Analysis of Preemptively Scheduled Hard Real-time Systems

Dissertation

zur Erlangung des Grades des

Doktors der Ingenieurwissenschaften

der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von

Sebastian Altmeyer

Saarbrücken
2012

Dekan:	Prof. Dr. Mark Groves
Prüfungsausschuß:	Prof. Dr. Sebastian Hack (Vorsitzender) Prof. Dr. Reinhard Wilhelm (Gutachter) Prof. Dr. Gerhard Fohler (Gutachter) Laurent Mauborgne, PhD (Gutachter) Dr. Daniel Grund (akademischer Mitarbeiter)
Tag des Kolloqiums:	25. Oktober 2012

Impressum

Druck und Verlag: epubli GmbH, Berlin, www.epubli.de
ISBN 978-3-8442-5161-6

Abstract

As timing is a major property of hard real-time, proving timing correctness is of utter importance. A static timing analysis derives upper bounds on the execution time of tasks, a scheduling analysis uses these bounds and checks if each task meets its timing constraints.

In preemptively scheduled systems with caches, this interface between timing analysis and scheduling analysis must be considered outdated. On a context switch, a preempting task may evict cached data of a preempted task that need to be reloaded again after preemption. The additional execution time due to these reloads, called cache-related preemption delay (CRPD), may substantially prolong a task's execution time and strongly influence the system's performance.

In this thesis, we present a formal definition of the cache-related preemption delay and determine the applicability and the limitations of a separate CRPD computation.

To bound the CRPD based on the analysis of the preempted task, we introduce the concept of definitely cached useful cache blocks. This new concept eliminates substantial pessimism with respect to former analyses by considering the over-approximation of a preceding timing analysis.

We consider the impact of the preempting task to further refine the CRPD bounds. To this end, we present the notion of resilience. The resilience of a cache block is a measure for the amount of disturbance of a preempting task a cache block of the preempted task may survive.

Based on these CRPD bounds, we show how to correctly account for the CRPD in the schedulability analysis for fixed-priority preemptive systems and present new CRPD-aware response time analyses: ECB-Union and Multiset approaches.

Zusammenfassung

Da das Zeitverhalten ein Hauptbestandteil harter Echtzeitsysteme ist, ist das Beweisen der zeitlichen Korrektheit von großer Bedeutung. Eine statische Zeitanalyse berechnet obere Schranken der Ausführungszeiten von Programmen, eine Planbarkeitsanalyse benutzt diese und prüft ob jedes Programm die Zeitanforderungen erfüllt.

In präemptiv geplanten Systemen mit Caches, muss die Nahtstelle zwischen Zeitanalyse und Planbarkeitsanalyse als veraltet angesehen werden. Im Falle eines Kontextwechsels kann das unterbrechende Programm Cachedaten des unterbrochenen Programms entfernen. Diese Daten müssen nach der Unterbrechung erneut geladen werden. Die zusätzliche Ausführungszeit durch das Nachladen der Daten, welche Cache-bezogene Präemptions-Verzögerung (engl. Cache-related Preemption Delay (CRPD)) genannt wird, kann die Ausführungszeit des Programm wesentlich erhöhen und hat somit einen starken Einfluss auf die Gesamtleistung des Systems.

Wir präsentieren in dieser Arbeit eine formale Definition der Cachebezogene Präemptions-Verzögerung und bestimmen die Einschränkungen und die Anwendbarkeit einer separaten Berechnung der CRPD.

Basierend auf der Analyse des unterbrochenen Programms präsentieren wir das Konzept der definitiv gecachten nützlichen Cacheblöcke. Verglichen mit bisherigen CRPD-Analysen eleminiert dieses neue Konzept wesentliche Überschätzung indem die Überschätzung der vorherigen Zeitanalyse mit in Betracht gezogen wird.

Wir analysieren den Einfluss des unterbrechenden Programms um die CRPD-Schranken weiter zu verbessern. Hierzu führen wir das Konzept der Belastbarkeit ein. Die Belastbarkeit eines Cacheblocks ist ein Maß für die Störung durch das unterbrechende Programm, die ein nützlicher Cacheblock überleben kann.

Basierend auf diesen CRPD-Schranken zeigen wir, wie die Cachebezogene Präemptions-Verzögerung korrekt in die Planbarkeitsanalyse

für Systeme mit statischen Prioritäten integriert werden kann und präsentieren neue CRPD-bewußte Antwortzeitanalysen: die ECB-Union und die Multimengen-Ansätze.

Acknowledgements

This thesis would not have been possible without the help and support of many people. Foremost, I like to thank Prof. Reinhard Wilhelm for his advise and guidance and for giving me the opportunity to write this PhD thesis. I have spent wonderful years at his chair at Saarland University and learned a lot during this time. I also owe thanks to Prof. Gerhard Fohler and Laurent Mauborgne for carefully reviewing my thesis and to Prof. Sebastian Hack and Dr. Daniel Grund as members of the examination board.

Substantial parts of this thesis originated from collaboration with other researchers: I like to express my gratitude to Claire Maiza, Jan Reineke and Robert Davis. It was a real pleasure to work with such bright and motivated colleagues and friends.

I thank Sascha El-Abed, Michael Gerke, Jörg Herter and Verena Kremer for carefully proofreading drafts of my thesis. I also like to thank my friends, and family for their support. I always had sufficient distraction to regain energy for the often tedious work. Last but not least, I thank Kim for her love and encouragement, especially during the last few months of my PhD studies.

Contents

Chapter 1

Introduction

Safety doesn't happen by accident.
Unknown

In our everyday life, we are surrounded by small computer systems, embedded into larger devices such as cars, trains, airplanes, but also mobile phones, washing machines and refrigerators. Most of these *embedded* computers are *real-time systems*, i.e., they are subject to real-time constraints. Correctness of such systems does not only depend on the correct result of the computation, but also on the timeliness of the result. An airbag controller, for instance, has not only to decide whether or not to inflate the airbag, but has to do so before the driver's head bangs on the steering wheel. A drive-by-wire system is not allowed to introduce long delays, but must translate the driver's steering motion and pass it to the wheels in time. A flight control system must quickly compensate external disturbances to prevent stalls and thus to keep the airplane in stable flight. Railroad signalling, pacemakers, monitoring in a nuclear power plant... This list can be extended to plenty of other areas and examples.

Real-time systems are divided into *soft* and *hard real-time systems* depending on whether a single deadline miss is tolerable (*soft*) or is considered a complete failure (*hard*). As timing behavior is a major property of such systems, proving timing correctness is of utter importance during the development process. Functionality of an embedded system is typically implemented by a set of tasks distributed to the available hardware and processing units. Proving timing correctness of the complete systems is thus traditionally a two-step approach:

1. deriving bounds on the execution times of tasks in isolation, and
2. distributing tasks to the available resources (e.g. processing units) guaranteeing that all tasks comply with their timing constraints.

Common denomination for theses steps are *timing analysis* and *scheduling analysis*.

Timing Analysis can either be static or dynamic. Which one to apply depends on the criticality of the system. *Dynamic timing analysis*, also known as measurement-based approach, explores a task's execution time for varying inputs and initial processor states. As exhaustive measurement is usually infeasible, it is unlikely that the actual *best-case* or *worst-case execution time* will be encountered. Hence, dynamic timing analysis does not provide guaranteed bounds on the execution times but only educated guesses. It is thus used for soft real-time systems or systems with less stringent timing constraints.

Static timing analysis employs an abstract model of the hardware to characterize the timing behavior of a task. It derives bounds on the execution time without any concrete execution of a task but based on the abstract model. A static timing analysis is required to be safe, i.e., execution time bounds must be conservative and is demanded to be precise, i.e., execution time bounds should be as close as possible to the actual best-case or worst-case timing behavior. In contrast to measurement-based approaches, static timing analyses can provide guaranteed bounds (assuming a sound abstract hardware model). It is thus more apt for hard real-time systems.

Scheduling Analysis determines if each task complies with its timing constraints when scheduled according to a predefined scheduling policy. Timing constraints are typically defined by a task's period and a task's deadline, both determined by the physical environment. Tasks are scheduled either *preemptively*, i.e., task's execution can be temporarily interrupted, or *non-preemptively*, i.e., once started each task runs to completion. Preemptive schedules are potentially more powerful in the sense that some task sets are only schedulable preemptively. Tasks with short deadlines—such as interrupts—usually can not postpone their execution until the completion of the currently running task.

An important class of schedulers are *priority driven schedulers* where each task is assigned (either statically or dynamically) a priority. The scheduler always executes the task with the highest priority among all currently available tasks.

Traditional Interface

Traditionally, the complete **interface between timing analysis and scheduling analysis** is given by bounds on the tasks' execution times. Other input to the scheduling analysis—such as deadlines or periods of tasks—are dictated by the system's environment and bypass the timing analysis. However, this interface is inherently pessimistic and can be considered outdated for modern hardware architectures. Since the advent of caches in embedded real-time systems, history-sensitive architectural components go beyond the scope of a task. This holds especially for preemptively scheduled systems. On a task preemption or *context switch*, the preempting task disrupts the current processor state and may evict useful cached data of the preempted task. After preemption, the execution time of the preempted task strongly depends on whether previously cached data is still resident in cache. The additional execution time due to preemption is denoted *context switch costs* and the portion of the context switch costs caused by additional cache reloads *cache-related preemption delay*.

Recent studies show a substantial increase of the execution times due to preemption [10, 14, 78]. Ignoring these costs is thus not an option as it leads to optimistic results. Instead, most scheduling analyses assume that the execution time bound already accounts for the additional context switch costs—as proposed in the seminal paper by Liu and Layland [53] that laid the foundation of the scheduling theory. This assumption however is inherently pessimistic for cached systems: A sound upper bound on the execution time must conservatively account for each scheduling scenario and each possible number of preemptions.

1.1. Contributions of this Thesis

This thesis advocates a different approach to the analysis of preemptively scheduled hard real-time systems: **Precise modelling and computation of the context switch costs.** Instead of limiting the interface between timing and scheduling analysis to a single scalar value (upper bound on the execution time), we propose to extend timing analysis to compute precise preemption costs and to integrate these costs into the scheduling analysis.

The first scheduling analysis explicitly considering preemption costs has been proposed by Busquets-Mataix et al. [18] (based on the assumption that each cache entry must be reloaded). This model has been refined by Lee et al. [49] by computing the worst-case impact on the preempted task, by Tomiyama et al. [90], and later Tan et al. [86] by considering preempting and preempted tasks.

We contribute to this line of research in the following ways and aspects:

Formal Model of the Cache-Related Preemption Delay

We provide a formal definition of the *cache-related preemption delay*. Advantages of this model are twofold: First, we are able to base all subsequent analyses on solid ground and prove the correctness of the proposed CRPD analyses. Second, we provide a clear identification of the applicability and the limitations of a separate computation of the context switch costs. We show that in contrast to prior beliefs, context switch costs of a single preemption can be unbounded in case of certain cache replacement policies and hardware features.

Concept of Definitely-Cached Useful Cache Blocks

We identify substantial pessimism in the first analysis of the context switch costs based on the preempted task and propose a new preciser model using the concept of *definitely-cached useful cache blocks (DC-UCBs)*. A useful cache block is a memory that is i) cached prior to a program point and ii) reused afterwards. In case of preemption at this program point, only useful cache blocks have to be reloaded and thus can increase the execution time. We improve on the basic concept by considering possible pessimism of the preceding timing analysis that derives an upper bound on the execution time. Furthermore, we prove the correctness of this new model and propose a static program analysis (based on the framework of abstract interpretation) to derive the set of DC-UCBs.

Resilience Analysis

We improve the bound on the cache-related preemption delay by considering the cache usage of the preempting task. This is especially problematic for set-associative caches. First we show that previous analyses for this setting are unsound and possibly optimistic. We then correct these bounds and present a new, precise, and sound CRPD analysis for set-associative caches. To this end, we introduce the notion of *resilience*. The resilience of a useful cache block is a metric for the amount of disturbance by preempting tasks a useful cache block may suffer without causing an additional cache reload due to preemption. We also prove the correctness of this concept and provide a static program analysis to compute the resilience of the cache blocks.

CRPD-Aware Response Time Analysis

We show how to incorporate bounds on the cache-related preemption delays in the scheduling analysis for *fixed-priority preemptive schedules*. Nested preemption requires particular attention. A preempting task may not only evict cache blocks of one task but of all nestedly preempted tasks. Vice versa, a preempted task may suffer eviction not only by the directly preempting task but also by tasks preempting the preempting task. We first provide a thorough review of the existing approaches and then present a new and preciser analysis: the so-called *ECB-Union*. The new analysis derives upper bounds on the effect of all preempting tasks to the preempted task. In addition, we show how to eliminate spurious preemption scenarios to improve the schedulability results even further. We also present how to correctly consider *mutual exclusive access* to *shared resources* when considering CRPD explicitly.

1.2. Structure

The thesis is structured as follows: We introduce the framework of *abstract interpretation* (on which we base all program analyses) in Chapter 2. Chapter 3 provides further background needed for the understanding of this thesis. The formal model and explanation of the *context switch costs* and the *cache-related preemption delay* are given in Chapter 4 and an overview of the related work in Chapter 5. Chapter 6 presents the concept and the analysis of *definitely-cached useful cache blocks*, Chapter 7 the *resilience analysis*, and Chapter 8 the CRPD-aware schedulability analyses. An evaluation of the new approaches can be found in Chapter 9 and Chapter 10 concludes this thesis.

Publications Contributing to this Thesis

Parts of the thesis have been published in peer-reviewed conferences, journals and workshops:

- Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. **Improved Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems.** In *Real-Time Systems* (to appear).

- Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. **Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems.** In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, (RTSS'11), pages 261–271, December 2011.

- Sebastian Altmeyer and Claire Maiza Burguière. **Cache-related preemption delay via useful cache blocks: Survey and redefinition.** *J. Syst. Archit.*, 57:707–719, August 2011.

- Sebastian Altmeyer, Claire Maiza, and Jan Reineke. **Resilience analysis: Tightening the CRPD bound for set-associative caches.** In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, (LCTES '10), pages 153–162, April 2010.

- Sebastian Altmeyer and Claire Burguière. **A new notion of useful cache block to improve the bounds of cache-related preemption delay.** In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, (ECRTS '09), pages 109–118, July 2009.

- Claire Burguière, Jan Reineke, and Sebastian Altmeyer. **Cache-related preemption delay computation for set-associative caches - pitfalls and solutions.** In *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2009.

Chapter 2

Abstract Interpretation

All exact science is dominated by the idea of approximation.
Bertrand Russell (1872 - 1970)

A program execution can be seen as a transformation of one state to another. The semantics of a program defines a) the set of states and b) how the program transforms a given state. Static program analysis [66] argues about certain properties of these states occurring during program execution, such as the sign of variables or the content of the cache memory. Many of these properties are undecidable in general such that the concrete semantics need to be approximated. Naturally, a program analysis must be sound in order to produce reliable results. *Abstract interpretation* [20, 21] provides means to design sound program analyses. In contrast to traditional program analysis that argues about program properties directly, abstract interpretation introduces a concrete and an abstract model of the semantics from which program properties can be derived. Correctness of the results is proven by showing that the abstract model is a sound approximation of the concrete model.

In this chapter, we introduce the basic notion of program analysis and abstract interpretation and provide the corresponding mathematical foundations. This chapter only serves as a basic introduction required for the understanding of this thesis. Further reading can be found in [21, 66, 44] and [75].

2.1. Program Analysis

Programs under examination are represented as *control-flow graphs*:

Definition 2.1 (Control-Flow Graph)
A control-flow graph *(CFG) is a directed graph $G = (V, E, p_s, p_e)$ with a finite set V of nodes, a set $E \subseteq V \times V$ of edges, a start node $p_s \in V$ and an end node $p_e \in V$. If $(p_n, p_m) \in E$, p_n is a predecessor of p_m (p_m is successor of p_n). Node p_s has no predecessor, p_e no successor. Note that we also refer to a node in the control flow graph as a program point.*

We assume in addition that a CFG is connected and that p_e is the only end node, i.e., all nodes can be reached from start node p_s and all nodes may reach p_e. If a program has several exits p_e^1 to p_e^n we enforce the second condition by introducing an artificial end node $p_{e'}$ and corresponding edges $(p_e^i, p_{e'})$. A path within the control-flow graph is defined as follows:

Definition 2.2 (Path)
Let $G = (V, E, p_s, p_e)$ be a CFG. A path *π from node p_1 to node p_k is a sequence of nodes $\pi = [p_1, p_2, \ldots, p_{k-1}, p_k]$ where $\forall i : (p_i, p_{i+1}) \in E$. The symbol ϵ represents the empty path, Π the set of all paths and $\pi_1 \cdot \pi_2$ concatenation of two paths.*

2.1.1. Collecting Semantics

We describe the semantics of a program as transitions between program states. To this end, we define *transfer function*, *path semantics*, and the *collecting semantics.* Examples of program semantics are cache behavior and variable assignment.

Definition 2.3 (Transfer functions)
Let $G = (V, E, p_s, p_e)$ be a CFG and $\mathbb{D}$ the set of program states. A transfer function *$tf : V \to (\mathbb{D} \to \mathbb{D})$ assigns the semantics of each node $p \in V$.*

The transfer function (or transformer) defines the local semantics of each node of the CFG. The semantics of a complete path within the CFG is defined as the composition of the transfer functions.

Definition 2.4 (Path Semantics)
The path semantics *$[\pi]_{tf_C}$ of a path π is a composition of a transfer function $tf : V \to (\mathbb{D} \to \mathbb{D})$ along the path π:*

$$[\pi]_{tf} = \begin{cases} id_{\mathbb{D}\to\mathbb{D}} & \text{if } \pi = \epsilon \\ [p_2, \ldots, p_n]_{tf} \circ tf(p_1) & \text{if } \pi = [p_1, \ldots, p_n] \end{cases}$$

Assume we are interested in the valuation of variables. A program state is defined as a function assigning each variable of the program a value and the transformer updates the program states according to the arithmetical operations of the program.

Sticky Collecting Semantics

This far, we defined the semantics of a program always using transitions from one state to another. If we apply, for instance, the path semantics to an initial state, the semantics is only valid for this specific execution of the program path. However, we are usually interested in deriving information valid for a set of possible initial states. For this reason, we define the *collecting sticky semantics.*

Definition 2.5 (Sticky Collecting Semantics)
The sticky collecting semantics *at node p for the set of initial states* $\mathrm{Init} \subseteq \mathbb{D}$ *is a function $Coll : V \to 2^{\mathbb{D}}$ defined as*

$$Coll(p) = \bigcup_{s \in \mathrm{Init}} \left(\bigcup \{ [\pi]_{tf}(s) | \pi \in \Pi \wedge \pi = [p_s, \ldots, p] \} \right)$$

The function $\mathrm{Coll}(p)$ delivers all possible states at node p, i.e., all states that may arise at p during the execution of the program with all possible initial states $s \in \mathrm{Init}$. In the case of variable valuation the sticky collecting semantics delivers all values a variable may obtain at program point p. Such information can be used for instance to statically exclude division-by-zero exceptions.

Path-Based Collecting Semantics

However, many program properties cannot be expressed using the sticky collecting semantics, at least not in a natural way. Liveness of variables for instance does not depend on the state at a program point p but on the paths reaching p. We therefore define the *path-based collecting semantics* [75].

Definition 2.6 (Path-Based Collecting Semantics)
The path-based collecting semantics *is the set of all paths ending in program point p (forward semantics)*

$$Coll_{\Pi}^{\rightarrow} : V \to 2^{\Pi}$$

$$Coll_{\Pi}^{\rightarrow}(p) = \{ \pi \mid \pi \in \Pi \wedge \pi = [p_s, \ldots, p] \} \tag{2.1}$$

or the set of all paths emanating from a program point p (backward semantics)

$$Coll_{\Pi}^{\leftarrow} : V \to 2^{\Pi}$$

$$Coll_{\Pi}^{\leftarrow}(p) = \{\pi \mid \pi \in \Pi \wedge \pi = [p, \ldots, p_n]\} \tag{2.2}$$

The path-based collecting semantics are also referred to as *second-order semantics* or *trace semantics* [75]. The liveness of a variable can be deduced from the backward path-based collecting semantics by checking if there exists at least one path on which the value of the variable is used.

Although some program properties require these collecting semantics, we can also define the path-based collecting semantics using the sticky collecting semantics. The concrete domain is the set of paths and the transfer function that appends, resp. prepends a program point to all paths $tf_C^{\leftarrow/\rightarrow}$:

$$tf^{\leftarrow/\rightarrow} : V \to (2^{\Pi} \to 2^{\Pi})$$

$$tf_{\pi}^{\rightarrow}(p)(S) := \{\pi \cdot p \mid \pi \in S\} \tag{2.3}$$

and

$$tf^{\leftarrow}(p)(S) := \{p \cdot \pi \mid \pi \in S\} \tag{2.4}$$

The initial program state is the set of paths containing only the empty path ϵ. Hence,

$$\begin{aligned}\mathrm{Coll}(p)^{\rightarrow} &= \bigcup_{s \in \{\epsilon\}} \left(\bigcup \{ [\pi]_{tf^{\rightarrow}}(s) | \pi \in \Pi \wedge \pi = [p_s, \ldots, p]\} \right) \\ &= \{\pi \mid \pi \in \Pi \wedge \pi = [p_s, \ldots, p]\} = \mathrm{Coll}_{\Pi}^{\rightarrow}(p) \end{aligned} \tag{2.5}$$

and

$$\begin{aligned}\mathrm{Coll}(p)^{\leftarrow} &= \bigcup_{s \in \{\epsilon\}} \left(\bigcup \{ [\pi]_{tf^{\leftarrow}}(s) | \pi \in \Pi \wedge \pi = [p_s, \ldots, p]\} \right) \\ &= \{\pi \mid \pi \in \Pi \wedge \pi = [p, \ldots, p_e]\} = \mathrm{Coll}_{\Pi}^{\leftarrow}(p) \end{aligned} \tag{2.6}$$

where $\mathrm{Coll}(p)^{\rightarrow}$ is the original sticky collecting semantics as in Definition 2.5 and $\mathrm{Coll}(p)^{\leftarrow}$ is the sticky collecting semantics used backwards, i.e., starting at the end node p_e.

This reduction allows us to consider only the sticky semantics for the following correctness proofs and to transfer the results to the path-based semantics. In the following, we thus only consider forward sticky collecting semantics.

2.2. Abstract Interpretation

Abstract interpretation is a formal, semantics-based framework to develop sound program analyses and to support the correctness proofs. It relates concrete semantics to abstract semantics. Within the framework of abstract interpretation, domains for the analyses are required to form *partial orders* to ensure that each subset of the domain has a *least upper bound* [23].

Preliminary Definitions

First, we provide basic definitions needed in the remainder of this chapter.

Definition 2.7 (Partial Order)
A binary relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ *over a set* $\mathbb{D}$ *is a* partial order*, if and only if* $\forall a, b, c \in \mathbb{D}$ *holds:*

$$\begin{aligned} &a \sqsubseteq a && \textit{(Reflexivity)} \\ &a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c && \textit{(Transitivity)} \\ &a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b && \textit{(Antisymmetry)} \end{aligned}$$

The partial order $\sqsubseteq$ is often used as a notion of precision. We say that $a \in \mathbb{D}$ is at least as precise as $b \in \mathbb{D}$ (or b approximates a) if $a \sqsubseteq b$.

Definition 2.8 (Complete Lattice)
A complete lattice $\mathbb{L}$ *is a partially ordered set* $\mathbb{D}$*, where each subset* $\mathbb{S} \subseteq \mathbb{D}$ *has a greatest lower bound* ($\bigsqcap$) *and a least upper bound* ($\bigsqcup$)*. The elements* $\bot = \bigsqcap \mathbb{D}, \top = \bigsqcup \mathbb{D}$ *are referred to as the* bottom *and the* top *element of* $\mathbb{D}$*. A complete lattice* $\mathbb{L}$ *is represented as a tuple* $\mathbb{L} = (\mathbb{D}, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$*. The operators* $\sqcup$ *and* $\sqcap$ *are called* join *or* meet*.*

Note that each powerset domain $2^{\mathbb{D}}$ together with subset ordering $\subseteq$ forms a complete lattice $(2^{\mathbb{D}}, \emptyset, \mathbb{D}, \subseteq, \bigcup, \bigcap)$.

In the context of program analysis and abstract interpretation functions are often required to exhibit certain properties.

Definition 2.9 (Monotonicity, Distributivity)
Let N *with* $\sqsubseteq$ *and* M *with* $\sqsubseteq'$ *be partially ordered sets. A function* $f : N \to M$ *is* monotone*, if and only if*

$$\forall a, b \in N : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq' f(b)$$

It is furthermore distributive*, if and only if*

$$\forall a, b \in N : f(a \sqcup b) = f(a) \sqcup' f(b)$$

Abstract Domain and its Relationship to the Concrete Domain

We now consider a powerset domain $2^{\mathbb{D}_C}$ of concrete states $\mathbb{D}_C$. The collecting semantics of this domain delivers the most precise information any analysis may derive. In general, however, the collecting semantics are not computable or prohibitively large [43]. Hence, we strive for another description of the concrete states, namely a domain of abstract states $\mathbb{D}_A$. Abstract interpretation provides means to relate an abstract domain with the concrete one. We first argue about correct relations between both domains. In classical abstract interpretation, a Galois connection is demanded:

Definition 2.10 (Galois Connection)
Let $(\mathbb{D}_C, \sqsubseteq)$ and $(\mathbb{D}_A, \sqsubseteq)$ be partially ordered sets and $\alpha : \mathbb{D}_C \to \mathbb{D}_A$, $\gamma : \mathbb{D}_A \to \mathbb{D}_C$ two monotone functions. The tuple $(\mathbb{D}_C, \alpha, \gamma, \mathbb{D}_A)$ is a Galois connection*, if and only if*

$$\forall d \in \mathbb{D}_C : d \sqsubseteq \gamma(\alpha(d)) \tag{2.7}$$

and

$$\forall d \in \mathbb{D}_A : \alpha(\gamma(d)) \sqsubseteq d \tag{2.8}$$

We call the function α abstraction *and γ* concretization*.*

A Galois connection establishes a relation between two domains [23]. Although one may lose precision by going back and forth between both domains, no elements are ‘lost’. Applied to the concrete and abstract domain, if $(\mathbb{D}_C, \alpha, \gamma, \mathbb{D}_A)$ forms a Galois connection, abstraction and concretization are sound and $\mathbb{D}_A$ is an actual representation of $\mathbb{D}_C$. Since an abstract domain typically exhibits a reduced size compared to the concrete domain, precision loss is often unavoidable. A weakness of the definition of Galois connection is that it allows to have two different abstract states to both represent the same set of concrete states. A stronger definition is the so called Galois insertion.

Definition 2.11 (Galois Insertion)
A Galois connection $(\mathbb{D}_C, \alpha, \gamma, \mathbb{D}_A)$ is called Galois insertion*, if and only if*

$$\forall d \in \mathbb{D}_A : d = \alpha(\gamma(d))$$

Concretising and then abstracting must yield the original abstract state.

