementation sequences; this is exactly what we do later when constructing the (initial) potential implementation sequence seq_I^0 from an interface trace of the hardware.

Dependency Graphs

We define the dependency graph $G = (V,E)$ of a potential implementation sequence seq_I with nodes $V = \mathbb{N}$ and edges $E \subseteq V \times V$. An edge leads from s_1 to s_2 (denoted by $(s_1,s_2) \in E$ or $s_1 \to s_2 \in E$) iff one of the following conditions holds:

1. Both nodes are for the same processor and s_1 belongs to an earlier phase than s_2, so

$$i_1 = i_2 \wedge j_1 < j_2 \; . \tag{6.82}$$

   We call such an edge a *processor-order dependency*. If $s_2 < s_1$, then, because of the ordering properties, j_1 must be even and hence, a *prefetch*.

2. Node s_1 precedes node s_2 and s_1 writes to the address used for memory access by s_2 or vice versa,

$$s_1 < s_2 \wedge (a_1 = a_2) \wedge (((j_1 \text{ odd}) \wedge ss_1) \vee ((j_2 \text{ odd}) \wedge ss_2)) \; . \tag{6.83}$$

   We call such an edge a *data dependency* (or, classically, *conflict* [SS88]). If, additionally, s_1 is a write and s_2 is a fetch, so, overall,

$$s_1 < s_2 \wedge (j_1 \text{ odd}) \wedge ss_1 \wedge (j_2 \text{ even}) \wedge (a_1 = a_2) \; , \tag{6.84}$$

   we call the edge a *data dependency requiring synchronization*.

Edges with $s_1 < s_2$ are called *forward edges* and edges with $s_2 < s_1$ are called backward edges. There are no self-cycles $s_1 \to s_1$. Edges between different processors are called inter-processor edges; they are all data dependencies and point in forward direction. Backward edges correspond to prefetching operations and thus the operation associated with s_1 and s_2 must be a load / store operation and a fetch operation of the same processor. Also note that processor order is already transitively closed.

We will use the regular definitions of paths, successors or predecessors of a node. In addition, we often use these notions with respect to certain boundary nodes $s < s'$. For example, a *local path* may only traverse nodes in these bounds. Local successors and predecessors are defined using local paths instead of regular paths.

◀ Definition 6.1
Equivalence

Two potential implementation sequences seq_I and seq'_I for the same initial configuration that are permutations of each other are called equivalent. *Formally, we require the existence of a permutation of the natural numbers / a bijective function $\pi : [\mathbb{N} \to \mathbb{N}]$ such that $seq'_I = seq_I \circ \pi$. We denote this fact by $seq_I \sim seq'_I$.*

The following lemma states how to equivalently transform a potential implementation using its dependency graph.

◀ Lemma 6.2

Consider a potential implementation sequence seq_I and two sequence indices s and s' with $s < s'$. Assume that no local simple path from s to s' exists in the dependency graph $G = (V,E)$ of seq_I. Then, the permutation π, which moves s and its local successors past s, induces an equivalent potential implementation sequence seq'_I:

- *The local successors of s before s' are given by $S = \{s_k \mid \exists \text{ path } (s_1,\ldots,s_k) \text{ in } G \wedge s_1 = s \wedge s_i < s'\}$. By assumption $s'' \to s' \notin E$ for all $s'' \in S$.*
- *Let $s_1 < \ldots < s_k$ enumerate the elements of S. Let $s'_1 < \ldots < s'_{s'-k}$ enumerate all other elements before s'.*

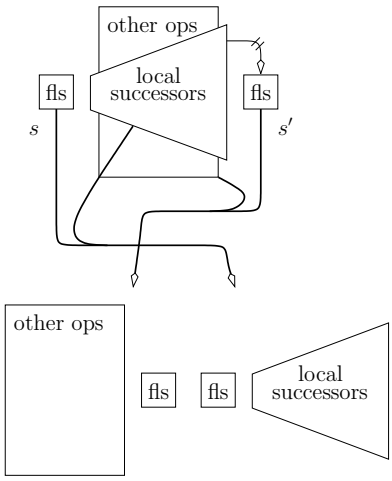

Figure 6.2 Reordering of a Potential Implementation Sequence According to Lemma 6.2. A box labeled 'fls' may be a fetch, a load, or a store operation. The elements left of s and right of s' do not change their position.

- *Define* π *by*

$$\pi(t) = \begin{cases} s'_{t+1} & \text{if } t < s' - k \text{ ;} \\ s' & \text{if } t = s' - k \text{ ;} \\ s_{t-(s'-k)} & \text{if } t > s' - k \wedge t < s' + 1 \text{ ;} \\ t & \text{otherwise.} \end{cases} \tag{6.85}$$

- *The sequence* $seq'_I = seq_I \circ \pi$ *is a potential implementation sequence and equivalent to* seq_I.

The proof is straightforward and not given here. The permutation does not reverse forward edges but may reverse backward edges. Therefore, the ordering properties still hold for seq'_I. Additionally, because of the data dependencies, the permutation preserves the history of write operations on each cell and the history of memory responses seen be each processor.

Figure 6.2 illustrates the definition of π. The permutation can be written more succinctly using the notation $(y_1, y_2, \ldots)$ to define sequences and an implicit conversion of sets (such as S) to sequences by ordering its elements:

$$\begin{aligned} \pi &= ([0 : s' - 1] \setminus S, s', S, [s' + 1 : \infty]) \\ &= ([0 : s'] \setminus S, s', S, [s' + 1 : \infty]) \end{aligned} \tag{6.86}$$

Defining an Initial Potential Implementation Sequence

In this section we show how to derive a potential implementation sequence from a multiprocessor computation. We assume the memory to be sequentially consistent and the processors to be locally correct. The essential part about this is to ignore any speculated memory operations observed at the processor interface.

The memory guarantees sequential consistency, i.e. for each trace trc_p we have a sequence $seq_p : [\mathbb{N} \to \mathbb{N} \times Port]$ enumerating event indices of a trace trc_p. The

sequence represents the (total) order of memory operations. Several restrictions must hold for seq_p, e.g. the order of operations must conform to the interfaces traces.

From trc_p we derive an augmented processor interface trace trc_p^+ with the additional information (phase indices, $dpc2$ and *sisr* flags) supplied by the processor.[7]

Ultimately, we want to construct a schedule for the specification that conforms to the observed event data of the trace. Apart from the structural difference of having twice as many memory ports, the implementation differs from the specification in two more important aspects: we may observe falsely speculated memory operations and prefetching in the interface trace.

Falsely speculated memory operations may be safely filtered out since they are never speculated writes, as guaranteed by the processor. Then, the main problem that remains is to show that prefetches return the same result as in some equivalent sequential execution.

We define now seq_I^0, the initial potential implementation sequence. For any event index (pty,i) in trc_p^+, let the predicate $fspec(t,(pty,i))$ indicate that the operation of processor i at its port pty at time t was falsely speculated (cf. the definition of $fspec$ at the end of Section 6.3.2). Let the function $filter : [\mathbb{N} \to \{s | \neg fspec(seq_p(s))\}]$ enumerate the elements of its domain, the sequence indices of correctly speculated memory operations, in ascending order. Abbreviate

$$seq_p(filter(s)) = (t,(pty,i)) \text{ and} \tag{6.87}$$

$$trc_p^+(t,(pty,i)) = (j,dpc2,sisr,1,mop,(a,din),1,dout)\ . \tag{6.88}$$

Define the flag $ss \in \mathbb{B}$ as true, if j is even and *sisr* or a `sync` or `rfe` instruction was fetched or j is odd and a write operation. Then, we set

$$seq_I^0(s) = (i,j,ss,dpc2,sisr,mop,(a,din),dout)\ . \tag{6.89}$$

The sequence seq_I^0 is a potential implementation sequence. ◀ Lemma 6.3

Again, the proof is straightforward and follows from decoupled processor correctness and sequential consistency of the memory. For example, to prove data consistency for seq_I^0 we use the latter fact and that, by decoupled processor correctness, falsely speculated operations are never writes; hence, the deletion of falsely speculated memory operations in seq_I^0 does not harm memory operation consistency.

Sequentialization

To show that seq_I^0 is equivalent to a computation with respect to the concurrent execution semantics we will reorder it by applying Lemma 6.2 repeatedly, thus removing prefetches while retaining equivalence. The code modification criterion is needed to show that the assumptions of this lemma are met for already sequentialized parts of the computation.

It is important to keep account of the fetch operations that appear out of processor order in a potential implementation sequence. We call those operations *unresolved prefetches* and define them with respect to the decoupled processor computation.

Consider a potential implementation sequence seq_I with respect to a starting configu- ◀ Definition 6.2 *Unresolved Prefetch*

[7] The processor correctness statement only gives us scheduling indices for each event at the instruction and the data port. However, for the instruction port, the phase index may be obtained by doubling the scheduling index; for the data port, it may be obtained by doubling the scheduling index and adding one.

ration $(p_{1,0},\ldots,p_{n,0},mm_0)$. *Let* $v_i : [\mathbb{N} \to \mathbb{N}]$ *enumerate the phases of processor* i *in ascending order, as in the definition of potential implementation sequences.*

A sequence index s_1 *is called* unresolved *up to a boundary* $s \geq s_1$ *iff, for* x_1 *with* $v_{i_1}(x_1) = s_1$, *there exists* $x_2 < x_1$ *such that* $v_{i_1}(x_2) > s$. *Otherwise,* s_1 *is called* resolved.

Since for a potential implementation sequence, s_1 *needs to denote a fetch operation to be unresolved, we also speak of* s_1 *being an* unresolved prefetch *up to* s. *We abbreviate this condition by* $upf(s,s_1)$.

By the effects that a synchronization condition has, a synchronization is never an unresolved prefetch. Once a fetch s_1 is unresolved with respect to a certain boundary s, all subsequent fetches before that boundary will also be unresolved, so, if $upf(s,s_1)$ than $upf(s,s_2)$ for any $s_1 \leq s_2 \leq s$ with $i_2 = i_1$ and j_2 even holds.

Because of decoupled processor consistency, unresolved prefetches must have witness in seq_I, i.e. operations that indicate their presence. Load / store operations $s_1 < s_2 \leq s$ for processor $i_2 = i_1$ satisfy $j_2 < j_1$ and are called *present* loads / stores with respect to s_1 and s. Load / store operations $s_2 > s$ for processor $i_2 = i_1$ with $j_2 < j_1$ are called *outstanding* loads / stores with respect to s_1 and s. In the dependency graph, unresolved prefetches are witnessed by backward edges $s_2 \to s_1$ crossing the boundary s.

We now formulate the central reordering theorem.

Theorem 6.4 ▶ *For all* $s \in \mathbb{N}$ *there exists a permutation* π^s *such that*

1. $\pi^s(s') = s'$ *for all* $s' > s$,
2. $seq_I^s = seq_I \circ \pi^s$ *is potential implementation sequence equivalent to* seq_I^0,
3. *in the dependency graph* $G^s = (V,E^s)$ *induced by* seq_I^s *there is no backward edge up to index* s, *so for all* $s_1 < s_2 \leq s$ *we have* $s_2 \to s_1 \notin E^s$.

Before we prove the theorem, we want to make some preliminary remarks. Consider some $s \in \mathbb{N}$, the sequence seq_I^s and its dependency graph G^s with the properties claimed in the theorem. Let upf^s denote the unresolved prefetch predicate for seq_I^s. In the first s operations all backward edges are already removed according to Condition 3. Therefore, there are no 'present' loads / stores with respect to an unresolved prefetch at position $s_1 < s$. For any processor i_1 with a minimal s_1 with $upf^s(s,s_1)$, only more unresolved prefetches may follow for the same processor between $s_1 + 1$ and s.

If we disregard all unresolved prefetches up to s, we may construct the prefix of a concurrent computation because data consistency holds for seq_I^s (due to Condition 2) and fetches do not change the memory configuration.

Clearly, the theorem will be proven by induction over s, starting with seq_I^0, a potential implementation sequence derived from the actual execution of the multiprocessor. In the induction step, we remove newly found backward edges by applying Lemma 6.2. We use the code modification criterion (for the sequentialized part of the computation) and the induction hypothesis to show that the assumptions of this lemma are met.

Then, we still need to show that every operation will eventually be resolved and stay resolved; this proves that the sequences seq_I^s converges and thus that seq_I^0 is sequentializable.

PROOF We prove the claim by induction over s. For the induction start $s = 0$ observe that the first operation requested and acknowledged after system reset must be an instruction fetch since all processors start in system mode and must complete their first instruction fetch before performing any data accesses. Of course, this operation is not an (unresolved) prefetch. With $\pi^0(s) = s$ all necessary properties follow easily.

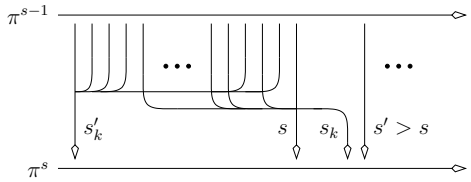

Figure 6.3 Definition of the New Permutation π^s with Operation s Witnessing Prefetching

We make the induction step from $s-1$ to s. Assume that the claim is correct for $s-1 \geq 0$. If operation s is not an outstanding load / store for an unresolved prefetch prior to s with respect to G^{s-1}, we keep the last permutation and set $\pi^s = \pi^{s-1}$. All necessary claims follow from the induction hypothesis.

Otherwise, we construct π^s from π^{s-1} as follows. Let $s^f_1 < \ldots < s^f_{k^f}$ enumerate all unresolved prefetches of s. We have backward edges $s \to s^f_i \in E^{s-1}$ and $upf^{s-1}(s-1, s_1)$. Furthermore, none of these fetches is locally synchronizing. Let

$$S' = \{y'_{k'} \mid \exists \text{ path } (y'_1, \ldots, y'_{k'}) \text{ in } G^{s-1} \wedge y'_1 = s^f_1 \wedge \forall i : y'_i < s\} \tag{6.90}$$

denote the set of local successors of s^f_1. Enumerate the elements $s_1 < \ldots < s_k$ of S'. By processor order and Condition 3 of the claim, we have $s^f_i \in S'$ and $s_1 = s^f_1$. For the new permutation π^s we move the elements in S' past s, preserving their order and the order elsewhere (cf. Lemma 6.2). Enumerate all elements $s'_1 < \ldots < s'_{s-k}$ prior to s and not in S'. So,

$$\{0, \ldots, s-1\} = \{s_1, \ldots, s_k\} \uplus \{s'_1, \ldots, s'_{s-k}\} \,. \tag{6.91}$$

Then, we define

$$\pi^s(t) = \begin{cases} \pi^{s-1}(s'_{t+1}) & \text{if } t < s-k \,; \\ \pi^{s-1}(s) & \text{if } t = s-k \,; \\ \pi^{s-1}(s_{t-(s-k)}) & \text{if } t > s-k \wedge t < s+1 \,; \\ \pi^{s-1}(t) & \text{otherwise.} \end{cases} \tag{6.92}$$

Figure 6.3 depicts the newly defined permutation. In alternative functional notation we may write π^s also as

$$\begin{aligned} \pi^s &= \pi^{s-1} \circ (\, s'_1, \ldots, s'_{s-k}, \quad s, s_1, \ldots, s_k, s+1, s+2, \ldots \,) \\ &= \pi^{s-1} \circ (\, [0 : s-1] \setminus S, s, S, \qquad\quad [s+1 : \infty] \,) \end{aligned} \tag{6.93}$$

Clearly, by the induction hypothesis and its definition, the new permutation π^s satisfies Condition 1 and 3 of the claim. We show that Condition 2 holds by proving equivalence of seq^s_I to $seq^{s-1}_I \sim seq^0_I$ with Lemma 6.2 applied for s_1 and s. The assumptions of the lemma require that there is no local path from s_1 to s. Let us assume otherwise for the purpose of constructing a contradiction. Observe that no local path from s_1 to s may exclusively visit nodes of processor i_1: if so, we would have a data dependency from a fetch s^f_i to the (store!) operation s_1, which would be in violation of the code modification criterion.[8]

[8] Technically, this proof has to be conducted in the same way as the one that follows.

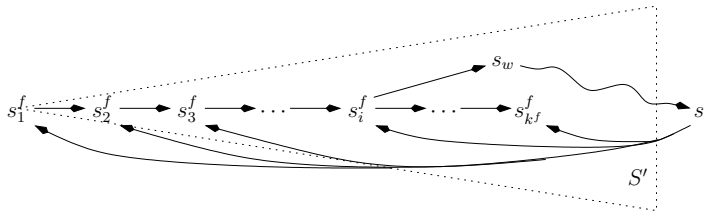

Figure 6.4 An Alleged Local Path p from s_1^f to s

Hence, all local paths have an edge $s_i^f \rightarrow s_w$ from an unresolved prefetch s_i^f to a write operation $s_w < s$ on a different processor. Let us consider the path p with an edge $s_i^f \rightarrow s_w$ of the above type such that (i) s_w is minimal and (ii) for this s_w, the operation s_i^f is also minimal. If we include every unresolved prefetch up to s_i^f in that path, it has the form $p = (s_1^f, s_2^f, \ldots, s_i^f, s_w, \ldots, s)$; this situation is depicted in Figure 6.4. Of course, all edges in that path are forward edges. We disprove the existence of p by constructing a computation in which s_i^f and s_w swap positions and all outstanding loads / stores up to s_i^f are inserted into the computation in processor order. More precisely, in the new computation, we perform the operations of seq_I^{s-1} in the following order:

1. Operations prior to s_1^f,
2. the local predecessors of and including s_w other than the operations s_j^f,
3. the remaining operations up to s_i^f, with the outstanding loads / stores for the operations s_j^f inserted in processor order into the computation, and
4. the remaining operations following s_i^f.

Formally, let $Pred(s)$ denote the predecessors of an operation s in G^{s-1} (including s). Let $OLS(s, s_j^f)$ denote the outstanding loads / stores for the unresolved prefetch s_j^f *without* outstanding loads / stores for previous unresolved prefetches $s_{j'}^f$ with $j' < j$. We define the permutation π', which reorders the operations of seq_I^{s-1} in the manner defined above. In the formula below, sets are meant to be implicitly converted to sequences by ordering them ascendingly:

$$\begin{aligned}
\pi' = (\ & [0 : s_1^f - 1], && \text{(Case 1)} \\
& Pred(s_w) \cap [s_1^f : s_w] \setminus \textstyle\bigcup_{j \le kf} \{s_j^f\}, && \text{(Case 2)} \\
& OLS(s, s_1^f),\ [s_1^f : s_2^f - 1] \setminus Pred(s_w), && \text{(Case 3; start)} \\
& OLS(s, s_2^f),\ [s_2^f : s_3^f - 1] \setminus Pred(s_w), \\
& \ldots, \\
& OLS(s, s_{i-1}^f),\ [s_{i-1}^f : s_i^f - 1] \setminus Pred(s_w), \\
& OLS(s, s_i^f),\ s_i^f && \text{(Case 3; end)} \\
& [s_i^f + 1 : \infty] \setminus Pred(s_w) \setminus \textstyle\bigcup_{j \le i} OLS(s, s_j^f)\ ) && \text{(Case 4)}
\end{aligned} \tag{6.94}$$

Consider $seq_I' = seq_I^{s-1} \circ \pi'$. Let us ignore temporarily, that for the inserted outstanding loads / stores and for the operations after $\pi'(s_i^f)$, the new position of operation s_i^f, this sequence need not be data-consistent anymore.

We observe that

- the operations leading up to $\pi'(s_w)$ in seq'_I are equivalent to the associated operations in seq_I^{s-1} by the choice of s_i^f and s_w;
- all the fetches $\pi'(s_j^f)$ for $j \leq i$ still fetch from their old locations, since the fetch address is guaranteed to be independent of outstanding loads / stores by Equation 6.81 for the potential implementation sequence seq_I^{s-1};
- none of the inserted outstanding loads / stores $s_o \in \pi'(\bigcup_{j \leq i} OLS(s, s_j^f))$ may write to the location of any of the fetches $\pi'(s_j^f)$ for $j \leq i$, because, as may be shown inductively for the outstanding loads / stores, that would be in violation of the code modification criterion;
- hence, all the fetches $\pi'(s_j^f)$ for $j \leq i$ remain non-synchronizing.

Therefore, there must still be a data dependency between $\pi'(s_w)$ and $\pi'(s_i^f)$. Since $\pi'(s_w) < \pi'(s_i^f)$ this is a data dependency requiring synchronization. However, from the last observation, we get that the required synchronization is missing. This leads to a violation of the code modification criterion.

Technically, to establish this, we have to fix up all the events in seq'_I up to position $\pi'(s_i^f)$ by applying Equation 6.77 to ensure that seq'_I is data-consistent up to this position. This does not interfere with the observations made above. Then, by filtering out the prefetches for other processors up to position $\pi'(s_i^f)$, we obtain the prefix of a concurrent computation that witnesses a violation of the code modification criterion. As this should universally hold, this contradicts the existence of the constructed computation and the assumption. Hence, there is no local path from s_1 to s and $seq_I^s \sim seq_I^{s-1}$ by Lemma 6.2.

In the proof of the theorem we have seen that any resolved operation will stay resolved. In the following we show that any unresolved prefetch will eventually be resolved.

No prefetch is forever unresolved in the sequences seq_I^s. ◀ Lemma 6.5

PROOF

First, for the any real design, the number of outstanding loads / stores for prefetches for any processor is bound by the maximum number of instructions in flight, i.e., in our case the size of the reorder buffer. Second, we have seen that in the induction step of the proof of the theorem the operations s and s_1^f swap their position. Therefore, the number of the outstanding load / store operations of s_1^f decreases by one and must eventually reach zero.

Sequentialization with Address Translation

We sketch the extensions that would be needed for the correctness proof for a multiprocessor with address translation. In this case, we have an additional interface at the lowest level: the cache interface. It is accessed by the MMUs on behalf of the processors. The MMUs in turn are accessed over the processor interface with an extended signature (translation flag and registers as additional inputs, translation exception as an additional output). The MMUs guarantee to perform untranslated and translated memory operations if the memory cells they access for a certain request stay unchanged

for the remainder of the request. As such, MMUs combine both the function of the translator and Bridge 2 (cf. Sections 4.5 and 5.2.2).

Hence, verification starts out with a cache interface trace $trc_c : [\mathbb{N} \to \mathbb{N} \times Iobs_c]$ for which the memory supplies an execution order $seq_c : [\mathbb{N} \to \mathbb{N} \times (\{I,D\} \times \{1,\ldots,n\})]$ mapping sequence indices to event indices. From the sequence seq_c we *construct* the sequence

$$seq_p : [\mathbb{N} \to \mathbb{N} \times (\{I,D\} \times \{1,\ldots,n\})] \tag{6.95}$$

of operations at the processor interface. If we neglect TLBs for simplicity, each processor operation is associated with a non-empty sequence of cache operations issued by the MMU. We define seq_p to preserve the order of the last one of these cache operations for any processor operation (these are the processor-type operations of Chapter 4).

In the same manner as before, we may derive a potential implementation seq_I^0 from trc_p and seq_p by filtering out falsely speculated operations. In contrast to the simple case, it is not known whether the data outputs $dout$ of seq_I^0 are equal to the outputs $dout'_s$ of an induced memory configuration / data output sequence. Therefore, in the reordering theorem we require data consistency with respect to complex memory operations only for resolved operations prior to s. In the induction step it must be shown that any newly resolved operations (operation s or newly resolved prefetches prior to s) are consistent, i.e. the assumptions of the MMU on the cache memory are met. This is only true under the translation persistence and the code modification criterion. As part of the proof it must be shown that intervals in which the system is flushed do not overlap with intervals in which an MMU performs a translated operation at the cache interface. To conduct this proof, potential implementation sequences must be extended with additional components, the translation flag t, the sequence of memory addresses T inspected for translation, and the translation exception flag.

6.4 Related Work

For related work on the VAMP architecture, we refer the reader to Section 5.5. Let us turn to TLB consistency. Multiprocessor TLB consistency is significantly more difficult to maintain than single-processor TLB consistency [Tel90]. The operation that deletes stale entries from other processors' TLBs is referred to as "TLB shoot-down". Vahalia [Vah96] contains a concise overview of the various software and hardware algorithms implementing TLB shoot-down. Without special hardware support, kernels make use of inter-processor interrupts, also called cross-processor interrupts, to perform consistent updates of the page tables. An example for such an algorithm is the Mach software TLB shoot-down algorithm [BRGH89]. Via IPIs, it implements a barrier mechanism in software. The interrupted processors ('responders') are notified of the required TLB flushes via messages in a special message queue. As Vahalia mentions, the responders must wait in a busy loop for the initiator of the shoot-down to update the page table before flushing its TLB; in addition to consistent TLBs, this also guarantees translation persistence (cf. Section 4.6.2). A faster algorithm, developed by Rosenburg et al. [Ros89], uses atomic update operations ("fetch&op") and software-locked updates of page tables (assuming page table are not updated architecturally, as is the case for hardware reference and dirty bits). Although the algorithm has user-visible consistency issues, the authors suggest that "operating system semantics should make the behavior of such activity [leading to inconsistencies] explicitly undefined."

For example, the authors state that "accessing a shared region while another thread is changing its mapping" may not a useful activity. We believe that this remark is valid and underlines the importance for a multiprocessor operating system semantics. An example of a hardware mechanisms that helps ensure TLB consistency is the architectured 'invalidate PTE' (IPTE) instruction of the S/390 architecture. As the principles of operation [IBM00] define, this instruction invalidates the designated page table entry in memory and removes it from all TLBs in the system.

Section 6.3.3 on coupling the processors with the memory is strongly related to work on weak memory models. For an overview on these consider [AG95, Gop04, SN04]. Although some of the work (e.g. [GMG91]) is concerned with establishing sequential consistency for weak memory models by means of software conditions on the use of synchronization instructions and, moreover, we seemingly consider a relatively simple case with a particularly strong software condition, we found it hard to adapt these approaches. In constructing the correctness proof (in particular of Theorem 6.4), the most difficult part was not to prove the actual sequentialization but (i) to construct and relate the assumptions on decoupled processor correctness to the sequentialization process and (ii) to apply the code modification criterion in an inductive argument for 'already sequentialized' parts of the computation.

The correctness proof for the multiprocessor we gave was based on a sequentially consistent cache. For blocking caches, as considered here, sequential consistency is equivalent to cache coherence (per-address sequential consistency). Many results have been reported on formal cache coherence protocol verification (cf. the survey of Pong and Dubois [PD97]). These are either carried out using theorem provers [LD91, PD96] or, for systems with a known numbers of nodes, with model checkers [PD97, Cho04]. As of late, fully automatic parameterized cache protocol verification has being tackled [EK03, CMP04] (based on certain, not mechanically formalized meta theorems). Yet, *gate-level* cache design verifications are few and only partially done or restricted in the number of participating nodes [Eir98, Bey05]. The gate-level verification of a parameterized cache protocol (in terms of cache coherence) remains a prominent research problem [PD97, Cho04]. We know of no mechanical proof that assumes such a result in order to establish correctness of the memory units of a multiprocessors in terms of a memory consistency model (stronger than cache coherence).

Chapter 7

An Exemplary Page Fault Handler

Contents

Section 7.1 introduces an exemplary page fault handler for which we show correctness in terms of a virtual memory simulation theorem in Section 7.2. In Section 7.3 we point out performance- and verification-related extensions.

7.1 Software

In this section we present a minimal operating system (OS) framework including a page fault handler. The OS framework provides initialization after reset, I/O functions, and an interrupt service routine (ISR). The ISR handles saving and restoration of (user)

processor configurations and dispatches the different interrupts to specific routines. One of these routines is the page fault handler, which we present in detail.

Concrete (assembler) code is given for almost all routines. Doing so, we may reason in detail on the execution path of a page fault handler call from the page-faulting instruction until its repetition.

We proceed as follows. In Section 7.1.1 we present the layout of the data structures and code in physical memory. In Section 7.1.2 we describe all data structures together with invariants and operations on them. Finally, Section 7.1.3 provides a detailed treatment of the OS and page fault handler code.

7.1.1 Overview of the Memory Map

The operating system enforces a memory organization on user programs which strictly separates memory used for OS purposes from user-accessible memory. We call the latter the *user memory* and the former—all the memory remaining—the *(operating) system memory*.

As shown in Figure 7.1, the main memory is divided on a more fine-grained level into the following sections:

- The first page (the *zero page*) of the physical memory is reserved for special use. In particular, the VAMP's the "host / external memory interface" [Mey02, Bey05], which we abstract from in this thesis, are controlled through memory cells in it. Moreover, the interrupt entry point (*SISR*) is at address 0. In order not to mix I/O ports with ISR code we will just place a jump to address 4096 at the start of the memory.

- The *system code* starts at address 4096; it consists of the interrupt service routine (entry & exit), the handlers for reset, page fault on fetch, and page fault on load / store. All the other interrupts will not be handled. The page fault on fetch and page fault on load / store handlers call the more general page fault handler function pfh, which takes up the rest of the code space.

- The *system data* section starts directly after the system code. The three most important data structures it contains are

  - the task control blocks (TCBs) holding user-visible and auxiliary configuration information for all active tasks,
  - the user memory page management (UMPM) controlling memory allocation of the user memory (see below), and
  - the page table space (PT space), which contains one (page-aligned) page table for each active task.

- The *user memory* (UM) is located at the high end of the physical memory. It starts with the first page $fuppx$ after the page table space and continues to the last page $luppx$ of the physical memory—8191 in our current configuration. The user memory is used as a cache for virtual pages. Its size may vary with $fuppx$ being adjusted dynamically. The size of the page table space is determined by the number of active tasks and their memory use.

  Let us briefly describe here the algorithm used by the page fault handler to allocate pages in the user memory. Pages of the user memory are managed with two

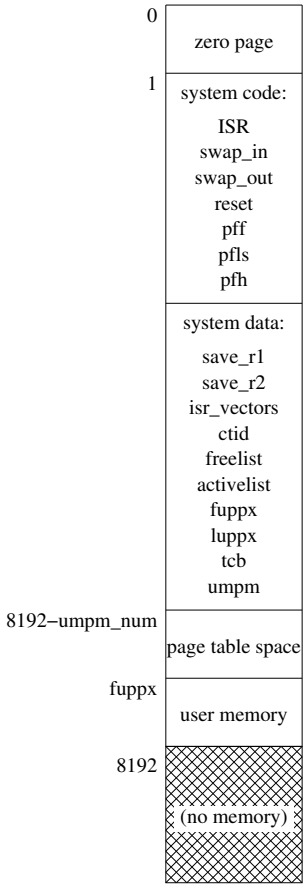

Figure 7.1 Overview of the Memory Map with Addresses Expressed in Physical Page Indices. The current VAMP configuration has $32M = 2^{13} \cdot 2^{12}$ bytes of physical memory. The index of the last accessible page is 8191.

lists, the free and the active list. Elements in the free list correspond to physical pages in user memory that are currently not used by any task. At start-up, the free list contains all user memory pages. Complementary, elements of the active list correspond to the allocated user pages. Every element in the active list is associated with a physical page index *ppx*, a task index *tid*, and a virtual page index *vpx*. We will see that for such an element in the active list, the page table entry *vpx* of task *tid* is valid and points to *ppx*.

The order of elements in the active list is significant, in fact, it is used as a queue: elements are created at its tail on swap-in, and they are taken away from its head on swap-out. Hence, the page fault handler implements a FIFO scheme, that always selects the oldest virtual page in user memory for eviction (swap-out). In Section 7.3 we present extensions to the algorithm.

7.1.2 Data Structures

This section presents the data structures maintained by the page fault handler and the interrupt service routine. The descriptions are interspersed with code examples illustrating typical operations on the data structures. All code in the remainder of this chapter is written in register-transfer language (RTL) instead of pure VAMP assembler language to simplify understanding. The most common RTL instruction type has the form

```
C := A ∘ B
```

and denotes the application of the operation $\circ$ on the operands `A` and `B` with the result being assigned to `C`. As operands, we use constants `a`, the general-purpose registers `R0` to `R31`, the program counter `PC'`, the special-purpose registers, and memory operands of the form M_4`[Rx+a]`. Operations include addition, subtraction, binary negation, conjunction, disjunction, and logical shifts. Constants `a` must be from the range $\{-2^{16},\ldots,2^{15}-1\}$, i.e. be representable as a bit vector of length 16 interpreted as a two's complement number. The special instructions for synchronization `SYNC`, return-from-exception `RFE`, and no-operation `NOP` $\equiv R0 := R0 + 0$ are used verbatim in RTL. Unindented words trailed by a colon as in

```
forever:
    PC' := (R0 = 0 ? forever : PC' + 4)
    NOP
```

are called *labels*; they are identified with the memory location of the succeeding program line and can be used as immediate operands or jump targets in expressions. As can be seen, the ternary operator `s?a:b` is used for branches and comparisons, and, to ease readability, jump targets are specified with absolute addresses, though the corresponding VAMP instruction may require a relative offset.

In all cases, an RTL instruction corresponds to a single instruction of the VAMP.

Variables

The system code maintains the following miscellaneous variables:

- In user mode, the current task identifier *ctid* holds the task identifier of the (active) task currently executing. In system mode, it holds the task identifier of the task most recently been executing or, if it has been updated, the task meant to be executed after return to user mode.
- We have seen that the user memory is placed at the high end of the physical memory. We keep the index of its first page $fuppx$ and its last page $luppx$ in variables. The latter is set according to the available physical memory to 8191 and will not be changed.[1] The former, though, is updated dynamically to adapt to the varying memory needs of the operating system's data structures in particular the page table space. It is may be increased with task creation and memory allocation and may be decreased with task destruction and memory deallocation.[2] Shrinking the user memory typically requires swapping out some

[1] There are systems like [FHPR01], that performs physical memory compression in hardware, for which this does not hold and $luppx$ needs dynamic adjustment.

[2] Note that due to the single-level address translation mechanism, there is a non-trivial allocation problem associated with growing page tables; these effects have been studied in Denning's classical paper [Den70]. The allocation problem gets simplifies significantly with multi-level address translation (or, in other terminology, the combination of segmentation and paging techniques).

user pages and thus may be computationally expensive. We do not describe such operations in detail.

- The entry part of the ISR uses the variables *save_r1* and *save_r2* to temporarily buffer registers $R1$ and $R2$ of the interrupted user task before they are properly saved into a higher-level data structure. Likewise, the exit part of the ISR uses them for restoration purposes.

  Both variables must be placed in the first $32K$ of the memory. Thereby, they are directly addressable using base register $R0$.

- The interrupt vectors table *isr_vectors* holds the start address of the interrupt-specific parts of the ISR for each interrupt level. To allow direct addressing—here: only for convenience—it is also placed in the first $32K$ of the physical memory.

Variables can be accessed in two ways. If their address is representable by a 16-bit sign-extended immediate constant one may use *direct addressing* using $R0$ as a base register and the address as an offset. Otherwise, the variable's address must be loaded into some register, which is subsequently used as a base register with an index of 0.

Loading a 32-bit constant into a register generally requires two instructions; the immediate constant of most instructions is 16 bits wide and sign-extended. Consider an address $a \in \mathbb{B}^{32}$. For $a' = a[31:16] \oplus a[15]^{16}$ the RTL instructions (with sign extension made explicit)

```
R1 := a' ≪ 16
R1 := R1 ⊕ (a[15]^16,a[15:0])
```

load a to register $R1$: for $j > 15$ we have $R1[j] = a'[j-16] = a[j] \oplus a[15]$ after the first and $R1[j] = (a[j] \oplus a[15]) \oplus a[15] = a[j]$ after the second instruction. The RTL instructions correspond directly to the VAMP instructions `lhi` (load high) and `xori` (exclusive or with an immediate). For readability, we define the functions $hi(a) = a[31:16] \oplus a[15]^{16}$ and $lo(a) = (a[15]^{16}, a[15:0])$ and write this code as follows:

```
R1 := hi(a) ≪ 16
R1 := R1 ⊕ lo(a)
```

In the following we identify a variable name with its address. For example, we will use the above code fragment with $a = ctid$ to load the $ctid$ variable into register $R1$.

Task Control Blocks

Structure. User-visible and system information of tasks is kept in a table of task control blocks (TCBs). The table has a fixed size of $task\_num = 128 = 2^7$ entries which is also the maximum number of supported tasks. The task control block for the task $tid \in \mathbb{B}^7$ is a structure with the following components:

- The variable $state \in \mathbb{B}^{32}$ has non-zero value iff task tid is active. If it is zero the other components are ignored.

- The page table origin $pto \in \mathbb{B}^{32}$ and the page table length $ptl \in \mathbb{B}^{32}$ specify placement and last index of the page table of task tid in the page table space. On a task switch $ctid = tid$ the pto and ptl SPRs are set to these values.

- The swap-memory origin variable $smo \in \mathbb{B}^{32}$ designates the starting page of the swap memory region of task tid. Its virtual pages are linearly mapped to the pages $\{smo, smo+1, \ldots, smo+ptl\}$ of the swap memory.
- The save area holds the user registers of task tid if $ctid \neq tid$ or the machine runs in system mode (and register save has already completed). The user register are the two program counters, 31 general-purpose registers ($R[0]$ needs not to be stored), 32 floating-point registers and three special-purpose registers related to floating-point operation (the rounding mode RM, the IEEE flags $IEEEf$ and the floating-point condition code FCC);[3] in total these are 68 registers of width 32 bits. For each register r to be stored there is a corresponding variable r in the TCB.

The components of a TCB can be accessed using the based-indexed addressing scheme with the base address of the TCB in some register and the offset of the component as immediate index. For ease of notation, for all components $comp$ we define symbolic constants tcb_comp equal to the offset of the component in the TCB:

$$\begin{aligned} tcb\_state &:= 0 & tcb\_r1 &:= 24 \\ tcb\_pto &:= 4 & tcb\_r2 &:= 28 \\ tcb\_ptl &:= 8 & &\ldots \\ tcb\_smo &:= 12 & tcb\_ieeef &:= 280 \\ tcb\_dpc &:= 16 & tcb\_fcc &:= 284 \\ tcb\_pc' &:= 20 \end{aligned}$$

Let us consider an example of a component access. Assume that register $R2$ holds the base address of a TCB. To clear the $r1$ components and to read the pto component the following two RTL instructions can be used:

```
mm4[R2 + tcb_r1] := R0
R3 := mm4[R2 + tcb_pto]
```

When a is known to be a TCB address we use the pseudo-typed notation $a.comp$ for $mm_4[a+tcb\_comp]$ for a given memory configuration mm.

The accumulated size of all TCB components is $4 \cdot 4 + 68 \cdot 4 = 288$ bytes. For further extensions and to ease indexing into the TCB table, we round this number up to the next power of two and denote it by

$$tcb\_size = 512 = 2^9 \,. \tag{7.1}$$

Let us consider an example of a TCB table lookup. Assume that register $R1$ holds the base address of the TCB table tcb and register $R2$ holds a task identifier (e.g. the current task identifier). Then, the following two RTL instructions compute the base address of the TCB for task $R2$ in register $R2$:

```
R2 := R2 ≪ log2(tcb_size)
R2 := R2 + R1
```

Figure 7.2 shows the TCB table.

[3] If IEEE exceptions are dispatched to user-specified code, then the IEEE interrupt masks $SR[11:7]$ must be made user-visible and -accessible, as well. We ignore this here.

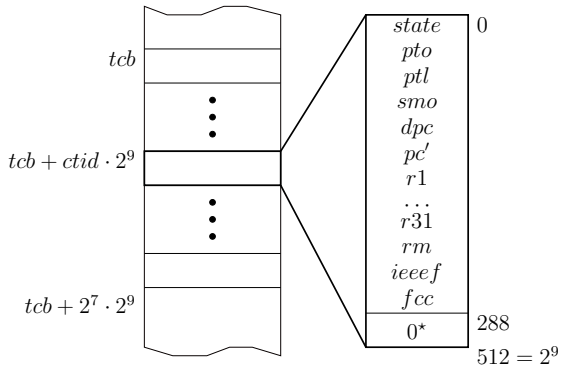

Figure 7.2 TCB Table. We have $tcb\_size = 2^9$ and $task\_num = 2^7$.

Invariants. Page tables must not overlap: for any given pair of page tables of active tasks, either one must end before the other starts or vice versa. Let tcb and tcb' be the addresses of two TCBs of active tasks, i.e.

$$tcb \neq tcb' \wedge mm_4[tcb + tcb\_state] \neq 0 \text{ and } mm_4[tcb' + tcb\_state] \neq 0\ . \tag{7.2}$$

We abbreviate

$$pto = \langle mm_4[tcb + tcb\_pto]\rangle\ , \tag{7.3}$$
$$ptl = \langle mm_4[tcb + tcb\_ptl]\rangle\ , \tag{7.4}$$
$$pto' = \langle mm_4[tcb' + tcb\_pto]\rangle\ , \text{ and} \tag{7.5}$$
$$ptl' = \langle mm_4[tcb' + tcb\_ptl]\rangle\ . \tag{7.6}$$

We derive the conditions indicating that both page tables do not overlap. All page tables start at page boundaries. Since a single page holds $2^{12}/4 = 2^{10}$ page table entries, the size of a page table in pages is the number of its elements divided by 2^{10} and rounded up.