Galois connections do not necessarily exist and are also not necessarily required for the correctness of an abstract domain. Instead, we require only Condition (2.7) of a Galois connection.

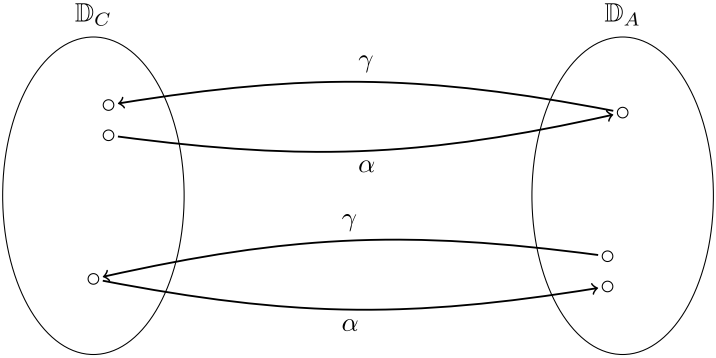


Figure 2.1.: Illustration of a Galois connection $(\mathbb{D}_C, \alpha, \gamma, \mathbb{D}_A)$. The order of the domains defines the vertical position of each element: the higher an element is, the more precise it is.

Definition 2.12 (Sound Abstraction)
Let $(\mathbb{D}_C, \sqsubseteq)$ and $(\mathbb{D}_A, \sqsubseteq)$ be partially ordered sets and $\alpha : \mathbb{D}_C \to \mathbb{D}_A$, $\gamma : \mathbb{D}_A \to \mathbb{D}_C$ two monotone functions. α and γ provide a sound abstraction *if and only if*

$$\forall d \in \mathbb{D}_C : d \sqsubseteq \gamma(\alpha(d))$$

This states that an element of the concrete domain d is conservatively approximated by $\alpha(d)$.

Correctness of the Abstract Transformer

In addition to a correct relation between two domains, we need to establish a correctness condition for the abstract transfer function

$$tf_A : V \to (\mathbb{D}_A \to \mathbb{D}_A)$$

which is also referred to as *abstract transformer.*

Definition 2.13 (Local Consistency)
Given two functions $tf_C : V \to (\mathbb{D}_C \to \mathbb{D}_C)$, $tf_A : V \to (\mathbb{D}_A \to \mathbb{D}_A)$ and a concretization $\gamma : \mathbb{D}_A \to \mathbb{D}_C$. The two functions are locally consistent, if and only if

$$\forall d \in \mathbb{D}_A : \forall p \in V : (tf_C(p))(\gamma(d)) \subseteq \gamma((tf_A(p))(d))$$

Applied to a concrete and an abstract transformer, local consistency ensures that the abstract transformer preserves all states although it may

$$\begin{tikzcd}
d_C \arrow[r, "tf_C(e)"] & d_C \arrow[d, phantom, "\rotatebox{90}{$\subseteq$}"] \\
 & {} \\
d_A \arrow[uu, "\gamma"] \arrow[r, "tf_A(e)"] & d_A \arrow[u, "\gamma"']
\end{tikzcd}$$

Figure 2.2.: Illustration of local consistency.

lose precision. In case of a Galois insertion, we can define the so-called *best abstract transformer* as $\gamma \circ tf \circ \alpha$. Note that the best transformer is often not computable in practice.

We can now define the abstract counterparts of the path semantics and the collecting semantics and prove the correctness. We simply have to replace the set-union $\bigcup$ by greatest-lower bound operator $\bigsqcup$ within the definitions and argue about an abstract domain.

Definition 2.14 (Abstract Semantics)
The abstract path semantics $[\pi]_{tf_A}$ *of a path* π *is a composition of the transfer function* $tf_A : V \to (\mathbb{D} \to \mathbb{D})$ *along the path:*

$$[\pi]_{tf_A} = \begin{cases} id_{\mathbb{D}\to\mathbb{D}} & \textit{if } \pi = \epsilon \\ [p_2, \ldots, p_n]_{tf_A} \circ tf_A((p_1)) & \textit{if } \pi = p_1, \ldots, p_n \end{cases}$$

The abstract collecting semantics *for the set of initial states* $Init_A \subseteq \mathbb{D}_A$ *at node* p *is a function* $Coll_A : V \to 2^{\mathbb{D}}$ *defined as*

$$Coll_A(p) = \bigsqcup_{s \in \; Init_A} \left(\bigsqcup \{[\pi]_{tf_A}(s) | \forall [p_s, \ldots, p] \in \Pi\} \right)$$

In traditional (not abstract-interpretation based) program analysis, the abstract collecting semantics is referred to as *meet-over-all paths (MOP)* [66].

We prove the correctness of the abstract semantics with respect to the concrete semantics starting with the correctness of the abstract path semantics:

Lemma 2.1 (Correctness of the abstract path semantics)
Given a Galois connection $(\mathbb{D}_C, \alpha, \gamma, \mathbb{D}_A)$, *the abstract path semantics is a sound over-approximation of the concrete path semantics if* tf *and* tf_A *are locally consistent.*

$$\forall d \in \mathbb{D}_A : [\pi]_{tf_C}(\gamma(d)) \subseteq \gamma([\pi]_{tf_A}(d))$$

Proof
We prove the claim by induction over the length of the path π. In case of an empty path ϵ, we have

$$\begin{aligned}[\epsilon]_{tf_C}(\gamma(d)) &= \gamma(d) \\ &= \gamma([\epsilon]_{tf_A}(d))\end{aligned}$$

Induction hypothesis:

$$\forall d \in \mathbb{D}_A : [\pi]_{tf_C}(\gamma(d)) \subseteq \gamma([\pi]_{tf_A}(d))$$

holds for all π of length $\leq l$.
Induction step $\pi \rightarrow \pi' = (\pi, p_{l+1})$

$$\begin{aligned}[\pi']_{tf_C}(\gamma(d)) &= ([\pi']_{tf_C} \circ \gamma)(d) & \\ &= ([\pi \cdot p_{l+1}]_{tf_C} \circ \gamma)(d) & \pi' = \pi \cdot p_{l+1} \\ &= ([\pi]_{tf_C} \circ tf(p_{l+1}) \circ \gamma)(d) & \textit{Def. 2.4} \\ &\subseteq ([\pi]_{tf_C} \circ \gamma \circ tf_A(p_{l+1}))(d) & \textit{Local Consistency} \\ &\subseteq (\gamma \circ [\pi]_{tf_A} \circ tf_A(p_{l+1}))(d) & \textit{Ind. Hypothesis} \\ &= (\gamma \circ [\pi \cdot p_{l+1}]_{tf_A})(d) & \textit{Def. 2.14} \\ &= \gamma([\pi']_{tf_A})(d) & \pi' = \pi \cdot p_{l+1}\end{aligned}$$

□

Correctness of the abstract collecting semantics remains to be shown. We therefore need to show that the abstract collecting semantics is a sound over-approximation of the concrete collecting semantics.

Lemma 2.2 (Correctness of the abstract collecting semantics)
Given a Galois connection $(\mathbb{D}_C, \alpha, \gamma, \mathbb{D}_A)$, the abstract collecting semantics is a sound over-approximation of the concrete collecting semantics

$$\forall p \in V : \mathit{Coll}(p) \subseteq \gamma(\mathit{Coll}_A(p))$$

if tf and tf_A are locally consistent and $Init_A$ is an over-approximation of $Init$ ($\gamma(Init_A) \supseteq Init$).

Proof

$$
\begin{aligned}
&\mathit{Coll}(p)\\
&= \bigcup_{s \in \ \mathit{Init}} \Big(\bigcup \{[\pi]_{tf_C}(s) | \forall [p_s, \dots, p] \in \Pi\}\Big) && \textit{Def. 2.5}\\
&\subseteq \bigcup_{s \in \ \mathit{Init}_A} \Big(\bigcup \{[\pi]_{tf_C}(\gamma(s)) | \pi = [p_s, \dots, p] \in \Pi\}\Big) && \gamma(\mathit{Init}_A) \supseteq \mathit{Init}_A\\
&\subseteq \bigcup_{s \in \ \mathit{Init}_A} \Big(\bigcup \{\gamma([\pi]_{tf_A}(s)) | \pi = [p_s, \dots, p] \in \Pi\}\Big) && \textit{Lemma 2.1}\\
&\subseteq \bigcup_{s \in \ \mathit{Init}_A} \gamma\Big(\bigsqcup \{[\pi]_{tf_A} | \pi = [p_s, \dots, p] \in \Pi\}\Big) && \textit{Monotonicity}\\
&\subseteq \gamma\Big(\bigsqcup_{s \in \ \mathit{Init}_A} \Big(\bigsqcup \{[\pi]_{tf_A}(s) | \pi = [p_s, \dots, p] \in \Pi\}\Big)\Big) && \textit{Monotonicity}\\
&= \gamma(\mathit{Coll}_A(p)) && \textit{Def. 2.14}
\end{aligned}
$$

□

Finally, we can formulate conditions each sound abstraction of the concrete semantics must fulfill:

- the abstract domain $\mathbb{D}_A$ is a partially ordered set,
- α and γ form a *sound abstraction* of the abstract domain, and
- tf and tf_A must be locally consistent.

2.3. Fixed-Point Analysis

Definition 2.5 and 2.14 define the (abstract) semantics of a program. In general however even the abstract collecting semantics is not efficiently computable [43]. The number of paths may be infinite or just too large in practice. Hence, the *minimal-fixed-point solution (MFP)* is computed [46, 43].

Definition 2.15 (Minimal-Fixed-Point Solution (MFP))
The minimal fixed-point (MFP) $\mathit{MFP} : V \to (\mathbb{D} \to \mathbb{D})$ *is the least fixed-point of the recursive equation system*

$$
\mathit{MFP}(p) = \begin{cases} \bot & p = p_s \\ \bigsqcup \{tf(p)(\mathit{MFP}(p_m)) \mid (p_m, p) \in E\} & p \neq p_s \end{cases}
$$

We now describe the conditions an abstraction must fulfill to ensure that i) the MFP is computable and ii) it results in an approximation of the collecting semantics Coll_A.

Theorem 2.1 (Knaster-Tarski [87])
In a complete lattice $\mathbb{L}$, each monotone function $f : \mathbb{L} \rightarrow \mathbb{L}$ has a minimal fixed-point.

According to Theorem 2.1, a minimal fixed-point as defined by MFP exists, if the transfer function tf is monotone [87].

Definition 2.16 (Ascending Chain)
An ascending chain *x is a sequence of elements $x_i \in \mathbb{D}$ of a partially ordered set $(\mathbb{D}, \sqsubseteq)$, such that $\forall i : x_i \sqsubseteq x_{i+1}$. A chain stabilizes if $\exists j : \forall l > j : x_l = x_j$.*

Theorem 2.2 (Kleenes Fixed-Point Theorem)
Let $\mathbb{L} = (\mathbb{D}, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ be a lattice and $f : \mathbb{D} \rightarrow \mathbb{D}$ be a monotone function. If all ascending chains in $\mathbb{L}$ finally stabilize, then

$$\exists k : \forall l > k : f^l(\bot) = f^k(\bot)$$

and $f^k(\bot)$ is the least fixed-point of f.

Kleenes' Theorem 2.2 defines the properties under which the iterative application of a function finally stabilizes and results in the minimal fixed-point. Note that there are several algorithms implementing the MFP Solution and therefore solving the data-flow problem. A detailed discussion on this topic can be found in [56].

Monotonicity holds for probably all meaningful transfer functions. However, there are some lattices where ascending chains do not stabilize. A prominent example is the *interval analysis* which tries to predict a value interval $[a, b]$ for each program variable v such that $v \in [a, b]$. For such analyses, *widening* and *narrowing* operators can be defined to restore computability [22]. A widening-operator sets the data-flow value to an upper bound (in case of interval analysis ∞) such that the algorithm results in a—possibly not-minimal but—valid fixed-point. Narrowing may then be used to reduce the pessimism introduced by widening.

So far, we only discussed computability of the MFP solution. The next theorems argue about soundness and precision of MFP compared to Coll_A. Remember that the precision is defined by the relation $\sqsubseteq$ (see Definition 2.7).

Theorem 2.3 (Coincidence)
Let $\mathbb{L} = (\mathbb{D}, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ be a lattice and $f : \mathbb{D} \to \mathbb{D}$ be a monotone function. If all ascending chains in $\mathbb{L}$ finally stabilize, then

$$\forall p \in V : \mathit{Coll}_A(p) \sqsubseteq \mathit{MFP}(p)$$

The Coincidence-Theorem 2.3 states that MFP approximates Coll_A, i.e., $\mathrm{Coll}_A \sqsubseteq$ MFP for all possible initial states and all nodes. According to Theorem 2.4 equality is given, if the transfer function is distributive [46].

Theorem 2.4 (Kildall [46])
Let $\mathbb{L} = (\mathbb{D}, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ be a lattice, $f : \mathbb{D} \to \mathbb{D}$ be a monotone function and . If f is furthermore distribute, then

$$\forall p \in V : \mathit{Coll}_A(p) = \mathit{MFP}(p)$$

(assuming that each node $p \in V$ is reachable from the starting node p_s).

We can conclude that a transfer function needs to be monotone in order to be able to compute an approximation of the collecting semantics. If all ascending chains within the corresponding lattice stabilize, fixed-point iteration will result in a correct approximation of the Coll_A. If not, widening and/or narrowing operators must be defined. If in addition, the transfer function is distributive, MFP equals Coll_A.

Chapter 3

Background

If I have been able to see further, it
was only because I stood on the
shoulders of giants.
Sir Isaac Newton (1643 - 1727)

This chapter presents the basics of different fields necessary for the understanding of this thesis: real-time scheduling, caches and timing analysis. Nevertheless, it is not meant as a general introduction but presents only the relevant information and notation. Parts familiar to the reader can be skipped.

3.1. Real-Time Scheduling

Embedded systems typically feature more tasks than processors. Thus, a *scheduler* is employed to distribute the available processing time to the tasks. In contrast to schedulers in general-purpose OS, where load-balancing or fairness is a main objective, real-time schedulers are faced with timing constraints such as deadlines of tasks. Depending on the penalty assigned to a deadline miss, real-time systems are divided into hard or soft. When a deadline miss is considered a complete failure of the system, it is considered a *hard* real-time system, when some deadline misses are tolerable, it is considered a *soft* real-time system.

Real-time scheduling denotes a large research area with plenty of different approaches and methods of which this section only presents the subset relevant for this thesis: uniprocessor preemptive priority-driven scheduling with static priority assignment as widely used in real-time systems. For a general overview on real-time scheduling see [19].

3.1.1. Sporadic Task Model

We assume a fixed set Γ of n tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$ to be executed on a single processor. Each task is assigned a worst-case execution time C_i, a relative *deadline* D_i and minimal inter-arrival time or *period* T_i with *release jitter* J_i. Each instance of task execution is called a job. *Arrival time* a_i^j denotes the time at which the j-th instance of task τ_i (or *job* j of τ_i) becomes *ready* to execute, and start time s_i^j the time the job is *dispatched* by the scheduler and starts to execute. Figure 3.1 shows the states and *transitions* of a task. The next job of task τ_i arrives at the earliest after a time span T_i minus jitter J_i:

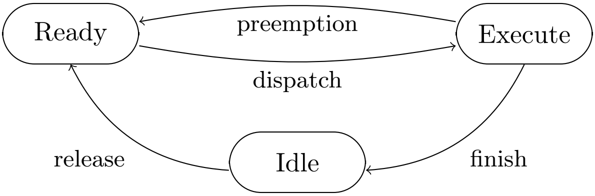


Figure 3.1.: State transitions of a task

$$a_i^{j+1} \geq a_i^j + T_i - J_i \tag{3.1}$$

Deadline d_i^j of a job is relative to its arrival time:

$$d_i^j = a_i^j + D_i \tag{3.2}$$

Finishing time f_i^j of a job denotes the time at which this job completes execution and the *response time* r_i^j denotes the time span from job arrival to job completion:

$$r_i^j = f_i^j - a_i^j \tag{3.3}$$

Response time of a task is given by the worst-case response time of its jobs:

$$R_i = \max_j \{r_i^j\} \tag{3.4}$$

Worst-case execution time C_i denotes the maximal execution demand *without* preemption cost. The fraction of time a task requires the processor is denoted as the task's utilization:

$$U_i = C_i/T_i \tag{3.5}$$

Utilization of a task set $\Gamma = (\tau_1, \ldots, \tau_n)$ is the sum of the utilizations of all tasks:

$$U_\Gamma = \sum_{i=1}^{n} C_i/T_i \tag{3.6}$$

A task τ_j is referred to as *schedulable* if each of its jobs finishes before their deadline:

$$R_i \leq D_i - J_i \Leftrightarrow \tau_i \text{ schedulable} \tag{3.7}$$

A task set Γ is said to be schedulable if each task $\tau_i \in \Gamma$ is schedulable.

Table 3.1.: Task model properties

$\Gamma = \{\tau_1, \ldots, \tau_n\}$	Task set of n tasks
C	Execution Time Demand (WCET)
T	Minimal Inter-arrival Time
D	Deadline
J	Release Jitter
U	Utilization
s	Starting Time
a	Arrival Time
f	Finishing time
r	Response Time (Job)
R	Response Time (Task)

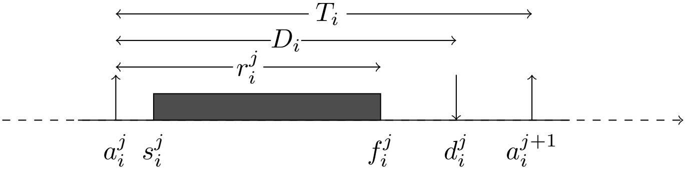
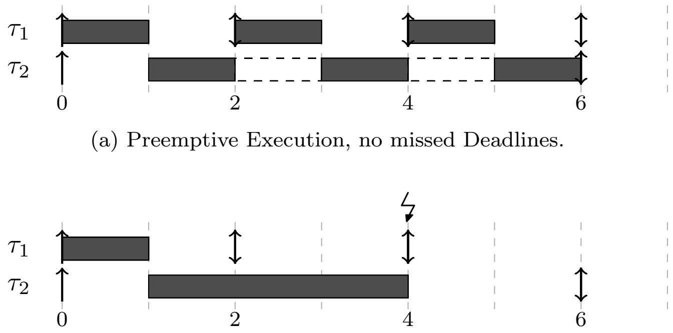


Figure 3.2.: Illustration of the sporadic task model and the associated variables

Table 3.1 summarizes and Figure 3.2 illustrates the parameters associated with each task/job. We assume independent tasks but weaken this restriction and allow tasks to access *mutual exclusive* sections via *semaphores* in Section 3.1.3

Why preemption?

Tasks can be executed either *preemptively* or *non-preemptively.* In a non-preemptively scheduled system, once started, each task runs to completion. In a preemptively scheduled system, a task's execution can be temporarily interrupted to execute another task. Preemptive schedules are potentially more powerful, i.e, some task sets are only schedulable preemptively. Tasks with short deadline usually can not postpone their execution until completion of the currently running task. See the task set shown in Figure 3.3.



(a) Preemptive Execution, no missed Deadlines.

(b) Non-Preemptive Execution, Deadline Miss of task τ_1 at 4.

Figure 3.3.: Preemptive versus Non-preemptive Scheduling: task set $\{\tau_1, \tau_2\}$ with $C_1 = 1$, $D_1 = T_1 = 2$ and $C_2 = 3$, $D_2 = T_2 = 6$.

3.1.2. Priority-Driven Scheduling

A priority-driven scheduler always executes the task τ_i with the highest *priority* $pr(\tau_i)$ among all tasks that are ready to execute. Priorities are assigned either *statically* or *dynamically. Earliest deadline first (EDF)* scheduling [53] and *deadline monotonic (DM)* scheduling—also called *Rate Monotonic (RM)* [53] in case of implicit deadlines ($\forall_i : D_i = T_i$)—are the main representatives for dynamic and static assignments, respectively. They are also optimal among scheduling algorithm of their class in the following sense: if there exists a feasible schedule with static, resp., dynamic priority assignment, *deadline monotonic*, resp., *earliest deadline* *first* finds one too. Note that these optimality results only hold for preemptive systems assuming negligible preemption cost.

Earliest Deadline First (EDF)

Earliest deadline first scheduling [53] always executes the task with the closest deadline. It maintains a priority queue of the ready tasks ordered by the proximity of each task's deadline. For the special case of implicit deadlines, EDF is always able to schedule a task set if the utilization of the task set is less than or equal to 1:

$$U_\Gamma \leq 1 \Leftrightarrow \Gamma \text{ schedulable} \tag{3.8}$$

In the general case with explicit deadlines, the schedulability test is more complex [40]. For each time span L, the requested processor time must be less than or equal to L:

$$\forall L > 0 : \sum_i \lfloor (L + T_i - D_i)/T_i \rfloor C_i \leq L \Leftrightarrow \Gamma \text{ schedulable} \tag{3.9}$$

Both schedulability tests are necessary and sufficient.

Deadline Monotonic (DM)

Deadline monotonic scheduling [52, 100] assigns task priorities in order of increasing deadlines, giving the task with the shortest deadline the highest priority. We assume w.l.o.g. that task indices express the priority order:

$$D_1 \leq D_2 \leq D_3 \leq \ldots \leq D_n \tag{3.10}$$

and

$$pr(\tau_1) > pr(\tau_2) > pr(\tau_3) > \ldots > pr(\tau_n) \tag{3.11}$$

For implicit deadlines ($T_i = D_i$), two linear-time sufficient schedulability tests based on the utilization of the task set exist. The first one was presented by Liu and Layland [53]:

$$U_\Gamma \leq n(2^{1/n} - 1) \Rightarrow \Gamma \text{ schedulable} \tag{3.12}$$

with

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln(2) \approx 0.6932 \tag{3.13}$$

The second schedulability test is the *hyperbolic bound* test by Bini et al. [15]:

$$\prod_{i=1}^{n} (U_i + 1) \leq 2 \Rightarrow \Gamma \text{ schedulable} \tag{3.14}$$

Note that both tests are only valid for implicit deadlines and are only sufficient, not necessary. For a performance evaluation of RM scheduling based on randomly generated tasks see [51].

Response Time Analysis

A sufficient and necessary test for static priorities is the *response time analysis* [8, 41, 25]. Equation (3.7) states that a task is schedulable, if its worst-case response time R_i is less than or equal to its deadline D_i less jitter J_i.

The response time R_i of a task necessarily contains its execution time C_i. In addition, τ_i suffers interference from tasks with higher priority than τ_i. Let τ_j be a task with priority higher than τ_i. Within the response time R_i, τ_j executes at most $\left\lceil \frac{R_i+J_j}{T_j} \right\rceil$ times, each time for at most C_j. Hence, the response time R_i of task τ_i is given by:

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j) \tag{3.15}$$

where $\mathrm{hp}(\tau_i)$ denotes the set of tasks with higher priority than τ_i. The response time R_i appears on the left-hand side and on the right-hand side. As the right-hand side is monotonically increasing in R_i, a fixed-point computation of R_i based on Equation (3.15) can be defined as follows:

$$R_i^0 = C_i \tag{3.16}$$

$$R_i^{l+1} = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i^l + J_j}{T_j} \right\rceil (C_j) \tag{3.17}$$

Eventually, for each task, the fixed-point computation either stabilizes, i.e., $R_i^l = R_i^{l+1}$, or the response time R_i^l of one task exceeds the deadline $R_i > D_i$. In the first case, we can conclude that each job of task τ_i will finish before its deadline, so each task is schedulable. In the second case, we can conclude that at least one job of task τ_j may miss its deadline, so the task set is not schedulable.

Note that the response time analysis is valid for any *static priority assignment* and also for explicit deadlines. It exhibits pseudo-polynomial runtime (compared to linear runtime for the Utilization and Hyperbolic bound), but forms a sufficient and necessary schedulability test (under the assumption of negligible context switch costs).

3.1.3. Mutual Exclusion

In this section, we extend the sporadic task model by *mutual exclusive* accesses to *shared resources*. A sequence of a task τ_i accessing such a shared resource is called *critical section*.

To ensure consistency of shared resources, at most one task can access a resource at a time. *Semaphores* are typically used to implement this *mutually exclusive* access: Each shared resource is assigned a semaphore S_i that implements two primitive operations

wait(S_i): a task requests access to a shared resource. If the resource's semaphore S_i is available, a task can enter its critical section and access the shared resource, while locking S_i. If S_i is not available, the task blocks on S_i and suspends its execution.

signal(S_i): a task that previously accessed a shared resource assigned to semaphore S_i finishes its critical section and unlocks S_i again.

Each critical section accessing a shared resource with semaphore S_i thus begins with *wait(S_i)* and ends with *signal(S_i)*. Task states and transitions are depicted in Figure 3.4.

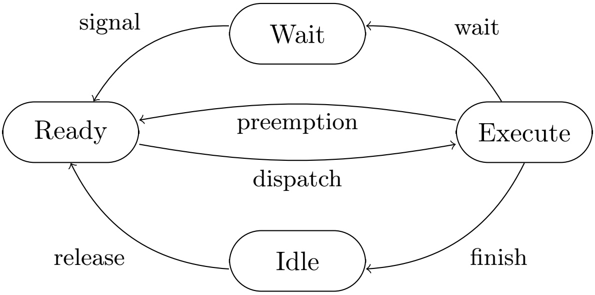


Figure 3.4.: State transitions of a task with shared resources

Priority Inversion Problem

Each task τ_i that shares a resource with a task τ_j with lower priority inevitably may suffer delay of the length of the critical section of τ_j accessing the shared resource. *Priority inversion* denotes the situation where a task's execution is prolonged by the execution of a task with lower priority even though both tasks do *not* share a common resource. See Figure 3.5 for an example. Task τ_1 and τ_3 both access a shared resource guarded by *semaphore S*. Task τ_3 first enters its critical section and locks S. Task τ_1 is activated at time 2, starts to executes but blocks on S at time 3. Now τ_3 resumes execution again, but is preempted by τ_2 at time 4. So, the τ_1 is not only delayed by the critical section of τ_3 but also by the execution time of τ_2. Several protocols have been proposed to solve the priority inversion problem. We will now shortly present the most prominent protocols: *priority inheritance*, *priority ceiling* and *stack resource protocol*. As only the latter one is able to handle dynamic priorities, we restrict the following description to *static priority assignment*.

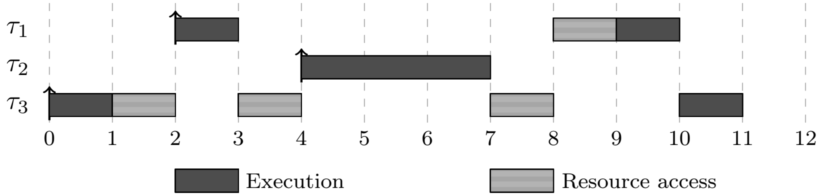


Figure 3.5.: Priority inversion problem

Table 3.2.: Shared resources: notation and terminology

SR_i	shared resource
S_i	semaphore guarding SR_i
$\mathrm{Access}(\mathrm{SR}_i)$	set of tasks accessing shared resource SR_i
pr	nominal priority
$\widehat{pr_i}$	active priority with $\widehat{pr_i} \geq pr_i$
$C(S_i)$	Ceiling priority of semaphore S_i (PCP)
S^*	Currently locked semaphore with highest $C(S)$ (PCP)
$cs_{i,k}$	critical section of task τ_i accessing resource SR_k
$C_i^{\mathrm{SR}_k}$	worst-case execution time of the critical section $cs_{i,k}$

Priority Inheritance Protocol (PIP)

To prevent priority inversion, the *priority inheritance protocol (PIP)* [77] increases the priority of a blocking task τ_i to the priority of the task with the highest priority currently blocked by τ_i. It therefore assigns each task an *active priority* $\widehat{pr_i}$ initially set to the *nominal priority* pr_i (the one assigned by the scheduling algorithm, e.g. deadline monotonic).