Either the page table designated by tcb must end before the page table designated by tcb' starts or vice versa. Therefore, the condition

$$(pto + \lceil (ptl + 1) \cdot 2^{-10} \rceil \leq pto') \vee (pto' + \lceil (ptl' + 1) \cdot 2^{-10} \rceil \leq pto) \tag{7.7}$$

must hold. Likewise, swap memory regions may also not overlap. Abbreviating additionally

$$smo = \langle mm_4[tcb + tcb\_smo]\rangle \text{ and } smo' = \langle mm_4[tcb' + tcb\_smo]\rangle \tag{7.8}$$

we demand

$$(smo + ptl < smo') \vee (smo' + ptl' < smo)\ . \tag{7.9}$$

Page Tables

The page tables of the active tasks are stored in a region called *page table space* just before the start of the user memory. As we have seen already, the page tables it contains may not overlap and, due to page table alignment, parts of the page table space are unused hence wasted.

Apart from the data interpreted by the hardware for address translation, each page table entry $pte[31:0]$ additionally encodes the logical rights for the virtual page:

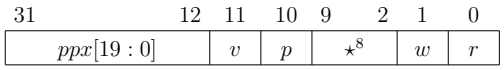

Figure 7.3 Page Table Entry with Bits for Logical Rights

- Bit $pte[0]$ indicates whether the task has a logical read right for the associated virtual page.
- Bit $pte[1]$ indicates whether the task has a logical write right for the associated virtual page. Since write-only pages cannot be implemented with the VAMP architecture $pte[1]$ implies $pte[0]$.

Overall, the format of a page table entry is shown in Figure 7.3.

Page tables are not arranged in any particular order in the page table space; resizing and deletion may require rearrangement of the page tables.

User Memory Page Management

User memory is managed with the user memory page management (UMPM) table whose entries form two double-linked lists, the active and the free list.

The UMPM Table. The user memory starts right after the end of the last page table. As the size of the page table space varies with the number of tasks and their memory requirements, the number of pages left for the user also varies. To keep account of this, we let the variable $fuppx$ denote the first physical page index of a user memory page; of course it has to satisfy $TCB(tid).pto + \lceil (TCB(tid).ptl + 1) \cdot 2^{-10} \rceil \leq fuppx$ for any active TCB but may not optimally do so (i.e. no active TCB satisfies the equality).

Starting from the page $fuppx$, the user memory continues to the upper end of the physical memory, which is in our current VAMP implementation at page 8192 (excluding). The system memory consists of all page indices not in this range:

$$Sys = \{0, \ldots, mm_4[fuppx] - 1\} \tag{7.10}$$

The so-called UMPM table is used to manage the user pages; one entry of the table corresponds to one physical page index of the user memory.

Entries of the UMPM table are structures with size $umpm\_size := 16$ bytes and have the following components:

- A pointer $next \in \mathbb{B}^{32}$ and a pointer $prev \in \mathbb{B}^{32}$ to other UMPM table entries.
- A task identifier $tid \in \mathbb{B}^{32}$ and a virtual page index $vpx \in \mathbb{B}^{32}$ that indicate to which task and to which virtual address a physical page belongs to if it is used. For tid only the lower 7 and for vpx only the lower 20 bits are used.

We determine a maximum size for the UMPM table by assuming that the page table space has size zero. The maximum size of the UMPM table is approximately the number of remaining bytes in physical memory divided by the sum of the page size 4096 and the UMPM entry size 16. We denote this number by the symbolic constant $umpm\_num$ and reserve space for the UMPM table starting at the address $umpm$.

The first entry of the UMPM table manages the physical page with index $(8192 - umpm\_num)$; the last entry of the UMPM table manages the physical page with index

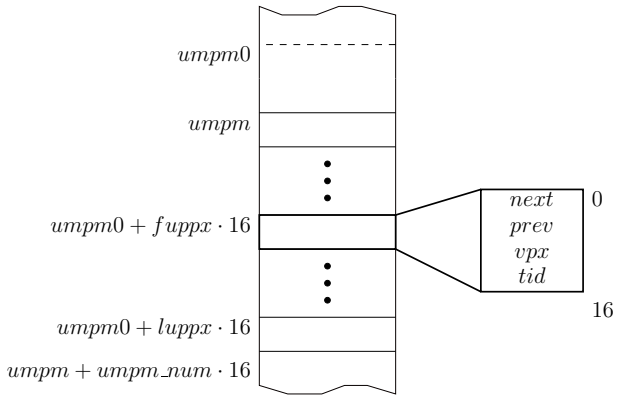

Figure 7.4 UMPM Table. Recall that $umpm\_size = 16$.

8191. To index the UMPM table by physical page indices rather than shifted page indices, we define the auxiliary constant

$$umpm0 := umpm - (8192 - umpm\_num) \cdot umpm\_size \; . \tag{7.11}$$

By the explanation above $8192 - umpm\_num$ is also the index of the first page to store any page table. Hence, we keep as an additional invariant that $TCB(tid).pto \geq 8192 - umpm\_num$ for any active TCB. From both conditions on page tables given here follows that page tables are placed inside the implemented memory and, in particular, do not wrap around.

Figure 7.4 shows the UMPM table with the entry for $fuppx$ highlighted.

Doubly-Linked Lists. As we have seen earlier, each element of the UMPM table is associated with a physical page. These pages fall into three categories:

- Pages storing page tables. These are located just before the user memory.
- Pages of the user memory that actually store a virtual page. These pages are called *active*.
- Free pages of the user memory.

While the first category is identified by the variable $fuppx$ pointing into the UMPM table, the latter two are managed using doubly-linked lists. This allows for an easy extension of the implemented page eviction algorithm (first-in first-out, FIFO) to a better one (FIFO with second chance).

We sketch our doubly-linked list implementation. Each list has a so-called *descriptor* that for simplicity has the same structure as an element of the UMPM table. However, the descriptor of a list only uses the *next* and the *prev* entries of the structure: the former points to the first element (the "head") of the list, the latter points to the last element (the "tail") of the list. A descriptor is initialized by having its *next* and *prev* pointers pointing to itself. An empty list is identified by checking the address of the list descriptor and its *next* pointer (or, symmetrically, its *prev* pointer) for equality.

Lists are traversed in forward direction by following the *next* pointers until reaching the list descriptor again. They are traversed in backward direction by following the *prev* pointers. Thus, traversing an empty list terminates directly.

Formally, let l_0 denote the address of a list descriptor. We call l_0 a *valid list descriptor*, abbreviated by $dllist?(l_0)$, iff there exists a natural number $n \in \mathbb{N}$ and a list of addresses $(l_1, \ldots, l_n)$ such that the following three conditions hold:

- All addresses, including the list descriptor, are pairwise distinct:

$$\forall i, j \in \{0, \ldots, n\} : i \neq j \Rightarrow l_i \neq l_j \tag{7.12}$$

- Chasing the *next* pointers returns the element addresses in ascending index order:

$$\forall i \in \{0, \ldots, n\} : mm_4[l_i + umpm\_next] = l_{(i+1) \bmod n+1} \tag{7.13}$$

- Chasing the *prev* pointers returns the element addresses in descending index order:

$$\forall i \in \{0, \ldots, n\} : mm_4[l_{(i+1) \bmod n+1} + umpm\_prev] = l_i \tag{7.14}$$

Address l_0 is called the descriptor address, address l_1 is called the head address, and address l_n is called the tail address. Since the element addresses must be distinct and arranged cyclically, the number n is uniquely defined and we call it the *size* of the list at l_0 and denote it by $size(l_0) = n$. We call the set $Elem(l_0) = \{l_1, \ldots, l_n\}$ the set of list elements and the set $Elem^+(l_0) = \{l_0, \ldots, l_n\}$ the set of list elements including the list descriptor. The sequence $elem(l_0) = (l_1, \ldots, l_n)$ is called the sequence of list elements; the sequence $elem^+(l_0) = (l_0, \ldots, l_n)$ is called the sequence of list elements including the list descriptor.

By symmetry, any of the addresses l_i is as well a valid list descriptor for a list of the same size n and $Elem^+(l_i) = Elem^+(l_0)$. Also, for two list descriptors l_0 and j_0, we have $Elem(l_0) = Elem(j_0)$ iff $Elem^+(l_0) = Elem^+(j_0)$ iff $elem^+(l_0)$ is a rotation of $elem^+(j_0)$.

We call the element l *woven* to the element j if

$$mm_4[j + umpm\_next] = l \text{ and } mm_4[l + umpm\_prev] = j \tag{7.15}$$

hold and denote this fact by $weave(j, l)$. For a valid list descriptor l_0 with $Elem^+(l_0) = (l_0, l_1, \ldots, l_n)$ we get $weave(l_i, l_{(i+1) \bmod n+1})$ for all $i \in \{0, \ldots, n\}$ directly from its definition.

List operations to add or delete elements can be understood in terms of *weaving*, i.e. establishing the $weave(j, l)$ predicate for certain list elements. Suppose, we want to insert a list element k *after* the element j. Assume l is woven to j, i.e. $weave(j, l)$.

Assuming that j, k, and l are stored in the registers $R1$, $R2$, and $R3$, by the RTL operations

```
mm4[R1 + umpm_next] = R2
mm4[R2 + umpm_prev] = R1
mm4[R2 + umpm_next] = R3
mm4[R3 + umpm_prev] = R2
```

we establish $weave(j, k)$ and $weave(k, l)$ and thus insert k into the list identified by j (or l, symmetrically). This works even if j and l are equal, i.e. starting with the empty list. Figure 7.5 shows an example of list insertion for the empty list and for a list with more than two elements.

To delete an element k one finds its predecessor $j := mm_4[k + umpm\_prev]$ and its successor $l := mm_4[j + umpm\_next]$ by pointer chasing. Assuming again that j, k, and l are stored in the registers $R1$, $R2$, and $R3$ by the RTL operations

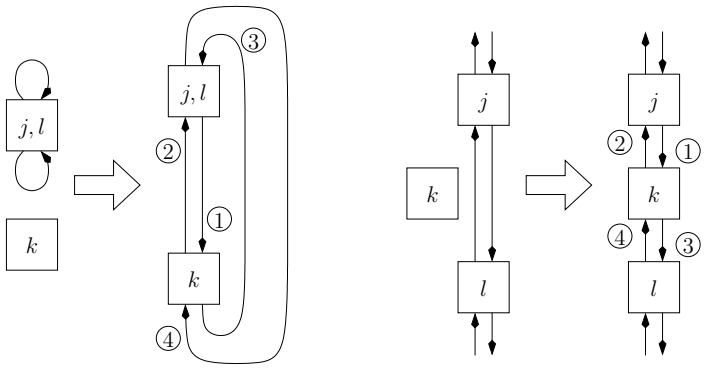

Figure 7.5 Doubly-Linked List Insertion. We use the following convention: An arrow with downward-pointing head from some element l to another element l' expresses the fact, that the *next* pointer of element l holds the address l'. An arrow with upward-pointing head expresses the same fact for the *prev* pointer. On the left-hand side an insertion of element k into an empty list is shown; on the right-hand side an insertion of element k between elements j and l is shown. The numbers at the arrowheads correspond to the numbers of the memory operations for the code given in the text.

```
mm4[R1 + umpm_next] = R3
mm4[R3 + umpm_prev] = R1
```

one establishes $weave(j,l)$, taking k out of the list with j and l.

Some UMPM Invariants. We will have the following invariants on the active and the free list:

- Both *activelist* and $freelist$ are valid list descriptors:

$$dllist?(activelist) \wedge dllist?(freelist) \tag{7.16}$$

- The elements of both lists are disjunct:

$$Elem(activelist) \cap Elem(freelist) = \emptyset \tag{7.17}$$

- The elements of both lists comprise the entries of the UMPM table corresponding to user memory pages:

$$Elem(activelist) \cup Elem(freelist) = \{umpm0 + i \cdot umpm\_size \mid mm_4[fuppx] \leq i < 8192\} \tag{7.18}$$

- Therefore the sum of both list sizes is equal to the number of user pages:

$$size(activelist) + size(freelist) = 8192 - fuppx \tag{7.19}$$

7.1.3 Code

Both the page fault on fetch and the page fault on load / store ISRs call the generic page fault handler pfh, which is the core part of our code. It takes the virtual page index and the memory operation that caused the exception as inputs. We refer to these two as the exception virtual page index and the exception memory operation. More inputs are provided implicitly by the special-purpose registers (page table origin and page table length) and system data (e.g. the current task identifier).

In the following sections, we outline the hardware interrupt service routine, the wrapper functions for page fault on fetch and on load / store, the stubs used to axiomatize swap memory access and the generic page fault handler itself.

Interrupt Service Routine

We describe the start of the interrupt service routine. By the architecture, the start address of the ISR is hardwired to address 0. As the zero page is already used for special purposes (memory-mapped I/O) we jump from address $sisr = 0$ directly to address $isr = 4096$, where the "real" code starts:

```
sisr:
    PC' = isr
    NOP
```

From the address $isr = 4096$ we proceed in six steps:

1. We save registers $R1$ and $R2$ to reserved locations in the memory. Thereafter both may be used for computations of the ISR.
2. We inspect the exception cause register ECA to determine whether the interrupt cause was power-up ($ECA[31:0] = 0^{32}$) or reset ($ECA[0] = 1$). In both cases we jump to the reset code described in a later section. Otherwise we continue.
3. All user registers are written to the save area of the current TCB with registers $R1$ and $R2$ being retrieved from their temporary save locations.
4. We compute the interrupt level, i.e. the index il of the first bit in ECA being set. We already know that $il > 0$. We call the interrupt-specific part of the handler at address l stored in table entry il of the interrupt vectors table $isr\_vectors$.
5. After its return we restore the user registers from the current TCB. With $ctid$ being changed, this might be a different TCB than before. Registers $R1$ and $R2$ are not yet restored but copied to their reserved memory locations.
6. Finally, registers $R1$ and $R2$ are also restored and the ISR ends with an `rfe` (return from exception) instruction.

We present the code for all these steps in detail.

Low-Level Register Save. At the beginning of the ISR we save register $R1$ and register $R2$ to special locations in main memory since we will need them as temporary registers. The symbolic constants $save\_r1$ and $save\_r2$ denote these locations. To allow base-indexed access using $R0$ as a base, both constants must be in the addressable range of a (sign-extended) 16-bit constant. We will assume $0 \leq save\_r1 < 2^{15}$ and $0 \leq save\_r2 < 2^{15}$. The corresponding RTL instructions are as follows:

```
isr_save:
    mm4[save_r1] := R1
    mm4[save_r2] := R2
```

Check for Reset / Power-Up. We check now whether we entered the ISR because of reset or power-up. In the latter case, the exception cause register is equal to zero; in the former case, bit 0 of the exception cause register is set.

Hence, if bits $[31:1]$ of the exception cause register are all zero, the exception cause must either be reset or power-up. In this case, we jump to the *reset* routine described later:

```
R1 := ECA
R1 := R1 ∧ 1^31 0
PC' := (R1 = 0 ? reset : PC' + 4)
```

The delay slot of this branch will be filled in the next section.

High-Level Register Save. On arriving here we already know that we have a regular interrupt to serve. We start by saving all user registers to the TCB of the currently running task.

To begin with, we load the contents of the *ctid* variable to register $R1$. To load the address of the *ctid* variable we use the standard 32-bit constant loading code (its first instruction being placed in the delay slot of the preceding branch):

```
R1 := hi(ctid) ≪ 0^16
R1 := R1 ⊕ lo(ctid)
R1 := mm4[R1]
```

Then, we compute the address of the current TCB by adding the current task identifier multiplied by the TCB size to the base address of the TCB table:

```
R1 := R1 ≪ log2(tcb_size)
R2 := hi(tcb) ≪ 16
R2 := R2 ⊕ lo(tcb)
R1 := R1 + R2
```

Using base-indexed memory writes we copy all user registers except $R1$ and $R2$ to the TCB. For each $elem \in \{r3, r4, \ldots, r31\}$ we have to execute an RTL instruction of the following form:

```
mm4[R1 + tcb_elem] := elem
```

Any register $elem \in \{rm, ieeef, fcc\}$ must first be copied into a general-purpose register. For each of them, we execute:

```
R2 := elem
mm4[R1 + tcb_elem] := R2
```

For the program counters the procedure is similar. They are, however, to be found in the exception program counters:

```
R2 := EDPC
mm4[R1 + tcb_dpc] := R2
R2 := EPC
mm4[R1 + tcb_pcp] := R2
```

Registers $R1$ and $R2$ have to copied from the locations they were saved in to the TCB:

```
    R2 := mm4[save_r1]
    mm4[R1 + save_r1] := R2
    R2 := mm4[save_r2]
    mm4[R1 + save_r2] := R2
```

Calling the Exception-Specific Part of the ISR. We determine the interrupt level by finding the least significant bit being set in the *ECA* register:

```
    R1 := ECA
    R3 := -1
isr_eca_loop:
    R2 := R1 ∧ 0^31 1
    R3 := R3 + 1
    PC' := (R2 = 0 ? isr_eca_loop : PC' + 4)
    R1 := R1 ≫ 1
```

This loop requires a linear number of iterations; this could be reduced to a logarithmic number of iterations by implementing it as a binary search.

On reaching the loop's branch after i complete executions of the loop body we have the invariant

$$R3 = i \text{ and } R1 = ECA \gg i \text{ and } R2 = 1 \Leftrightarrow ECA[i:0] = 10^i \,. \tag{7.20}$$

Hence, on exiting the loop register $R3$ contains the interrupt level $il \in \{1,\ldots,31\}$. We multiply it by four in order to use it as an offset into the ISR vectors table, which stores the start locations of the interrupt-specific parts of the ISRs:

```
    R3 := R3 ≪ 2
```

The base address *isr_vectors* of this table is assumed to be inside the first $32K$ of the physical memory (for simplicity but not out of necessity). We load the appropriate table entry and jump to it storing the return address in register $R31$ (the `jalr` instruction has this semantics):

```
    R1 := mm4[isr_vectors + R3]
    PC' := R1, R31 := PC' + 4
    NOP
```

The interrupt-specific parts of the ISR are described later; as we have already saved the complete user task register set, these routines may use all general-purpose registers at will.

High-Level Register Restore. Eventually, the interrupt-specific part returns. We begin to restore the user registers of the current task, which might have changed in the meantime. We load the current task identifier to register $R1$:

```
isr_restore:
    R1 := hi(ctid) ≪ 16
    R1 := R1 ⊕ lo(ctid)
    R1 := mm4[R1]
```

Then, we compute the address of the current TCB as we did for the register save part:

```
R1 := R1 ≪ log2(tcb_size)
R2 := hi(tcb) ≪ 16
R2 := R2 ⊕ lo(tcb)
R1 := R1 + R2
```

Using the appropriate entries of the TCB we are now able to restore all but registers $R1$ and $R2$. First, we restore the exception program counters:

```
R2 := mm4[R1 + tcb_dpc]
EDPC := R2
R2 := mm4[R1 + tcb_pcp]
EPC := R2
```

Second, we restore the user-visible special-purpose registers $elem \in \{rm, ieeef, fcc\}$:

```
R2 := mm4[R1 + tcb_elem]
elem := R2
```

Third, most of GPRs are restored. For each $elem \in \{r3, r4, \ldots, r31\}$ we execute an RTL instruction of the following form:

```
elem := mm4[R1 + tcb_elem]
```

Finally, registers $R1$ and $R2$ are copied to their separate, special locations:

```
R2 := mm4[R1 + tcb_r2]
mm4[save_r2] := R2
R1 := mm4[R1 + tcb_r2]
mm4[save_r1] := R1
```

Note that the order of these two operations guarantees that we do not destroy the contents of the addressing register $R1$ prematurely.

Low-Level Register Restore. We restore registers $R1$ and $R2$ from their save locations and issue a return from exception instruction.

```
R1 := mm4[save_r1]
R2 := mm4[save_r2]
RFE
```

The `rfe` instruction has no delay slot.

Reset / Initialization

Ignoring hardware-specific initializations (e.g. setting up devices) our reset routine can be divided into two parts: (i) setting up the task control blocks and page tables by decoding initial task images stored in the swap memory and (ii) setting up the data structure for user page management and the page fault handler. We specify how the both parts are done and additionally present the code of the second part.