The protocol works as follows:

- the task τ_i with the highest active priority $\widehat{pr_i}$ among all ready tasks is executed.
- when the running task τ_i requests a semaphore S_l:

  if S_l is free, τ_i locks S_l and enters its critical section.

  if S_l is blocked by another task τ_j, τ_j *inherits* the priority of τ_i: $\widehat{pr_j} := \widehat{pr_i}$.
- when a task τ_i exits its critical section, it releases the corresponding semaphore S_l and:

  if τ_i blocks no other task, the active priority of τ_i is set to the nominal priority $\widehat{pr_i} := pr_i$.

  otherwise, the active priority of τ_i is set to the active priority of the task τ_j with the highest priority of all tasks still blocked by τ_i: $\widehat{pr_i} := \widehat{pr_j}$.

Note that priority inheritance is transitive.

Although PIP prevents priority inversion, a task may still suffer substantial delay by the blocking time of lower-priority tasks. In the worst-case, τ_i may suffer m blocking delays where m denotes the number of shared resources τ_i accesses. Such a situation is called chained blocking. In addition, nested shared resources may lead to a deadlock situation [19].

Priority Ceiling Protocol (PCP)

Priority Ceiling Protocol (PCP) by Sha et al. [77] improves over PIP in the sense that a) each task can be delayed by at most one critical section and b) deadlocks are prevented. Instead of assigning priorities only to tasks, each semaphore S_i is statically assigned a ceiling priority $C(S_i)$. The ceiling priority is the highest priority of any task accessing the shared resource guarded by S_i:

$$C(S_i) = \max\{pr_j \mid \tau_j \in \text{Access}(\text{SR}_i)\} \tag{3.18}$$

where $\text{Access}(\text{SR}_i)$ denotes the set of tasks accessing shared resource SR_i. In addition, PCP keeps track of the currently locked semaphore S^* with the highest ceiling priority $C(S_l)$ of all locked semaphores. The protocol prevents a task τ_i from entering a critical section guarded by S_l not only if S_l is locked, but also if there is any semaphore currently locked that could lead to blocking of τ_i.

The protocol works as follows:

- the task τ_i with the highest active priority $\widehat{pr_i}$ among all ready tasks is executed.
- when the running task τ_i requests a semaphore S_l:

  if $\widehat{pr_i} > C(S^*)$, τ_i locks S_l and enters its critical section.

  otherwise τ_i is blocked by another task τ_j holding semaphore S^* and τ_j inherits the priority of τ_i: $\widehat{pr_j} := \widehat{pr_i}$.
- when a task τ_i exits its critical section, it releases the corresponding semaphore S_l and updates S^*. Furthermore,

  if τ_i blocks no other task, the active priority of τ_i is set to the nominal priority $\widehat{pr_i} := pr_i$.

  otherwise, the active priority of τ_i is set to the priority of the task τ_j with the highest priority of all tasks still blocked by τ_i: $\widehat{pr_i} := \widehat{pr_j}$.

With PCP, the maximal blocking delay a task τ_i may suffer is bounded by the maximal execution time $C_j^{\mathrm{SR}_k}$ of a critical section SR_k of a lower priority task τ_j that shares a common resource SR_k guarded by a semaphore with ceiling priority $C(S_k)$ higher than or equal to the priority τ_i:

$$B_i = \max\{C_j^{\mathrm{SR}_k} \mid pr_j < pr_i \wedge C(S_k) \geq pr_i\} \tag{3.19}$$

Stack Resource Protocol (SRP)

The *stack resource protocol (SRP)* [9] denotes a set of extensions to PCP. SRP is applicable in case of dynamic priorities, allows for multi-unit resources, i.e., resources that enable up to l parallel accesses, and enables sharing of runtime stack resources. In the simple case we consider here, where we are only interested in static priority assignment and binary semaphores that are either locked or free, the improvement of SRP merely breaks down to the advantage that SRP exhibits less context switches than PCP. This, however, can also be achieved in the priority ceiling protocol by adding the following rules:

- When a task enters a critical section, its priority is set to $C(S^*)$.
- When a task leaves the critical section, its former priority is restored.

Thus, a task τ_i with priority less than or equal to $C(S^*)$ is prevented from being activated and not only from entering a critical section. This results in less context switches besides otherwise equal behavior compared to the original priority ceiling protocol.

Mutual Exclusion and Response Time Analysis

The response time Equation (3.15) can be extended as follows to include the blocking delay τ_i may suffer:

$$R_i = C_i + B_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j) \tag{3.20}$$

with B_i defined by Equation (3.19).

3.2. Memory Hierarchy and Caches

Fast and large memory are desirable, but infeasible due to technical (and economical) limitations. Instead, existing storages are either large and slow or small and fast. To emulate a memory which is fast and large at the same time, *memory hierarchies* were introduced based on memories with varying speeds and sizes: Small but fast memories on top, slower but larger memories below. Each memory level contains a subset of the data stored in the level below. Scratchpads, caches or buffers are examples for such intermediate memories. See Figure 3.6 for a simple memory hierarchy of an embedded architecture.

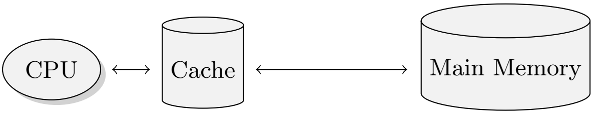


Figure 3.6.: Typical memory hierarchy often to be found in embedded systems

3.2.1. Principle of Locality

To decide which memory blocks to store, a memory hierarchy utilizes a heuristic known as the *locality principle* [26]:

Spatial Locality neighboring memory blocks are likely to be accessed contemporary.

Temporal Locality recently accessed memory blocks are likely to be accessed in the near future again.

The rationale behind these locality principles is the exploitation of common characteristics of task executions. Memory accesses are not uniformly distributed over a task's data. Sequential code alignment and clustering of data (arrays etc.) increase the likelihood of accessing neighboring memory blocks. Loop structures increase the likelihood of reusing recently accessed memory blocks. Spatial locality is realized by storing not only the currently accessed data but larger chunks of contiguous memory containing the accessed blocks. Temporal locality is realized by storing recently accessed data.

In the following, we focus on uni-level processor caches common in typical embedded systems. For an overview on memory hierarchies and caches, we refer to Hennesy and Patterson [37]. We also skip description of memory management units and virtual memory management, as both concepts are not common in hard real-time systems.

3.2.2. Processor Caches

Typically located on the die of the microprocessor, *processor caches* serve data much faster than the main memory which is connected by the memory bus. Caches operate completely transparently to the processor semantics, only influencing timing behavior and performance. On a memory access (no matter if read or write), data is requested from the cache. In case the accessed memory block is resident in cache, a situation called *cache hit*, the processor directly reads from/writes to cache. Only in case of a *cache miss*, i.e., data not resident in cache, the requested data is retrieved from the main memory and then loaded to the cache. There are three types of cache misses [38]:

Compulsory Misses misses on the first access of a memory block. As caches are initially empty, the first access to a datum always inflicts a cache miss.

Capacity Misses misses due to the limited cache capacity. Such misses occur if the amount of accessed data exceeds the cache size.

Conflict Misses misses due to an unbalanced cache usage, i.e., misses due to eviction in one cache set, while cache lines of other sets are still empty.

The delay to retrieve data from main memory is referred to as *cache-miss penalty* or *block-reload time*.

Cache Organization

To implement spatial locality, caches do not only store the accessed data, but load memory blocks of *line size* L usually substantially larger than the accessed element. Caches are partitioned into S *cache sets*, where each memory block (of size L) maps to exactly one of the cache sets. Each cache set in turn may contain up to K different memory blocks at once, where K is referred to as the *associativity* of the cache. *Cache size* is thus given by $L \cdot S \cdot K$. Such a cache is called a set-associative cache. See Figure 3.7. The set of all memory blocks is denoted by M. We introduce symbol $\perp$ to represent empty or invalid cache lines.

Table 3.3.: Cache parameter and domains

Line Size	$L \in \mathbb{N}$
Associativity	$K \in \mathbb{N}$
Number of Sets	$S \in \mathbb{N}$
Cache Size	$CS \in \mathbb{N}; CS = L \cdot S \cdot K$
Policy	$P \in \{\text{LRU}, \text{FIFO}, \text{PLRU}\}$
Cache-Set State	$\zeta \in Z$
Set of Memory Blocks	M
Invalid Line	$\perp$
Content of a cache Line	$M_\perp = M \cup \{\perp\}$
Cache-Set Update	$t^P : M \times \zeta^P \rightarrow \zeta^P$
Content of a Cache Set	$\overline{\zeta} \in 2^{M_\perp}$
Number of misses on path π and on an initial cache state ζ	$\mathit{miss} : \Pi \times Z \rightarrow \mathbb{N}$
Number of hits on path π and on an initial cache state ζ	$\mathit{hits} : \Pi \times Z \rightarrow \mathbb{N}$

There are two special cases of set-associative caches:

Direct-Mapped Caches ($K = 1$) Each memory block can reside in exactly one cache line. Direct-mapped caches exhibit low hardware implementation cost. However, memory accesses are usually not uniformly distributed over all cache sets resulting in unnecessary conflict misses.

Fully-Associative Caches ($K = S$) Each memory block can reside in any cache line. Fully-associative caches only need to evict data if all cache lines of the whole cache are filled. Hardware implementation of such caches is costly so that only small caches are fully-associative.

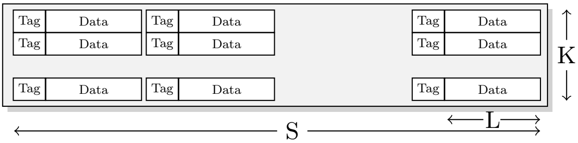


Figure 3.7.: Cache organization on a K-way set associative cache

Cache Tags and Block Addresses

To identify which parts of the main memory are currently cached, caches must store not only the actual data but also tags to identify the memory address the data belongs to.

A *tag* is the smallest possible portion of the *memory address* sufficient to reconstruct the original address. As caches always store blocks of size L, the last $\log(L)$ bits of the address do not need to be considered. The same holds for the previous $\log(S)$ bits, as this part of the address can be reconstructed from the index of the cache set in which data is stored. The remaining bits form the tag of an address and need to be stored together with the data. See Figure 3.8.

Tag	Set Index	Offset

Figure 3.8.: Address Computation

Write-Through vs. Write-Back

On a write access, a changed datum must be written back to the main memory eventually. When to perform this write back depends on the write policy. The two alternatives are:

Write-Through datum is directly written to main memory, or,

Write-Back datum is marked as *dirty* and written to main memory when evicted from cache.

Write policies form another trade-off between hardware cost and performance. A write-back cache requires book-keeping of all dirty cache lines. However, it benefits from fewer memory accesses compared to a write-through cache, which sends data to main memory on each write access. Another decision related to write accesses is the question whether or not to store the written data in the cache. The alternatives are called *write-allocate* and *no write-allocate*. Typical combinations are *write-through* with *no write-allocate* (as each write incurs a main memory access) and *write-back* with *write-allocate*.

For timing-critical embedded systems, write-through caches are considered beneficial as the point in time of each main-memory write depends statically on the instruction. Write-back caches do not allow for a precise determination of these memory-writes—or at least, no precise analysis is known yet.

3.2.3. Replacement Policy

Eventually, all ways of a cache set are filled. On a cache miss, the currently accessed element has to be stored in the cache while evicting another one. Which element to replace is determined by the *replacement policy*. The most common replacement policies are *least-recently used (LRU)*, *first-in first-out (FIFO)* and *pseudo least-recently used (PLRU)*. Note that no replacement policy is needed for direct-mapped caches as each set has only one way and each memory block maps to a unique cache position. In the following, we refer to a position in a cache as the *age* of a cache block. These are always meant as logical concepts and do not refer to the physical position in the cache hardware.

Least-Recently Used (LRU)

LRU policy keeps a list of cached memory blocks ordered by the last use of each memory block. To keep this order, it conceptually assigns each cached memory block an *age* indicating its position in the order. The most-recently used element has age 0, the least-recently used age $K-1$. It treats misses and hits uniformly. A property considerably useful for timing analysis and predictability of a cache. Starting with a completely unknown initial cache state, precise information about the cache content (and the age of each memory block) can be derived after accessing K different memory blocks. LRU caches are predominant in academia, but

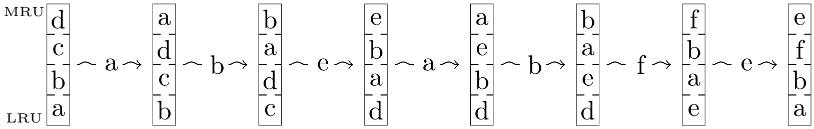


Figure 3.9.: Access sequence on a LRU replacement policy

due to the implementation cost rather unpopular in modern CPUs for higher associativities—even though efficient LRU implementations are feasible [1].

We represent an LRU cache state $\zeta \in Z$ as ordered lists of K elements $[x_1, x_2, \dots, x_K]$ with the MRU element at front:

$$\zeta^{\text{LRU}} : \underbrace{M_\perp \times M_\perp \times \ldots \times M_\perp}_{K} \tag{3.21}$$

The LRU-update is formally defined as follows:

$$\mathbb{U}^{\text{LRU}} : M \times \zeta^{\text{LRU}} \rightarrow \zeta^{\text{LRU}}$$

The currently accessed element is put at the first position in the cache (no matter if hit or miss). All younger elements are moved to the right, i.e., aged by one, possibly evicting the *least-recently used* element. All older elements retain their position.

$$\mathbb{U}^{\text{LRU}}(m, [x_1, \dots, x_K]) := \\ \begin{cases} [m, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_K] & m = x_i \\ [m, x_1, \dots, x_{K-1}] & \textit{otherwise} \end{cases} \tag{3.22}$$

First-In First-Out (FIFO)

In FIFO caches, elements are ordered by the time when the element was loaded to the cache. Although FIFO replacement policy is commonly implemented by means of a pointer directing to the element evicted next, we depict FIFO states as ordered list with the first element on top and the last at bottom—similar to the representation of LRU cache states.

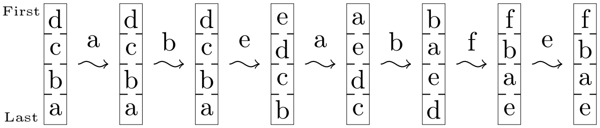


Figure 3.10.: Access sequence on a FIFO replacement policy

$$\zeta^{\text{FIFO}} : \underbrace{M_\perp \times M_\perp \times \ldots \times M_\perp}_{K} \tag{3.23}$$

In case of a miss, the currently accessed element is inserted at the first position, shifting all others to the right while evicting the right-most one.

Hits do not change the cache state.

$$\mathbb{U}^{\text{FIFO}} : M \times \zeta^{\text{FIFO}} \rightarrow \zeta^{\text{FIFO}}$$

$$\mathbb{U}^{\text{FIFO}}(m, [x_1, \ldots, x_K]) := \begin{cases} [x_1, \ldots, x_K] & m = x_i \\ [m, x_1, \ldots, x_{K-1}] & \textit{otherwise} \end{cases} \tag{3.24}$$

Pseudo Least-Recently Used (PLRU)

PLRU mimics LRU caches with lower implementation cost. PLRU caches can be best explained using a graphical representation (see Figure 3.11). Instead of storing an age for each cache line, a binary pointer tree (with $K-1$ bits) determines the element evicted next. On a cache hit, all pointers on the path to the accessed element m are flipped away from m to rejuvenate this element. On a cache miss, the indicated element is replaced while flipping away all pointers on this path. Due to the structure, PLRU policies requires K to be a power of two, i.e., $K = 2^j$. We define PLRU update and domain recursively starting with $K = 1$:

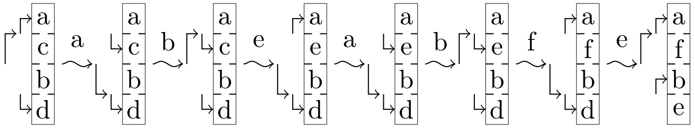


Figure 3.11.: Access sequence on a PLRU replacement policy

$$\zeta^{\text{PLRU}(1)} : M_{\perp} \tag{3.25}$$

$$\mathbb{U}^{\text{PLRU}(1)}(m, \zeta^{\text{PLRU}(1)}) := m \tag{3.26}$$

As K is a multiple of 2, we can define the domain PLRU for $K > 1$ by means of PLRU$(K/2)$:

$$\zeta^{\text{PLRU}(1)} : \{0, 1\} \times \zeta^{\text{PLRU}(K/2)} \times \zeta^{\text{PLRU}(K/2)} \tag{3.27}$$

and also the cache update:

$$\mathbb{U}^{\text{PLRU}(K)}(m, (b, \zeta_0, \zeta_1)) := \begin{cases} (1, t^{\text{PLRU}(K/2)}(m, \zeta_0), \zeta_1) & m \in \zeta_0 \\ (0, \zeta_0, t^{\text{PLRU}(m,K/2)}(\zeta_1)) & m \in \zeta_1 \\ (1, t^{\text{PLRU}(K/2)}(\zeta_0), \zeta_1) & \perp \in \zeta_0 \\ (0, \zeta_0, t^{\text{PLRU}(K/2)}(\zeta_1)) & \perp \in \zeta_1 \\ (1, t^{\text{PLRU}(K/2)}(m, \zeta_0), \zeta_1) & b = 0 \\ (0, \zeta_0, t^{\text{PLRU}(K/2)}(m, \zeta_1)) & b = 1 \end{cases} \tag{3.28}$$

The first two cases denote cache hits, either in case of a cache hit in sub-state ζ_0 where the pointer is set to 1 or in case of a cache hit in ζ_1 where the pointer is set to 0. The last two cases denote cache misses where the element is added to ζ_0 or ζ_1 depending on the value of the pointer b. PLRU policy requires a special treatment of invalid lines. In contrast to FIFO and LRU, an invalid line in an PLRU cache may even survive an arbitrarily large access sequence [74]. To prevent such permanent invalid lines, and thus, loss of capacity, refilling invalid lines is preferred in case of a cache miss.

Sensitivity of Replacement Policies

Reineke [72, 73] has recently examined properties of different replacement policies regarding timing analysis. Among these properties, the *sensitivity* plays an important role in our context. Sensitivity refers to the possible variation of the number of hits or misses depending on the initial cache state. The sensitivity of a replacement policy is determined by a multiplicative and an additive factor. For instance, LRU policy is (1,K) miss-sensitive, where K is the associativity. This means that the number of misses on path π and initial cache state ζ_1 is at most K plus one time the number of misses on the same path but with another initial cache state ζ_2. In general, a policy is (c, s) miss-sensitive, if for any path and any two initial cache states the following holds:

Table 3.4.: Sensitivity of LRU, PLRU, and FIFO for associativity 2,4, and 8.

	Miss-Sensitivity			Hit-Sensitivity		
	2	4	8	2	4	8
LRU	$(1,2)$	$(1,4)$	$(1,8)$	$(1,2)$	$(1,4)$	$(1,8)$
FIFO	$(2,2)$	$(4,4)$	$(8,8)$	$(0,0)$	$(0,0)$	$(0,0)$
PLRU	$(1,2)$	∞	∞	$(1,2)$	$(\frac{1}{3},\frac{5}{3})$	$(\frac{1}{11},\frac{19}{11})$

$$miss^P(\pi, \zeta_1) \leq c * miss^P(\pi, \zeta_2) + s \tag{3.29}$$

where $miss^P(\pi, \zeta)$ denote the number of misses of policy P on path π and cache state ζ. Note that the bound in Equation (3.29) is tight, i.e., if policy P is (c, s) miss-sensitive, then there exists a path π and two cache states ζ_1, ζ_2, such that $miss(\pi, \zeta_1) = c * miss(\pi, \zeta_2) + s$. The hit-sensitivity is defined accordingly; a policy is (c, s) hit-sensitive, if for any path and any two initial cache states the following holds:

$$hits^P(\pi, \zeta_1) \geq c * hits^P(\pi, \zeta_2) - s \tag{3.30}$$

where $hits^P(\pi, \zeta)$ denote the number of hits of policy P on path π and cache state ζ. Table 3.4 shows the hit-, and miss-sensitivity of different replacement policies for associativities 2,4 and 8. ∞ indicates that the variation in the number of misses/hits is not bounded. Note that the values are given assuming a fully-associative cache but can be lifted to set-associative caches by multiplying the additive (subtractive) factor by the number of sets. The multiplicative factor remains the same.

The sensitivity is related to the analysis of preemptively scheduled systems, since a preemption results in a change of the cache state of the preempted task, while the subsequent execution path after preemption remains unchanged.

3.3. Timing and Cache Analysis

Real-time systems are subject to timing constraints, induced by the surrounding environment. In case of hard real-time systems, failing these constraints is considered a system failure. Hence, considerable effort is taken in order to prove the timing correctness.

This is usually done in two steps: a timing analysis derives bounds on the execution time of tasks which are then used in a subsequent scheduling analysis to prove the correctness of the system's timing behavior. This section provides an introduction to timing analysis relevant for this thesis. For a complete overview see [97] or [101].

3.3.1. Components of a Timing Analysis

A task's timing behavior depends on the initial processor state (including the cache) and on the input. This dependency results in a variation of the possible execution of the task. In general, it is computationally infeasible to explore all different executions and thus, to derive the exact *worst-case execution time (WCET)*, resp. *best-case execution time (BCET)*. Instead, a *static timing analysis* can be used to derive bounds on the execution time based on abstract models of the system. Note that the term WCET is often used inconsistently as the worst-case execution time and as a bound on the worst-case execution time. To clearly distinguish between the value and its bound, we use WCET for the value and WCET^B for the upper bound; the same holds for BCET and BCET^B.

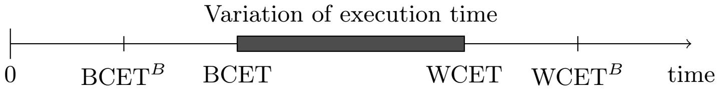


Figure 3.12.: Variation of the execution times, BCET, WCET and bounds on BCET and WCET

For the sake of simplicity we assume a) that the initial hardware state only depends on the initial cache state ζ and that b) the input variation is reflected by a variation of the execution path π. Let $ET(\pi, \zeta)$ be the execution time of the path π on initial state ζ:

$$WCET = \max_{\zeta \in Z, \pi \in \Pi} (ET(\pi, \zeta)) \tag{3.31}$$

$$BCET = \min_{\zeta \in Z, \pi \in \Pi} (ET(\pi, \zeta)) \tag{3.32}$$

$$\forall \zeta \in Z, \pi \in \Pi : \mathrm{BCET}^B \leq ET(\pi, \zeta) \leq \mathrm{WCET}^B \tag{3.33}$$

Conceptually, timing analysis abstracts from the concrete program semantics to reduce the state space and thus, to render a derivation of timing bounds feasible. Two main abstractions are given by the *micro-architectural analysis* and the *cache analysis*:

Micro-architectural (pipeline) Analysis The microarchitectural analysis forms an abstract pipeline model that abstracts from all non-timing relevant features. For instance, the concrete operands of an add-instruction do not influence the timing, while the type of the operands (floating point, integer) does. The complexity of the model strongly depends on the complexity of the pipeline. In the simplest case, *counting* instructions suffices while complex hardware features such as speculative execution and branch prediction, require a more sophisticated hardware model [30, 36] (cf. Timing Anomaly).

Cache Analysis The cache analysis [62, 95, 63, 31] forms an abstract model of the cache and the cache content. It aims at a classification of memory accesses into hits or misses. As the cache is of utter importance for a system's performance, a precise cache analysis is required for tight timing bounds. We detail an LRU cache analysis in the next section.

Both analyses in combination are used to derive bounds on the execution times of basic blocks.

Further components of a static timing analysis

Besides the microarchitectural-analysis and the cache-analysis, a typical timing analysis framework contains the following components:

- *Control Flow Reconstruction* that reconstructs the control-flow graph (CFG) of the analyzed program from the executable [44, 88][1].
- *Value Analysis* that computes effective addresses of memory accesses [22] and bounds on the number of loop iterations or recursions [58].
- *Path Analysis* that combines execution time bounds on the basic blocks with the loop bounds to compute the longest path within the CFG (typical using integer linear programming [91, 89]).

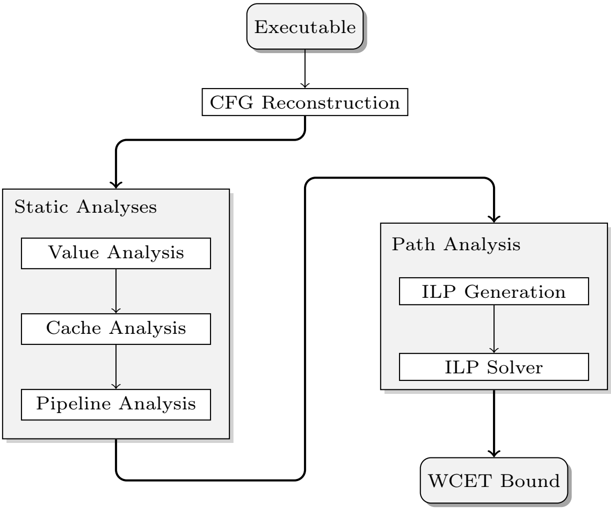


Figure 3.13.: Structure of a timing analysis

Figure 3.13 depicts the tool-chain of a typical timing analysis [29, 30], such as the **aiT timing analyzer**[2], a static timing analysis used in the automotive and aeronautics industry. For further details on this approach to static timing analysis see [35] and for an overview of currently available

[1] A precise static timing analysis must resort to the level of the linked executable as only this level contains all timing relevant properties.

[2] `http://www.absint.com/`

timing analyses, see [39, 97]. Note that bounding the execution time of a task is infeasible in general as it includes solving the halting problem. Hence, static timing analysis is only applicable to a subset of all programs. Fortunately, hard real-time tasks are often designed with analysability in mind avoiding certain programing features such as function pointer, unbounded recursion, and dynamic memory allocation.

Timing Anomaly

A *timing anomaly* [55, 74] denotes counter-intuitive behavior of a processor architecture where a local best-case entails a global worst-case. Typical example is a cache miss leading to a globally shorter execution time than a cache hit. Figure 3.14 depicts a timing anomaly due to branch misprediction. In case of a cache hit, the result of the branch prediction is only available after the execution of the next, falsely predicted instruction has started. A roll-back of this instruction is necessary and leads to a longer execution time compared to a situation of a cache miss, where the result of the branch prediction is available before the memory access was served. The additional execution time may not be bounded by a constant but be proportional to the total execution time, a situation referred to as *domino effect.* An architecture can be classified depending on whether it exhibits timing anomalies. [98]: A *fully timing composable architecture* exhibits no timing anomaly, a *compositional architecture with constant-bounded effect* exhibits timing anomalies but no domino effects and a *non-compositional architecture* exhibits timing anomalies and domino effects. **ARM7**, **TriCore** and **PowerPC** are claimed to be representatives of these classes of architectures. Timing anomalies have been proven to exist for **TriCore** and **PowerPC**, the latter also with domino effects [12]. A formal proof of the classification of **TriCore** and **ARM7**, however, is yet to be found.

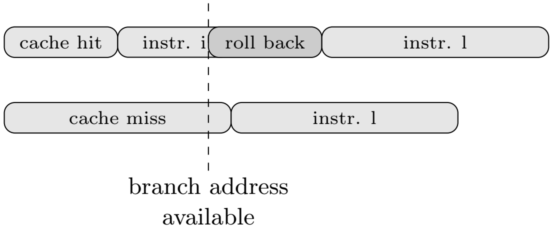


Figure 3.14.: Example of a timing anomaly

3.3.2. Cache Analysis

The cache analysis is a main component of the timing analysis. We present Ferdinand's LRU cache analysis [31] based on *abstract interpretation*. For the sake of simplicity, we assume a fully-associative cache. An analysis for a set-associative cache is then given by S parallel analyses for S fully-associative caches. Note that most cache analyses [31, 47, 91, 62, 95, 80] assume LRU replacement. Only recently, cache analyses for FIFO and PLRU have been published [32, 33].

Before we start with the domain and the abstraction of the analysis, we need to discuss how the cache analysis is used within the timing analysis and what its requirements are.

Classification of Memory Accesses

To determine the execution time of basic blocks, timing analysis needs to classify memory accesses into cache hits or cache misses. As timing analysis is required to deliver runtime guarantees for all possible executions, it can assume a cache hit (miss) for a memory access, only if this memory access always results in a hit (miss). The more memory references classified as always hits, the lower the upper bound WCET^B and the more accesses classified as always misses, the higher the lower bound BCET^B. Such a classification can not be complete; for one thing, some accesses inflict a cache miss in one execution trace and a hit in another trace, and for another thing, cache analysis relies on abstraction as deriving all possible concrete cache states is computationally infeasible. We thus end up with the classification described in Table 3.5 and a classification function with the following signature:

$$\text{Classify} : M \times V \to \{ah, am, nc\} \tag{3.34}$$

Table 3.5.: Memory access classification

always hit (ah)	memory access always results in a cache hit
always miss (am)	memory access always results in a cache miss
not classified (nc)	no classification to always hit or always miss

Effective Memory Addresses

Most cache analyses rely on exact knowledge about the referenced memory blocks; in order to decide if a memory block is cached or not, one usually has to know the memory block. Hence, a cache analysis for instruction caches is thus often easier to implement. The *effective memory address* solely depends on the instruction and is static during program execution. A prior address analysis to determine the effective address of a memory reference is not required.

Concrete Cache Semantics

Recall the definition of an LRU cache state described in Section 3.2: an LRU cache state $\zeta \in Z$ is an ordered list of K elements $[x_1, x_2, \ldots, x_K]$ with the most-recently used (MRU) element x_1 at the first position.

$$\zeta : \underbrace{M_\bot \times M_\bot \times \ldots \times M_\bot}_{K} \tag{3.35}$$

The cache update on an access to M was defined as follows:

$$\mathbb{U}^{\text{LRU}} : M \times Z^{\text{LRU}} \to Z^{\text{LRU}}$$

$$\begin{aligned} &\mathbb{U}^{\text{LRU}}(m, [x_1, \ldots, x_K]) := \\ &\qquad \begin{cases} [m, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_K] & m = x_i \\ [m, x_1, \ldots, x_{K-1}] & \textit{otherwise} \end{cases} \end{aligned} \tag{3.36}$$

The newly accessed element is put to the first position in the cache (no matter if hit or miss). All younger elements are moved to the right, i.e., aged by one, possibly evicting the least-recently used element. All older elements retain their position. We also refer to the position of an element in the cache as the *age* of an element:

$$\text{age} : \underbrace{M \to \{0, \ldots, K-1, \infty\}}_{\text{Age}} \tag{3.37}$$

where ∞ indicates that an element is not cached. We define a helper function to extract the ages from a given cache state:

$$\widehat{\text{age}} : Z^{\text{LRU}} \to \text{Age}$$

with

$$\widehat{\text{age}}(\zeta) = \lambda m. \begin{cases} i & \text{if } m = x_i \\ \infty & \textit{otherwise} \end{cases} \tag{3.38}$$

We can define the LRU update function as a function on program points $p \in V$:

$$tf_p^{\text{LRU}} : V \to (Z^{\text{LRU}} \to Z^{\text{LRU}})$$

$$tf_p^{\text{LRU}}(p) = \lambda\zeta.\mathbb{U}^{\text{LRU}}(\sharp(p), \zeta) \tag{3.39}$$

(where $\sharp(p)$ extracts the accessed memory block at program point p) and lift this function to complete paths $\pi \in \Pi$:

$$tf^{\text{LRU}} : \Pi \to (Z^{\text{LRU}} \to Z^{\text{LRU}})$$

$$tf^{\text{LRU}}(\pi) = \lambda\zeta. \begin{cases} \zeta & \text{if } \pi = \epsilon \\ tf^{\text{LRU}}([p_2 \ldots, p_n])(tf_p^{\text{LRU}}(p_1)(\zeta)) & \text{if } \pi = [p_1, \ldots, p_n] \end{cases} \tag{3.40}$$

The *collecting cache semantics* are defined as follows:

$$\text{Coll}^{\text{LRU}} : V \to 2^Z$$

$$\text{Coll}^{\text{LRU}}(p_n) = \bigcup_{s \in \text{ Init}} \left(\bigcup \{ [\pi]_{tf^{\text{LRU}}}(s) | \forall \text{ paths } \pi \text{ from } p_s \text{ to } p_n \} \right) \tag{3.41}$$

where $\text{Init} \subset Z^{\text{LRU}}$ is the set of possible initial cache states. Equation (3.42) provides the classification based on the collecting cache semantics.

$$\text{Classify}(m, p) := \begin{cases} ah & \forall\zeta \in \text{Coll}^{\text{LRU}}(p) : \exists i : m = x_i \\ am & \forall\zeta \in \text{Coll}^{\text{LRU}}(p) : \nexists i : m = x_i \\ nc & otherwise \end{cases} \tag{3.42}$$

A memory block m is classified as always hit at program point p, if m is cached in all cache states of the collecting cache semantics at p. It is classified as always miss, if there is no cache state that contains m. The collecting cache semantics delivers the most precise classification. Computing the *collecting semantics*, however, is computationally infeasible for realistically sized programs.

Abstract Semantics

The abstract LRU cache domain uses bounds on the age of memory blocks instead of concrete ages. The abstract cache analysis consists of two separate analyses, a *may-cache* analysis and a *must-cache* analysis. The may-cache contains all memory blocks that *may be cached* at p and the must-cache contains all memory blocks that *are definitely cached* at p whenever execution reaches program point p. Hence, the may-cache is used to predict cache misses by guaranteeing the absence of memory blocks while the must-cache is used to predict cache hits by guaranteeing the presence of memory blocks in the cache.

May-Cache To bound the set of possibly cached elements, the may-cache keeps for each memory block an *upper bound*

$$\mathrm{age}^{\mathrm{may}} : \mathrm{Age}$$

on its concrete age. An abstract may-cache $\mathrm{age}^{\mathrm{may}}$ represents the set of concrete cache states where the concrete age of each element m is at *least* $\mathrm{age}^{\mathrm{may}}(m)$:

$$\gamma^{\mathrm{may}}(\mathrm{age}^{\mathrm{may}}) = \{\zeta \mid \forall m \in M : \widehat{\mathrm{age}}(\zeta)(m) \geq \mathrm{age}^{\mathrm{may}}(m)\} \tag{3.43}$$

with the corresponding abstraction function α:

$$\alpha^{\mathrm{may}}(C) = \lambda m. \min_{\zeta \in C}(\widehat{\mathrm{age}}(\zeta)(m)) \tag{3.44}$$

On control flow joins, i.e., when two abstract cache states are combined, the may-cache assumes the youngest age of an element:

$$\bigsqcup : \mathrm{Age} \times \mathrm{Age} \rightarrow \mathrm{Age}$$

$$\mathrm{age}_1 \bigsqcup \mathrm{age}_2 = \lambda m. \min(\mathrm{age}_1(m), \mathrm{age}_2(m)) \tag{3.45}$$

The transformer of the abstract may-cache is defined as follows:

$$tf^{\mathrm{may}} : V \rightarrow \mathrm{Age} \rightarrow \mathrm{Age}$$

$$tf^{\mathrm{may}}(p)(\mathrm{age}) := \lambda m. \begin{cases} 0 & m = \sharp(p) \\ \mathrm{age}(m) & \mathrm{age}(m) > \mathrm{age}(\sharp(p)) \\ \mathrm{age}(m) + 1 & \mathrm{age}(m) \leq \mathrm{age}(\sharp(p)) \wedge \mathrm{age}(m) < K - 1 \\ \infty & \textit{otherwise} \end{cases} \tag{3.46}$$

where $\sharp(p)$ is the accessed memory block at program point p. The newly accessed element is given age bound 0 as it is now the MRU element (first case). All older elements retain their age bound (second case) and all younger elements age by one (third case). The interesting case is the treatment of elements with the same age bound as the accessed one, i.e., $\mathrm{age}(\sharp(p)) = \mathrm{age}(m)$. In a may-cache, these elements age by one: In any concrete cache state, only one element is cached at position $age(\sharp(p))$. If this element was $\sharp(p)$, all other elements are at a later position and will be after the access to $\sharp(p)$. If any other element m was cached at *age*$(\sharp(p))$, we can be sure that $\sharp(p)$ was older and so m ages by one on the access to $\sharp(p)$. Note that the analysis only needs to keep track of elements with finite age (fourth case).

We say that control flow information age_1 is more precise than age_2, if all age bounds in age_1 are at least the age bounds in age_2.

$$\mathrm{age}_1 \sqsubseteq^{\mathrm{may}} \mathrm{age}_2 \Leftrightarrow \forall m \in M : \mathrm{age}_1(m) \geq \mathrm{age}_2(m) \tag{3.47}$$

Must-Cache To derive a set of definitely cached elements, the must-cache keeps for each memory block a *lower bound*

$$\mathrm{age}^{\mathrm{must}} : \mathrm{Age}$$

on its concrete age. Conversely to the may-cache analysis, an abstract must-cache state $\mathrm{age}^{\mathrm{must}}$ represents the set of concrete cache states where the concrete age of each element m is at *most* $\mathrm{age}^{\mathrm{must}}(m)$:

$$\gamma(\mathrm{age}^{\mathrm{must}}) = \{\zeta \mid \forall m \in M : \widehat{\mathrm{age}}(\zeta)(m) \leq \mathrm{age}^{\mathrm{must}}(m)\} \tag{3.48}$$

with the corresponding abstraction function α:

$$\alpha(C) = \lambda m. \max_{\zeta \in C}(\widehat{\mathrm{age}}(\zeta)(m)) \tag{3.49}$$

On control flow joins, the must-cache conservatively keeps the oldest age of an element:

$$\bigsqcup : \mathrm{Age} \times \mathrm{Age} \to \mathrm{Age}$$

$$\mathrm{age}_1 \bigsqcup \mathrm{age}_2 = \lambda m. \max(\mathrm{age}_1(m), \mathrm{age}_2(m)) \tag{3.50}$$

The transformer of the abstract must-cache is defined as follows:

$$tf^{\mathrm{must}} : V \to \mathrm{Age} \to \mathrm{Age}$$

$$tf^{\mathrm{must}}(p)(\mathrm{age}) :=$$
$$\lambda m. \begin{cases} 0 & m = \sharp(p) \\ \mathrm{age}(m) & \mathrm{age}(m) \geq \mathrm{age}(\sharp(p)) \\ \mathrm{age}(m) + 1 & \mathrm{age}(m) < \mathrm{age}(\sharp(p)) \wedge \mathrm{age}(m) < K - 1 \\ \infty & \textit{otherwise} \end{cases} \tag{3.51}$$

The only difference to the may-cache transfer function (Equation (3.46)) is the treatment of elements that have the same age bound as the accessed element. As the must-cache analysis computes upper bounds on the ages, the accessed element $\sharp(p)$ was either younger than any element m with the same age bound and hence, accessing $\sharp(p)$ has not changed the position of m, or m was younger and ages by one on the access to $\sharp(p)$. In both cases, the age bound of m does not need to be increased.

We say that control flow information age_1 is more precise than age_2, if all age bounds in age_1 are at most the age bounds in age_2.

$$\text{age}_1 \sqsubseteq^{\text{must}} \text{age}_2 \Leftrightarrow \forall m \in M : \text{age}_1(m) \leq \text{age}_2(m) \tag{3.52}$$

Initial Cache States: If the cache can be assumed to be empty prior to program execution (either at system start-up or after a cache flush), the initial cache state for both analyses is given by:

$$\text{age}_{\text{Init}}^{\text{may/must}} = \lambda m.\infty \tag{3.53}$$

Otherwise, the initial cache is given by the abstraction of the set of initial concrete states Init.

$$\text{age}_{\text{Init}}^{\text{may/must}} = \alpha^{\text{may/must}}(\text{Init}) \tag{3.54}$$

If the set of concrete initial cache states is not restricted in any way, the initial may-cache state assumes that all elements are cached

$$\text{age}_{\text{Init}}^{\text{may}} = \lambda m.0 \tag{3.55}$$

and the initial must-cache state can not guarantee any element to be cached.

$$\text{age}_{\text{Init}}^{\text{must}} = \lambda m.\infty \tag{3.56}$$

Classification based on the abstract cache analysis Given the upper and lower bounds on the ages of the memory blocks, the memory access classification is given as follows:

$$\text{Classify}(m, p) := \begin{cases} ah & \text{age}_p^{\text{must}}(m) \leq K - 1 \\ am & \text{age}_p^{\text{may}}(m) > K - 1 \\ am & \textit{otherwise} \end{cases} \tag{3.57}$$

where $\text{age}_p^{\text{may/must}}(m)$ is the lower/upper age bound of m at p derived by the must/may-cache analysis.

Let $ET^{\text{Cl}}(\pi, \zeta)$ be the execution time of the path π on initial state ζ assuming a cache hit at each memory reference classified as always hit and assuming a cache miss at each memory reference classified as always miss. As static timing analysis relies on this abstract cache model, the following holds:

$$\forall \zeta \in Z, \pi \in \Pi : \text{BCET}^B \leq ET^{\text{Cl}}(\pi, \zeta) \leq \text{WCET}^B \tag{3.58}$$

Example

Consider for example the following program (Collatz-conjecture test for value n):

```
    do {
a:    if (n%2) {
b:       n = n +1;
c:       n=n/2
      } else {
d:       n=n/2
      }
e:  } while (n>1)
```

We assume a 4-way fully associative LRU cache and that each instruction (a to e) is stored in its own memory block. An abstract cache state is represented by 4 sets: $[\{b\}, \{a\}, \emptyset, \emptyset]$ indicates that b is assigned abstract age 0, a abstract age 1 and all other memory blocks are assigned age ∞. The control-flow graph of the example program is depicted in Figure 3.15, with the result of the must-cache analysis associated to each program point. The upper value denotes the must-cache information before, the lower value after execution of the program point.

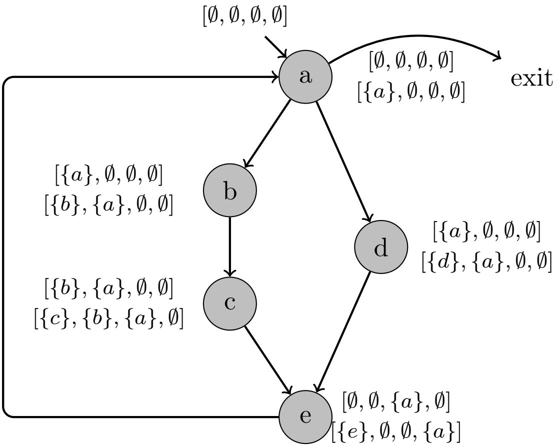

$[\emptyset, \emptyset, \emptyset, \emptyset]$
a
$[\emptyset, \emptyset, \emptyset, \emptyset]$
$[\{a\}, \emptyset, \emptyset, \emptyset]$
exit
$[\{a\}, \emptyset, \emptyset, \emptyset]$
$[\{b\}, \{a\}, \emptyset, \emptyset]$
b
d
$[\{a\}, \emptyset, \emptyset, \emptyset]$
$[\{d\}, \{a\}, \emptyset, \emptyset]$
$[\{b\}, \{a\}, \emptyset, \emptyset]$
$[\{c\}, \{b\}, \{a\}, \emptyset]$
c
e
$[\emptyset, \emptyset, \{a\}, \emptyset]$
$[\{e\}, \emptyset, \emptyset, \{a\}]$

Figure 3.15.: Example of the must-cache analysis: control flow graph of the example program and associated must-cache information

Virtual Unrolling

The must-cache analysis is not able to predict any cache hits in this example, which is correct as we have to conservatively assume an empty initial cache. Although, after one iteration of the loop, a will be cached and will remain so for the rest of the execution. The same holds for e. However, the analysis is not able to distinguish between the first loop iteration (cold misses) and the nth iteration where only conflict misses occur. To remedy this problem and to increase the precision of the analysis, *virtual loop unrolling* [57] can be applied. Virtual loop unrolling artificially increases the control flow graph of the program in order to separate the first i iterations of a loop from the remaining $i+x$ iterations. Figure 3.16 depicts the control flow graph of the example program after applying virtual loop unrolling with depth one. The second accesses are now correctly predicted as cache hits.

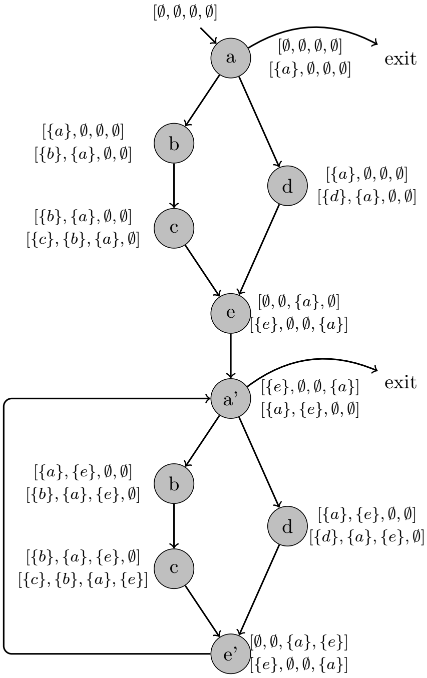

$[\emptyset,\emptyset,\emptyset,\emptyset]$
a
$[\emptyset,\emptyset,\emptyset,\emptyset]$
$[\{a\},\emptyset,\emptyset,\emptyset]$
exit
$[\{a\},\emptyset,\emptyset,\emptyset]$
$[\{b\},\{a\},\emptyset,\emptyset]$
b
d
$[\{a\},\emptyset,\emptyset,\emptyset]$
$[\{d\},\{a\},\emptyset,\emptyset]$
$[\{b\},\{a\},\emptyset,\emptyset]$
$[\{c\},\{b\},\{a\},\emptyset]$
c
e
$[\emptyset,\emptyset,\{a\},\emptyset]$
$[\{e\},\emptyset,\emptyset,\{a\}]$
a’
$[\{e\},\emptyset,\emptyset,\{a\}]$
$[\{a\},\{e\},\emptyset,\emptyset]$
exit
$[\{a\},\{e\},\emptyset,\emptyset]$
$[\{b\},\{a\},\{e\},\emptyset]$
b
d
$[\{a\},\{e\},\emptyset,\emptyset]$
$[\{d\},\{a\},\{e\},\emptyset]$
$[\{b\},\{a\},\{e\},\emptyset]$
$[\{c\},\{b\},\{a\},\{e\}]$
c
e’
$[\emptyset,\emptyset,\{a\},\{e\}]$
$[\{e\},\emptyset,\emptyset,\{a\}]$

Figure 3.16.: Example of virtual loop unrolling: must-cache information after virtual loop unrolling.

Chapter 4

Context-Switch Costs

Mathematics allows for no hypocrisy
and no vagueness.
Stendhal (1783 - 1843)

In preemptive scheduling, a *context switch* occurs when a high priority task preempts a low priority one. In such a case, the execution time of the preempted task is increased. This increase is referred to as *context switch costs*.

This chapter provides a formal definition of the *context switch costs*, the *cache-related preemption delay* and its relation to the total execution time of a task under preemption. Furthermore, we present the requirements and limitations of a separate computation of the context switch costs. Note that this chapter is partially published in [7] and [17].

4.1. The Impact of a Context Switch

Context switch costs denote the additional execution time of a task due to the effects of preemption. First approaches in the area of schedulability analysis for hard real-time systems assumed they are negligible or subsumed by the execution time bound:

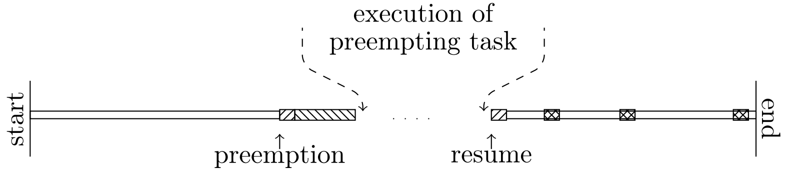
> *The runtime [...] can be interpreted as the maximum processing time for a task. In this way [...] the cost of preemptions can be taken into account.*
>
> Liu & Layland [53]

This assumption simplified substantially the analysis and the design of preemptive systems, but is invalid for modern systems employing caches. Context switches can have a major impact on the performance of preemptively scheduled tasks.

In case of a context switch, three main factors increase the execution time (see Figure 4.1):

- the pipeline has to be flushed at preemption and refilled afterwards,
- the scheduler is invoked and decides which task to execute next, and
- preempting tasks may evict cache entries of the preempted one, which have to be reloaded later on.

Note that the context switch costs do not include the execution time(s) of the preempting task(s).



▨ pipeline flush/pipeline refill ▧ invocation of the scheduler
⊠ additional cache-miss penalty

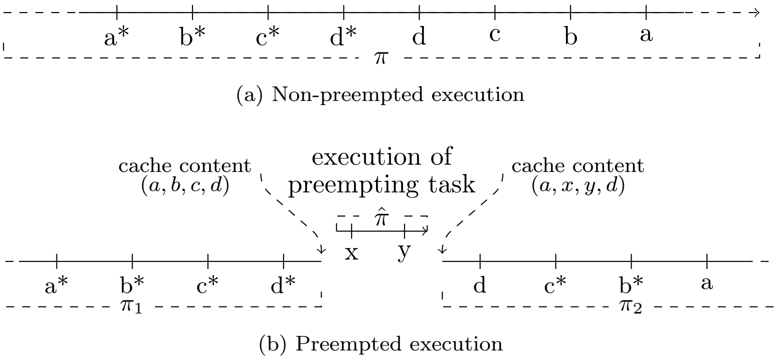
Figure 4.1.: Schematic view of a context switch

The first two sources (pipeline/scheduler) are considered constant (at least for some processors, see Section 3.3) and occur along with preemption. The cost for the scheduler invocation can be seen as part of the preempting task as it occurs even in case of the non-preemptive execution.

The effect of cache eviction, however, is postponed until a later point in time. Thus, correlations between a cache miss and a prior preemption are hard to identify. The additional execution time caused by additional cache misses due to preemption is usually referred to as *cache-related preemption delay* (CRPD).

4.2. Cache-related Preemption Delay

The impact of a preemption on the cache is explained in Figure 4.2. Figure 4.2a shows a non-preempted sequence of memory accesses given a direct-mapped cache with four cache sets. In the middle of the sequence π (between the two accesses to d), the cache contains all memory blocks a to d. Hence, the last four accesses will result in cache hits. Except for the compulsory misses when data is accessed for the first time, no conflict misses occur. Figure 4.2b shows the same sequence preempted in the middle. Due to the cache disturbance caused by the preempting task, e.g., accesses to x and y, two cache blocks have been evicted and need to be reloaded when accessed next. Two more misses occur compared to the non-preempted execution. The example also shows that the cache-related preemption delay occurs at a later point in the task execution.



(a) Non-preempted execution

(b) Preempted execution

Figure 4.2.: Non-preempted (a) versus preempted (b) execution trace assuming a direct-mapped cache of size 4. Letters a to d denote memory blocks, a star ($*$) marks a cache miss.

4.2.1. Early Work on CRPD

In 1994, as caches became common in real-time systems, Basumallick and Nilsen [11] identified the need to consider and estimate the *extrinsic cache behavior*, as they name the context switch costs. Busquets-Mataix et al. [18] proposed the first schedulability analysis (that we detail in Chapter 8) including context switch costs. Although they identify different possibilities to precisely bound the context switch costs, they focus on the scheduling analysis only and use a very pessimistic bound: the delay to refill the entire cache. The first precise bound on the context switch costs based on a static analysis was proposed by Lee et al. [49]. They introduced the concept of *useful cache blocks* and coined the term *cache-related preemption delay.* A static analysis of the preempting task was presented by Tomiyama and Dutt [90], a combined analysis of preempted and preempting task by Tan and Mooney [86].

These papers can be regarded as the milestones or the initial work in this research area. We will detail the different approaches to bound the cache-related preemption delay in Chapter 5 and the CRPD aware response time analysis in the first section of Chapter 8.

4.2.2. Formal Definitions

This section provides formal definitions of *preemption*, *cache-related preemption delay* and *execution time under preemption* that we later on need to prove the correctness of the different analyses presented in this thesis.

Let π be the execution trace of the preempted task τ_2, the sub-trace before preemption point p is denoted by π_1 and the sub-trace after p by π_2 ($\pi = \pi_1 \cdot \pi_2$). Furthermore, let ζ be the cache state of task τ_2 at point p before preemption and ζ' after preemption, i.e., the resulting cache state after the preempting execution trace $\hat{\pi}$ of τ_1.

Cache-related preemption delay is given by the number of additional misses of the execution of trace π_2 on cache state ζ times *cache-miss penalty* or *block-reload time* CRT (compared to the execution of trace π_2 in cache state ζ').

$$CRPD_P = (miss(\pi_2, \zeta') - miss(\pi_2, \zeta)) \cdot \mathrm{CRT} \tag{4.1}$$

where $miss(\pi, \zeta)$ gives the number of misses of path π on initial cache state ζ.

In the following, we provide definitions for preemptions and cache-related preemption delay.

Definition 4.1 (Preemption)
A preemption *pr is a pair consisting of a preemption point $p \in V$ and preempting access sequence $\hat{\pi}$:*

$$pr : V \times 2^{\pi}$$

We define a preemption point to denote always the last instruction of the preempted task that is still executed prior to task suspension. Thus, the preemption at point p occurs in fact directly after the execution of program point p.