Initially, the swap memory contains the following data. Page 0 called the *blank page* contains all zeroes.[4] Word 0 of page 1 contains the number of tasks $0 < n <$

[4]This simplifies proofs later on. Apart from this, a blank page in main memory can be used to optimize implementations of sparsely occupied arrays in virtual memory.

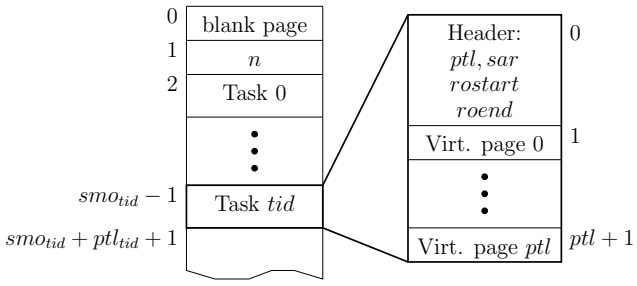

Figure 7.6 Structure of the Task Images in the Swap Memory. The addresses shown are swap page indices.

task_num that are stored in swap memory. Page 2 and the following contain the initial configurations for all tasks $tid \in \{0,\ldots,n-1\}$. The image of task tid starts at page $smo_{tid} - 1$. Page $smo_{tid} - 1$ contains the initial values for the user registers (including the task's start address in the program counters), the index of the last accessible page ptl_{tid}, and the indices $rostart_{tid}$ and $roend_{tid}$ of the first and last page of a read-only segment (typically the code segment). Pages smo_{tid} to $smo_{tid} + ptl_{tid}$ contain the initial contents of the task's virtual memory mapped out linearly. We set $smo_0 := 4$ and define $smo_{tid+1} = smo_{tid} + ptl_{tid} + 2$ for $0 < tid < n$ inductively. Figure 7.6 shows this structure.

The tasks are loaded in a loop. For each task $tid < n$, the user registers in the TCB are initialized and the TCB's state component is set to active. The ptl and smo components are set to ptl_{tid} and smo_{tid} (which means that the task image file is *live*, i.e. it is also used to store swapped-out pages). The page table origin is set to $(8192 - umpm\_num) + \sum_{i<tid}(ptl_i + 1)$. By construction, swap memory regions and page tables do not overlap. All PTEs are initialized invalid and have the logical read right. Additionally, a PTE has the logical write right if it is outside the task's read-only segment.

The TCBs of the remaining tasks are set to inactive in the TCB table. Variable $ctid$ and the PTO and PTL special-purpose registers are set up for task 0.

With the above loading procedure user memory may start after the last page occupied by the page table of task $n-1$. The variable $fuppx$ is set accordingly, i.e. $M_4[fuppx] = (8192 - umpm\_num) + \sum_{i<n}(ptl_i + 1)$. The variable $luppx$ is set to 8191. This is the starting point for the initialization of the UMPM data structures: the free list is initialized to be *full* holding all user pages, and the active list is set to the empty list. We present the code with its invariants.

We compute the number of user pages *minus one* in register *R*3 as the difference of $luppx$ and $fuppx$:

```
R5 := hi(fuppx) ≪ 16
R5 := R5 ⊕ lo(fuppx)
R3 := hi(luppx) ≪ 16
R3 := R3 ⊕ lo(luppx)
R3 := R3 - R5
```

We multiply $fuppx$ by the size of an UMPM element and add it to the address *umpm*0, thus obtaining the address of the UMPM element corresponding to the physical page $fuppx$:

```
R5 := R5 ≪ log2(umpm_size)
R1 := hi(umpm0) ≪ 16
R1 := R1 ⊕ lo(umpm0)
R1 := R1 + R5
```

Then we load the address $freelist$ into register $R5$:

```
R5 := hi(freelist) ≪ 16
R5 := R5 ⊕ lo(freelist)
```

Having the descriptor's address in register $R5$ and the address of the (designated) head in register $R1$, we start building the free list by weaving $R5$ to $R1$:

```
mm4[R5 + umpm_next] := R1
mm4[R1 + umpm_prev] := R5
```

We initialize all but one of the remaining entries of the free list in a loop. In each iteration, we weave the element at address $R1$ to the element at address $R2$. For initialization, by

```
R2 := R1 + umpm_size
```

register $R1$ still points to the head and register $R2$ points to its designated successor, the UMPM element for page $fuppx+1$. We use $R3$ as a decrementing loop counter until it reaches zero. In each iteration, we weave the element at $R1$ to the element at $R2$ and increment $R1$ and $R2$ by the size of an UMPM element:

```
reset_loop1:
    mm4[R1 + umpm_next] := R2
    mm4[R2 + umpm_prev] := R1
    R3 := R3 - 1
    R1 := R1 + umpm_size
    PC' := (R3 = 0 ? reset_loop1 : PC' + 4)
    R2 := R2 + umpm_size
```

The loop satisfies the following invariant: after i executions of the loop body, the equations

$$R3 = luppx - fuppx - i\,, \tag{7.21}$$

$$R1 = umpm0 + (fuppx + i) \cdot umpm\_size\text{ , and} \tag{7.22}$$

$$R2 = umpm0 + (fuppx + i + 1) \cdot umpm\_size \tag{7.23}$$

hold and for all j with $fuppx \le j < fuppx + i$ we have

$$weave(umpm0 + j \cdot umpm\_size, umpm0 + (j+1) \cdot umpm\_size)\,. \tag{7.24}$$

After the completion of the loop, $R1$ contains the address of the last UMPM element. We weave this element to the descriptor $freelist$:

```
mm4[R1 + umpm_next] := R5
mm4[R5 + umpm_prev] := R1
```

This completes the construction of the free list establishing $dllist?(freelist)$ and

$$elem(freelist) = \{umpm0 + j \cdot umpm\_size \mid fuppx \le j \le luppx\}\,. \tag{7.25}$$

Figure 7.7 shows the fully constructed free list.

For the active list we store the descriptor's address in its *next* and *prev* pointers:

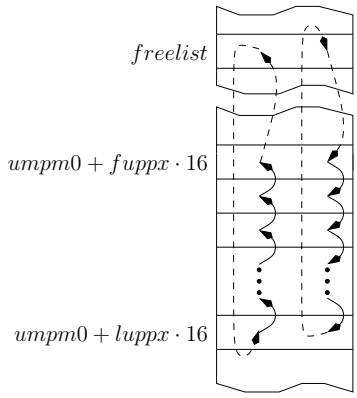

Figure 7.7 Initialization of the Free List. Next pointers are on the right side, heads pointing downward; previous pointers on the left side, heads pointing upward. The two dashed short arrows near the top are established before the loop, solid arrows in it, and the two dashed long arrows after it.

```
R1 := hi(activelist) ≪ 16
R1 := R1 ⊕ lo(activelist)
mm4[R1 + umpm_next] = R1
mm4[R1 + umpm_prev] = R1
```

This guarantees $dllist?(activelist)$ and $elem(activelist) = \emptyset$.

Finally, the reset routine sets $ctid = 0$, loads the page table origin and the page table length register for task $ctid = 0$ (usually done by the scheduler), and continues execution with high-level register restore (cf. Section 7.1.3).

Wrapper for Page Fault on Fetch

The wrapper for page fault on fetch is called whenever a page fault on fetch occurs (i.e. its address pff is stored in entry three of the interrupt vectors table). It computes the arguments for the generic page fault handling procedure pfh and calls it. The implementation is straightforward. The exception delayed PC *EDPC* right-shifted by twelve positions yields the exception virtual page index. The page fault handler expects this input in register $R1$:

```
pff:
    R1 := EDPC
    R1 := R1 ≫ 12
```

The memory operation for fetches is zero (for reads). This argument is expected in register $R3$. We execute the instruction, setting this argument in the delay slot of the unconditional branch to the pfh function:

```
    PC' := pfh
    R3 := 0
```

Note, that the return address to the ISR handler is still held in register $R31$ and used by the page fault handler as return address.

Wrapper for Page Fault on Load / Store

The wrapper for page fault on load / store is called whenever a page fault on load / store occurs (i.e. its address $pfls$ is stored in entry four of the interrupt vectors table). It is a little more complicated to implement since we need to find out, whether the operation that caused the exception was a load or a store instruction.[5] To do this, we have to emulate the translated fetch of the exception instruction. Inspection of the instruction opcodes shows that all store instructions have bit 29 set to one and may thus be distinguished from load instructions.

We copy the exception delayed PC to register $R1$:

```
pfls:
    R1 := EDPC
```

Afterwards, we compute the offset into the page table in $R2$:

```
    R2 := R1 ≫ 10
    R2 := R2 ∧ 1^30 0^2
```

By adding the page table origin multiplied by 2^{12} to this offset we compute the page table entry address. We load the page table entry to register $R3$:

```
    R3 := PTO
    R3 := R3 ≪ 12
    R3 := R2 + R3
    R3 := mm4[R3]
```

We compute the physical address of the fetch by concatenating the ppx field of the PTE with the byte offset of the exception delayed PC still stored in register $R1$. Afterwards, we fetch the instruction word into register $R3$.

```
    R3 := R3 ∧ 1^20 0^12
    R1 := R1 ∧ 0^20 1^12
    R3 := R1 ∨ R3
    R3 := mm4[R3]
```

Shifting and masking out bit 29 of the instruction word (see above) we compute the memory operation input to pfh in register $R3$—zero for page fault on load and one for page fault on store:

```
    R3 := R3 ≫ 28
    R3 := R3 ∧ 0^31 1
```

By definition of the instruction set architecture, the exception effective address is stored in the $EDATA$ register. We store its page index in register $R1$ and jump unconditionally to the pfh function. Again, we make use of the delay slot and remark that the page fault handler will use the unchanged return address in $R31$:

```
    R1 := EDATA
    PC' := pfh
    R1 := R1 ≫ 12
```

[5] The current architecture fails to provide this information in a special-purpose register.

I/O Functions

The swap memory $sm : [\mathbb{B}^{27+9} \rightarrow \mathbb{B}^{64}]$ can be accessed by the functions *swap_out* and *swap_in*, the former copying a page from the main to the swap memory and the latter copying a page from the swap to the main memory. We only specify the behavior of both functions. Both take as inputs a physical page index ppx (in register $R10$) and a swap memory page index $smpx$ (in register $R11$). Let t and t' denote the call and return time of the function.

The function *swap_out* copies the main memory page ppx to the swap memory page $smpx$:

$$sm^{t'}_{4096}[smpx, 0^{12}] = mm^{t}_{4096}[ppx, 0^{12}] \tag{7.26}$$

The function *swap_in*, inversely, copies the swap memory page $smpx$ to the main memory page ppx:

$$mm^{t'}_{4096}[ppx, 0^{12}] = sm^{t}_{4096}[smpx, 0^{12}] \tag{7.27}$$

Other than by calling *swap_out*, the swap memory contents stay unchanged; other than by calling *swap_in*, the swap memory may not be read.

As their names indicate, the I/O functions are used for regular swapping operations but also, as was already described, for initially loading the tasks. The pages of a task (which cannot be shared) are stored linearly from a certain offset in the swap memory. Such a convention is of course simplistic and utterly inflexible. Realistic implementations / algorithms are too complex to be described here and for efficiency need to maintain complex data structures in main memory.

The swap memory holds $2^7 \cdot 2^{20}$ pages and may thus accommodate all pages of up to 2^7 tasks.

Page Fault Handler

The generic page fault handler function starting at address pfh receives two input arguments: The exception virtual page index in register $R1$ and the memory operation identifier in register $R3$, which is 0 for reads and 1 for writes.

We divide the execution of the page fault handler into three steps:

- First, we check whether the exception operation was legal or illegal. If the operation was illegal, we abort the execution of the handler (in our simple scenario, we just enter an endless loop at address *abort*). Otherwise, we continue.

- Second, we determine the physical page index in user memory that we intend to use for swap-in. There are two possibilities for choosing this index. If there are still unused / free pages in user memory we take one of their indices. Otherwise, we select and swap-out one of the pages in user memory; by the first-in first-out strategy (FIFO strategy) we take the page associated with the head of the active list.

- Third, we swap in the exception page and update the corresponding page table entry.

Figure 7.8 shows the flow chart of the page fault handler.

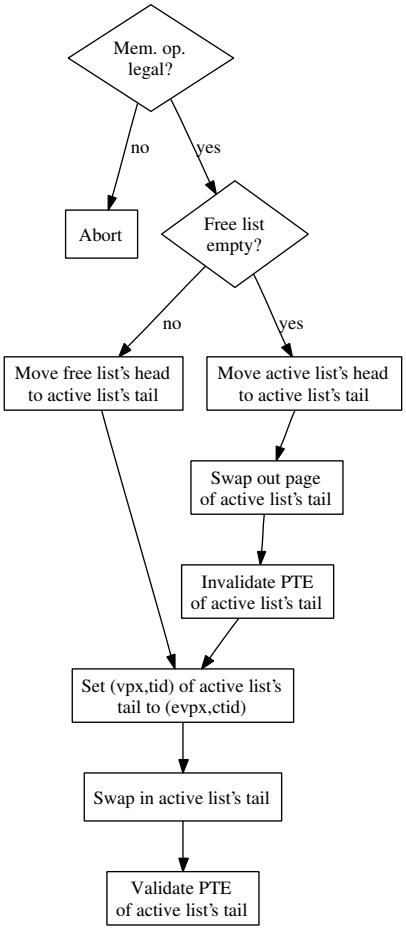

Figure 7.8 Flow Chart of the Page Fault Handler

Logical Rights Check. By convention, an operation is illegal if we have a page table length exception or if its logical right is absent.

Initially, we check whether a length violation occurred by comparing the exception virtual page index to the page table length register:

```
pfh:
    R4 := PTL
    R4 := (R1 > R4 ? 1 : 0)
    PC' := (R4 ≠ 0 ? abort : PC' + 4)
```

The delay slot of the branch is filled in the next code chunk. We fetch the page table entry that caused the exception. This code is similar to the code we have already seen for the wrapper for page fault on load / store.

We compute the PTE address in register *R*4 by adding the exception virtual page index multiplied by four to the page table origin. Then, we load the word stored at this address to register *R*5. We will later need both the PTE address and the PTE.

```
    R4 := PTO
```

```
R4 := R4 ≪ 12
R5 := R1 ≪ 2
R4 := R4 + R5
R5 := mm4[R4]
```

Now we check for a logical rights violation. By adding one to the memory operation register *R*3, it equals to one for read operation and to two for write operations. By software convention, the PTE holds the logical read right in bit 0 and the logical write right in bit 1. Additionally, a logical write right implied a logical read right. Hence, a logical right for the exception memory operation is present iff the bit-wise conjunction of register *R*3 with the PTE right-shifted by 5 is non-zero. If this is the case, we jump to the *abort* routine.

```
R3 := R3 + 1
R6 := R6 ∧ R3
PC' := (R6 ≠ 0 ? abort : PC' + 4)
```

The delay slot of this branch will be filled in the next section.

Check the Free List. We now check whether the free list is empty or not. Recall, that an empty list is indicated by the descriptor's *next* pointer equalling its *prev* pointer. We load the address of the free list descriptor to register *R*7 by the VAMP standard constant loading code:

```
R7 := hi(freelist) ≪ 16
R7 := R7 ⊕ lo(freelist)
```

Then, we load the next and the previous pointer to registers *R*6 and *R*8 and compare them. The symbolic constants *umpm_next* and *umpm_prev* are defined to be the offset to the *next* and *prev* pointer of an UMPM structure. If the next pointer of the list descriptor points to itself, the free list is empty and we jump to the code which does some swapping out. Otherwise, we continue.

```
R6 := mm4[R7 + umpm_next]
R8 := (R6 = R7 ? 1 : 0)
PC' := (R8 = 1 ? pfh_swap_out : PC' + 4)
NOP
```

Choose a Page from the Free List. We know at this point that the free list is not empty; hence we take its first element and add it to the tail of the active list. This is done by the standard *weaving* procedure described in Section 7.1.2. Figure 7.9 illustrates this; its sub-figures correspond to the labeled lines in the code below.

We already have the address of the list head in *R*6 and the address of the list descriptor in *R*7. We chase the next pointer of the first element to load the address of the second element to *R*8:

```
R8 := mm4[R6 + umpm_next]        ①
```

Afterwards, we weave the head to the second element:

```
mm4[R7 + umpm_next] := R8
mm4[R8 + umpm_prev] := R7        ②
```

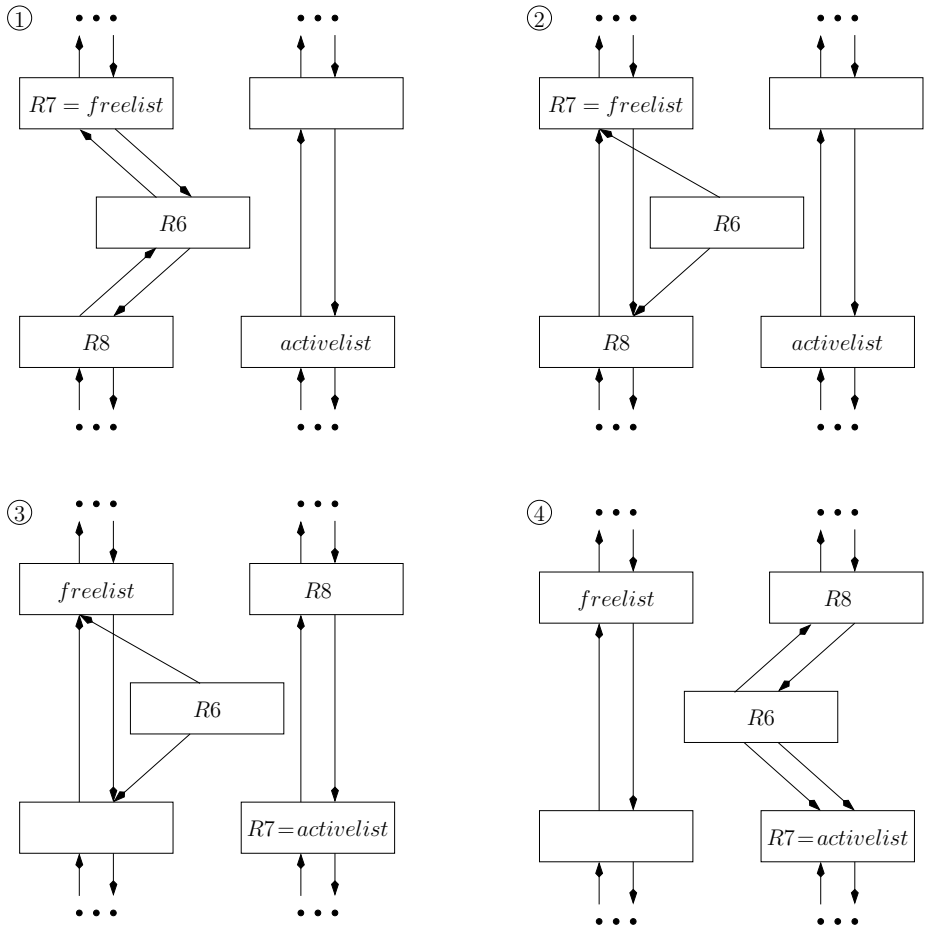

Figure 7.9 Selection of a Page from the Free List. The head of the free list is moved to the tail of the active list.

Now, we add the first element (still pointed to by *R*6) to the tail of the active list. To prepare this operation we load the address of the head of the active list to *R*7 and the address of its tail to *R*8.

```
R7 := hi(activelist) ≪ 16
R7 := R7 ⊕ lo(activelist)
R8 := mm4[R7 + umpm_prev]        ③
```

Then, we weave the new tail to the head and the old tail to the new tail:

```
mm4[R7 + umpm_prev] := R6
mm4[R6 + umpm_next] := R7
mm4[R8 + umpm_next] := R6
mm4[R6 + umpm_prev] := R8        ④
```

For later use, we compute the physical page index of the page to which the UMPM element corresponds to in register *R*7. This can be done by subtracting the address

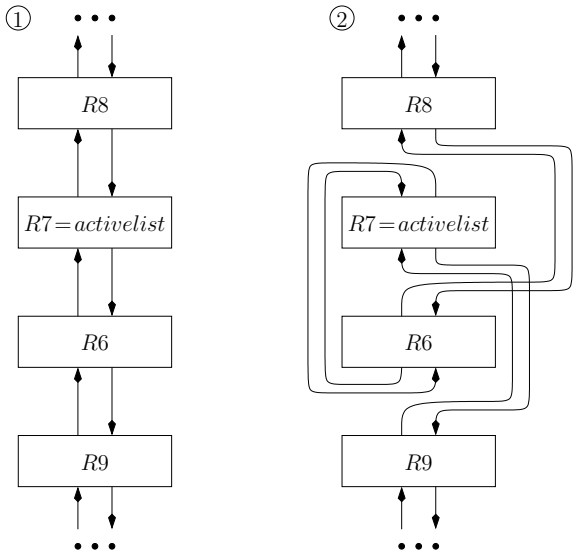

Figure 7.10 Selection of a Page from the Active List. The head of the active list is made the tail of the active list.