A preempted execution of task τ is a triple consisting of an execution trace π of τ, a set of preemptions $PR \subseteq 2^{Pr}$ and an initial cache state ζ. The set of n preemptions $PR = \{(q_1, \hat{\pi}_1), (q_2, \hat{\pi}_2), \ldots, (q_n, \hat{\pi}_n)\}$ divides the execution trace $\pi = [p_s, \ldots, p_e]$ into a sequence of $n+1$ sub-traces π_0 to π_n:

$$\forall_{i=0}^{n} : \pi_i = [q_i, \ldots, q_{i+1}] \tag{4.2}$$

with $q_0 = p_s$ and $q_{n+1} = p_e$.

We are now interested in the different cache states each sub-trace executes on (for both cases, preempted (P) and non-preempted (NP) execution). The first sub-trace always executes on the initial cache state ζ_{init}. The resulting cache state $(tf(\pi_0)(\zeta_{init}))$ determines the initial cache state for the next sub-trace. We thus end up with a recursive definition:

$$\begin{aligned} \zeta_0^{NP} &= \zeta_{init} \\ \zeta_i^{NP} &= tf(\pi_i)(\zeta_{i-1}^{NP}) \end{aligned} \tag{4.3}$$

The first sub-trace executes on the initial cache state ζ also in case of preemptive execution. As there have not been any preemptions yet, there are no differences. The difference comes with the next step, where the initial cache state for the next sub-trace depends not only on the result of the prior sub-trace but also on the preemption trace $\hat{\pi}$

$$\begin{aligned} \zeta_0^{P} &= \zeta_{init} \\ \zeta_i^{P} &= tf(\hat{\pi}_{q_i})(tf(\pi_{i-1})(\zeta_{i-1}^{P})) \end{aligned} \tag{4.4}$$

Note that cache state transformer $tf : \Pi \to (Z \to Z)$ was defined in Chapter 3.3.2.

Definition 4.2 (Cache-related Preemption Delay)
Cache-related preemption delay of execution trace $\pi \in 2^{\pi}$ on initial cache state ζ preempted by $PR \in 2^{Pr}$ is given by:

$$CRPD^T(\pi, \zeta, PR) = \left(\sum_{i=0}^{n} (miss(\pi_i, \zeta_i^P) - miss(\pi_i, \zeta_i^{NP})) \right) \cdot CRT$$

In prior work, CRPD often refers to the delay for one preemption only. Such a definition is inherently imprecise as preemptions may *interact*, i.e., the costs of one preemption may depend on prior preemptions. We use the superscript T to denote the total CRPD and to clearly state the difference to the CRPD for a single preemption.

Remember that $ET(\pi, \zeta, PR)$ denotes the execution time of trace π on initial cache state ζ assuming non-preemptive execution. We now introduce $ET^P(\pi, \zeta, PR)$ to denote the execution time with preemption by PR.

Equation (4.5) presents the actual use of the cache-related preemption delay.

$$ET^P(\pi, \zeta, PR) \leq ET(\pi, \zeta) + \mathrm{CRPD}^T(\pi, \zeta, PR) \tag{4.5}$$

The total execution time under preemption is determined by the time of non-preempted execution plus the additional delay due to preemption. However, the actual execution time under preemption may be *less than* the value given by Equation (4.5). The delay of cache-reloads may overlap with other delays and thus, does *not* or does *not fully* contribute to the total execution time. In case of a pipelined processor for instance, a reload may happen in parallel to a time-intensive floating point operation without further increasing the total execution time.

Note that the constant cost for a single preemption as caused by the scheduler or the pipeline (see Section 4.1) is attributed and thus added to the execution of the preempting task. This simplifies the subsequent schedulability analysis. Note furthermore that a separate computation of the cache-related preemption delay entails a strong restriction on the processor, namely the restriction to *timing composable architectures* or *compositional architectures with constant-bounded effect*. For complex architectures exhibiting timing anomalies and domino effects, Equation (4.5) may be optimistic and thus lead to unsound results.

4.3. Limitations of the CRPD Approach

A separate computation of the CRPD is restricted to *fully timing composable architecture* or *compositional architectures with constant-bounded effects*. Only for such architectures, we can assign each additional cache miss a corresponding timing penalty, denoted by CRT in Equation (4.5). The classification of architectures [98] depends on the complexity of the processor components and on the cache.

4.3.1. Classification of Architectures

The occurrence of timing anomalies and domino effects determines the classification of the architectures. According to Wilhelm et al. [98] there are three types of architectures (see Section 3.3):

Fully timing composable architectures
No timing anomalies may occur. The cache-miss penalty CRT for additional misses due to preemption is solely determined by the cache latency and thus, given by the time to reload a cache block.

Compositional architectures with constant-bounded effect
Timing anomalies but no domino effects can occur. The penalty CRT contains at least the block-reload time but must also account for any additional timing penalty a cache miss may inflict.

Non-compositional architectures
As domino effects may occur, no constant bound for an additional cache miss exist. A separate computation of the CRPD is not applicable [76].

Besides the need to determine the cache-miss penalty for preemption misses, we also need to be able to bound the total number of additional misses. This mostly depends on the cache and the cache replacement policies. Note that the domino effects are mostly caused by cache-replacement policies.

4.3.2. CRPD and Cache Replacement Policies

First CRPD analyses either assumed direct-mapped caches or LRU replacement policy. Computation for other replacement schemes have been ignored [49] or declared to be trivial [79].

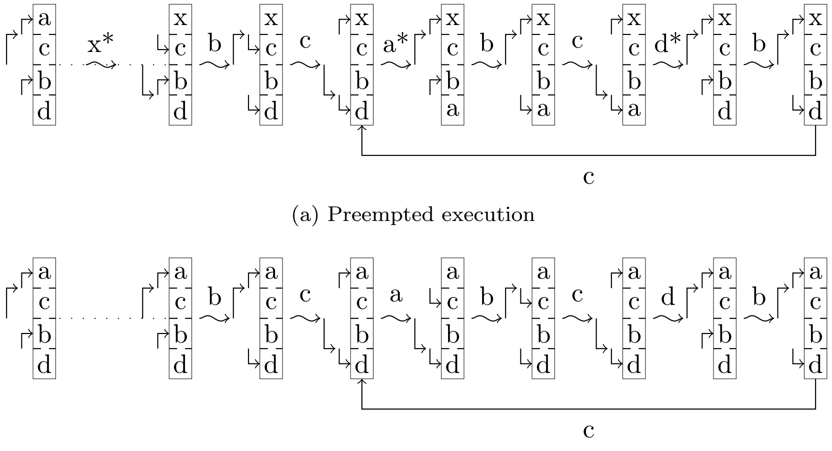
Observation 4.1
In case of FIFO and PLRU, the number of additional misses due to a single preemption is not bounded by the associativity of the cache but may be proportional to the total execution time.

Observation 4.1 inhibits a separate computation of the cache-related preemption delay. The methods considered and published so far always assume that each preemption has a bounded impact only—a property that only holds for LRU caches where a bound on a single preemption is always given by the associativity of the cache. In general, the *miss-sensitivity* [72, 73] of the replacement policy determines if a bound exists. If a replacement policy has a *miss-sensitivity* greater than one, no fixed CRPD bound exists (see Section 3.2).

In the following, we will justify Observation 4.1.

Pseudo Least Recently Used (PLRU)

We start with pseudo least-recently used policy that mimics LRU with lower implementation overhead. A pointer tree is used to indicate the element evicted next. See Section 3.2 for a detailed description. The corresponding access sequence for PLRU caches that causes an arbitrarily high CRPD is described by the regular expression



(a) Preempted execution

(b) Non-preempted execution

Figure 4.3.: Access sequences with unbounded preemption delay for PLRU. A $*$ marks a cache miss. Sub-figure (a) shows the evolution on the cache in case of preemption with access to x and (b) without preemption.

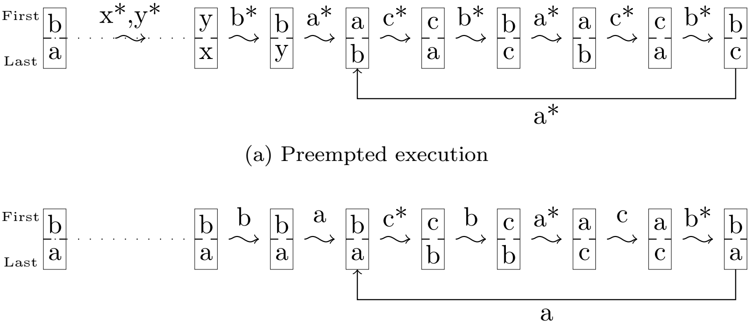
$$(bcabcd)^* \tag{4.6}$$

As the sequence contains only elements already cached prior to preemption, no cache misses occur in case of non-preempted execution. During preemption, however, the element x replaces a. Since the sequence *rejuvenates* the element x such that it remains cached, the other four elements a, b, c, d have to compete for the remaining three places. This leads to an arbitrary number of misses.

First-In First-Out (FIFO)

Similar to PLRU, we can easily construct an access sequence and a preemption scenario that leads to an arbitrarily high CRPD in case of

FIFO replacement policy (see Figure 4.4). The corresponding access sequences for FIFO caches that causes an arbitrarily high CRPD is described by the regular expression



(a) Preempted execution

(b) Non-preempted execution

Figure 4.4.: Access sequence with unbounded preemption delay (FIFO). A $*$ marks a cache miss. Sub-figure (a) shows the evolution on the cache in case of preemption with accesses to x and y and (b) without preemption.

$$ba(cba)^* \tag{4.7}$$

After the access to a and b, the same elements are cached in both cases, but the order differs. This little change on the order of cached elements will never even out and eventually cause an arbitrarily high number of additional misses due to preemption. Note that the access sequence also leads to misses in the non-preemptive case. However, the number of misses per iteration differs (3 in case of non-preempted execution versus 5 in case of preempted execution).

Least Recently Used (LRU)

Each of the access sequences with unbounded preemption delay in case of PLRU, FIFO and MRU have a bounded delay in case of LRU. Two different LRU states converge after at most K distinct accesses, where K is the cache associativity. The LRU replacement policy does not distinguish between a cache miss and a cache hit in the sense that the least recently accessed memory block is always moved to the (logical) first position, all younger elements are aged by one. In addition, the age of an element can only decrease if this element was accessed—in all other cases, the age of a memory block is monotonically increasing. Thus, two different cache states converge after at most K different accesses, and so, at most K additional misses due to preemption can occur. For further information on cache-replacement policies, see [72]

4.4. Other Approaches to the Analysis of Preemptive Systems

Besides a separate computation of the CRPD, other approaches exist to handle preemption overhead due to caches. These approaches can be divided into the following groups:

(i) avoid inter-task cache interference by cache partitioning.

(ii) derive timing bounds including the context switch costs.

(iii) adapt the scheduling policy to reduce number of preemptions.

(iv) replace caches by more *predictable* hardware.

The following sections details these different approaches.

Cache Partitioning/Cache Locking

Cache partitioning [61, 99, 69, 48] and *cache locking* [70, 92, 93] are techniques to completely avoid cache-related preemption delay. In the first case, each task is assigned a dedicated part of the cache to guarantee that a preempting task can not evict cache blocks of another one. Cache partitioning is implemented either in hardware by means of a memory management unit (which is not common in hard-real time systems) or in software with the help of adapted compilers, often resulting in substantial code-changes required to ensure that the task only accesses its dedicated cache-partition [61]. In the second case, cache locking [92], cache lines are locked such that a preempting task can not evict locked data. In static cache locking, data is loaded to cache and locked for the entire task execution while dynamic cache locking only locks data for some predefined code regions. Both techniques trade *inter-task* with *intra-task* cache conflicts, hence paying for the reduced cache-related preemption delay by a possible increase of the worst-case execution time of some tasks. To our best knowledge, cache partitioning, resp. cache locking techniques have not yet been compared to a precise CRPD analysis.

Instead, techniques have been compared against each other [70, 54] and against uncached systems [92, 93]. Note that cache partitioning and cache locking only avoid cache-related preemption delay. Timing anomalies and domino effects may also be caused by other history sensitive processor features such as branch-target buffers. Hence, both techniques are also not applicable in case of *non-compositional architectures*.

Preemption Cost as Part of the Execution Time Bound

Liu and Layland's [53] assumption of negligible context switch costs can be restored by subsuming these costs in the execution time bound. An analysis to compute such an extended timing bound has been proposed by Schneider [76]. The analyses relies on the pessimistic assumption that a preemption occurs at each program point, evicting each *useful cache block* (the last assumption was weakened in [5] considering the specific memory mapping). The execution time bound is valid for each preemption scenario, but overly pessimistic. However, this is the only approach to the analysis of preemptive systems in case of *non-compositional architectures*.

Reducing Number of Preemptions

Obviously, non-preemptive scheduling does not suffer from additional preemption costs. As some task sets, however, are only schedulable preemptively, completely avoiding preemption is not an alternative. Instead, the overall preemption costs can be reduced by reducing the number of preemptions. Scheduling policies aiming at such a reduction employ adapted scheduling policies [27] or settle in between fully preemptive and non-preemptive systems. Examples of the latter class are *non-preemption groups* [24], *preemption thresholds* [45, 71, 94], *FP-FIFO scheduling* [59] and *co-operative scheduling* where preemption is restricted to a set of possible preemption points or disabled for a predefined time (also often referred to as *deferred preemption*). Recently, Bertagno [13, 14] et al. extended co-operative scheduling by an algorithm that selects optimal preemption points for co-operative scheduling. However, these approaches constitute an alternative only to the applied scheduling policy, not to the computation of CRPD bounds. As preemptions may still occur, bounds on the CRPD are still required.

Hardware Alternatives–Scratchpads

Recent research aims at the design of more predictable hardware architectures. Examples are the Pret Machine [28] and the Prompt Design

Principles [96]. The use of scratchpad memories [67] as an alternative to caches is common in many of these approaches. Initially meant to reduce power consumption of an embedded device [82], *scratchpads* are nowadays advocated as means to increase a system's predictability. Similar to caches, scratchpads are small but fast memories located close to the processor. Contrary to the dynamic behavior of caches, the decision which data to store in a scratchpad is taken statically at pre-runtime. Hence, scratchpads are not transparent to a system designer (such as caches) but map to the address space resulting in non-uniform access latencies. Scratchpads were used preferably in static non-preemptive systems. Recent implementations, however, also allow efficient use in preemptive systems [83]. Nevertheless, scratchpad memories have not yet replaced caches—and probably will not in the near future. For one thing, many systems employ caches and scratchpads side by side, and for another thing, scratchpads are only used in architectures specifically designed for embedded systems. Many system designs, however, must resort to low-cost, off-the shelf general-purpose hardware.

Chapter 5

Bounding Cache-Related Preemption delay (Related Work)

From the errors of others, a wise
man corrects his own.
Publilius Syrus (1st century BC)

This chapter provides a survey of the state of the art in the computation of the *cache-related preemption delay (CRPD)*. It also serves as starting point for the CRPD analyses presented in Chapter 6 and Chapter 7 as well as for the CRPD-aware response time analyses in Chapter 8. Note that this chapter is published in [7]

5.1. Useful Cache Blocks; Lee's Original Approach

In 1996, Lee et al. [49] introduced the concept of *useful cache blocks* to bound the CRPD: in case of preemption at program point p, a memory block m that may be cached at p and may still be cached at its next reuse may cause an additional cache miss.

Definition 5.1 (Useful Cache Block (UCB))
A memory block m is called useful at program point p, if

a) m may be cached at p and

b) m may be reused at program point p' that may be reached from p without eviction of m on this path.

The function

$$UCB : V \rightarrow 2^M$$

assigns each program point the corresponding set of UCBs, i.e., $UCB(p)$ is the set of useful cache blocks at program point p, and

$$UCB^s : V \rightarrow 2^M$$

picks those UCBs at a program point that map to cache set s with

$$\bigcup_s UCB^s(p) = UCB(p)$$

Note that Definition 5.1 adheres to the original definition of UCBs by Lee at al. (relying on the rather vague information of *may* be cached, *may* be reused). The refined version of useful cache blocks that we present in Chapter 6 resorts to the path semantics.

From this definition, we can directly step forward to a bound on the cache-related preemption delay as given by Equation (5.1).

$$\mathrm{CRPD}_P^{\mathrm{UCB}} : (V \rightarrow 2^M) \times V \rightarrow \mathbb{N}$$

$$\mathrm{CRPD}_P^{\mathrm{UCB}}(\mathrm{UCB}, P) = \mathrm{CRT} \cdot |\mathrm{UCB}(P)| \tag{5.1}$$

The cardinality of the set of useful cache blocks at p gives an upper bound on the number of additional cache misses due to preemption. Each of these misses incurs an additional delay of CRT cycles. A global bound on the preemption cost for preemption at any point in the program is given by the worst-case preemption point $p \in V$, i.e., the preemption point causing the highest delay:

$$\mathrm{CRPD}^{\mathrm{UCB}} : (V \rightarrow 2^M) \rightarrow \mathbb{N}$$

$$\mathrm{CRPD}^{\mathrm{UCB}}(\mathrm{UCB}) = \max_{p \in V}\{\mathrm{CRPD}_P^{\mathrm{UCB}}(\mathrm{UCB}, P)\} \tag{5.2}$$

Without any further ado, we can plug-in the bound on the CRPD to determine a bound on the execution time of a task under preemption:

$$\mathrm{WCET}^{B} + n \cdot \mathrm{CRPD}^{\mathrm{UCB}}(\mathrm{UCB}) \tag{5.3}$$

where n is the number of preemptions. The correctness of the formula relies on two assumptions. First, *timing analysis* provides a valid bound on the uninterrupted execution time of a task, here denoted by WCET^{B}, independent of the initial cache state ζ and the actual execution trace π:

$$\mathrm{WCET}^{B} \geq ET(\pi, \zeta) \tag{5.4}$$

Second, Equation (5.2) provides a bound on the CRPD for any single preemption and so, n times this value bounds the total CRPD:

$$|PR| \cdot \mathrm{CRPD}^{\mathrm{UCB}}(\mathrm{UCB}) \geq \mathrm{CRPD}^{T}(\pi, \zeta, PR) \tag{5.5}$$

5.2. Evicting Cache Blocks

Each memory block of the preempting task that is cached during the task's execution may potentially evict cache blocks of the preempted task. We call such memory blocks *evicting cache blocks.*

Definition 5.2 (Evicting Cache Blocks (ECB))
A memory block of the preempting task on path $\hat{\pi}$ is called an evicting cache block*, if it is accessed during the execution of the path $\hat{\pi}$. The function*

$$ECB^{\Pi} : \Pi \to 2^{M}$$

extracts the evicting cache blocks on a path, i.e., $ECB^{\Pi}(\hat{\pi})$ delivers the set of all memory blocks accessed on path $\hat{\pi}$. The set of all evicting cache blocks of a task τ_j is then defined as the union of the ECBs of all paths of task τ_j:

$$ECB_{\tau_j} = \bigcup_{\hat{\pi} \in \Pi_{\tau_j}} ECB^{\Pi}(\hat{\pi})$$

We denote the set of evicting cache blocks that map to cache set s by $ECB^{s} \subseteq 2^{M}$ with $\bigcup_{s} ECB^{s} = ECB.$

Although direct-mapped caches can be considered a special case of a K way set-associative cache with $K = 1$, deriving a precise bound on the cache-related preemption delay for direct-mapped caches turns out to be much easier. Tomiyama et al. [90] proposed to count the number of cache sets to which at least one evicting cache block maps.

$$\mathrm{CRPD}^{\mathrm{ECB}} : 2^M \rightarrow \mathbb{N}$$

$$\mathrm{CRPD}^{\mathrm{ECB}}(\mathrm{ECB}) = \mathrm{CRT} \cdot |\{s \mid \mathrm{ECB}^s \neq \emptyset\}| \tag{5.6}$$

Equation (5.6) is only valid for direct-mapped caches. To our best knowledge, no approach for set-associative caches solely based on the number of evicting cache blocks has been proposed.

Busquets-Mataix et al. [18] identified first the possibility to use the number of cache blocks of the evicting task to compute a bound on the CRPD, yet Tomiyama et al. [90] proposed the first analysis based on the evicting cache blocks.

5.3. Combining ECBs and UCBs

Again, we start with direct-mapped caches. Negi [64] and Tan [86] proposed to refine Equation (5.6) by counting only those cache sets that may contain a useful cache block of the preempted task τ_i (where τ_j denotes the preempting task):

$$\mathrm{CRPD}_P^{\mathrm{UCB\&ECB}} : (V \rightarrow 2^M) \times 2^M \times V \rightarrow \mathbb{N}$$

$$\begin{aligned}\mathrm{CRPD}_P^{\mathrm{UCB\&ECB}}(\mathrm{UCB}_{\tau_i}, \mathrm{ECB}_{\tau_j}) = \\ \mathrm{CRT} \cdot |\{s \mid \mathrm{ECB}^s_{\tau_j} \neq \emptyset \ \wedge \mathrm{UCB}^s_{\tau_i}(p) \neq \emptyset\}|\end{aligned} \tag{5.7}$$

$$\mathrm{CRPD}^{\mathrm{UCB\&ECB}} : (V \rightarrow 2^M) \times 2^M \rightarrow \mathbb{N}$$

$$\begin{aligned}\mathrm{CRPD}^{\mathrm{UCB\&ECB}}(\mathrm{UCB}_{\tau_i}, \mathrm{ECB}_{\tau_j}) = \\ \max_{p \in V}\{\mathrm{CRPD}_P^{\mathrm{UCB\&ECB}}(\mathrm{UCB}_{\tau_i}, \mathrm{ECB}_{\tau_j})\}\end{aligned} \tag{5.8}$$

Tan and Mooney [86] presented a generalization of Equation (5.7) for LRU caches. For each cache set s and each program point p, they compute the minimum of the UCBs at p, the ECB and the associativity (K) of the cache:

$$\mathrm{CRPD}^{\mathrm{UCB\&ECB}} : (V \rightarrow 2^M) \times 2^M \rightarrow \mathbb{N}$$

$$\mathrm{CRPD}_P^{\mathrm{UCB\&ECB}}(\mathrm{UCB}_{\tau_i}, \mathrm{ECB}_{\tau_j}) = \mathrm{CRT} \cdot \sum_s \min(|\mathrm{UCB}_{\tau_i}^s(p)|, |\mathrm{ECB}_{\tau_j}^s|, K) \quad (5.9)$$

Although Equation (5.9) seems to be logical generalization of Equation (5.7), it does not provide a valid upper bound on the cache-related preemption delay.

Observation 5.1
In case of set-associative caches, the number of ECBs is not a bound on the number of additional misses due to preemption.

The example from Figure 5.1 proves Observation 5.1 and also provides a counterexample to Equation (5.9). We assume a 4 way set-associative LRU cache. After preemption with only one ECB x, the cache contains all but one cached element of before. However, each new access will lead to a cache miss and evict the element accessed next. This domino effect leads to a total of 4 cache misses and all of them due to preemption with one ECB only.

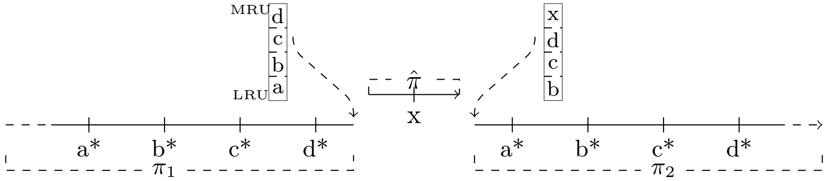


Figure 5.1.: Optimism of the naive UCB/ECB combination for LRU caches: execution trace on a 4 way LRU cache with preemption with one ECB but 4 additional cache misses.

5.4. Deriving the Set of UCBs/ECBs

UCB Analysis

To determine whether a memory block is useful at a given program point p, we need to ascertain whether there exists a path on which p is useful. To this end, we split each path through p into two parts: one that ends in p and one that starts in p.

For each memory block m, we derive for the first path (the one that ends in p), the number of accesses from the last use of m to program point p, and for the second path (the one that starts in p), the number of accesses from program point p to its next reuse. If the number of accesses to different memory blocks on the path ending in p is larger than the associativity of the cache, we can conclude that m is not cached at p. If the number of accesses to different memory blocks on the path starting in p is larger than the associativity of the cache, we can conclude that m will not survive in the cache (if it was cached at p). We denote the number of accesses on path to, resp. on path from program point p as the forward age ($\mathrm{age}^{\rightarrow}$), resp. backward age ($\mathrm{age}^{\leftarrow}$). As we are interested in an over-approximation of the set of UCBs, we keep the minimal ages at the joins. Hence, forward and backward ages are computed according to the may-cache analysis described in Chapter 3.3 (in case of the backward age, the may analysis is employed as a backwards analysis). The set of UCBs is then given as follows:

$$\mathrm{UCB}(p) \subseteq \{m \mid \mathrm{age}^{\leftarrow}(p)(m) < K\} \cap \{m \mid \mathrm{age}^{\rightarrow}(p)(m) < K\} \quad (5.10)$$

Lee's original UCB-analysis employs abstract cache states very similar to the abstract cache semantics (see Section 3.3). For the sake of completeness, we describe the complete UCB analysis in Appendix A.

Negi et al. [64] and later Staschulat et al. [80] proposed analyses based on the *collecting cache semantics* (see Section 3.2, Equation (3.41)). Such analyses are potentially more precise, yet computationally infeasible for realistically sized programs [43].

ECB Analysis

The analysis of ECBs is comparably simple. It suffices to collect all memory blocks accessed during a task's execution. Although the set of evicting cache blocks is defined as the set of *all* memory blocks possibly accessed during a task's execution, the impact of the preempting task on the useful cache blocks in each cache set is always limited by the associativity. Hence, it suffices to consider only up to associativity-many ECBs per cache set and so, the may-cache analysis can be used for the derivation of the ECBs.

Chapter 6

Definitely-Cached Useful Cache Blocks

All truths are easy to understand once they are discovered; the point is to discover them.

Galileo Galilei (1564 - 1642)

In 1996, Lee et al. [49] proposed the first static analysis of the cache-related preemption delay. To this end, they introduced the concept of *useful cache blocks* to denote cache blocks of the preempted task that may need to be reloaded in case of preemption. Since then, this concept has been applied and extended in various way while keeping the basic idea unmodified.

In this chapter, we identify substantial pessimism in the CRPD analyses based on useful cache blocks. We introduce a new, precise concept for the CRPD analysis, the *definitely-cached useful cache blocks (DC-UCB)*, and prove the correctness of the analysis. All CRPD equations provided in Chapter 5 are also valid for DC-UCBs. Note that the results presented in this chapter are partially published in [2].

6.1. Pessimism in Lee's Approach

Timing analysis for non-preemptive execution and the CRPD analysis are treated in isolation. Although this approach provides some natural separation of concerns, it is inherently imprecise. To detail this imprecision, we have to take a close look at the *cache analysis* used in both cases.

As presented in Section 3.3, a static timing analysis uses an under-approximation of the cache contents to predict the number of cache hits and an over-approximation to predict the number of cache misses (see Table 3.5). The corresponding analyses are the *must-cache analysis* (under-approximation) and the *may-cache analysis* (over-approximation). A cache miss can only be excluded if the memory access is classified as *always hit* by the *must-cache analysis*:

always hit (ah) memory access always results in a cache hit

always miss (am) memory access always results in a cache miss

not classified (nc) no classification as always hit or always miss

To provide a conservative bound on the cache-related preemption delay, an over-approximation of the cache content is needed. Each memory block that *may* be cached may lead to an additional cache miss due to preemption. Hence, Lee et al. [49] rely on the results of a may-cache analysis.

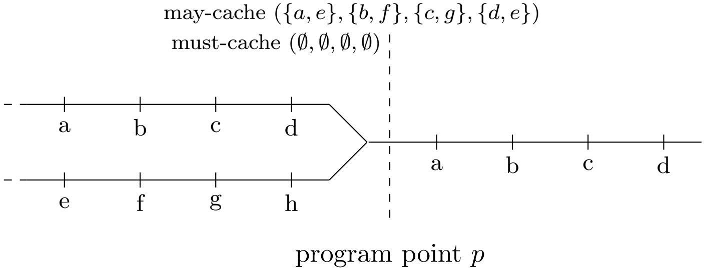

may-cache $(\{a,e\},\{b,f\},\{c,g\},\{d,e\})$
must-cache $(\emptyset,\emptyset,\emptyset,\emptyset)$
a
b
c
d
e
f
g
h
a
b
c
d
program point p

Figure 6.1.: Over-approximation of WCET and CRPD analysis. Execution of a preempted task and a direct-mapped cache with 4 cache sets.

What happens to the memory accesses part of the *may*, but not of the *must-cache* (i.e., neither classified as *always miss* nor as *always hit*)? Such an access is considered a cache miss both within timing analysis and within CRPD analysis. Hence, these accesses may contribute twice to the overall execution time bound under preemption. The imprecision of the cache analysis accumulates and results in a pessimistic total bound. See Figure 6.1 for an example. We assume a direct-mapped cache with 4 cache sets and preemption at program point p. The second accesses to a, b, c and d are considered to inflict cache misses by the timing analysis and by the CRPD analysis.

6.2. Definitely-Cached UCBs

A bound on the cache-related preemption delay is used in most cases—if not in all—in combination with a bound on the execution time. Remember that we are aiming at a bound on the execution time of a task under preemption. We are thus only interested in the *additional* delay due to preemption (with respect to the non-preempted execution time). To this end, we change the definition of useful cache blocks to compute only this additional delay. We need to restrict the set of useful cache blocks to accesses considered hits by the timing analysis.

Definition 6.1 (Definitely-cached UCB on a path)
Let π be a execution path, p and p' two program points on π with p prior to p' and ζ a cache state. A memory block m is called a definitely-cached useful cache block *(DC-UCB) at program point p on path π with initial cache state ζ, if*

a) m is cached at p,

b) m is reused at program point p' that is reached from p and is cached along the path to its reuse, and,

c) m is considered a cache hit at p' by the timing analysis.

The function

$$\textit{DC-UCB}^{\Pi} : \Pi \times Z \times V \rightarrow 2^{M}$$

assigns each program point on a path the corresponding set of DC-UCBs, i.e., $\textit{DC-UCB}^{\Pi}(\pi, \zeta, p)$ is the set of definitely cached useful cache blocks at program point p on path π and initial cache state ζ.

Definition 6.2 (Definitely-cached UCB at a program point)
A memory block m is called a definitely-cached useful cache block *(DC-UCB) at program point p, iff there is at least one path π such that m is a DC-UCB at p on path π. The function*

$$\textit{DC-UCB} : V \rightarrow 2^{M}$$

assigns each program point the corresponding set of definitely-cached useful cache blocks, i.e., DC-UCB(p) is the set of definitely cached useful cache blocks at program point p.

$$m \in \mathit{DC\text{-}UCB}(p) \Leftrightarrow \exists \pi \in \Pi, \zeta \in Z : m \in \mathit{DC\text{-}UCB}^{\Pi}(\pi, \zeta, p)$$

$\mathit{DC\text{-}UCB}^s : V \rightarrow 2^M$: *assigns each program point the set of DC-UCBs mapping to cache set s.*

Definitely-cached useful cache blocks rely on the memory access classification of the timing analysis (see Section 3.3). A cache block may only be considered to inflict an additional cache miss, if it was treated as a cache hit within the timing analysis. Obviously, the set of DC-UCBs is a subset of the set of UCBs at any program point p: DC-UCB(p) $\subseteq$ UCB(p). Consider the example illustrated in Figure 6.1. The set of UCBs at p contains all four memory accesses a to d, the set of DC-UCBs none. A preemption at p may inflict four misses not present in the non-preempted execution. But all of them are already considered cache misses by the timing analysis.

Remark

Note that Lee et al. [50] implicitly use the notion of DC-UCB in UCB analysis of data caches: they focus only on static addressing of data accesses; dynamic accesses are considered as cache misses by their timing analysis. Hence, the computed CRPD bound only denotes the additional cache misses due to preemption in case of unknown memory accesses (e.g., if no precise effective memory address is available, see Section 3.3).

The CRPD bounds based on DC-UCBs are defined analogously to those based on UCBs. Equation (6.1) defines a bound on the CRPD for preemption after program point p

$$\mathrm{CRPD}_P^{\mathrm{UCB}} : (V \rightarrow 2^M) \times V \rightarrow \mathbb{N}$$

$$\mathrm{CRPD}_P^{\mathrm{UCB}}(\text{DC-UCB}, p) = \mathrm{CRT} \cdot |\text{DC-UCB}(p)| \tag{6.1}$$

A global bound on the preemption cost for preemption at any point in the program is described by Equation (6.2)

$$\mathrm{CRPD}^{\mathrm{UCB}} : (V \rightarrow 2^M) \rightarrow \mathbb{N}$$

$$\mathrm{CRPD}^{\mathrm{UCB}}(\text{DC-UCB}) = \max_{p \in V}\{\mathrm{CRPD}_P^{\mathrm{UCB}}(\text{DC-UCB}, p)\} \tag{6.2}$$

The bound on the execution time under preemption is defined analogously:

$$\mathrm{WCET}^B + n \cdot \mathrm{CRPD}^{\mathrm{UCB}}(\text{DC-UCB}) \tag{6.3}$$

6.2.1. Correctness

A preemption-cost bound based on the set of definitely-cached useful cache blocks does not necessarily bound the actual preemption cost. This means that there exists an execution path, an initial cache state and a set of preemptions such that the actual preemption costs are larger than the bound computed using DC-UCBs:

$$\exists \pi, \zeta, PR : n \cdot \mathrm{CRPD}^{\mathrm{UCB}}(\text{DC-UCB}) < \mathrm{CRPD}^{T}(\pi, \zeta, PR) \tag{6.4}$$

Thus, to prove the correctness of the DC-UCB approach, we have to show that Equation (6.3) bounds the execution time of a task under preemption. Note that we assume LRU or direct-mapped caches.

Theorem 6.1
The execution of a task under preemption is bounded by the execution time bound for non-preemptive execution and n times DC-UCB CRPD-bound as derived by Equation (6.2) (where n denotes the number of preemptions):

$$\forall \pi, \zeta, PR : ET^{P}(\pi, \zeta, PR) \leq WCET^{B} + |PR| \cdot CRPD^{UCB}(DC\text{-}UCB)$$

We prove this theorem in two steps. First, we prove the correctness of the UCB approach and then establish the connection to DC-UCBs.

Theorem 6.2
Each additional cache miss m at p' due to preemption by the set PR is contained in the set of UCBs of at least one preemption point.

Proof
Let p be the earliest preemption point prior to program point p'. Such a preemption point prior to the additional cache miss must exist as we assume that both execution traces (non-preemptive and preemptive execution) start with the same initial cache state. The first difference in execution can thus only occur after the first preemption. We need to prove that

a) m has not been accessed on path $[p, \dots, p']$

b) m has been cached at p

c) m is cached along the path to its reuse $[p, \dots, p']$ in case of non-preempted execution

We prove claim a) by contradiction. Assume there is an access to m at program point $\hat{p}$ within the path from p to p'. In this case, m would have *the same cache position at $\hat{p}$ and so also p'. After the access to m at $\hat{p}$, m is stored at the first position in the cache (remember that we assume LRU or direct-mapped caches). From $\hat{p}$ on, the cache position of m is equal in both cases, preempted and non-preempted execution—again due to the restriction to LRU/direct-mapped caches and the property that a cache hit and cache miss result in the same cache update. Thus, as we have a cache hit in the non-preemptive execution, we also have a cache hit in the preemptive case. This is a contradiction to the assumption that m is an additional cache miss at p'. From a), we can deduce b) and c) directly: as there has been no access to m from p to p' but a cache hit at p', m must be cached at p and can not be evicted on this path. Thus m is a UCB at p.* □

Hence, each memory block that causes an additional cache miss due to preemption is contained in the set of UCBs of at least one preemption point. As the proof relies on specific properties of LRU caches, Theorem 6.1 is not valid for FIFO or PLRU caches: Misses and hits may result in different cache updates and thus, cache change at preemption point p may have an unbounded effect as shown in Section 4.3. We can now proof the correctness of Theorem 6.1

Proof

By construction of the timing analysis, we know that the derived timing bound is an upper approximation of the execution time of a task under each possible initial cache state and actual execution path:

$$\forall \pi, \zeta : ET(\pi, \zeta) \leq WCET^B \tag{6.5}$$

Recall the classification of memory accesses to always hit, always miss *and* not classified *from Section 3.3. Based on this classification, a timing analysis conservatively assumes that each cache access that can not be proven to inflict a cache hit may also inflict a cache miss. Hence, the timing bound $WCET^B$ is also valid for any execution of the task, where each cache block not classified as always hit inflicts a cache miss. Let ET^{Cl} denote the execution time of such an execution:*

$$\forall \pi, \zeta : ET^{Cl}(\pi, \zeta) \leq WCET^B \tag{6.6}$$

We now compare the execution trace π of a task where assuming a cache miss for each cache access to an element not classified as always hit with the same execution trace assuming preemption at PR. Let m be a memory block accessed at p' that inflicts an additional miss due to preemption. We can conclude that m must have been considered a cache hit at p' by *the timing analysis. Furthermore, Theorem 6.1 states that there is a preemption point p, such that $m \in UCB(p)$.*

$$\begin{aligned}
&ET^{P}(\pi, \zeta, PR) \\
&\quad \leq ET^{Cl}(\pi, \zeta) + \sum_{(p,\pi) \in PR} CRT \cdot |DC\text{-}UCB(p)| \\
&\quad \leq WCET^{B} + \sum_{(p,\pi) \in PR} CRT \cdot |DC\text{-}UCB(p)| && \text{Eq. (6.6)} \\
&\quad = WCET^{B} + \sum_{(p,\pi) \in PR} CRPD^{\scriptscriptstyle UCB}_{P}(DC\text{-}UCB, p) && \text{Eq. (6.1)} \\
&\quad \leq WCET^{B} + \sum_{(p,\pi) \in PR} CRPD^{\scriptscriptstyle UCB}(DC\text{-}UCB) && \text{Eq. (6.2)} \\
&\quad = WCET^{B} + |PR| \cdot CRPD^{\scriptscriptstyle UCB}
\end{aligned}$$

□

6.3. Deriving the Set of DC-UCBs

If an element is classified as *always hit (ah)* at its reuse, it is also classified as always hit along the path to its reuse. This insight allows for a simple DC-UCB analysis. It suffices to intersect the set of UCBs at program point p (as derived for instance by the UCB analysis presented in Appendix A) with the *must-cache* information at p (i.e., the set of memory blocks with upper age bound < associativity).

$$\text{DC-UCB}(p) \subseteq \text{UCB}(p) \cap \{m \mid \text{Classify}(m, p) = ah\} \tag{6.7}$$

As the example depicted in Figure 6.2 shows, this approach may be imprecise. Timing analysis needs to consider both paths reaching the second access to memory block a. Thus, it can not guarantee a to be a hit. However, on the upper path after the first access to a, a is considered to be cached on the sub-path. If we compute the set of DC-UCBs using Equation (6.7), we would consider a to be a DC-UCB on this path, although it is already considered a cache miss at its next access.

Therefore, we present an alternative derivation of the DC-UCBs. A backwards analysis that computes at each program point p a set of memory blocks, such that for each memory block $m \in \text{DC-UCB}(p)$ there

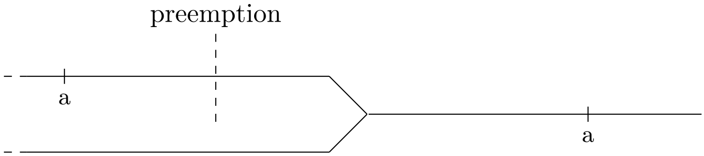


Figure 6.2.: Pessimism of a naive DC-UCB analysis: In case the set of DC-UCBs is computed by an intersection of the set of UCBs with the must-cache, memory block a is falsely considered a DC-UCB.

exists a path on which m is reused at its next access p_n and m is classified as *always hit* at p_n:

$$\begin{aligned} m \in \text{DC-UCB}(p_i) \Leftrightarrow \exists [p_i, \ldots, p_n] \in \Pi : \\ \sharp(p_n) = m \wedge \text{Classify}(m, p_i) = ah \wedge \forall_{j=i}^{n-1} \sharp(p_j) \neq m \end{aligned} \tag{6.8}$$

As this property is important for the DC-UCB analysis, we define a function Els that checks for this property:

$$\text{Els} : V \times M \to \{\text{true}, \text{false}\}$$

$$\text{Els}(\pi, m) = \begin{cases} \text{true} & \text{if } \pi = [p_1, \ldots, p_n] \cdot \pi' \wedge \sharp(p_n) = m \\ & \qquad \wedge \text{Classify}(m, p) = ah \wedge \forall_{j=i}^{n-1} \sharp(p) \neq m \\ \text{false} & \text{otherwise} \end{cases} \tag{6.9}$$

We formulate this analysis as an abstract interpretation.

Concrete Semantics

The set of DC-UCBs at program point p is the set of memory blocks, for which Property (6.8) holds on at least one concrete path $\pi \in \Pi$ starting in p. As we argue about paths, we resort to the *path-based backwards collecting semantics*, i.e., the set of all concrete paths that emanate from a program point:

$$\text{Coll}_{\Pi}^{\leftarrow} : V \to 2^{\Pi}$$

$$\text{Coll}_{\Pi}^{\leftarrow}(p) = \{\pi \mid \pi \in \Pi \wedge \pi = [p, \ldots, p_n]\} \tag{6.10}$$

For the sake of completeness, we also present the concrete transformer defined as a function that prepends the current program point to each path of the control flow information:

$$tf_C : V \to (2^{\Pi} \to 2^{\Pi})$$

$$tf_C(p)(S) := \{p \cdot \pi \mid \pi \in S\} \tag{6.11}$$

Given the collecting semantics at program point p, the precise set of DC-UCBs can be extracted by means of the function Els:

$$\text{DC-UCB}(p) = \{m \mid \exists \pi \in \text{Coll}_{\Pi}^{\leftarrow}(p) : \text{Els}(\pi, m)\} \tag{6.12}$$

Abstract DC-UCB Analysis

As the collecting semantics are prohibitively large, we present an abstract analysis using an efficient representation. Instead of computing all paths emanating from a program point and then extracting the set of DC-UCBs, the abstract analysis computes an approximation of the DC-UCBs directly. The domain of the analysis is the powerset of the set of memory blocks:

$$\mathbb{D} : 2^M \tag{6.13}$$

with subset ordering $\subseteq$. The abstract DC-UCB analysis is a backwards analysis with the following transformer:

$$tf : V \rightarrow (2^M \rightarrow 2^M)$$

$$tf(p)(S) = \begin{cases} S & \sharp(p) = \bot \\ S \cup \{\sharp(p)\} & \text{Classify}(\sharp(p), p) = ah \\ S \setminus \{\sharp(p)\} & \text{Classify}(\sharp(p), p) \neq ah \end{cases} \tag{6.14}$$

In case the instruction at program point p accesses no memory block, i.e., $\sharp(p) = \bot$, we do not change the control flow information. If program point p accesses an element $\sharp(p)$ classified as *always hit* at p, we add this element to the set of DC-UCBs; from now on, there is a path satisfying Property (6.8) for the accessed memory block. If the accessed element $\sharp(p)$ is not classified as always hit, we remove this element from the set of DC-UCBs as each new path (with p prepended) violates Property (6.8) for memory block $\sharp(p)$.

Theorem 6.3 (Monotonicity)
The abstract transformer of the DC-UCB analysis tf is monotone:

$$\forall p \in V : \forall a, b \in 2^M : a \subseteq b \Rightarrow tf(p)(a) \subseteq tf(p)(b)$$

See Appendix B for a proof of Theorem 6.3. As we are interested in a safe over-approximation of the set of DC-UCBs at each program point, we need to compute the union of the set of DC-UCBs at all joins.

Connecting Abstract and Concrete Domain

The *concretization* γ for the abstract domain is given by a set of all paths, such that Property (6.8) only holds for elements that are contained in the set of DC-UCBs:

$$\gamma : 2^M \rightarrow 2^\Pi$$

$$\gamma(S) := \{\pi \ \mid \ \pi \in \Pi : \forall m \in (M \setminus S) : \neg \mathrm{Els}(\pi, m)\} \tag{6.15}$$

In other words, $\gamma(S)$ contains all paths from Π except those on which memory blocks not contained in S are considered a DC-UCB. The function γ allows us to state the *local consistency* of the abstract DC-UCB transformer (with respect to the path-based semantics):

Theorem 6.4 (Local Consistency)
The abstract transformer tf and the concrete transformer tf_C are locally consistent:

$$\forall S \in 2^M : \forall p \in V : (tf_C(p))(\gamma(S)) \subseteq \gamma((tf(p))(S))$$

Proof of Theorem 6.4 can be found in Appendix B.

The *abstraction* of a set of paths is given by the set of memory blocks for which Property (6.8) holds on at least one path:

$$\alpha : 2^\Pi \rightarrow 2^M$$

$$\alpha(S) := \{m \ \mid \ \exists \pi \in S : \mathrm{Els}(\pi, m)\} \tag{6.16}$$

The functions α and γ are *sound*, i.e., abstracting a set of paths and then concretizing the results yields a superset of the initial set of paths.

Theorem 6.5 (Soundness of the Abstraction)
The tuple $(\gamma, 2^{M_\Pi})$ is a sound abstraction *of $(2^\Pi, \alpha)$ with subset-ordering, i.e.,*

$$\forall S \in 2^\Pi : S \sqsubseteq \gamma(\alpha(S))$$

holds.

Hence, a set of paths S is conservatively approximated by $\alpha(S)$. Thus, Theorem 6.5 relates the concrete to the abstract semantics and allows to compute the set of DC-UCBs in the abstract domain.

Given the soundness of the abstraction and local consistency of tf and tf_C, we can conclude that the results of the abstract analysis are a sound over-approximation of the set of DC-UCBs at program point p. Note that a fixed-point is computable as the domain M (resp. M_Π) is finite and tf is monotone (see Theorem 2.1 and 2.2).

Example

Figure 6.3 depicts an example of the DC-UCB analysis on a virtual unrolled control-flow graph (see Section 3.3). Memory blocks a and e are DC-UCBs after the first access to a, resp. to e; from these points on, a path exists to a next reference to a, resp. b such that this next access is classified as *always hit* (second accesses to a and e). According to the previous notion, a, e and b, c, d were useful.

Implementation Issues

In contrast to the basic UCB analysis, we do not need to adapt this analysis to set-associative caches. The complete cache structure is hidden in the memory access classification queried by the DC-UCB analysis, i.e., we do not need to run S DC-UCB analyses in parallel (where S is the number of cache sets). The DC-UCB analysis is part of the **aiT timing analyzer** for the **ARM7** target architecture [42].

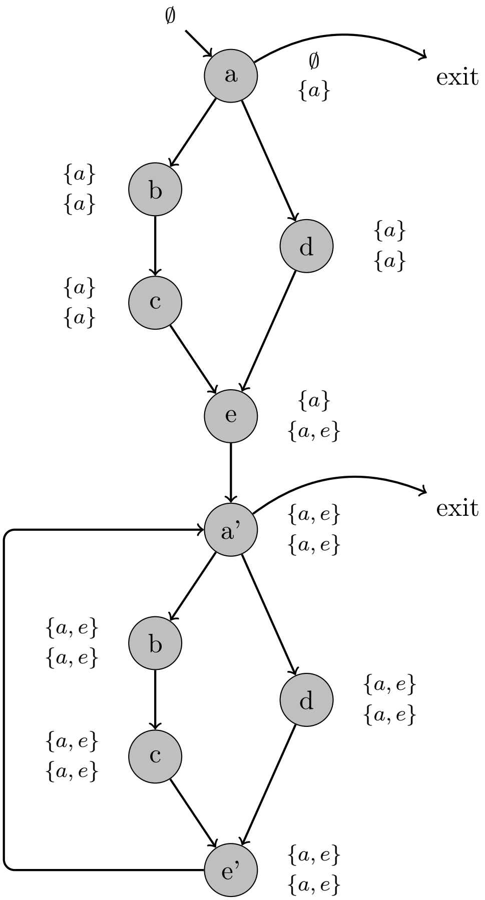


Figure 6.3.: Example of the DC-UCB Analysis

Chapter 7

CRPD for LRU Caches—Resilience Analysis

It's so much easier to suggest solutions when you don't know too much about the problem.
Malcolm Forbes (1919 - 1990)

We have seen in Chapter 5 that no valid bound on the cache-related preemption delay for LRU caches based on ECBs and UCBs has been published so far. Not even a bound based solely on the ECBs has been proposed; only for the special case of direct-mapped caches valid and sound bounds exist.

In this chapter, we tackle this problem and present valid CRPD analyses for LRU caches that incorporate both sides, the preempting task and the preempted task. We start with a simple, shallow combination assuming that one ECB always displaces all UCBs of the same cache set. This assumption is sound but pessimistic; a useful cache block may survive a preemption even if the preempting task uses the same cache set. Hence, we present a second, in depth combination of UCBs and ECBs and introduce the notion of *resilience* of a UCB. The *resilience* determines the amount of disturbance due to preemption, i.e., the number of additional accesses such that a cache block remains useful.

In addition to the notion of resilience and the CRPD bounds for LRU caches, we present a resilience analysis based on abstract interpretation and prove its correctness. Note that the approaches in this chapter can be applied to both notions of UCBs, the original concept by Lee et al. [49] and the concept of DC-UCBs presented in the previous chapter. The results presented in this chapter are partially published in [6].

7.1. CRPD for LRU Caches

From Observation 5.1 we know that a single evicting cache block suffices to displace all useful cache blocks of the same cache set. If we have only the set of ECBs at hand to determine an upper bound on the number of additional reloads, we have to assume that up to associativity-many reloads happen in each cache set to which at least one ECB maps:

$$\begin{gathered}\mathrm{CRPD}^{\mathrm{ECB}} : 2^{M} \rightarrow \mathbb{N}\\ \mathrm{CRPD}^{\mathrm{ECB}}(\mathrm{ECB}) = \mathrm{CRT} \cdot \left(\sum_{s} x^{s}\right)\\ x^{s} = \begin{cases} K & \text{if } \mathrm{ECB}^{s} \neq \emptyset \\ 0 & \text{if } \mathrm{ECB}^{s} = \emptyset \end{cases}\end{gathered} \tag{7.1}$$

Note that considering only the effect of the preempting task, i.e., the evicting cache blocks of the preempting task, abstracts from the actual preemption point. Hence, we do not need to compute the maximum over all preemption points to derive a global CRPD bound. Correctness of Equation (7.1) is obvious. If the preempting task accesses no cache blocks mapping to cache set s, the cache state of set s remains unchanged and no additional cache miss in s can occur. If there is an ECB mapping to s, we assume the worst-case, i.e., each cached memory block of the preempted task is evicted and causes an additional reload. This assumption can be easily relaxed by just considering the actual set of useful cache blocks at cache set s:

$$\begin{gathered}\mathrm{CRPD}_{P}^{\mathrm{UCB\&ECB}} : (V \rightarrow 2^{M}) \times 2^{M} \times V \rightarrow \mathbb{N}\\ \mathrm{CRPD}_{P}^{\mathrm{UCB\&ECB}}(\mathrm{UCB}, \mathrm{ECB}, p) = \mathrm{CRT} \cdot \left(\sum_{s} x^{s}\right)\\ x^{s} = \begin{cases} |\mathrm{UCB}^{s}(p)| & \text{if } \mathrm{ECB}^{s} \neq \emptyset \\ 0 & \text{if } \mathrm{ECB}^{s} = \emptyset \end{cases}\end{gathered} \tag{7.2}$$

$$\mathrm{CRPD}^{\mathrm{UCB\&ECB}} : (V \rightarrow 2^{M}) \times 2^{M} \rightarrow \mathbb{N}$$

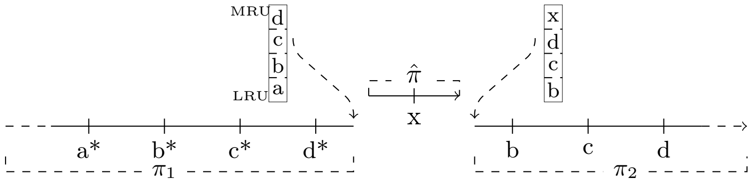


Figure 7.1.: Pessimism of the shallow ECB/UCB combination: Execution trace on a 4-way LRU cache with preemption with one ECBs and three UCBs, but no additional cache misses.

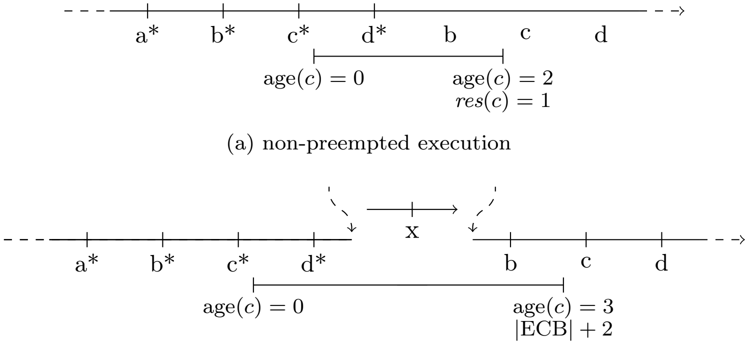
$$\mathrm{CRPD}^{\mathrm{UCB\&ECB}}(\mathrm{UCB},\mathrm{ECB}) = \max_{p \in V}\{\mathrm{CRPD}_P^{\mathrm{UCB\&ECB}}(\mathrm{UCB},\mathrm{ECB},p)\} \tag{7.3}$$

Note that in contrast to Equation (7.1), we now consider different preemption points and thus, need to compute the worst-case preemption point again.