*umpm*0 from *R*7 and dividing the size of an UMPM element. A jump to pfh_swap_in completes this case; we will describe the swap-in procedure shortly.

```
R7 := hi(umpm0) ≪ log2(umpm_size)
R7 := R7 ⊕ lo(umpm0)
R7 := R6 - R7
PC' := pfh_swap_in
R7 := R7 ≫ log2(umpm_size)
```

Choose a Page from the Active list, Swap It Out, and Update PTE. In the complementary case (free list is empty) we need to find a candidate for eviction / swap-out. In the presented page fault handler we use a very simple strategy: we swap out the oldest page in the memory with respect to swap-in time. This is the head of the active list; we move it to the tail. Figure 7.10 illustrates this list operation; its sub-figures correspond to the labeled lines in the code below.

We load its address into register *R*6:

```
pfh_swap_out:
    R7 := hi(activelist) ≪ 16
    R7 := R7 ⊕ lo(activelist)
    R6 := mm4[R7 + umpm_next]
```

Since we use this element to correspond to the newest page after swap-in, we move it to the end of the active list. To prepare this operation, we additionally load the address of the last element and the address of the second element to the registers *R*8 and *R*9:

```
    R8 := mm4[R7 + umpm_prev]
    R9 := mm4[R6 + umpm_next]        ①
```

Now *R*8, *R*7, *R*6, and *R*9 point to consecutive elements of the active list (where *R*8 is the last element of the active list). The addresses *R*8 and *R*9 are equal iff the active list consists of two elements. We re-weave the elements to arrange them in the order *R*8, *R*6, *R*7, and *R*9, which corresponds to moving the first element to the last element of the active list. The following operations are from left to right:

```
mm4[R8 + umpm_next] := R6
mm4[R6 + umpm_prev] := R8
mm4[R6 + umpm_next] := R7
mm4[R7 + umpm_prev] := R6
mm4[R7 + umpm_next] := R9
mm4[R9 + umpm_prev] := R7        ②
```

Now that we have the UMPM element, we want to swap out the corresponding physical page by calling the swap-out routine. We obtain the physical page index in the same way as we did for the former case: we compute the element's distance to *umpm*0 and divide it by the size of the UMPM element.

```
R7 := hi(umpm0) ≪ 16
R7 := R7 ⊕ lo(umpm0)
R7 := R6 - R7
R7 := R7 ≫ log2(umpm_size)
```

The swap-out routine was specified in Section 7.1.3: it takes the source physical page index in register *R*10 and the destination swap memory index in register *R*11 as inputs. The swap memory index is the sum of the swap memory origin and the virtual page index. We compute the address of the TCB in register *R*8:

```
R8 := mm4[R6 + umpm_tid]
R8 := R8 ≪ log2(tcb_size)
R9 := hi(tcb) ≪ 16
R9 := R9 ⊕ lo(tcb)
R8 := R8 + R9
```

Then, we load the swap memory origin to register *R*11 and the virtual page index to register *R*9 and add both:

```
R11 := mm4[R8 + tcb_smo]
R9 := mm4[R6 + umpm_vpx]
R11 := R9 + R11
```

We save / restore the link register *R*31 temporarily in register *R*9. Note the use of the delay slot to load the PPX input register.

```
R9 := R31
PC' := swap_out, R31 := PC' + 4
R10 := R7
R31 := R9
```

After the swap-out has taken place, we have to update the PTE corresponding to the swapped out page to reflect the new situation. The update consists in invalidating the PTE.

We load the PTO stored in the TCB into register *R*8:

```
R8 := mm4[R8 + tcb_pto]
```

From the page table origin and the virtual page index stored in the UMPM entry we may now compute the page table entry address in register $R8$. Thereafter, we load the page table entry to register $R9$.

```
R8 := R8 ≪ 10
R9 := mm4[R6 + umpm_vpx]
R8 := R8 + R9
R8 := R8 ≪ 2
R9 := mm4[R8]
```

We clear the invalid bit and write back the PTE. Especially, the logical rights bits will remain intact in the PTE.

```
R9 := R9 ∧ 1^20 01^11
mm4[R8] := R9
```

This completes the swap-out procedure.

Swap In Exception Page and Update PTE. By now, we have already determined an UMPM entry and put it at the tail of the active list. This code section has the following inputs: Register $R1$ holds the exception virtual page index, register $R3$ is equal to one for reads and to two for writes, register $R4$ holds the address of the exception PTE, register $R5$ holds the exception PTE, register $R6$ holds the address of the UMPM entry, and register $R7$ holds the physical page index of the page corresponding to the UMPM entry.

First, we update the UMPM entry to hold the exception virtual page index and the current task ID. As was mentioned, the exception virtual page index is still stored in register $R1$. We have to load the current task ID from the corresponding variable in the system memory.

```
pfh_swap_in:
    mm4[R7 + umpm_vpx] := R1
    R8 := hi(ctid) ≪ 16
    R8 := R8 ⊕ lo(ctid)
    R8 := mm4[R8]
    mm4[R7 + umpm_tid] := R8
```

Second, we call the swap-in function. In register $R10$ it expects the physical page index and in register $R11$ the swap memory page index.

The swap memory index is the sum of the swap memory origin and the virtual page index. We will load the former to register $R11$ using the standard TCB table lookup procedure: we multiply the current TID with the TCB entry size, add to it the TCB table origin and load the *smo* entry:

```
R8 := R8 ≪ log2(tcb_size)
R9 := hi(tcb) ≪ 16
R9 := R9 ⊕ lo(tcb)
R8 := R8 + R9
R11 := mm4[R8 + tcb_smo]
```

The exception virtual page index is still stored in the register $R1$, so we may directly add it to the swap memory origin:

```
R11 := R11 + R1
```

We copy the physical page index from register $R7$ in the delay slot of the function call. Before calling *swap_in* we save the link register $R31$ temporarily in $R8$ and restore it after the call:

```
R8 := R31
PC' := swap_in, R31 := PC' + 4
R10 := R7
R31 := R8
```

Third, we must update the exception PTE. The PPX field of the PTE is set to the PPX register.

```
R5 := R5 ∧ 0^20 1^12
R7 := R7 ≪ 12
R5 := R5 ∨ R7
```

The valid bit is activated and the protection bit is activated if there is no logical write right. We first activate set the protection bit as well and invert it afterwards if the logical write right bit is active (via an exclusive-or operation).

```
R5 := R5 ∨ 0^20 1^2 0^10
R8 := R5 ∧ 0^30 10
R8 := R8 ≪ 9
R5 := R5 ⊕ R8
```

Finally, we write back the PTE and jump to the return address stored in register $R31$.

```
PC' := R31
mm4[R4] := R5
```

7.2 Simulation Theorem

In this section we prove a simulation theorem for the presented code. In Section 7.2.1 we shortly review the computation model of the user. Section 7.2.2 shows how hardware configurations are mapped to user configurations. In Section 7.2.3 we formulate and prove a lemma tailored to the page fault handler implementation. As we see in Section 7.2.4, this lemma implies the attachment invariant that was our main assumption in the simulation theorem in Chapter 4. Based on this observation, we show liveness of the simulated user machines in Section 7.2.5 and their functional correctness in Section 7.2.6.

7.2.1 Virtual Processor Model

A multitasking machine configuration $c_{mt} = (mem, atid, p, tr, r)$ consists of six entries:

- A (conceptually) shared memory $mem : [La \to \mathbb{B}^8]$.
- An active task ID predicate $atid(c) : [Tid \to \mathbb{B}]$ that is true for active tasks. It is also used to model the creation and termination of tasks.

- Let $R_{mt} := \{r0, r1, \ldots, r31, fpr0, fpr1, \ldots, fpr31, rm, ieeef, fcc\}$ denote the set of user-visible register identifiers. The processor configuration function p holds a value for all registers of all tasks. It has the signature

$$p : Tid \times R_{mt} \to \mathbb{B}^{32} . \tag{7.28}$$

- The translation function $tr : [Tid \times Va \to La]$ holds relocation information for each address; it can be used to introduce sharing. Here, we set $La = Tid \times Va$ and always have the translation function equal to the identity.

- The rights function $r : [Tid \times Va \to \{\emptyset, \{R\}, \{R,W\}\}$ holds the access rights for each task and for each virtual address. It is used to check for illegal operations; if a task tries to perform an illegal operation, it is killed (set to non-active).

We describe the computation of the machine. Without an active task, the multitasking machine does nothing (and hence enters an endless loop). Otherwise, the machine computes a step of an active task. Let *ctid* denote this task. The computation step uses the current register values $p(ctid)$ and may access the memory at virtual addresses, which are translated to physical addresses using the translation function $tr(ctid)$. If a task tries to perform a memory operation without a right for it, it will be deactivated. Hence, the next-state function for task *ctid*, denoted by $ns_{mt}(ctid) : C_{mt} \to C_{mt}]$, operates only on *mem*, $p(ctid)$, $tr(ctid)$, and $r(ctid)$.

7.2.2 Decode and Projection Functions

Based on the data structures of the operating system code, we now define the more abstract decode functions that we will use to formulate invariants on the code in the same way as in Chapter 4.

The current task identifier of some memory configuration *mm* is stored in the current task variable. We define

$$dec_{ctid}(mm) = mm_4[ctid] . \tag{7.29}$$

The active task identifier function dec_{atid}, the save area function dec_{sar}, and the first part of the implementation translation function dec_{itr1}, which determines the page table origin and the page table length of a task, are all defined by selecting parts of the TCB table. We define

$$dec_{atid}(mm,tid) = mm_4[tcba + tcb\_state][0] , \tag{7.30}$$

$$dec_{sar}(mm,tid)(R) = mm_4[tcba + tcb\_R] \text{ , and} \tag{7.31}$$

$$dec_{itr1}(mm,tid) = (\langle mm_4[tcba + tcb\_pto]\rangle, \langle mm_4[tcba + tcb\_ptl]\rangle) \tag{7.32}$$

where $tcba = tcb + tid \cdot tcb\_size$ is the base address of the TCB for task *tid*. Recall that dec_{itr1} models the software part of the implementation translation function (cf. Section 5.1.2). We maintain the (user mode) invariant

$$(pto, ptl) = dec_{itr1}(mm,tid) \tag{7.33}$$

for the special-purpose registers *pto* and *ptl*.

Let *tid* still denote a task identifier and $va = px \cdot 2^{12} + bx$ a virtual address. Furthermore, let $(pto, ptl) = dec_{itr1}(mm,tid)$ and let $pte = mm_4[pto \cdot 2^{12} + 4 \cdot px]$ abbreviate the page table entry used for the translation of *va*.

Logical rights are stored in the page table. Virtual pages with indices greater than the page table length have no logical rights. We define

$$dec_r(mm,tid)(va) = \begin{cases} \emptyset & \text{if } pte[1:0] = 0^2 \vee px > ptl\ , \\ \{R\} & \text{if } pte[1:0] = 01\ , \text{ and} \\ \{R,W\} & \text{if } pte[1]\ . \end{cases} \tag{7.34}$$

As was pointed out, we have no sharing of addresses, no two virtual addresses of arbitrary tasks are identified with each other. Hence, our logical address space corresponds to the cross product of the sets of task identifiers and the virtual addresses:

$$La = Tid \times Va \tag{7.35}$$

The logical translation function is the identity on its address inputs:

$$dec_{tr}(mm,tid,va) = (tid,va) \tag{7.36}$$

Hence, the logical translation does not depend on the memory configuration and we will silently replace it by the identity whenever it occurs.

The location of a logical address is encoded in the page table. Addresses outside of the page table are assumed to be located in the zero-page of the swap memory (which is, by convention, always filled with zeroes). If the page table entry is valid, the location is in main memory at the location given by the physical page index. Otherwise, the location is in swap memory; the swap memory page index can be computed by adding the virtual page index to the swap memory origin of that task. We set

$$dec_{laloc}(mm,(tid,va)) = \begin{cases} (0,0,0) & \text{if } px > ptl\ , \\ (1,0,\langle pte[31:20]\rangle \cdot 2^{12} + bx) & \text{if } pte[11]\ , \\ (0,(smo+px)\cdot 2^{12} + bx, 0) & \text{if } \neg pte[11]\ . \end{cases} \tag{7.37}$$

By the projection function Π, hardware configurations are mapped to multitasking configurations. Let $c_h = (mm,sm,p_h)$ be a hardware configuration consisting of a main memory configuration mm, a swap memory configuration sm, and a (hardware) processor state p_h. Let $\Pi(c_h) = c_{mt} = (mem,atid,p,tr,r)$. Then, we define

$$atid = dec_{atid}(c_h)\ , \tag{7.38}$$

$$tr = dec_{tr}(c_h)\ , \text{ and} \tag{7.39}$$

$$r = dec_r(c_h)\ . \tag{7.40}$$

Simulated processor registers are (among other implementation specific places) either to be found in the hardware registers or in the save area. The former case applies for the currently executing task if the hardware processor is in user mode. Otherwise, the latter case applies. Hence, we define

$$p(tid,r) = \begin{cases} p_h(r) & \text{if } tid = ctid \wedge \neg mode(p_h) \text{ and} \\ dec_{sar}(tid,r) & \text{otherwise.} \end{cases} \tag{7.41}$$

The memory configuration is defined with the help of the dec_{laloc} function. We set

$$mem(tid,va) = \begin{cases} mm(ma) & \text{if } inm \text{ and} \\ sm(sa) & \text{otherwise} \end{cases} \tag{7.42}$$

where $(inm,ma,sa) = dec_{laloc}(mm,tid,va)$.

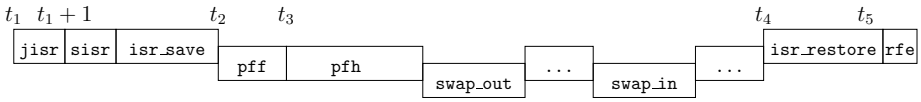

Figure 7.11 Call Structure for a Page Fault on Fetch

7.2.3 Implementation-Specific Page Fault Handler Correctness

Let us briefly sketch what should happen on a page fault. Assume we have a page fault on fetch or a page fault on load / store caused by a user mode instruction in cycle t_1. In this case, the machine enters system mode in cycle t_1+1. After completion of the entry part of the ISR we eventually reach cycle t_2, in which one of the wrappers pff or $pfls$ starts to execute, and subsequently a cycle t_3 which marks the entering of the page fault handler pfh. If the page fault was a memory management exception (i.e. not due to an illegal memory operation) we require the existence of a cycle t_4 when we return from pfh and of a cycle t_5 when we complete the ISR exit part, entering user mode in cycle t_5+1. Figure 7.11 depicts this situation for a page fault on fetch. If the page fault was a violation exception (i.e. for an illegal exception operation) the current task must be deactivated.

In this section we concentrate on memory management exceptions and on the execution of the pfh function, i.e. the interval from t_3 to t_4. Correctness of this part is expressed via pre- and post-conditions.

Some of the conditions are purely technical. For example, we want to state that the page fault handler is *entered* at time t_3, terminates, and *returns* at a later time t_4 (possibly calling the *swap_out* and *swap_in* functions in between). We denote these conditions by $pfh\_enter^{t_3}$ and $pfh\_call(t_3,t_4)$. Both might be defined as follows:

$$pfh\_enter^{t_3} = (dpc^{t_3} = pfh) \wedge (pcp^{t_3} = pfh+4) \wedge (mode^{t_3} = 0) \tag{7.43}$$

$$pfh\_call(t_3,t_4) = pfh\_enter^{t_3} \wedge (pcp^{t_4} = R31^{t_3}) \tag{7.44}$$

With regards to the input arguments in $R1$ and $R3$ (exception page index and operation) we define the pre-condition $pfh\_mme^{t_3}$ identifying this case by

$$\begin{aligned} pfh\_mme^{t_3} = \; & \neg att(mm^{t_3}, dec_{ctid}(mm^{t_3}), R1^{t_3} \cdot 2^{12}, R3^{t_3}) \\ & \wedge R3^{t_3} \in dec_r(mm^{t_3}, dec_{ctid}(mm^{t_3}), R1^{t_3} \cdot 2^{12}) \; . \end{aligned} \tag{7.45}$$

where the attachment predicate $att(mm,tid,vpx,mop)$ is true iff no exception is indicated by the translation $dec_{itr}(mm,tid,vpx,mop)$ (cf. Section 4.6.1).

The invariant $pfh_1^{t_3}$ is both a pre- and a post-condition. It states that the data structures of the page fault handler are in a "correct" state, e.g. the UMPM lists are intact and there is exactly one entry for each user page. As we prove in Section 7.2.4 it entails the attachment invariant (cf. Section 4.6.1).

Finally, we have the post-condition $pfh_2(t_3,t_4)$, which is the conjunction of the following four conditions:

- The active list is of non-zero size at time t_4 and its tail entry corresponds to the exception page (according to the inputs at time t_3). Together with pfh_1 this requirement is equivalent to the attachment of the exception operation.
- If the active list was of non-zero size at time t_3 as well, the tail of the active list at time t_3 is still present in the active list at time t_4. Together with pfh_1

this requirement is equivalent to the attachment of the penultimate exception operation and used to prove the absence of an infinite loop of page faults.

- Certain data structures must not be changed. These include the TCBs and the current task ID.
- The projected user memory must not be changed; this condition is called tamper-freeness.

In the following sections, we formally define the conditions pfh_1 and pfh_2 and prove the correctness lemma for management exceptions:

The page fault handler satisfies the following property: ◀ Lemma 7.1

$$pfh\_enter^{t_3} \wedge pfh\_mme^{t_3} \wedge pfh_1^{t_3} \Rightarrow \exists t_4 > t_3 : pfh\_call(t_3,t_4) \wedge pfh_1^{t_4} \wedge pfh_2(t_3,t_4) \quad (7.46)$$

Afterwards we apply this result to show that for a complete ISR execution memory management exceptions are handled correctly. A similar pair of lemmas can be proven for violation exceptions.

The Invariant pfh_1

In Section 7.1.2, Equations 7.16 to 7.18, we have already defined three UMPM invariants: (i) *activelist* and *freelist* are valid list descriptors, (ii) their elements are disjunct, and (iii) they comprise all the (used) entries of the UMPM table.

Additionally, the elements stored in the active list correspond exactly to those memory operations with full attachment (yet no over-attachment) and the translated address of those memory operations is determined by the address of the UMPM entry. We formalize this.

Let $a \in Elem(activelist)$ be an element of the active list, i.e. the address of an UMPM entry. By its *tid* and *vpx* entries, this entry is associated with the logical address $(a.tid, a.vpx \cdot 4096)$; we require that *tid* belongs to an active task and *vpx* is a page with non-empty rights:

$$dec_{atid}(a.tid) \wedge dec_r(a.tid, a.vpx \cdot 4096) \neq \emptyset \quad (7.47)$$

Furthermore, the UMPM entry with address a is associated with a physical page index determined by its offset in the UMPM table. We defined the (imaginary) base address *umpm*0 as belonging to the physical page with index 0. By subtracting *umpm*0 and dividing by the size of an UMPM entry, we determine the physical page index

$$ppx = (a - umpm0)/umpm\_size\ , \quad (7.48)$$

which is associated with the UMPM entry at address a. We require that all legal memory operations for $(a.tid, a.vpx \cdot 4096)$ are attached and translated to the physical address $ppx \cdot 4096$:

$$mop \in dec_r(a.tid, a.vpx \cdot 4096) \Rightarrow dec_{itr}(a.tid, a.vpx, mop) = (0, ppx \cdot 4096) \quad (7.49)$$

Furthermore, there is no over-attachment, memory operations without an existing right must not be attached:

$$mop \notin dec_r(a.tid, a.vpx \cdot 4096) \Rightarrow \neg att(a.tid, a.vpx, mop) \quad (7.50)$$

Now, we need to express the property that virtual addresses without entry in the active list have no attachment. The contraposition of this property is, that all attached memory operations have a corresponding entry in the active list. Let mop denote a memory operation, tid a task identifier and vpx some virtual page index. If the operation mop on the address $vpx \cdot 4096$ is attached for task tid,

$$att(mop, tid, vpx \cdot 4096)\,, \tag{7.51}$$

we need to find an entry $a \in Elem(activelist)$ with

$$a.vpx = vpx \wedge a.tid = tid\,. \tag{7.52}$$

Some more conditions are necessary to guarantee correct behavior of the page fault handler:

- The current task must be active, i.e. $dec_{atid}(mm, dec_{ctid}(mm))$ holds.
- The user memory contains at least two pages:

  $$8192 - mm_4[fuppx] \geq 2 \tag{7.53}$$

  Otherwise, it is impossible to guarantee liveness for load / store instructions that need to access up to two pages to execute without a page fault.