Equation (7.2) provides a valid upper bound on the CRPD for LRU caches based on UCBs and ECBs, but with unsatisfactory precision. Figure 7.1 depicts a simple scenario in which Equation (7.2) computes a total CRPD bound of 3, but no additional cache misses occur. One ECB simply does not suffice to evict a useful cache block at preemption point p and also does not lead to a subsequent additional miss. In the next section, we provide an analysis that aims at getting rid of this pessimism.

7.2. Resilience of a Cache Block

To improve the bound on the cache-related preemption delay for LRU caches, we need to identify useful cache blocks that are not evicted even if some evicting cache blocks map to the same cache set. To this end, we introduce the *resilience* of a cache block as a measure for the amount of *disturbance* a cache block may suffer without causing an additional miss due to preemption. The disturbance a cache block suffers is upper bounded by the set of evicting cache blocks. So, the resilience $res(m)$ of a cache block m is the maximal number of additional misses, such that m remains cached and useful until its next access. Hence, if the disturbances, i.e., the number of ECBs, is less than or equal to the resilience of UCB m, m remains useful after preemption and will not inflict an additional cache miss due to preemption. Obviously, the resilience of a useful cache block on a path depends on the associativity of the cache and the age of the



(a) non-preempted execution

(b) preempted execution

Figure 7.2.: Notion of the Resilience of an UCB: Execution trace on an 4 way LRU cache and preemption with one ECBs and three UCBs, but no additional cache misses. All UCBs are 1-resilient.

cache block directly before its next reuse. Consider the example depicted in Figure 7.2. Memory block c is loaded to the cache at its first access and later on reused on its second access. At this second access, memory block c has age 2 (see Figure 7.2a). In a cache with associativity 4, c can survive one more access but not two. Hence, resilience of c at this position is 1. In case of preemption with one ECB memory block c is still cached at its next reuse (see Figure 7.2b).

Definition 7.1 (Resilience on a Path)
Let π be an execution path of the preempted program and ζ a cache state. The resilience function on a path

$$res^{\Pi} : \Pi \times Z \times V \times M \to \mathbb{N}$$

is defined as follows

$$res^{\Pi}(\pi, \zeta, p', m) =$$
$$\begin{cases} K - \widehat{age}(tf^{LRU}([p_1, \ldots, p'])(\zeta))(m) - 1 & \textit{if } m \in UCB(\pi, \zeta, p) \\ & \wedge \pi = [p_1, \ldots, p_n] \\ \infty & \textit{if } m \notin UCB(\pi, \zeta, p) \end{cases}$$

where p' is the next reuse of m (if this reuse exists in π), K the associativity of the cache and $\widehat{age}(tf^{LRU}([p_1, \ldots, p'])(\zeta))(m)$ the cache-position of m at p' on path π and initial cache state ζ.

Remember that tf^{LRU} is the LRU cache transformer and $\widehat{\text{age}}$ the helper function that extracts the ages from a cache state as defined in Section 3.3. Note that we define resilience for any memory block and not only for UCBs. As only UCBs may inflict additional cache misses due to preemption, the resilience of any non-useful block is infinite: No matter how many ECB map to the same cache set, no additional cache miss happens.

Definition 7.2 (Resilience on a Program Point)
The resilience $res(p, m)$ of memory block m at program point p is the minimum over the resiliences of m on all paths running through p and all initial cache states:

$$res : V \times M \to \mathbb{N}$$

$$res(p, m) = \min_{\pi \in \Pi, \zeta \in Z} \{res^{\Pi}(\pi, \zeta, p, m)\}$$

A bound on the cache-related preemption delay based on resilience at program point p is given as follows:

$$\text{CRPD}_P^{\text{RES}} : (V \to 2^M) \times (V \times M \to \mathbb{N}) \times 2^M \times V \to \mathbb{N}$$

$$\begin{aligned} &\text{CRPD}_P^{\text{RES}}(\text{UCB}, res, \text{ECB}, p) = \\ &\qquad \text{CRT} \cdot \sum_s |\text{UCB}^s(p) \setminus \{m \mid res(p, m) \geq |\text{ECB}^s|\}| \end{aligned} \tag{7.4}$$

$$\text{CRPD}^{\text{RES}} : (V \to 2^M) \times (V \times M \to \mathbb{N}) \times 2^M \to \mathbb{N}$$

$$\text{CRPD}^{\text{RES}}(\text{UCB}, res, \text{ECB}) = \max_{p \in V}\{\text{CRPD}_p^{\text{RES}}(\text{UCB}, res, \text{ECB}, p) \tag{7.5}$$

Remember that ECB^s and UCB^s denote the evicting/useful cache blocks mapping at cache set s. The set of useful cache blocks is reduced by those memory blocks that are not evicted by $|\text{ECB}^s|$ many evicting cache blocks. We consider once again the example from Figure 7.1. The set $\text{UCB}(p)$ contains the memory blocks b, c, d, all of them have resilience of 1, memory block a has an infinite resilience and there is only one cache set. We now plug these values into Equation (7.5):

$$\begin{aligned}
\mathrm{CRPD}_P^{\mathrm{RES}}&(\mathrm{UCB}, \mathit{res}, \mathrm{ECB}, p)\\
&= \mathrm{CRT} \cdot |\mathrm{UCB}(P) \setminus \{m \mid \mathit{res}(p, m) \geq |\mathrm{ECB}|\}|\\
&= \mathrm{CRT} \cdot |\{b, c, d\} \setminus \{m \mid \mathit{res}(p, m) \geq |\{x\}|\}|\\
&= \mathrm{CRT} \cdot |\{b, c, d\} \setminus \{m \mid \mathit{res}(p, m) \geq 1\}|\\
&= \mathrm{CRT} \cdot |\{b, c, d\} \setminus \{a, b, c, d\}|\\
&= \mathrm{CRT} \cdot |\emptyset|\\
&= 0
\end{aligned}$$

We can conclude that Equation (7.5) computes a precise CRPD bound—in contrast to all former approaches.

7.2.1. Multiple Preemptions

As mentioned in Chapter 4, preemptions may *interact*. See the example in Figure 7.3. Both preemptions considered in isolation do not inflict any additional cache misses. However, in combination two more misses due to preemption occur. Remember, we have defined the concrete CRPD as the total number of additional misses due to *all* preemptions and not just due to a single preemption (see Chapter 4).

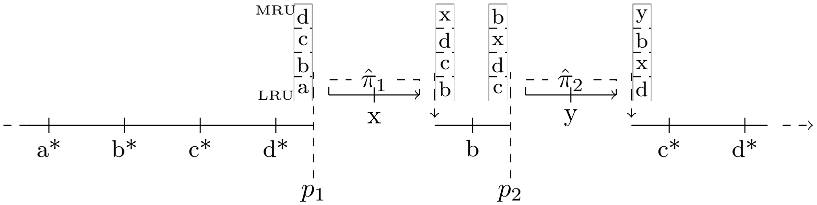


Figure 7.3.: Interacting preemptions: each preemption (at p_1 and p_2) has individual cost of 0, but combined cost of 3 (assuming a 4-way LRU cache)

There are different ways to include multiple preemptions and to account for interacting preemptions. We aim for a simple solution where we simulate nested preemptions. In the example from Figure 7.3, we can use the original CRPD bound for the preemption at p_1 as no prior preemption may interact with the first one:

$$\mathrm{CRPD}_P^{\mathrm{RES}}\left(\mathrm{UCB}, \mathit{res}, \left(\mathrm{ECB}^{\Pi}(\hat{\pi}_1)\right), p_1\right) = 0 \tag{7.6}$$

We assume that the second preemption was a nested preemption by $\hat{\pi}_1$ and $\hat{\pi}_2$. Hence, we use the union of the ECBs of $\hat{\pi}_1$ and $\hat{\pi}_2$ to compute the preemption cost:

$$\mathrm{CRPD}_P^{\mathrm{RES}}\left(\mathrm{UCB}, \mathit{res}, \left(\mathrm{ECB}^{\Pi}(\hat{\pi}_1) \cup \mathrm{ECB}^{\Pi}(\hat{\pi}_2)\right), p_2\right) = 2 \tag{7.7}$$

To generalize this, we assume that the i-th preemption interacts with all prior preemptions 1 to $i-1$.

$$\mathrm{CRPD}^T(\pi, \zeta, PR) \leq \sum_{i=1}^{|PR|} \mathrm{CRPD}_P^{\mathrm{RES}}\left(\mathrm{UCB}, \mathit{res}, \bigcup_{j \leq i} \mathrm{ECB}^{\Pi}(\hat{\pi}_j), p_i\right) \tag{7.8}$$

This approach assumes knowledge about the order of the preempting tasks and the preemptions points. Scheduling analysis, as we will see in Chapter 8, usually can not deliver such information. What it usually can deliver is an over-approximation of the preempting tasks and of the number of preemptions.

Theorem 7.1
For any initial cache state ζ, any execution path π and any set of preemptions PR:

$$\mathit{CRPD}^T(\pi, \zeta, PR) \leq |PR| \cdot \mathit{CRPD}^{RES}\left(\mathit{UCB}, \mathit{res}, \bigcup_{\tau_k \in PT} \mathit{ECB}_{\tau_k}\right) \tag{7.9}$$

where PT is the set of preempting tasks.

Equation (7.9) does not provide the most precise CRPD bound, but uses only information given by the scheduling analysis. More precise bounds require more information.

7.2.2. Correctness

We prove Theorem 7.1 in two steps. We first prove that each cache miss due to preemption is accounted for in the preemption cost attributed to the nearest preemption point prior to the cache miss (assuming nested preemption by all preempting tasks) and then proceed to the total CRPD bound for all preemptions in PR.

Theorem 7.2
Each additional cache miss at p' accessing m due to preemption by the set

$$PR = \{(q_1, \hat{\pi}_1), \ldots, (q_n, \hat{\pi}_n)\}$$

is contained in the set

$$\mathit{UCB}^s(p_i) \setminus \{m \mid \mathit{res}(p_i, m) \geq \big|\bigcup_{j \leq i} \mathit{ECB}^s(\hat{\pi}_j)\big|\}$$

of the nearest preemption point p_i prior to p'.

Proof
We prove Theorem 7.2 by contradiction. Assume there exists a cache miss at m such that

$$m \notin \mathit{UCB}^s(p_i) \setminus \{n \mid \mathit{res}(p_i, n) \geq \big|\bigcup_{j \leq i} \mathit{ECB}(\hat{\pi}_j)\big|\}$$

Let p'' be the last access to m prior to preemption point p, p' the next access to m after p, and $PT \subseteq \{1, \ldots, i\}$ be the indices of preemptions with preemption points between p'' and p_i. By Theorem 6.2, we know that $m \in \mathit{UCB}(p_i)$. Hence,

$$\mathit{res}(p_i, m) \geq \big|\bigcup_{j \leq i} \mathit{ECB}(\hat{\pi}_j)\big|$$

As m inflicts a cache miss at p', we know that the number of accesses to distinct elements on path $[p'', \ldots, p']$ and on the access sequence of the preempting tasks is larger than the associativity. Note that the age of m at p' $\widehat{age}(tf^{\mathit{LRU}}([p_1, \ldots, p'])(\zeta))(m)$ is independent of the initial cache state ζ as p'' accesses m.

$$\begin{aligned}
&K < \widehat{age}(tf^{\mathit{LRU}}([p'', \ldots, p'])(\zeta))(m) + \big|\bigcup_{j \in PT} \mathit{ECB}^{\Pi}(\hat{\pi}_j)\big| \\
\Leftrightarrow &K - \widehat{age}(tf^{\mathit{LRU}}([p'', \ldots, p'])(\zeta))(m) < \big|\bigcup_{j \in PT} \mathit{ECB}^{\Pi}(\hat{\pi}_j)\big| \\
\Rightarrow &K - \widehat{age}(tf^{\mathit{LRU}}([p'', \ldots, p'])(\zeta))(m) < \big|\bigcup_{j \leq i} \mathit{ECB}^{\Pi}(\hat{\pi}_j)\big| \\
\Rightarrow &\mathit{res}(p_i, m) < \big|\bigcup_{j \leq i} \mathit{ECB}^{\Pi}(\hat{\pi}_j)\big|
\end{aligned}$$

This contradicts our assumptions. □

We can now proceed to the proof of Theorem 7.1.

Proof

$$
\begin{aligned}
&\mathit{CRPD}^T(\pi, \zeta, PR) \\
&\quad \leq \sum_{i=1}^{|PR|} \mathit{CRPD}_P^{\mathit{RES}}\left(\mathit{UCB}, \mathit{res}, \bigcup_{j \leq i} \mathit{ECB}(\hat{\pi}_j), p_i\right) && \textit{Theo. 7.2} \\
&\quad \leq \sum_{i=1}^{|PR|} \mathit{CRPD}^{\mathit{RES}}\left(\mathit{UCB}, \mathit{res}, \bigcup_{j \leq i} \mathit{ECB}(\hat{\pi}_j)\right) && \textit{Eq. (7.5)} \\
&\quad \leq |PR \cdot |\mathit{CRPD}^{\mathit{RES}}\left(\mathit{UCB}, \mathit{res}, \bigcup_{j} \mathit{ECB}(\hat{\pi}_j)\right) \\
&\quad \leq |PR \cdot |\mathit{CRPD}^{\mathit{RES}}\left(\mathit{UCB}, \mathit{res}, \bigcup_{\tau_k \in PT} \mathit{ECB}_{\tau_k}\right) && \textit{Def. 5.2}
\end{aligned}
$$

where PT is the set of preempting tasks. □

7.3. Resilience Analysis

The resilience of a useful cache block m at a node p is determined by the age of m at its next access p':

$$K - (\widehat{\mathrm{age}}(tf^{\mathrm{LRU}}([p_1, \ldots, p'])(\zeta))(m) + 1) \tag{7.10}$$

Note that we need to increment the age by one since $\mathrm{age}(m) \in \{0, \ldots K - 1, \infty\}$, i.e., 0 denotes the first position in the cache, $K - 1$ the last and ∞ that the element is not cached. As we are interested in a lower bound on the resilience (in order not to classify too many blocks as surviving), we must compute upper bounds on the age. However, we only need to consider the age of a memory block on paths on which this block is considered useful. Consider the control flow depicted in Figure 7.4. Memory block m is only reused on the upper right, not on the lower right path. We thus only need to consider the upper path for the computation of the resilience, even for the resilience of m on the common sub-path on the left side.

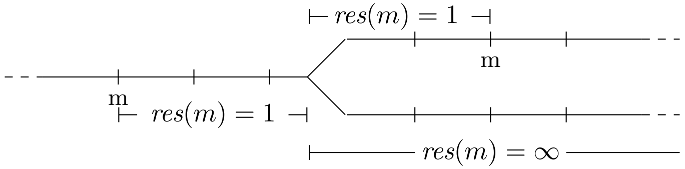


Figure 7.4.: Resilience under different paths, assuming a 4 way LRU cache.

Concrete Semantics

For the computation of the resilience, we start again with the concrete semantics. Similar to the basic UCB analysis (see Appendix A), we split all paths through p in two sets: a set of paths ending in p and a set of paths emanating from p. The first set is given by the path-based *forwards* collecting semantics:

$$\mathrm{Coll}_{\Pi}^{\rightarrow} : V \rightarrow 2^{\Pi}$$

$$\mathrm{Coll}_{\Pi}^{\rightarrow}(p) = \{\pi \mid \pi \in \Pi \wedge \pi = [p_s, \ldots, p]\} \tag{7.11}$$

while the latter set is given by the path-based *backwards* collecting semantics:

$$\mathrm{Coll}_{\Pi}^{\leftarrow} : V \rightarrow 2^{\Pi}$$

$$\mathrm{Coll}_{\Pi}^{\leftarrow}(p) = \{\pi \mid \pi \in \Pi \wedge \pi = [p, \ldots, p_n]\} \tag{7.12}$$

with the concrete transformer $tf_C^{\leftarrow/\rightarrow}$:

$$tf_C^{\leftarrow/\rightarrow} : V \rightarrow (2^{\Pi} \rightarrow 2^{\Pi})$$

$$tf_C^{\rightarrow}(p)(S) := \{\pi \cdot p \mid \pi \in S\} \tag{7.13}$$

and

$$tf_C^{\leftarrow}(p)(S) := \{p \cdot \pi \mid \pi \in S\} \tag{7.14}$$

that appends/prepends a program point to each path of the incoming flow information.

To derive the resilience of a cache block from the collecting path-based semantics, we need two auxiliary functions:

$$\widehat{\mathrm{age}}^{\leftarrow/\rightarrow} : \Pi \rightarrow \mathrm{Age}$$

The function $\widehat{\text{age}}^{\leftarrow}$ counts all different elements on path π backwards (i.e., starting at the last program point in p) to the last access to m. In case no such access exists, it returns ∞:

$$\widehat{\text{age}}^{\leftarrow}(\pi)(m) = \begin{cases} |\bigcup_{j<i} \sharp(p_j)| & \text{if } \pi = [p_1, \ldots, p_i] \cdot \pi' \wedge \sharp(p_i) = m \\ & \quad \wedge \forall j \in \{1, \ldots, i-1\} : \sharp(p_j) \neq m \\ \infty & \pi = [p_1, \ldots, p_n] \\ & \quad \wedge \forall j \in \{1, \ldots, n\} : \sharp(p_j) \neq m \end{cases} \tag{7.15}$$

Conversely, $\widehat{\text{age}}^{\rightarrow}$ counts all memory blocks on path π forwards to the first next access to m and returns ∞ if no such access exists.

$$\widehat{\text{age}}^{\rightarrow}(\pi)(m) = \begin{cases} |\bigcup_{j>i} \sharp(p_j)| & \text{if } \pi = \pi \cdot [p_i, \ldots, p_n] \cdot \pi' \wedge \sharp(p_i) = m \\ & \quad \wedge \forall j \in \{1, \ldots, i-1\} : \sharp(p_j) \neq m \\ \infty & \pi = [p_1, \ldots, p_n] \\ & \quad \wedge \forall j \in \{1, \ldots, n\} : \sharp(p_j) \neq m \end{cases} \tag{7.16}$$

We refer to $\widehat{\text{age}}^{\leftarrow/\rightarrow}(\pi)(m)$ as the concrete forward/backward age of m on path π.

A lower bound on the resilience of block m is thus given as by the associativity minus the sum of forward and backward age:

$$\begin{aligned} res_p(m) \geq K - (&\max\{\widehat{\text{age}}^{\leftarrow}(\pi)(m) \mid \pi \in \text{Coll}_{\Pi}^{\leftarrow}(p) \wedge m \in \text{UCB}(p)\} + \\ &\max\{\widehat{\text{age}}^{\rightarrow}(\pi)(m) \mid \pi \in \text{Coll}_{\Pi}^{\rightarrow}(p) \wedge m \in \text{UCB}(p)\} + 1) \end{aligned} \tag{7.17}$$

where K is the associativity. Again, we need to increment the sum of the ages by one as age $\in \{0, \ldots K-1, \infty\}$. Note that Equation (7.17) is only well defined for useful memory block $m \in \text{UCB}_p$.

Abstract Domain

Instead of computing all paths to/from p and then computing the resilience, we directly bound for each memory block m the concrete ages $\widehat{\text{age}}^{\leftarrow}$ and $\widehat{\text{age}}^{\rightarrow}$. Note that we omit the direction of the analysis in the following as forward analysis and backward analysis are equivalent.

In order to correctly bound the age of a UCB m, we need to derive a bound on the age of block m under the constraint that m is useful. We refer to this bound as the *constrained age* of m. In addition, we need an *unconstrained age* bound of m in order to correctly update the ages of the other memory blocks. We refer to this bound as the unconstrained age of m.

We start with the transformer for the unconstrained age. As we are interested in upper bounds on the ages, we employ a *must-cache analysis* using the very same transfer function (see Section 3.3):

$$tf_{ua} : V \to \text{Age} \to \text{Age}$$

$$tf_{ua}(p)(ua) := \\ \lambda m. \begin{cases} 0 & m = \sharp(p) \\ ua(m) & ua(m) \geq ua(\sharp(p)) \\ ua(m) + 1 & ua(m) < ua(\sharp(p)) \wedge ua(m) < K - 1 \\ \infty & \textit{otherwise} \end{cases} \tag{7.18}$$

The accessed element $\sharp(p)$ is assigned age zero, all younger elements (as the accessed element) age by one. The ages of all older elements or elements of the same age remain unchanged.

The transformer for the constrained age also takes the unconstrained age as input.

$$tf_{ca} : V \to (\text{Age} \times \text{Age} \to \text{Age})$$

$$tf_{ca}(p)(ca, ua) := \\ \lambda m. \begin{cases} 0 & m = \sharp(p) \vee m \notin \text{UCB}(p) \\ ca(m) & ca(m) \geq ua(\sharp(p)) \vee ca(m) = K - 1 \\ ca(m) + 1 & ca(m) < ua(\sharp(p)) \end{cases} \tag{7.19}$$

As the constrained age may under-approximate the actual age, we require the unconstrained age of the accessed element to update the constrained ages of the other elements. The constrained age only matters for useful cache blocks. We can therefore assign the age of other blocks arbitrarily which we do by assigning each non-useful cache block the constrained age zero. Furthermore, we know that the age of a useful cache block m is at least $K - 1$ (as we assume the next access to m is a hit). We thus have the additional constrain $ca(m) = K - 1$ in the second case.

Both ages, unconstrained and constrained, are upper bounds. Hence, $\sqsubseteq$ and $\bigsqcup$ for unconstrained and constrained age are defined according to the *must-cache analysis*:

$$ua_1 \sqsubseteq ua_2 \Leftrightarrow \forall m \in M : ua_1(m) \leq ua_2(m) \tag{7.20}$$

$$ua_1 \bigsqcup ua_2 = \lambda m. \max(ua_1(m), ua_2(m)) \tag{7.21}$$

and

$$ca_1 \sqsubseteq ca_2 \Leftrightarrow \forall m \in M : ca_1(m) \leq ca_2(m) \tag{7.22}$$

$$ca_1 \bigsqcup ca_2 = \lambda m.\, \max(ca_1(m), ca_2(m)) \tag{7.23}$$

The domain of the resilience analysis is a tuple of the two age-bound functions:

$$\mathbb{D} = \text{Age} \times \text{Age} \tag{7.24}$$

with combined transfer function defined as follows:

$$tf_{res}(p)(ca, ua) = (tf_{ua}(p)(ua), tf_{ca}(p)(ca, ua)) \tag{7.25}$$

and $\sqsubseteq$ and $\bigsqcup$ of the domain as the corresponding pairwise operator on the ages:

$$(ua_1, ca_1) \sqsubseteq (ua_2, ca_2) \Leftrightarrow ua_1 \sqsubseteq ua_2 \wedge ca_1 \sqsubseteq ca_2 \tag{7.26}$$

$$(ua_1, ca_1) \bigsqcup (ua_2, ca_2) = (ua_1 \bigsqcup ua_2, ca_1 \bigsqcup ca_2) \tag{7.27}$$

We also need *monotonicity* of the abstract transformer in order to show that a fixed-point of the resilience analysis exists.

Theorem 7.3 (Monotonicity)
The abstract transformer of the resilience analysis tf is monotone, i.e.,

$$\forall p \in V : \forall a, b \in \mathbb{D} : a \sqsubseteq b \Rightarrow tf_{res}(p)(a) \sqsubseteq tf_{res}(p)(b)$$

The proof can be found in Appendix B.

Using the constrained ages, we can define a bound on the resilience of block m as the maximum of zero and the associativity K minus the sum of forward and backward constrained age:

$$res_p(m) \geq \max(K - (ca_p^{\leftarrow}(m) + ca_p^{\rightarrow}(m) + 1), 0) \tag{7.28}$$

Connecting Abstract and Concrete Domain

The set of concrete paths represented by a pair of constrained and unconstrained ages is given by the set of all paths that *respect* the age bound. For the *concretization*, we have to distinguish again between forward and backward analysis, as the concretization of the forward analysis considers all paths emanating from a program point p:

$$\begin{aligned}\gamma^{\rightarrow}((ua, ca)) = \{(p \cdot \pi) \in \Pi \mid \forall m \in M : (\widehat{\text{age}}^{\rightarrow}(p \cdot \pi)(m) \leq ca^{\rightarrow}(m))\\ \vee (\widehat{\text{age}}^{\rightarrow}(p \cdot \pi)(m) \leq ua^{\rightarrow}(m) \wedge m \notin \text{UCB}_p(m))\}\end{aligned} \tag{7.29}$$

while the backward analysis considers all paths ending in a program point p:

$$\begin{aligned}\gamma^{\leftarrow}((ua, ca)) = \{(\pi \cdot p) \in \Pi \mid \forall m \in M : (\widehat{\text{age}}^{\leftarrow}(\pi \cdot p)(m) \leq ca^{\leftarrow}(m)) \\ \vee (\widehat{\text{age}}^{\leftarrow}(\pi \cdot p)(m) \leq ua^{\leftarrow}(m) \wedge m \notin \text{UCB}_p(m))\}\end{aligned} \quad (7.30)$$

Using $\gamma^{\rightarrow}$ and $\gamma^{\leftarrow}$, we can state the *local consistency* of the abstract transformer.

Theorem 7.4 (Local Consistency)
The abstract transformer tf and the concrete transformer tf_C are locally consistent:

$$\forall (ua, ca) \in \mathbb{D} : \forall p \in V : (tf_C(p))(\gamma(ua, ca)) \subseteq \gamma((tf_{res}(p))(ua, ca))$$

The proof can be found in Appendix B.

For the *abstraction* functions, we also have to distinguish between forward and backward analysis. The unconstrained age of a memory block m is given by the maximal age over all paths, while the constrained only considers the paths on which the memory block m is considered useful. The constrained age of m is zero if no such paths exists, i.e., m is not useful on any path:

$$\begin{aligned}\alpha^{\rightarrow}(S) = (\lambda m. \max\{\widehat{\text{age}}^{\rightarrow}(p \cdot \pi)(m) \mid (p \cdot \pi) \in S\}, \\ \lambda m. \max\{\widehat{\text{age}}^{\rightarrow}(p \cdot \pi)(m) \mid (p \cdot \pi) \in S \wedge m \in \text{UCB}(p)\}) \cup \{0\})\end{aligned} \quad (7.31)$$

$$\begin{aligned}\alpha^{\leftarrow}(S) = (\lambda m. \max\{\widehat{\text{age}}^{\leftarrow}(\pi \cdot p)(m) \mid (\pi \cdot p) \in S\}, \\ \lambda m. \max\{\widehat{\text{age}}^{\leftarrow}(p \cdot \pi)(m) \mid (\pi \cdot p) \in S \wedge m \in \text{UCB}(p)\}) \cup \{0\})\end{aligned} \quad (7.32)$$

We now formulate the soundness of the abstraction, i.e., that a set of concrete paths S is conservatively approximated by $\alpha(S)$. Again, the proof can be found in Appendix B.