The invariant pfh_1 is not complete as shown here. We are (deliberately) leaving out low-level details like absence of interrupts, absence of self-modification and non-tampering with data structures of other parts of the operating system.

The Postcondition pfh_2

We want to establish four postconditions in addition to the invariants. These conditions are formulated in terms of the starting time t and the ending time t' of the page fault handler execution.

- By the operation restrictions $or(t,t')$ we require, that the page fault handler must not modify certain data structures. In particular, the logical rights and translations, the active tasks, and the save area must not be modified:

  $$dec_{ctid}(mm^t) = dec_{ctid}(mm^{t'}) \tag{7.54}$$
  $$dec_r(mm^t) = dec_r(mm^{t'}) \tag{7.55}$$
  $$dec_{tr}(mm^t) = dec_{tr}(mm^{t'}) \tag{7.56}$$
  $$dec_{atid}(mm^t) = dec_{atid}(mm^{t'}) \tag{7.57}$$
  $$dec_{sar}(mm^t) = dec_{sar}(mm^{t'}) \tag{7.58}$$

- Tamper-freeness $tf(t,t')$ states that the projection of the hardware configuration (which represents configuration visible to the user) does not change:

  $$\Pi(c_h^t) = \Pi(c_h^{t'}) \tag{7.59}$$

- If there was a tail of the active list in time t, then it is still present in the active list in time t':

  $$size^t(activelist) > 0 \Rightarrow activelist.prev^t \in elem(activelist^{t'}) \tag{7.60}$$

The task identifier and virtual page index entries of the tail must not change,

$$(a.tid^{t'} = a.tid^{t}) \wedge (a.vpx^{t'} = a.vpx^{t}) \tag{7.61}$$

where $a = activelist.prev^{t}$.

- The exception page index (parameter $R1$ at time t) is at the tail of the active list in time t', i.e.

$$size^{t'}(activelist) > 0 \wedge a.tid^{t'} = dec^{t'}_{ctid} \wedge a.vpx^{t'} = R1^{t'} \tag{7.62}$$

where $a = activelist.prev^{t'}$.

Proof Sketch

Termination of the page fault handler can be easily shown: under the assumption that the swap-in and swap-out functions are terminating, the page fault handler must be terminating since is does not contain loops.

Assume $elem(mm^{t}, freelist) = (f_1, \ldots, f_n)$ and $elem(mm^{t}, activelist) = (a_1, \ldots, a_m)$ with n and m being the respective list sizes. By the pfh_1 invariant, we know that $n + m = 8192 - mm^{t}_{4}[fuppx]$ and $f_i \neq a_j$ for all i, j.

Consider now two cases.

First, assume that the free list was not empty in time t, so $n > 0$. In this case, possibly $m = 0$ (meaning we are currently fixing the first page fault after initialization).

Since we have a legal access by assumption, we will execute the "logical rights check", the "free list check", the "choose a page from the free list", and the "swap in exception page and update PTE" parts, the last with a nested *swap_in* call.

The page fault handler takes the head of the free list f_1 and puts it to the tail of the active list. Hence,

$$elem(mm^{t'}, freelist) = (f_2, \ldots, f_n) \text{ and} \tag{7.63}$$

$$elem(mm^{t'}, activelist) = (a_1, \ldots, a_m, f_1)\ . \tag{7.64}$$

Clearly, the list structure at time t' is still intact and the elements of both lists are disjunct. The former tail of the active list is still present as the second-last element of the active list (and unchanged).

The only page table entry that is modified is that of the exception page. It is attached with full logical rights and its ppx entry is set to $(f_1 - umpm0)/umpm\_size$. The page fault handler also sets $f_1.tid = dec_{ctid}(mm^{t})$ and $f_1.vpx = R1^{t}$.

Since we know that the exception operation was not attached in time t by assumption and we have full attachment, there is no element a_i in the active list that has the same task identifier and virtual page index entries. Therefore, all properties involving the old elements a_i of the active list and their corresponding operations are still satisfied.

As the page fault handler only writes to the UMPM table (and a single page table entry), the list descriptors and the page tables, the current task identifier, the logical translation, the active task identifiers, and the save area are unchanged. Also, the page fault handler leaves the two lower bits of a page table entry untouched and therefore the logical rights do not change, either. Hence, the operation restrictions are satisfied.

For tamper-freeness we must argue about the swap-in call the page fault handler makes. The page fault handler copies a page from the swap-memory address to some

physical page and then attaches the page, thus also updating the logical address location function. The claim is that it swaps the right pages, i.e.

$$mm^{t'}_{4096}(ma(dec_{laloc}(mm^{t'}, R1^{t} \cdot 4096))) = sm^{t}_{4096}(sa(dec_{laloc}(mm^{t}, R1^{t} \cdot 4096))) \; . \quad (7.65)$$

The projection thus does not change. This proves all claims for the first case.

Second, assume the free list was empty, so $n = 0$. In this case, the page fault handler will detach all operations associated with a_1 and swap-out the page to the swap-space location. Hence, the projection does not change. The remaining arguments are similar to the previous case.

Calling of the Page Fault Handler

In this section we show a lemma on memory management exceptions in terms of executions of the ISR. We do this by (i) identifying properties of the ISR's entry and exit part and (ii) combining these with Lemma 7.1.

To reason on the ISR, we have to "translate" the properties of the page fault handler first. The condition $pfh\_enter^{t_3}$ translates to $isr\_pfh\_enter^{t_1}$, which detects a transition to system mode with an interrupt level of 3 (page fault on fetch) or 4 (page fault on load / store) by

$$isr\_pfh\_enter^{t_1} = \neg mode^{t_1} \wedge il^{t_1} \in \{3,4\} \; . \quad (7.66)$$

The condition $pfh\_mme^{t_3}$ on having a memory management exception translates to

$$\begin{aligned} isr\_pfh\_mme^{t_1} = \; & \neg att(mm^{t_1}, dec_{ctid}(mm^{t_1}), evpx^{t_1} \cdot 2^{12}, emop^{t_1}) \\ & \wedge emop^{t_1} \in dec_r(mm^{t_1}, dec_{ctid}(mm^{t_1}), evpx^{t_1} \cdot 2^{12}) \end{aligned} \quad (7.67)$$

where the exception page index $evpx^{t_1}$ and operation $emop^{t_1}$ are defined as

$$evpx^{t_1} = \begin{cases} dpc^{t_1}[31:12] & \text{if } il^{t_1} = 3 \; , \\ ea^{t_1}[31:12] & \text{if } il^{t_1} = 4 \text{ and} \end{cases} \quad (7.68)$$

$$emop^{t_1} = \begin{cases} 0 & \text{if } il^{t_1} = 3 \vee \neg ir^{t_1}[29] \; , \\ 1 & \text{if } il^{t_1} = 4 \vee ir^{t_1}[29] \; . \end{cases} \quad (7.69)$$

The save part of the ISR must store the user registers into the current TCB and store registers $R1$ and $R2$ additionally into the save locations. The memory must not be modified elsewhere. We abbreviate this condition as $isr\_save(t_1, t_3)$ without defining it formally.

Overall, for the entry part of the ISR we require

$$\begin{aligned} & isr\_pfh\_enter^{t_1} \Rightarrow \exists t_3 > t_1 : \\ & \quad isr\_save(t_1, t_3) \wedge pfh\_enter^{t_3} \wedge (R1^{t_3} = evpx^{t_1}) \wedge (R3^{t_3} = emop^{t_1}) \; . \end{aligned} \quad (7.70)$$

The exit part starts to execute at time $t_4 + 2$ after return of the page fault handler; we denote this fact by $isr\_exit\_enter^{t_4+2}$. We demand the existence of a time t_5 such that the user registers are restored from the current TCB, the processor runs in user mode again, and, apart from the save locations $save\_r1$ and $save\_r2$, the memory stays unchanged. We abbreviate this condition as $isr\_restore(t_4 + 2, t_5)$ without defining it formally. Overall, we demand for the exit part

$$isr\_exit\_enter^{t_4+2} \Rightarrow \exists t_5 > t_4 + 2 : isr\_restore(t_4 + 2, t_5) \; . \quad (7.71)$$

Equation 7.70, Equation 7.71, and Lemma 7.1 are strong enough to allow for a compositional proof of a new result, the correctness of the interrupt service routine for memory management exceptions.

The interrupt service routine satisfies the following property:

$$isr\_pfh\_enter^{t_1} \wedge isr\_pfh\_mme^{t_1} \wedge pfh_1^{t_1} \Rightarrow \exists t_5 > t_1 : \\ pfh_1^{t_5} \wedge pfh_2(t_1,t_5) \wedge \neg mode^{t_5+1} \wedge \forall r \in R_{mt} : r^{t_5} = r^{t_1} \quad (7.72)$$

Key to the proof are the restrictions on the memory updates of the ISR entry part, the page fault handler, and the ISR exit part. The proof is straightforward and therefore omitted here.

7.2.4 The Attachment Invariant

In this section we prove that the page fault handler invariant pfh_1 implies the attachment invariant (cf. Sections 4.4.3 and 4.6.1).

Definition
The attachment invariant states that all attached memory operations satisfy the access conditions, i.e.

$$\forall mop, tid, va : att(mm, mop, tid, va) \Rightarrow ac(mm, mop, tid, va) \quad (7.73)$$

where

$$ac(mm, mop, tid, va) = \begin{array}{l} sys(mm, mop, tid, va) \wedge rc(mm, mop, tid, va) \wedge \\ lc(mm, mop, tid, va) \wedge cow(mm, mop, tid, va)\ . \end{array} \quad (7.74)$$

Assume that $dec_{itr}(mm, tid, va, mop) = (0, ma)$. The access conditions are the following:

- The system memory condition *sys* states, that the translated address must not be inside the system memory:

  $$sys(mm, mop, i, va) \Leftrightarrow px(ma) \notin Sys \quad (7.75)$$

- The rights consistency condition *rc* states, that there is a right for the operation *mop*:

  $$rc(mm, mop, tid, va) \Leftrightarrow mop \in dec_r(mm, tid, va) \quad (7.76)$$

- The *laloc* consistency condition *lc* states that the translated address *ma* must equal the address returned by the logical address location function. Let

  $$(inm, ma_f, sa_f) = dec_{laloc}(mm, dec_{tr}(mm, tid, va))\ . \quad (7.77)$$

  We define:

  $$lc(mm, mop, tid, va) \Leftrightarrow inm \wedge ma_f = ma \quad (7.78)$$

  Observe that by definition of the logical address location function, this property may not be violated in our easy scenario: attachment already implies that $ma_f = ma$.

- The copy-on-write condition states that the address *ma* must not be shared by a logically different address.

$$cow(mm,mop,tid,va) \Leftrightarrow (mop = W \Rightarrow \neg sh_{pnl}(mm,tid,va)) \tag{7.79}$$

where

$$\begin{aligned} &sh_{pnl}(mm,tid,va) \Leftrightarrow \\ &\quad \exists tid', va' : \ dec_{tr}(mm,tid,va) \neq dec_{tr}(mm,tid',va') \wedge \\ &\qquad dec_{laloc}(mm, dec_{tr}(mm,tid,va)) = \\ &\qquad\quad dec_{laloc}(mm, dec_{tr}(mm,tid',va')) \ . \end{aligned} \tag{7.80}$$

Since dec_{tr} is the identity function (we do not have address sharing yet), we may simplify

$$\begin{aligned} &sh_{pnl}(mm,tid,va) \Leftrightarrow \\ &\quad \exists tid', va' : \ (tid',va') \neq (tid,va) \wedge \\ &\qquad dec_{laloc}(mm,(tid,va)) = dec_{laloc}(mm,(tid',va')) \ . \end{aligned} \tag{7.81}$$

Establishing the Attachment Invariant

Lemma 7.3 ▶ *The list invariant is stronger than the attachment invariant:*

$$\forall mm : pfh_1(mm) \Rightarrow I(mm) \tag{7.82}$$

PROOF The attachment invariant is quantified over all memory operations. Let (mop,tid,va) be an attached memory operation, so $att(mm,mop,tid,va)$ holds. Let vpx and bx be the page and byte index of va. Thus, va may be written as $va = vpx \cdot 4096 + bx$. Also, since the operation mop is attached, we let ppx denote the translated physical page index, it satisfies $dec_{itr}(mm,tid,va,mop) = (0, ppx \cdot 4096 + bx)$.

We also note that $att(mm,mop,tid,vpx \cdot 4096)$, and $dec_r(mm,tid,vpx \cdot 4096) = dec_r(mm,tid,va)$ because of granularity; we will need these simple facts later on.

All attached memory operations have a corresponding element in the active list by the pfh_1 invariant. Let $a \in Elem(activelist)$ be this element, satisfying $a.tid = tid$ and $a.vpx = vpx$. The element a is uniquely defined. If we assume otherwise that $a' \neq a$ with $a'.tid = tid$ and $a'.vpx = vpx$ would exist, then we immediately get a contradiction against the inequality by the ppx condition (Equation 7.48) through $(a - umpm0)/umpm\_size = (a' - umpm0)/umpm\_size$.

By the same condition, we obtain $ppx = (a - umpm0)/umpm\_size$. Since

$$Elem(activelist) \subseteq \{umpm0 + i \cdot umpm\_size \mid mm_4[fuppx] \leq i < 8192\} \tag{7.83}$$

we have $a = umpm0 + j \cdot umpm\_size$ for some $j \geq mm_4[fuppx]$. Hence,

$$\begin{aligned} ppx &= (a - umpm0)/umpm\_size && (7.84)\\ &= j && (7.85)\\ &\geq mm_4[fuppx] \ . && (7.86) \end{aligned}$$

This is equivalent to $ppx \notin Sys$. This shows the system memory condition.

Now we show $mop \in dec_r(mm,tid,va)$. Let us assume otherwise. Since we do not allow over-attachment, we obtain

$$mop \notin dec_r(a.tid, a.vpx \cdot 4096) \Rightarrow \neg att(a.tid, a.vpx, mop) \ , \tag{7.87}$$

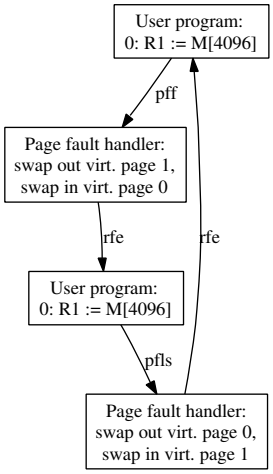

Figure 7.12 An Infinite Loop of Page Faults

which contradicts our assumption. Thus, the attached operation is consistent with the given rights.

Attachment implies that the valid bit v of the PTE is set. Therefore, by definition, the dec_{laloc} function returns an active in-main-memory flag and the same address as the address translation:

$$dec_{laloc}(mm,(tid,va)) = (1,0,ppx \cdot 4096 + bx) \tag{7.88}$$

Hence, the *laloc* consistency condition holds.

Finally, we show that the copy-on-write condition holds because there is no *pnl*-sharing (physical but not logical sharing) in our system. Assume otherwise, i.e. there exists (tid',va') which is a witness for $sh_{pnl}(mm,tid,va)$. Hence,

$$(tid',va') \neq (tid,va) \wedge dec_{laloc}(mm,(tid,va)) = dec_{laloc}(mm,(tid',va'))\ . \tag{7.89}$$

Since $dec_{laloc}(mm,(tid,va)) = (1,0,ppx \cdot 4096 + bx)$, we may assume that

$$dec_{laloc}(mm,(tid',va')) = (1,0,ppx' \cdot 4096 + bx)\ . \tag{7.90}$$

By definition of the dec_{laloc} function, the valid bit of the PTE for the address (tid',va') must be set, the address (tid',va') must at least be attached for the read operation. Therefore, a uniquely defined address $a \in Elem(activelist)$ exists that corresponds to it, i.e. $a'.tid = tid'$ and $a'.vpx = px(va')$. Since $(tid',va') \neq (tid,va)$ we have $a' \neq a$. By the ppx condition, different active list element addresses are associated with different physical page indices. Therefore, we get the contradiction $ppx \neq ppx'$.

Hence, $att(mm,mop,tid,va) \Rightarrow \neg sh_{pnl}(mm,tid,va)$, which suffices to show the copy-on-write condition.

7.2.5 Liveness

VAMP load / store instructions can generate up to two page faults. Because instruction repeats always start over by fetching the instruction again, in extremely tight memory

situations (only one page in user memory) or for a badly written handler an infinite loop of page faults may occur (as in Figure 7.12). We prove that our page fault handler does not exhibit such a behavior.

Lemma 7.4 ▶ *If tasks perform only legal memory operations, there are infinitely many user mode steps without interrupt, i.e. the set* $\{t \mid mode^t \wedge \neg JISR^t\}$ *is infinite.*

PROOF Currently, the only interrupts which occur in the computation are a reset interrupt at the beginning of the computation and page faults. Since both the initialization routines and the page fault handlers (for legal operations) are terminating, there are infinitely many user mode steps and the set $\{t \mid mode^t\}$ is infinite.

Assume that $\{t \mid mode^t \wedge \neg JISR^t\}$ is finite. Then, we can find three indices $0 < t_1 < t_2 < t_3$ denoting user mode steps with only system mode steps in between:

$$\begin{aligned} &mode^{t_1} \wedge mode^{t_2} \wedge mode^{t_3} \wedge \\ &JISR^{t_1} \wedge JISR^{t_2} \wedge JISR^{t_3} \wedge \\ &\forall t_1 < t' < t_3, t' \neq t_2 : \neg mode^{t'} \end{aligned} \tag{7.91}$$

Since $t_1 > 0$, the intervals t_1 to t_2 and t_2 to t_3 each correspond to a page fault handler execution. We contradict $JISR^{t_3}$.

Since the page fault handler does not switch tasks, the current task identifier and the user mode program counter remain unchanged in any of the indexed user mode steps:

$$dpc^{t_1} = dpc^{t_2} = dpc^{t_3} \tag{7.92}$$

Recall that all elements of the active list are fully attached. Since the page fault handler guarantees on return that the exception page is the tail of the active list and that all elements of the active list are fully attached, the same page fault cannot directly repeat. Therefore, the types of page faults must alternate (this is not necessary for the proof, we just use it for illustration purposes). Since the exception at t_1 is either a page fault on fetch or a page fault on load / store, only two possible sequences of page faults may occur, $pff^{t_1} \wedge pfls^{t_2} \wedge pff^{t_3}$ or $pfls^{t_1} \wedge pff^{t_2} \wedge pfls^{t_3}$.

The page fault handler also guarantees that the tail of the active list is still present in the active list after its execution. Hence, $JISR^{t_3}$ cannot hold, contradicting our assumption and proving that $\{t \mid mode^t \wedge \neg JISR^t\}$ is infinite.

Note that this lemma holds with the introduction of an (external) timer interrupt. Since this interrupt is of type *continue* and would be of lower priority than the page faults, repeats of page-faulting instructions are either completed without a page fault or directly enter the page fault handler again. So, even then the page fault handler guarantees that eventually all page faults of a single instruction will be fixed.

Of course, the above proof also shows that any given legal memory operation eventually completes:

Lemma 7.5 ▶ *If a task tries to execute an instruction with legal memory operations at time* t_1 *(the fetch is legal and, if it is a load / store instruction, the read / write is legal, too), a time* t_3 *exists when this instruction does not cause a page fault.*

This property is also called *guaranteed forward progress*.

7.2.6 Correctness

Consider a hardware computation $(c_h^0, c_h^1, \ldots)$. We mark out certain hardware configurations by defining a strictly monotonic index function $idx : [\mathbb{N} \to \mathbb{N}]$. With $idx(0) := \min\{t > 0 | mode^{t+1}\}$ we mark the end of the initialization, the configuration before entering user mode for the first time. Thereafter, we inductively mark every hardware configuration in which an exception-free user mode step is performed by

$$idx(i+1) = \min\{t > idx(i) \mid mode^t \wedge \neg JISR^t\} \; . \tag{7.93}$$

Clearly, idx is strictly monotonic.

We assume that the current task identifier may change in after returning from the interrupt handler of a non-page fault interruptions; we have not explicitly modeled this behavior, typically caused by the periodic calling of an operating system's scheduler through (cf. the lecture notes [PDM04] for details).

Consider again the projection function Π defined in Section 7.2.2. Let $c_{mt}^0 := \Pi(c_h^{idx(0)+1})$. With $ctid(i) = dec_{ctid}(idx(i))$ we denote the task scheduled for execution in step i. The multitasking computation is defined inductively by

$$c_{mt}^{i+1} := ns_{mt}(ctid(i))(c_{mt}^i) \tag{7.94}$$

where the next state function $ns_{mt}(tid)$ executes one step for task tid.

◀ Theorem 7.6

If (i) the reset line is active in cycle 0 *(so* $reset^0 = 1$*), (ii) the main memory contains the operating system code with a valid interrupt vectors table initially (in cycles* 0 *and* 1*), (iii) the swap functions terminate and are functionally correct, and (iv) the only interrupts caused in user mode are page faults, then the projected hardware configurations (as selected by the index function idx) correspond to the multitasking configurations:*

$$\forall i : c_{mt}(i) = \Pi(c_h(idx(i)+1)) \tag{7.95}$$

PROOF

The proof is very similar to the simulation theorem for the multiprocessor case; therefore we will just provide a proof sketch and highlight the differences. We prove the claim by induction over i. The induction start is satisfied by the initialization.

For $i > 0$, assume $c_{mt}(i) = \Pi(c_h(idx(i)+1))$ and we want to show $c_{mt}(i+1) = \Pi(c_h(idx(i+1)+1))$.

First, we claim that the projection of the memory did not change from step $idx(i)+1$ to step $idx(i+1)$. If $idx(i)+1 = idx(i+1)$ there is nothing to show. Otherwise, the time interval between $idx(i)+1$ and $idx(i+1)$ is a sequence of page-faulting user mode instructions and page fault handler calls which both do not change the projected memory configuration. Hence,

$$idx(\Pi(c_h(idx(i+1)))) = \Pi(c_h(idx(i)+1)) \; . \tag{7.96}$$

Second, we apply the step lemma (cf. Section 4.4.4). Since $idx(i+1)$ denotes a user mode step without a page fault and the attachment invariant holds in cycle $idx(i)$ by the page fault handler correctness, the memory operations of the user mode instructions are all attached and satisfy the access conditions. Therefore, for both the instruction fetch and any load / store operation the step lemma applies yielding correctness of the simulation.

```
        R1 := 0xac00     // bits [31:16] of mm4[R0+x] := R0
        mm2[loc+2] := R1  // modify bits [31:16] of next instr.
    loc:
        R2 := mm4[x]      // page fault on load
```

Figure 7.13 Malevolently Self-Modifying User Program. Suppose this code was placed in the first 32*K* of the program's virtual memory, that means *loc* is directly addressable. The first two instructions modify the instruction at address *loc*. Due to the missing `sync`, the old instruction at *loc* might be fetched (depending on the processor pipeline). If the old instruction causes a page fault on load, it is mistaken as a page fault on store by the page fault handler, which only sees the new instruction.

7.3 Extensions

We conclude with a short list of extensions to our system. Since optimization approaches for memory management and page fault handling are too numerous to be discussed here, we focus on two goals. First, we show how to use virtual memory methods to simplify the user's computational model and ease verification. Second, we show (in three subsections) how to develop the simplistic page fault handler into a competitive one.

We do not cover sharing techniques (logical or copy-on-write) as these would require substantial additions to data structures and algorithms.

7.3.1 Dealing with Unrestricted Self-Modification

We have seen that prefetching has forced us to introduce the synced code property on the architecture level that regulates the use of self-modification. The hardware correctness proof was explicitly performed only for programs conforming to this property. Unless this property is proven to hold, the result of any program running on the actual hardware is, formally, unpredictable.

Moreover, it is not known, whether (malevolent) programs violating the synced code property can do any 'harm'. In an operating system environment, 'harm' means compromising other tasks or the operating system itself. As an example, consider the system presented in this chapter. To distinguish a page fault on load from a page fault on store, the handler had to emulate instruction fetch. Hence, an unsynchronized self-modifying program can make the page fault handler mistake one for the other (cf. Figure 7.13). In this particular case, though, no damage can be done.

To handle both problems, we propose to write the operating system in such a way that it enforces the synced code property on user programs. User programs, on the other hand, may then be based on regular sequential semantics without restrictions on modifications; this simplifies software verification.[6]

Recall that in Section 5.1.4 we identified three causes that "synchronize" the processor. Either a special synchronization instruction is executed, or a return-from-exception instruction is executed, or an interrupt is detected. The idea of the approach is to exploit the last condition, to somehow make user programs cause an interrupt on

[6]Otherwise, the architecture has to be strongly tailored to the processor implementation imitating its pipeline behavior or it operate indeterministically. Both approaches seem much worse.

self-modification.

This is most easily done on a three-rights machine (cf. Section 3.3.2) with a separate right for fetching an instruction. As an invariant, the operating system keeps the rights to write to and fetch from a page exclusive. Whenever an exception is encountered, the rights are toggled. This way, before data written by the user can be executed, a (pseudo-) exception is inserted in the computation of the machine. There is one case requiring special care: if an instruction writes to the page it currently fetches from, the toggling of rights leads to an infinite loop. In this case, the operating system has to resort to *emulation*, so that the next state of the user program is computed in system mode / in software. The performance impact of such a policy resembles, on a larger scale, that of the synchronization instruction: for code loading and compilation the impact is acceptable, whereas highly localized self-modification and low-level programming tricks are discouraged.

For the two-rights machine *without* write-only rights (as the one presented), the situation is much worse and all modifications must be emulated.

We remark that self-modification is used in implementations of virtual-machine based languages to increase performance by just-in time compilation, the dynamic translation of virtual into machine code. This technique was invented for Smalltalk-80 [DS84] and is in wide use today (e.g. for Java).

7.3.2 Dirty Bits

Dirty bits are associated with virtual pages in main memory. They are cleared on swap-in and set on modification of a page. A page that has a dirty bit equal to zero is called *clean*. If a clean page is chosen for eviction, it needs not to be swapped-out; this saves one I/O operation.

While some architectures have hardware support for dirty bits, they can also be emulated in software without significant performance loss [Dra91]. In order to do so, we store a software-managed dirty bit $d = pte[9]$ in each page table entry $pte[31:0]$. On swap-in, the dirty bit is cleared, the valid bit $v = pte[11]$ and the protection bit $p = pte[10]$ are set. Thus, the first user write to the page generates a page fault. On seeing the page valid ($v = 1$), the page fault handler sets the dirty bit, clears the protection bit, and returns without performing any swapping. Subsequent user write operations proceed at full speed.

As a short-cut, a page swapped-in due to a page fault on store can directly be marked dirty.

With respect to correctness, the page fault handler invariants must be changed since elements of the active list might not be fully attached. Additionally, clean valid pages in main memory are equal to their versions in swap memory. If the above short-cut is implemented, no instruction may generate more than two page faults in a row, so liveness properties remain unchanged.

7.3.3 Reference Bits

Like dirty bits, reference bits are associated with virtual pages in main memory, too. However, they are not only set on modification of a page but additionally on reading a page. If cleared under software control at certain intervals, reference bits thus indicate whether a page has recently been used or not. This information helps the page fault

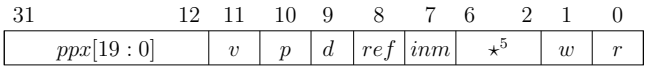

Figure 7.14 Page Table Entry with Reference, Dirty, and In-Main-Memory Bits

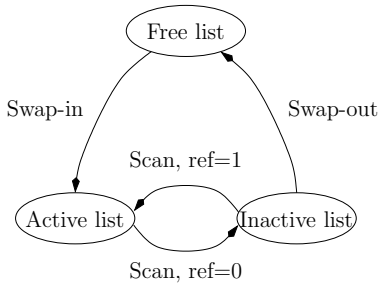

Figure 7.15 UMPM Lists for the FIFO with Second Chance Algorithm

handler to swap-out pages that are less likely to be used in the near future.

We show how to emulate reference bits in software [Dra91]. A reference bit $ref = pte[8]$ and an in-main-memory bit $inm = pte[7]$ are stored in each page table entry $pte[31:0]$. Figure 7.14 shows the extended format of the page table entry. On swap-in, the in-main-memory bit, the reference bit, and the valid bit are set. On clearing (initiated as described below), both the valid bit and the reference bit are reset. Thus, any user access to the page generates a page fault, a so-called *reference fault*. On seeing the page in main memory ($inm = 1$), the page fault handler sets both the valid and reference bit and returns without performing any swapping. Subsequent user operations proceed at full speed. On swap-out, the in-main-memory bit *inm* must be cleared. Hence, with reference bit emulation, not the valid bit but the in-main-memory bit indicates whether some page is in main or in swap memory.

To make use of reference bits, the page fault handler algorithm must be changed. Popular algorithms are the 2-hand clock algorithm [LMKQ89] and FIFO with second chance [TL81, Dra91]. We sketch the latter, an extension of the FIFO algorithm. Instead of two lists, the page fault handler maintains three lists: the free list, the active list, and the inactive lists. The elements of these lists are associated with free, frequently-used and less frequently-used pages in user memory. Candidates for swap-in are chosen from the free list and then placed on the active list. Candidates for swap-out are taken from the inactive list and then placed on the free list. If the free list or the inactive list are insufficiently filled (in worst case they are empty), then the active list and the inactive list are *scanned*:

- A page is moved from the inactive to the active list if its reference bit is set. Otherwise, it is moved from the inactive list to the free list (performing the necessary swap-out operation first).
- A page is moved from the active list to the inactive list if its reference bit is cleared.

In any case, reference bits are cleared on scanning. Scanning may be stopped if the free list or the inactive list have reached a certain minimum size. The different page moves are depicted in Figure 7.15.

With this algorithm, pages that are often referenced almost always have their reference bits set and thus stay in the active list. Less referenced pages will be moved to the inactive list and swapped out eventually. Benchmarks may be used to justify this choice; the results of such benchmarks are typically compared against an LRU replacement policy, which performs good but is far too expensive to implement.

In contrast to implementing dirty bits, reference bits have very little impact on the correctness arguments. It must be shown however that scanning the lists does not impede liveness.

7.3.4 Asynchronous Paging

Asynchronous paging uses non-blocking I/O operations to perform swapping operations "in background" thus increasing CPU utilization. It strongly depends on details of the scheduler and the I/O subsystem.

With asynchronous paging, the page fault handler must maintain information on the ongoing swap operations. Abstractly, such information can be represented as a set $\mathcal{S}$ of quadruples (dir,tid,vpx,ppx) where $dir \in \mathbb{B}$ denotes the direction of the swapping operation (0 for swap-out, 1 for swap-in), tid and vpx denote the task identifier and the virtual address, and ppx denotes the physical page index.

Let us briefly characterize the swapping set $\mathcal{S}$ further.

- A task tid with some element $(1,tid,vpx,ppx) \in \mathcal{S}$ waits for a swap-in to complete and must not be scheduled.
- Any physical page ppx with an associated element in $\mathcal{S}$ is used for swap-in or swap-out and must not be chosen for eviction.
- Elements are put into $\mathcal{S}$ at the start of a swapping operation and are taken out of $\mathcal{S}$ at the completion of a swapping operation.
- When a quadruple $(0,tid,vpx,ppx)$ is put into the set, the page table entry for vpx of tid is invalidated.
- When a quadruple $(1,tid,vpx,ppx)$ is taken out of the set, the page table entry for vpx of tid is validated (modulo dirty and reference bit emulation).
- If a page fault for a task tid for page vpx is detected and a corresponding entry $(0,tid,vpx,ppx) \in \mathcal{S}$ exists, the page fault handler might still decide to keep the page ppx in memory.

Liveness proofs are typically more difficult for asynchronous paging. They are fairly easy, in case the page fault handler may re-schedule a page-faulting task immediately after the completion of its swap-in operation. If the page fault handler is not allowed to do so (the scheduler might not be willing to allocate the exception task a full time slice), instruction emulation techniques may be of use: emulating the exception instruction in the page fault handler is comparatively cheap, guarantees page fault handler liveness, and helps to keep scheduler and page fault handler proof obligations apart.

7.4 Related Work

We have presented a simple page fault handler and showed its correctness in the form of a virtual memory simulation theorem. Though the literature on paging and virtual memory is vast (as even the early bibliography [Smi78] indicates), the focus is on page replacement algorithm and virtual memory engine design rather than correctness criteria and verification. We sketch related work in the design and verification areas.

In addition to the basic features that have been presented, modern virtual memory engines support a number of additional visible or internal features. Most of these have already been noted at the end of Chapters 3 and 4. Copy-on-write (also known as copy-on-reference) techniques delay physical copying of pages as long as possible, maybe saving the copy operation altogether, and thus allow for low latency implementations of task duplication [BBMT72] and of message passing [RR81]. Zero-copy mechanisms also save memory-to-memory copy operations in relation with I/O operations [Chu96, CP98]. Memory sharing between tasks is used for shared libraries (i.e. code sharing) and communication [BCD72, IEE01a]. Modern Unices also support mapping files into the memory as a means of I/O and, for security-relevant and real-time applications, the allocation of locked pages, which are not to be written to the swap devices [IEE01a]. Microkernel virtual memory engines do not support such a great variety of features. Instead, they introduce external paging, moving the actual paging procedure / policy outside the kernel, [YTR+87]. Finally, as we have seen in Section 7.3, the virtual memory engine might also compensate for missing architectural features in software, such as hardware reference bits [BJ81, Dra91].[7]

From the above list it may be guessed that the virtual memory engines are complex systems with intricate interactions with the remaining kernel code, e.g. process management, scheduler, and the I/O sub system.[8] The text books [Vah96, Tan01] contain descriptions of various (mostly Unix) virtual memory engines with sufficient level of detail. The Linux virtual memory engine is the best documented, thanks to Gorman's thesis and book devoted solely to that subject [Gor04a, Gor04b]. The paper of Cranor and Parulkar describing the zero-copy mechanisms of their virtual memory engine UVM [CP98] gives a good overview on how virtual memory system design is driven by requirements from I/O and IPC sub systems of the kernel. Instead of the ad-hoc optimizations often used in that areas, the authors try to develop a clean VM system design. In most virtual memory engines, a hardware abstraction layers for translation mechanisms and TLB control has been introduced to encapsulate architectural features and simplify the task of porting an operating system to another architecture. An important aspect for portability is if there is a clean interface abstracting architectural operations on translation data structures and TLBs. The first such interface was the Mach pmap interface that is in use in a variety of BSD Unices (including the afore-mentioned UVM [CP99]), an extended Mach virtual memory engine [RTY+87]. We remark, however, that address translation mechanisms vary greatly and the currently implemented hardware abstraction layers are considered too coarse by some researchers [JM98a].

Judging from current research, the subject of page replacement policies is not so important anymore; this argument seems to have been settled, with only a few algorithm being in use (e.g. the one- and two-hand clock algorithm [Cor69, LMKQ89] or FIFO with second chance [TL81, Dra91]) in current systems. The abundance of papers

[7] In the extreme case [JM97], the only architectural feature is a cache miss interrupt.

[8] Notably, the identification of these components of an operating system or a kernel was once also a matter of research and goes back to Dijkstra's seminal work on layered system design [Dij68] and Brinch Hansen's work on *nuclei* [BH70].

on that subject thirty years ago (cf. again [Smi78]) has ceased and now research in that area concentrates on page replacement strategies for special applications—such as multimedia or databases (cf. for an overview for literature on physical memory management with regards to file systems and databases [McN96]). Bottlenecks and performance optimizations currently being described focus on the overall virtual memory system design—how to support all the above feature runtime- and space-efficient?—and on architecture-dependent optimizations—how to make good use of caches and TLBs.

The latter range from macro optimizations to micro optimizations. For example, a page replacement algorithm with page coloring tries to distribute virtual pages evenly on certain sets of physical pages. In a non-thrashing system, this results in a reduction of overall runtime and runtime variance by optimizing the cache utilization [TDF90, KH92b]. Similar techniques are being used for the management of small memory buffers [Bon94] and for the separation of real-time and non-real-time applications inside an operating system [LHH97]. Another architecture-dependent optimizations is related to the management of TLBs. While it was thought that TLBs provide cost-effective solution for the latency problem of address translation, in the mid-1990s the view changed and the opinion was then that "TLBs must be studied again because of current workload and processor trends" [TH94, NUS+93]. With the reduced cycle time it was not possible to boost the number of entries in a TLB to compensate for the quickly growing address space size of programs. Jacob and Mudge report that the VM runtime overhead including cache pollution is up to 20 percent [JM98b] instead of the acceptable runtime overhead of up to 10 percent [Den96]. To tackle the problem, current architecture have varying page sizes (for example "large pages") and, correspondingly, different TLB entries for the different sizes. To make use of the extended architecture, the virtual memory engine has to be extended as well. The transparent and efficient support of super pages is complex and requires pervasive modification and extension of the virtual memory engine [NIDC02]. Still, the increased efforts seems worthwhile and are one of the rare practical approaches to bring down TLB miss rates again.

Architecture-dependent micro optimizations are especially relevant for architectures with software-managed TLBs. These micro optimizations deal with bringing down the instruction count of kernel execution paths in a number of code transformations [Lie95]. As Liedtke shows convincingly, architecture-awareness can make the difference in overall execution speed for microkernels that typically have far more context switches than monolithic kernels [LE96, Lie96].

Finally, let us briefly discuss the issue connected with a kernel running in translated mode. In most modern architectures, system mode need not correlate with untranslated memory access. For a translated kernel we must assert that the page fault handler's code and data structures (including the page tables) do not get paged out. Furthermore, to allow manipulation of the page tables in physical memory, a known interval $[a : a + memsize - 1]$ of the kernel's virtual addresses must map to the physical memory addresses $[0 : memsize - 1]$. The advantage of using a translated kernel is that, with a clever setup, it is possible to avoid software address translations in kernel code (as we have seen in the wrapper code for page fault on load / store in Section 7.1.3), software-generated page fault handler calls, and, on some architecture, TLB flushes. These benefits were first described in [BCD72]; modern systems like the Linux kernel use similar approaches (in typical Linux configurations for 32-bit machines, the lower 3GB are reserved for the user and the upper 1GB for the kernel [Tor04, Gor04a]).

In contrast to system design, there is almost no work on 'systems verification'.

This term, referring to the formal verification of a system throughout all layers of abstraction, was coined by J Moore and his group [BHMY89]. They report on such a verification for a processor, an assembler, a higher-level language and its compiler up to a (simple) operating systems kernel. The kernel 'Kit' relies on additional protection features of the processor (base and limit registers, a very simple form of address relocation) that have not been implemented in the verified processor. Later, Bevier, who verified 'Kit', and Smith formalized large parts of the Mach kernel [BS94a]. No verification was done using this (partial) specification. See Section 3.4 for a discussion of their work.

Chapter 8

Summary and Future Work

8.1 Summary

In this thesis we have examined memory management and address translation techniques, an area where hardware and software are coupled extremely tightly. These intricate interactions have been lacking formal verification and indeed even mathematical formalization.

We presented a formal model of address spaces and computation for tasks running concurrently and in parallel on a multiprocessor. For its implementation, we abstractly modeled a multiprocessor hardware with main and swap memory. Universal page fault handler correctness conditions (the access and the runtime conditions) have been defined and used to prove the correctness of the page fault handling mechanisms on this multiprocessor. In the more practical part we have shown where exactly these condition are established in a computer system comprising of hardware and software. For the concrete system, it turned out (as in [SH98, BJK+03]) that the pipelined instruction fetch, formally constituting prefetching, requires special attention with regards to self-modifying programs. This also held true for the concrete multiprocessor hardware we designed; a strongly revised proof architecture had to be developed to prove its correctness, even with the conservative, sequentially-consistent memory model. To release user programmers from the burden of obeying (and hackers from the 'duty' of exploiting) architectural restrictions related to self-modification, we opted for an operating system that enforces these restrictions through virtual memory techniques.

Multiprocessor correctness, despite its archaic connotation, is of extreme relevance today. As chip companies fear to break Moore's law for the publicity that it causes [Sla04, Hei04], in addition to traditional symmetric multiprocessing (SMP) systems, hyper-threaded and multi-cored processors are constantly being announced. As good as multitasking increases processor utilization by hiding the latency of I/O operations, hyper-threading and related techniques increase processor utilization even further by hiding misprediction and cache miss penalties. Similar correctness proofs to the one given should apply for such systems.

8.2 Future Work

We sketch directions of future research.

- Most of the work presented here was only done mathematically. The proofs for the single-processor VAMP extended with address translation in the theorem prover PVS have been completed by Dalinger in January 2005 [Dal05, DHP05]. In the Verisoft project, sub project 2, the results of this thesis are applied to establish virtual execution environments for tasks running on a microkernel [Ver03].

- Only a sketch of the correctness proof has been presented for the multiprocessor VAMP with address translation and a barrier mechanism. This sketch has to be elaborated to a full paper and pencil proof and formalized.

  As we explained, multiprocessing is on the rise (though differently named). Hence, any formalization of correctness proofs for such systems seems worthwhile.

- As we have noted in Chapter 6 the gate-level verification of a sequentially consistent cache (even a coherent cache) with an arbitrary number of nodes is a prominent research problem [PD97, Cho04]. Probably, Beyer's work in PVS [Bey05] on the gate-level verification of a single-processor instruction- and data-cache system can be extended and generalized for such cache systems.

  With the previous result (formalization of Chapter 6), this would make the complete, gate-level verification of a multiprocessor feasible.

- Our multiprocessor correctness proofs have been based on a conservative, strong memory model. Because of the incurred performance delays, multiprocessors implementing sequential consistency are rare today (e.g. [IBM00]). So, for most current multiprocessors, our proofs must be redone for weak memory models. This may not be trivial and probably requires adaptations of one of the existing formalisms for weak memory models for processor correctness.

- IBM and its competitors implement recursive virtual machines [IBM05, HP01]. For these, the RMM formalism has to be extended to support a hierarchy of relocated machines. The VMM framework must be adapted to represent the extended state of such a machine (although the basic means of simulation, address translation and main and swap memory will not change). As, for reasons of symmetry, 'pure' recursive virtual machines do not not share data across machine boundaries, we would expect the correctness proof of such a machine to lengthen but not to get more difficult.

- Last, but not least, we have argued that the RMM machine model forms the basis of a formalization of operating systems. Among the features / completions lacking most we identified a concrete system call architecture with a system call convention, a virtual processor supporting an exception handling mechanisms, which may be used for hardware-generated exception (overflows, IEEE exceptions, I/O) but also Unix-style signal handling, and a formalization of I/O devices, either as external processes [BH70] or with asynchronous interruptions. A carefully crafted instantiation of the RMM machine may serve as a machine-independent layer for the implementation of operating systems.

  Work on this has been started with the definition of communicating virtual machines [PDM04, GHLP05] and as part of the Verisoft project [Ver03].

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new foundation for Unix development. In *Proceedings of the USENIX Summer Conference*, pages 93–112, 1986.

[ABJ+93] Mustaque Ahamad, Rida A. Bazzi, Ranjit John, Prince Kohli, and Gil Neiger. The power of processor consistency. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 251–260. ACM Press, 1993.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report WRL-TR 95/7, Digital Western Research Laboratory, September 1995.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering: A new definition. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 2–14. ACM Press, 1990.

[AIM95] Apple Computer, Inc., IBM Corporation, and Motorola, Inc. *PowerPC Microprocessor Common Hardware Reference Platform: A System Architecture*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1995.

[AM03] Bernhard K. Aichernig and T. S. E. Maibaum, editors. *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, volume 2757 of *Lecture Notes in Computer Science*. Springer, 2003.

[BBMT72] Daniel G. Bobrow, Jerry D. Burchfiel, Daniel L. Murphy, and Raymond S. Tomlinson. TENEX, a paged time sharing system for the PDP-10. *Communications of the ACM*, 15(3):135–143, 1972.

[BCD72] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: concepts and design. *Communications of the ACM*, 15(5):308–318, 1972.

[BCDM86] Michael C. Browne, Edmund M. Clarke, David L. Dill, and Bud Mishra. Automatic verification of sequential circuits using temporal logic. *IEEE Transactions on Computers*, 35(12):1035–1044, December 1986.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177. ACM Press, 2003.

[Bel66] Laszlo A. Belady. A study of replacement algorithms for virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Ber01] Christoph Berg. Formal verification of an IEEE floating point adder. Master's thesis, Saarland University, Computer Science Department, 2001.

[Bey05] Sven Beyer. *Putting It All Together: Formal Verification of the VAMP*. PhD thesis, Saarland University, Computer Science Department, 2005.

[BH70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[BH75] Gerald Belpaire and Nai-Ting Hsu. Hardware architecture for recursive virtual machines. In *Proceedings of the 1975 annual conference*, pages 14–18. ACM Press, 1975.

[BHMY89] William R. Bevier, Warren A. Hunt, Jr., J S. Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989. In [Boy89].

[BJ81] Özalp Babaoğlu and William Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 78–86. ACM Press, 1981.

[BJK+03] Sven Beyer, Christian Jacobi, Daniel Kroening, Dirk Leinenbach, and Wolfgang Paul. Instantiating uninterpreted functional units and memory system: functional verification of the VAMP processor. In Geist and Tronci [GT03], pages 51–65.

[BM88] Robert S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[Bon94] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer*, pages 87–98, 1994.

[Boy89] Robert S. Boyer, editor. *Special Issue on System Verification*, volume 5 of *Journal of Automated Reasoning*. Kluwer Academic Publishers, 1989.

[BRGH89] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. Translation lookaside buffer consistency: a software approach. In *Proceedings of the third international conference on Architectural support for programming languages and operating systems*, pages 113–122. ACM Press, 1989.

[BS93a] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Entities and relations. Technical Report 88, Computational Logic, Inc., February 1993.

[BS93b] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Atomic actions and locks. Technical Report 89, Computational Logic, Inc., February 1993.

[BS94a] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel. Technical Report 102, Computational Logic, Inc., December 1994.

[BS94b] William R. Bevier and Lawrence M. Smith. A mathematical model of the Mach kernel: Kernel requests. Technical Report 103, Computational Logic, Inc., December 1994.

[CD97a] Michel Cekleov and Michel Dubois. Virtual-address caches, part 1: Problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, 1997.

[CD97b] Michel Cekleov and Michel Dubois. Virtual-address caches, part 2: Multiprocessor issues. *IEEE Micro*, 17(6):69–74, 1997.

[Cho04] Ching-Tsun Chou. How to specify and verify cache coherence protocols: An elementary tutorial. In Hu and Martin [HM04].

[Chu96] H. K. Jerry Chu. Zero-copy TCP in solaris. In *USENIX Annual Technical Conference*, pages 253–264, 1996.

[CMP04] Ching-Tsun Chou, Phanindra K. Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In Hu and Martin [HM04], pages 382–398.

[Cor69] F. J. Corbató. A paging experiment with the Multics system. In *In Honor of P. M. Morse*, pages 217–228. MIT Press, Cambridge Massachusetts, 1969.

[CP98] Charles D. Cranor and Gurudatta M. Parulkar. Zero-copy data movement mechanisms for UVM. Technical report, Washington University Department of Computer Science, December 1998.

[CP99] Charles D. Cranor and Gurudatta M. Parulkar. The UVM virtual memory system. In *Proceedings of the USENIX Annual Technical Conference*, pages 117–130, June 1999.

[CWH03] Matthew Chapman, Ian Wienand, and Gernot Heiser. Itanium page tables and TLB. Technical Report UNSW-CSE-TR-0307, University of New South Wales, Sydney, Australia, May 2003.

[Dal05] Iakov Dalinger. *Mechanical Verification of a Processor with Address Translation (Draft)*. PhD thesis, Saarland University, Computer Science Department, 2005.

[Den65] Jack B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, 1965.

[Den67] Peter J. Denning. The working set model for program behavior. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12. ACM Press, 1967.

[Den70] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[Den80] Peter J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, January 1980.

[Den96] Peter J. Denning. Virtual memory. *ACM Computing Surveys*, 28(1):213–216, 1996.

[DHP05] Iakov Dalinger, Mark Hillebrand, and Wolfgang Paul. On the verification of memory management mechanisms. In D. Borrione and W. Paul, editors, *Proceedings of the 13th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2005)*, Lecture Notes in Computer Science. Springer, 2005. To appear.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[Dij68] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, 1968.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dra91] Richard P. Draves. Page replacement and reference bit emulation in Mach. In *Proceedings of the USENIX Mach Symposium*, pages 201–212, 1991.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the eleventh ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM Press, 1984.

[DSB86] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*, pages 434–442. IEEE Computer Society Press, 1986.

[Eir98] Ásgeir Th. Eiríksson. The formal design of 1M-gate ASICs. In Ganesh Gopalakrishnan and Phillip J. Windley, editors, *FMCAD*, volume 1522 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 1998.

[EK03] Allen E. Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In Geist and Tronci [GT03], pages 247–262.

[EP97] Guy Even and Wolfgang J. Paul. On the design of IEEE compliant floating point units. In *Proceedings of the 13th symposium on Computer arithmetic*, pages 54–63. IEEE Computer Society Press, 1997.

[FHPR01] Peter A. Franaszek, Philip Heidelberger, Dan E. Poff, and John T. Robinson. Algorithms and data structures for compressed-memory machines. *IBM Journal of Research and Development*, 45(2), March 2001.

[FK99] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 1999.

[FR86] Robert Fitzgerald and Richard F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Computing Surveys*, 4(2):147–177, 1986.

[GD91] David B. Golub and Richard P. Draves. Moving the default memory manager out of the Mach kernel. In *Proceedings of the USENIX Mach Symposium*, pages 177–188, 1991.

[GHLP05] Mauro Gargano, Mark Hillebrand, Dirk Leinenbach, and Wolfgang Paul. On the correctness of operating system kernels. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, Lecture Notes in Computer Science. Springer, 2005. To appear.

[GLL+90] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessor. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 15–26. ACM Press, 1990.

[GMG91] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories (extended abstract). In *SPAA '91: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 292–303. ACM Press, 1991.

[Gol73] R. P. Goldberg. Architecture of virtual machines. In *Proceedings of the workshop on virtual computer systems*, pages 74–112, 1973.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, March 1989.

[Gop04] Ganesh Gopalakrishnan. Shared memory consistency models: A broad survey. In Hu and Martin [HM04].

[Gor] Mel Gorman. VM regress – A regression, test and benchmark suite. available under `http://www.skynet.ie/~mel/projects/vmregress/`.

[Gor04a] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice-Hall, 2004.

[Gor04b] Mel Gorman. Understanding the linux virtual memory manager. Master's thesis, University of Limerick, 2004.

[GT03] Daniel Geist and Enrico Tronci, editors. *Correct Hardware Design and Verification Methods, 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volume 2860 of *Lecture Notes in Computer Science*. Springer, 2003.

[Hei04] Intels neueste Fahrplan-Änderung: Pentium 4 mit 4 GHz abgesagt. `http://www.heise.de/newsticker/meldung/52187`, October 2004.

[HM04] Alan Hu and Andrew Martin, editors. *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, TX, USA, November 14-17, 2004, Proceedings*, volume 3312 of *Lecture Notes in Computer Science*. Springer, 2004.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996.

[HP01] Hewlett-Packard. *HP-UX Virtual Partitions (vPars) - White Paper*, 2001.

[HP03] Mark A. Hillebrand and Wolfgang J. Paul. Virtual memory simulation theorems. `http://www-wjp.cs.uni-sb.de/publikationen/vmsimtheorems.ps`, July 2003.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[IBM00] IBM. *z/Architecture Principles of Operation*. Poughkeepsie, NY, December 2000.

[IBM05] IBM. *IBM z/VM General Information*. Poughkeepsie, NY, January 2005.

[IC78] R. N. Ibbett and P. C. Capon. The development of the MU5 computer system. *Communications of the ACM*, 21(1):13–24, 1978.

[IEE85] IEEE. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, 1985.

[IEE01a] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. Institute of Electrical and Electronics Engineers, 2001.

[IEE01b] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) System Interfaces, Issue 6*. Institute of Electrical and Electronics Engineers, 2001.

[IEE01c] IEEE. *IEEE Std 1003.1-2001 Standard for Information Technology — Portable Operating System Interface (POSIX) Shell and Utilities, Issue 6*. Institute of Electrical and Electronics Engineers, 2001.

[Int02] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual*, volume 2: System Architecture. Intel Corporation, October 2002.

[Int04] Intel Corporation. *Intel IA-32 Intel Architecture Software Developer's Manual*, volume 3: System Programming Guide. Intel Corporation, Denver, CO, USA, 2004.

[Jac02] Christian Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, Saarland University, Computer Science Department, 2002.

[JJ01] A. Jaleel and B. Jacob. In-line interrupt handling for software-managed TLBs. In *ICCD*, pages 62–67, Washington - Brussels - Tokyo, September 2001. IEEE.

[JM97] B. Jacob and T. Mudge. Software-managed address translation. In *HPCA '97: Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA '97)*, pages 156–167. IEEE Computer Society, 1997.

[JM98a] Bruce Jacob and Trevor Mudge. Virtual memory in contemporary microprocessors. *IEEE Micro*, 18(4):60–75, 1998.

[JM98b] Bruce L. Jacob and Trevor N. Mudge. A look at several memory management units, TLB-refill mechanisms, and page table organizations. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 295–306. ACM Press, 1998.

[Kan88] Gerry Kane. *MIPS RISC architecture*. Prentice-Hall, Inc., 1988.

[KH92a] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.

[KH92b] R. E. Kessler and Mark D. Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems*, 10(4):338–359, 1992.

[KHPS61] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner. The Manchester University Atlas operating system part I: Internal organization. *The Computer Journal*, 4(3):222–225, October 1961.

[KMP00] Daniel Kroening, Silvia M. Mueller, and Wolfgang Paul. Proving the correctness of processors with delayed branch using delayed PCs. In I. Althoefer, N. Cai, G. Dueck, L. Khachatrian, M. Pinsker, A. Sarkozy, I. Wegener, and Zhang Z., editors, *Proceedings of the Symposium on Numbers, Information and Complexity, Bielefeld*, pages 579–588. Kluwer Academic Publishers, 2000.

[Kro01] Daniel Kroening. *Formal Verification of Pipelined Microprocessors*. PhD thesis, Saarland University, Computer Science Department, 2001.

[Lam74] Leslie Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[Lam97] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.

[Lau] James Laurus. SPIM: A MIPS32 Simulator. `http://www.cs.wisc.edu/~larus/spim.html`.

[LBB+91] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a microkernel based OS. *SIGOPS Operating Systems Review*, 25(2):51–62, 1991.

[LD91] Paul Loewenstein and David L. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic. In Edmund M. Clarke and Robert P. Kurshan, editors, *CAV*, volume 531 of *Lecture Notes in Computer Science*, pages 302–311. Springer, 1991.

[LE96] Jochen Liedtke and Kevin Elphinstone. Guarded page tables on MIPS R4600 or an exercise in architecture-dependent micro optimization. *ACM SIGOPS Operating Systems Review*, 30(1):4–15, 1996.

[Lee69] F. F. Lee. Study of 'look-aside' memory. *IEEE Transactions on Computers*, 18(11):1062–1064, November 1969.

[Lee89] Ruby B. Lee. Precision architecture. *IEEE Computer*, 22(1):78–91, 1989.

[Lei02] Dirk Leinenbach. Implementierung eines maschinell verifizierten prozessors. Master's thesis, Saarland University, Computer Science Department, 2002.

[Lev00] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000.

[LHH97] Jochen Liedtke, Hermann Härtig, and Michael Hohmuth. OS-controlled cache predictability for real-time systems. In *IEEE Real Time Technology and Applications Symposium*, pages 213–223. IEEE Computer Society, 1997.

[Lie95] Jochen Liedtke. On micro-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating systems principles*, pages 237–250. ACM Press, 1995.

[Lie96] Jochen Liedtke. Toward real microkernels. *Communications of the ACM*, 39(9):70–77, 1996.

[LMKQ89] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The design and implementation of the 4.3BSD operating system*. Addison Wesley Longman Publishing Co., Inc., 1989.

[Loe91] Keith Loepere. Mach 3 kernel interface. Technical report, Open Software Foundation, May 1991.

[LS84] Johnny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Transactions on Computers*, 17(1):6–22, January 1984.

[LW73] Hugh C. Lauer and David Wyeth. A recursive virtual machine architecture. In *Proceedings of the workshop on virtual computer systems*, pages 113–116, 1973.

[McN96] Dylan McNamee. Flexible physical memory management, January 1996.

[Mey02] Carsten Meyer. Entwicklung einer Laufzeitumgebung für den VAMP-Prozessor. Master's thesis, Saarland University, Computer Science Department, 2002.

[Mis86] J. Misra. Axioms for memory access in asynchronous hardware systems. *ACM Trans. Program. Lang. Syst.*, 8(1):142–153, 1986.

[MLP04] Samuel P. Midkiff, Jaejin Lee, and David A. Padua. A compiler for multiple memory models. *Concurrency and Computation: Practice and Experience*, 16(2-3):197–220, 2004.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[Moo03] J Strother Moore. A grand challenge proposal for formal methods: A verified stack. In Aichernig and Maibaum [AM03], pages 161–172.

[MP00] Silvia M. Mueller and Wolfgang J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.

[NIDC02] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Operating Systems Review*, 36(SI):89–104, 2002.

[NUS+93] David Nagle, Richard Uhlig, Tim Stanley, Stuart Sechrest, Trevor Mudge, and Richard Brown. Design tradeoffs for software-managed TLBs. In *Proceedings of the 20th annual international symposium on Computer architecture*, pages 27–38. ACM Press, 1993.

[OSR92] S. Owre, N. Shankar, and J. M. Rushby. PVS: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 1992.

[Pat85] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.

[PD96] Seungjoon Park and David L. Dill. Verification of FLASH cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'96)*, pages 288–296, Padua, Italy, June 1996. SIGARCH, ACM.

[PD97] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1):82–126, 1997.

[PDM04] Wolfgang Paul, Dilyana Dimova, and Mario Mancino. Skript zur Vorlesung Systemarchitektur. `http://www-wjp.cs.uni-sb.de/publikationen/Skript.pdf`, July 2004.

[Ros89] Bryan S. Rosenburg. Low-synchronization translation lookaside buffer consistency in large-scale shared-memory multiprocessors. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 137–146. ACM Press, 1989.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the eighth ACM symposium on Operating systems principles*, pages 64–75. ACM Press, 1981.

[RTY+87] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, and Robert Baron. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. In *Proceedings of the second international conference on Architectual support for programming languages and operating systems*, pages 31–39. IEEE Computer Society Press, 1987.

[SGG00] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne. *Applied operating system concepts*. John Wiley & Sons, Inc., 2000.

[SH98] Jun Sawada and Warren A. Hunt, Jr. Processor verification with precise exeptions and speculative execution. In *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 135–146. Springer-Verlag, 1998.

[Sla04] Intel scraps plan for 4 GHz P4 chip. `http://slashdot.org/articles/04/10/14/2227212.shtml`, October 2004.

[Smi78] Alan Jay Smith. Bibliography on paging and related topics. *ACM SIGOPS Operating Systems Review*, 12(4):39–56, 1978.

[Smi82] Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[SN04] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, 1988.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, second edition, 2001.

[TDF90] George Taylor, Peter Davies, and Michael Farmwald. The TLB slice: A low-cost high-speed address translation mechanism. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 355–363. ACM Press, 1990.

[Tel90] Patricia J. Teller. Translation-lookaside buffer consistency. *IEEE Computer*, 23(6):26–36, 1990.

[TH94] Madhusudhan Talluri and Mark D. Hill. Surpassing the TLB performance of superpages with less operating system support. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 171–182. ACM Press, 1994.

[TL81] Rollins Turner and Henry Levy. Segmented FIFO page replacement. In *SIGMETRICS '81: Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 48–51. ACM Press, 1981.

[Tor04] Linus Torvalds et. al. The Linux kernel archives. `http://www.kernel.org/`, 2004.

[Vah96] Uresh Vahalia. *UNIX internals: The new frontiers*. Prentice Hall Press, 1996.

[VAM03] The verified architecture microprocessor (VAMP). `http://www-wjp.cs.uni-sb.de/forschung/projekte/VAMP/`, 2003.

[Ver03] The Verisoft Project. `http://www.verisoft.de/`, 2003.

[WB92] Bob Wheeler and Brian N. Bershad. Consistency management for virtually indexed caches. In *Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 124–136. ACM Press, 1992.

[YGLS03] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and SAT. In Geist and Tronci [GT03], pages 81–95.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, and J. Eppinger. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 63–76. ACM Press, 1987.