Theorem 7.5 (Soundness of the Abstraction)
The tuple $(\gamma, \mathbb{D})$ is a sound abstraction *of $(2^{\Pi}, \alpha)$ with subset-ordering:*

$$\forall S \in 2^{\Pi} : S \sqsubseteq \gamma(\alpha(S))$$

The abstract domain and α do not represent the most precise abstraction. Assume a program that consists of a single program point p, and so the set of paths Π contains only the empty path and the path $[p]$:

$$\Pi = \{\epsilon, [p]\} \quad (7.33)$$

Yet, an abstract unconstrained age ua may assign the element $\sharp(p)$ accessed at p age 2:

$$ua(\sharp(p)) = 2 \tag{7.34}$$

Although on all paths in Π, the age of $\sharp(p)$ is at most zero. So, the value (ua, ca) is less precise than $\alpha(\gamma((ua, ca)))$.

Example

Figure 7.5 depicts an example of the resilience analysis (forward age, backward age, and pair of UCB/resilience). We represent the domain of the analysis as a set of mappings $\{(x, (ca, ua))\}$ of memory blocks to pair of ages (constrained and unconstrained) with default mapping (∞, ∞). I.e., each memory block that does not map to a pair of ages explicitly, maps to (∞, ∞) by default. Figure 7.5 only depicts the control-flow information *after* a program point.

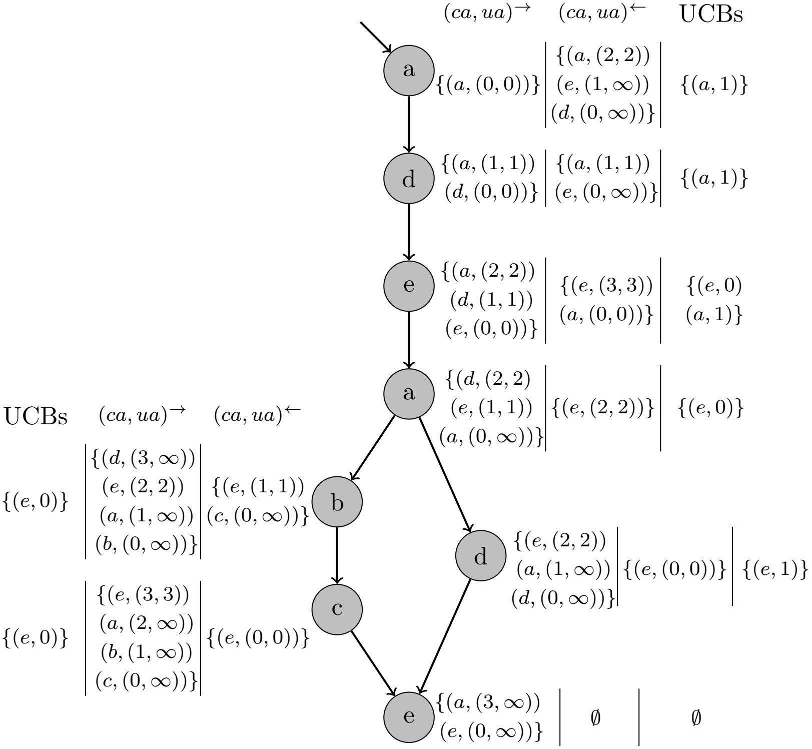


Figure 7.5.: Example of the Resilience Analysis; Forward and Backward Ages (Constrained, Unconstrained), UCBs and corresponding resilience.

Chapter 8

Preemption cost aware Response Time Analysis

I love deadlines. I like the whooshing sound they make as they fly by.
Douglas Adams (1952 - 2001)

In the previous chapters, we have discussed how to derive sound and precise bounds on the cache-related preemption delay. This chapter now presents how to incorporate these bounds in the *response time analyses* for *fixed-priority based schedules.* We start with a discussion and thorough review of the existing approaches and then proceed to a new, more precise analysis: the so-called ECB-Union approach that derives an upper bounds of the effect of all preempting tasks on the preempted task. In addition, we show how to eliminate spurious preemption scenarios to improve the schedulability results even further.

In the following, we use the notation introduced in Chapter 3.1 (execution time C_i, period P_i, deadline D_i) extended by Table 8.1. The new symbols are explained (if necessary) at their first occurrence. Note that the results presented in this chapter are partially published in [3] and [4].

8.1. Existing Approaches

In Chapter 3.1, we have discussed *response time analysis* [8, 41] as the standard schedulability test for fixed-priority preemptive systems. Recursive Equation (8.1) repeats the basic principle of the fixed-point iteration to derive a task's response time R. If the response time of a task is less than or equal to its deadline, we can conclude that this task will never miss its deadline. If this holds for all tasks, we say that the task set is schedulable.

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j) \tag{8.1}$$

Equation (8.1) does not consider any cache-related preemption delay explicitly. Hence, such delay must be subsumed by the execution demand bound C implicitly.

To include the preemption cost in the response time analysis, Busquets and Wellings [18] proposed to extend Equation (8.1) by a value $\gamma_{i,j}$ to represent the preemption cost of a job of task τ_j executing during the response time of task τ_i (with $j < i$):

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j + \gamma_{i,j}) \tag{8.2}$$

Petters et al. [68] and later Staschulat et al. [81] based their analysis on the following Equation:

$$R_i = C_i + \sum_{\forall j \in \mathrm{hp}(i)} \left(\left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j + \gamma_{i,j} \right) \tag{8.3}$$

where $\gamma_{i,j}$ denotes the preemption cost of *all* jobs of task τ_j executing during the response time of task τ_i (with $j < i$). Although the difference to Equation (8.2) is only subtle, it proves beneficial—as we show later—to use $\gamma_{i,j}$ as a bound on the *overall impact* of all jobs of τ_j on the response time R_i instead of a bound of *just one* job of τ_j. Note that when preemption costs are considered explicitly, the worst-case scenario is not necessarily given by a synchronous release of all higher priority tasks [60] and Equation (8.2) and (8.3) form only sufficient, not exact schedulability tests.

The following sections present a review and analysis of the existing approaches. We use a notation that may differ strongly from some of the presented papers. Furthermore, some of the presented approaches are based on flawed CRPD equations. In these cases, we have directly replaced flawed equations by the correct CRPD formulas presented in the previous chapters.

Regarding the examples, we always assume a *direct-mapped cache* with 4 cache sets. So, useful/evicting cache blocks are represented as 4-tuples. For instance, $\mathrm{UCB}_i = (a, b, -, -)$ means that a, resp. b is a useful cache block of task τ_i mapping to cache set 1, resp. 2 while no UCBs of task τ_i map to cache set 3 or 4. We also abstract from the preemption points and define one set of UCBs for the whole task (except for one example where we explicitly mention different preemption points). Note that these simplifications are used only to simplify the example but do not impose actual restriction to the presented approaches.

Table 8.1.: Sets of tasks: notation and terminology

$\mathrm{hp}(i)$	$\{j \mid j < i\}$	tasks with higher priority than τ_i
$\mathrm{hep}(i)$	$\{j \mid j \leq i\}$	tasks with higher or equal priority as τ_i
$\mathrm{lp}(i)$	$\{j \mid j > i\}$	tasks with lower priority than τ_i
$\mathrm{lep}(i)$	$\{j \mid j \geq i\}$	tasks with lower or equal priority as τ_i
$\mathrm{aff}(i, j)$	$\mathrm{hep}(i) \cap \mathrm{lp}(j)$	tasks affected by τ_j during execution of τ_i

8.1.1. ECB-Only & UCB-Only

We already mentioned Busquets and Wellings [18] as the first to extend the basic response time equation. They proposed to use only the set of *evicting cache blocks* of the preempting task as a bound on the preemption cost:

$$\gamma_{i,j}^{\mathrm{ecb}} = \mathrm{CRPD}^{\mathrm{ECB}}(\mathrm{ECB}_{\tau_j}) \tag{8.4}$$

We refer to this approach as ECB-Only. Note that in [18] Busquets and Wellings conservatively bound the set of ECBs and assume that the complete cache content was evicted due to preemption and must be refilled.

Lee et al. [49] tackled the problem from the side of the preempted task. They used the number of *useful cache blocks* of the preempted task. However, the highest preemption cost may not necessarily occur when the current task τ_i, for which the response time is computed, is preempted. It is possible that the preemption cost of τ_j preempting a task τ_k with intermediate priority $j < k < i$ forms the worst-case scenario. The set of tasks that may affect the response time of τ_i but have lower priority than

τ_j is denoted with aff(i,j). As a result, one has to take the maximum of the preemption cost over all tasks from the set aff(i,j):

$$\gamma_{i,j}^{\text{ucb}} = \max_{\forall k \in \text{aff}(i,j)} \{\text{CRPD}^{\text{ECB}}(\text{UCB}_{\tau_i})\} \tag{8.5}$$

We refer to this approach as UCB-Only.

ECB-Only and UCB-Only share the same disadvantage. By considering one side only, either the preempting task or alternatively the preempted task, precision is lost. Not each useful cache block is necessarily evicted due to preemption and not each evicting cache block necessarily evicts a useful cache block. Figure 8.1 illustrates this pessimism.

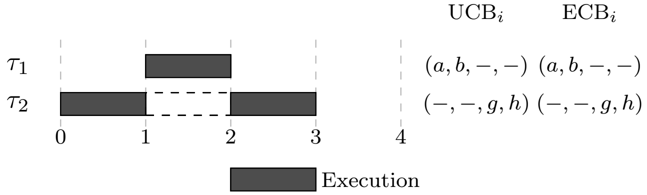


Figure 8.1.: Ganttchart demonstrating the over-approximation of ECB-Only and UCB-Only: two tasks τ_1 and τ_2 with $C_1 = 1$ and $C_2 = 2$. As the ECBs of task τ_1 map to different cache sets as the UCBs of τ_2, no useful cache blocks are evicted due to preemption and so, CRPD = 0. ECB-Only and UCB-Only derive bounds of $2 \cdot \text{CRT}$ in this example.

Why not using a simple combination?

A naive approach to reduce the pessimism of the UCB-Only and ECB-Only approaches is to combine both and to use the precise CRPD bound of task τ_j preempting task τ_i:

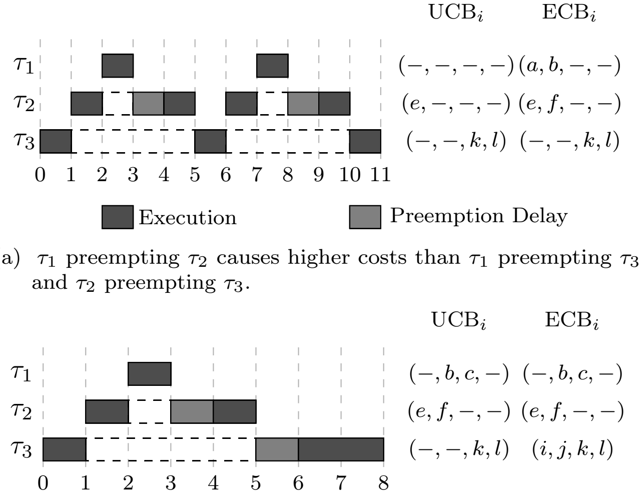
$$\gamma_{i,j} = \text{CRPD}^{\text{UCB\&ECB}}(\text{UCB}_{\tau_i}, \text{ECB}_{\tau_j}) \tag{8.6}$$

Computing the response time using Equation (8.6) is optimistic. Consider the example depicted in Figure 8.2a. When computing the response time R_3, only the cost of τ_1 preempting τ_3 and τ_2 preempting τ_3 are considered. The worst-case scenario, however, occurs when τ_1 preempts τ_2.

Equation (8.7) presents a slightly more complex combination of UCBs and ECBs that covers the previous optimism by computing the maximum preemption cost of task τ_j preempting task τ_i or any intermediate task ($\in \text{aff}(i,j)$).

$$\gamma_{i,j} = \max_{\forall k \in \text{aff}(i,j)} \left\{ \text{CRPD}^{\text{UCB\&ECB}}(\text{UCB}_{\tau_i}, \text{UCB}_{\tau_j}) \right\} \qquad (8.7)$$

However, also this approach is optimistic. It misses the situation of a nested preemption constituting the worst-case preemption scenario as depicted in Figure 8.2b. Task τ_1 evicts useful cache blocks from both tasks τ_2 and τ_3. Response time analysis using Equation (8.7) results in optimistic total CRPD of 1, as it considers only τ_1 preempting τ_2 or τ_3 with cost 1 each and τ_2 preempting τ_3 with cost 0. However, 2 additional reloads due to preemption are necessary in the worst-case. Note that Equation (8.7) was proposed by Tan and Mooney [84].



(a) τ_1 preempting τ_2 causes higher costs than τ_1 preempting τ_3 and τ_2 preempting τ_3.

(b) Nested preemption: τ_1 preempting τ_2 preempting τ_3, causes higher costs than any non-nested preemption.

Figure 8.2.: Ganttchart demonstrating pitfalls due to nested preemption: task set $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = 2$, $C_3 = 3$ and a block reload time of 1.

8.1.2. UCB Union

To include nested preemption, Tan and Mooney proposed in [85] to use an upper bound on the number of useful cache blocks (of all preempted tasks) a preempting task τ_j may evict. In this case, the worst-case preemption point of a task does not only depend on its useful cache blocks and the evicting cache blocks of the preempting task, but also on the UCBs of all other preempted tasks. Assume for instance an execution of three tasks on a system with direct-mapped cache with 4 cache sets and the following sets of UCBs/ECBs:

$$\begin{aligned} \mathrm{ECB}_1 &= (a, b, c, d) \\ \mathrm{UCB}_2 &= (e, f, -, -) \\ \mathrm{UCB}_3(p_1) &= (i, j, -, -) \\ \mathrm{UCB}_3(p_2) &= (-, -, k, -) \end{aligned} \tag{8.8}$$

Task τ_3 has two different possible preemption points with two disjoint sets of UCBs. Task τ_1 may evict 3 UCBs (one of τ_3 and two of τ_2), worst case preemption point for a direct preemption of τ_3 by τ_1, however, is p_1 with two evicted useful cache blocks. This means that it is insufficient to consider only one preemption point of task τ_3.

Tan and Mooney avoid a precise derivation of the worst-case preemption points by computing the union of all UCBs at all preemption points of the preempted task:

$$\mathrm{UCB}^{\cup} = \bigcup_{p \in V} \mathrm{UCB}(p) \tag{8.9}$$

In this case, a slight variation of Equation (7.2) delivers a valid CRPD bound:

$$\begin{aligned} &\mathrm{CRPD}^{\text{ucb-u}} : 2^M \times 2^M \to \mathbb{N} \\ &\mathrm{CRPD}^{\text{ucb-u}}(\mathrm{UCB}^{\cup}, \mathrm{ECB}) = \mathrm{CRT} \cdot \left(\sum_s x^s \right) \\ &x^s = \begin{cases} |\mathrm{UCB}^{\cup,s}| & \text{if } \mathrm{ECB}^s \neq \emptyset \\ 0 & \text{if } \mathrm{ECB}^s = \emptyset \end{cases} \end{aligned} \tag{8.10}$$

Note that Tan and Money originally based their approach on the flawed CRPD bound Equation (5.9) presented in Chapter 5.

We can now proceed to the computation of $\gamma^{\text{ucb-u}}$. As only useful cache blocks of tasks with equal or higher priority than τ_i may increase the response time R_i, we only need to consider tasks with intermediate priority, i.e., tasks from the set $\mathrm{aff}(i, j)$):

$$\gamma_{i,j}^{\text{ucb-u}} = \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{ucb-u}} \left(\left(\bigcup_{k \in \mathrm{aff}(i,j)} \mathrm{UCB}_k^{\cup} \right), \mathrm{ECB}_j \right) \tag{8.11}$$

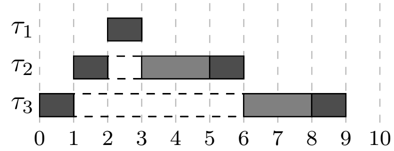
So, $\gamma_{i,j}^{\text{ucb-u}}$ represents the worst-case impact a job of task τ_j can have on all (useful cache blocks of) tasks with lower priority than τ_j down to τ_i. We refer to this approach as UCB-Union.

This approach has two main deficiencies. First, a single preemption affects only the useful cache blocks at one preemption point. Equation (8.11) considers all useful cache blocks of all possibly affected tasks. Second, some UCBs that may be evicted at most once are considered to be evicted several times. Figure 8.3 shows such an example. Consider the response time of task τ_3.

$$\begin{aligned}\gamma_{3,1}^{\text{ucb-u}} &= \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{UCB\&ECB}}\left(\left(\bigcup_{k \in \mathrm{aff}(3,1)} \mathrm{UCB}_k^{\cup}\right), \mathrm{ECB}_1\right)\\ &= \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{UCB\&ECB}}\left(\left(\mathrm{UCB}_2^{\cup} \cup \mathrm{UCB}_3^{\cup}\right), \mathrm{ECB}_1\right)\\ &= \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{UCB\&ECB}}\left(\{e, f, k, l\}, \{a, b, c, d\}\right) = 4\end{aligned}$$

$$\begin{aligned}\gamma_{3,2}^{\text{ucb-u}} &= \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{UCB\&ECB}}\left(\left(\bigcup_{k \in \mathrm{aff}(3,2)} \mathrm{UCB}_k^{\cup}\right), \mathrm{ECB}_2\right)\\ &= \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{UCB\&ECB}}\left(\mathrm{UCB}_3^{\cup}, \mathrm{ECB}_1\right)\\ &= \mathrm{CRT} \cdot \mathrm{CRPD}^{\text{UCB\&ECB}}\left(\{k, l\}, \{g, h\}\right) = 2\end{aligned}$$

The UCB-Union approach derives a total preemption cost of 6, actual cost are 4. Either UCBs $\{e, f, k, l\}$ are evicted once (nested preemption) or UCBs $\{k, l\}$ are evicted twice.



	UCB_i	ECB_i
τ_1	(a, b, c, d)	(a, b, c, d)
τ_2	$(e, f, -, -)$	(e, f, g, h)
τ_3	$(-, -, k, l)$	(i, j, k, l)

Figure 8.3.: Pessimism of the UCB-Union approach. The task set consists of 3 tasks $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = C_3 = 2$, and CRT $=$ 1. For the response time R_3, Equation (8.11) computes total preemption costs of 6, whereas the actual cost is only 4.

Although stated otherwise in [3], UCB-Union does not necessarily dominate ECB-Only. This only holds if for each task, there is at least one program point such that all UCBs of the whole task are also UCBs at this program point, i.e., for the simplified model of one set of UCBs per task.

$$\forall_i \exists p : \mathrm{UCB}_i(p) = \mathrm{UCB}_i^{\cup}$$

8.1.3. Multiset Approaches

Instead of assigning each job of a task the preemption cost this job inflicts, Petters et al. [68] and later also Staschulat et al. [81] assign an overall bound to each task directly: $\gamma_{i,j}$ does not refer to the cost of a single preemption by task τ_j but to the cost of all preemptions of task τ_j during the response time R_i of task τ_i. So, Equation (8.3) is used to compute the response time analysis in this case. To simplify the notation in the following equations, we use

$$E_k(R_j) = \left\lceil \frac{R_i + J_j}{T_j} \right\rceil \tag{8.12}$$

as a bound on the number of jobs of task τ_k during the response time R_j. Figure 8.4 illustrates the advantage of this approach. During the response time R_3, at most one job of task τ_2 ($E_2(R_3) = 1$) is activated and during the response time R_2 at most one job of task τ_1 ($E_1(R_2) = 1$). Assume that solely task τ_1 preempting task τ_2 inflicts additional cache reloads. Only if we assign preemption cost to each task and not to each job, we can correctly take into account that the preemption cost of τ_1 preempting τ_2 delays the finishing time of a job of task τ_3 at most once.

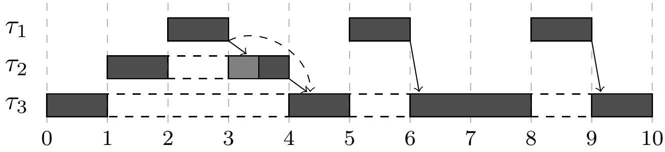


Figure 8.4.: Advantages of the Multiset approach: Assigning preemption cost to the jobs of a task (instead of to the task directly) may lead to an over-approximation.

Petters' Approach

First, we build a *multiset M* containing the cost of each preemption as often as this preemption can occur. As Petters et al. [68] use only the set of UCBs of the preempting task as a bound on the CRPD, we have the following equation:

$$M = \bigcup_{k \in \mathrm{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} |\mathrm{CRPD}^{\text{UCB}}(\mathrm{UCB}_k)| \right) \tag{8.13}$$

Since at most $E_k(R_i)$ jobs of task τ_k can be released during the response time R_i, preemption cost of τ_j preempting τ_k can contribute at most $E_j(R_k)E_k(R_i)$ times.

The cost of all preemptions by jobs of task τ_j during the response time R_i is then given by the $E_j(R_i)$ largest values of M:

$$\gamma_{i,j}^{\text{ecb-m}} = \mathrm{CRT} \cdot \sum_{l=1}^{E_j(R_i)} |M^l| \tag{8.14}$$

where M^l denotes the l-th largest element from the multiset M.

Staschulat's Approach

Staschulat et al. [81] proposed to exploit the fact that each additional preemption of task τ_i may result in a smaller preemption cost than the last one: The worst-case preemption point is given by the program point with the largest number of UCBs. For a second preemption, program point with the second largest number of UCBs can be assumed, for the third preemption the third largest and so on. Furthermore, Staschulat et al. [81] aim for an inclusion of the precise cost of τ_j preempting τ_k. Hence, they proposed to define the multiset as follows:

$$M = \bigcup_{k \in \mathrm{aff}(i,j} \left(\bigcup_{E_k(R_i)} \left\{ \mathrm{CRPD}^{\text{UCB\&ECB}(l)}(\mathrm{UCB}_k, \mathrm{ECB}_j) | n \in [1; E_j(R_k)] \right\} \right) \tag{8.15}$$

where $\mathrm{CRPD}^{\text{UCB\&ECB}(l)}(\mathrm{UCB}_k, \mathrm{ECB}_j)$ denotes the preemption cost for the l-th preemption of task τ_k by τ_j. In order to correctly account for nested preemptions, Staschulat et al. [81] increase the number of elements taken from the set M. They need to account for each job of a task that may suffer eviction by the preempting task τ_j. Hence, they do not only consider the $E_j(R_i)$ highest preemption cost, but the q highest ones, where q is given by the number of jobs of tasks from the set $\text{aff}(i, j)$ that are executed during the response time R_i:

$$q = \sum_{\forall k \in \text{aff}(i,j)} E_k(R_i) \tag{8.16}$$

with Equation (8.17) to compute $\gamma_{i,j}^{\text{sta}}$:

$$\gamma_{i,j}^{\text{sta}} = \text{CRT} \cdot \sum_{l=1}^{q} |M^l| \tag{8.17}$$

where M^l is the l-th largest element from the multiset M.

Improvement of the l-th preemption does not compensate for the over-approximation of the number of preemptions taken into account. As our evaluation and other measurements have shown [14], considering the l-largest CRPD bound in case of the l-th preemption only improves over the largest CRPD bound in case of very large l. First, program points close to each other have a similar number of useful cache blocks. They may not only be one single preemption point with the maximal number of UCBs but plenty. Second, preemption points within a loop have to be considered as often as the loop iterates.

8.2. ECB Union

We have this far seen different approaches to derive the response time of a task. Especially accounting for nested preemption contains several pitfalls. The UCB-Union approach provides a sound response time analysis but suffers (besides other pessimism) from the over-approximation of the set of useful cache blocks $\text{UCB}^{\cup}$. We will now present the ECB-Union approach that complements UCB-Union.

Instead of considering the precise set of ECBs of a preempting task and bounding all possibly affected UCBs (as UCB-Union does), ECB-Union considers the precise number of UCBs of the preempted task. It then always assumes that the preempting task τ_j has itself already been preempted by all tasks with higher priority. This nested preemption of the preempting task is represented by the union of the ECBs of all tasks with higher priority than τ_j:

$$\gamma_{i,j}^{\text{ecb-u}} = \max_{\forall k \in \text{aff}(i,j)} \left\{ \text{CRPD}^{\text{UCB\&ECB}} \left(\text{UCB}_i, \left(\bigcup_{h \in \text{hep}(j)} \text{ECB}_h \right) \right) \right\} \tag{8.18}$$

Similarly to the UCB-Only approach, we need to compute the maximal CRPD cost over the set $\text{aff}(i,j)$, as task τ_j may preempt any task with lower priority and all tasks with priority higher than or equal to τ_i may increase the response time R_i. So, $\gamma_{i,j}^{\text{ecb-u}}$ represents the preemption cost of a job of τ_j together with all tasks with priority higher than τ_j preempting any job τ_h with intermediate priority: $j < h \leq i$. ECB-Union is a direct improvement over UCB-Only and dominates this approach. However, it is incomparable to UCB-Union. There are examples such that one approach outperforms the other and vice-versa. For the task set depicted in Figure 8.3, the ECB-Union approach correctly computes a precise CRPD bound of 2. As each of the two tasks τ_1 and τ_2 occupy all 4 cache sets alone, computing the union of ECB_1 and ECB_2 does not introduce any pessimism.

$$\begin{aligned}\gamma_{3,1}^{\text{ecb-u}} &= \max_{\forall k \in \{2,3\}} \left\{ \text{CRPD}^{\text{UCB\&ECB}} \left(\text{UCB}_i, \left(\bigcup_{h \in \text{hep}(j)} \text{ECB}_h \right) \right) \right\} \\ &= \max\{\text{CRPD}^{\text{UCB\&ECB}}\left(\text{UCB}_2, \text{ECB}_1\right), \\ &\qquad \text{CRPD}^{\text{UCB\&ECB}}\left(\text{UCB}_3, \text{ECB}_1 \cup \text{ECB}_2\right)\} \\ &= \max\{2, 2\} = 2\end{aligned}$$

$$\begin{aligned}\gamma_{3,2}^{\text{ecb-u}} &= \max_{\forall k \in \{3\}} \left\{ \text{CRPD}^{\text{UCB\&ECB}} \left(\text{UCB}_i, \left(\bigcup_{h \in \text{hep}(j)} \text{ECB}_h \right) \right) \right\} \\ &= \max\{\text{CRPD}^{\text{UCB\&ECB}}\left(\text{UCB}_3, \text{ECB}_1 \cup \text{ECB}_2\right)\} \\ &= \max\{2\} = 2\end{aligned}$$

Figure 8.5 presents a task set, for which ECB-Union over-approximates the preemption cost (and UCB-Union is precise). The latter approach assumes that UCBs i and k are evicted twice although only task τ_1 can evict these UCBs and τ_1 is executed at most once during the response time of τ_3.

$$\begin{aligned}\gamma_{3,1}^{\text{ecb-u}} &= \max_{\forall k \in \{2,3\}} \left\{ \text{CRPD}^{\text{UCB\&ECB}} \left(\text{UCB}_i, \left(\bigcup_{h \in \text{hep}(j)} \text{ECB}_h \right) \right) \right\} \\ &= \max\{\text{CRPD}^{\text{UCB\&ECB}}\left(\text{UCB}_2, \text{ECB}_1\right), \\ &\qquad \text{CRPD}^{\text{UCB\&ECB}}\left(\text{UCB}_3, \text{ECB}_1 \cup \text{ECB}_2\right)\} \\ &= \max\{0, 2\} = 2\end{aligned}$$

$$
\begin{aligned}
\gamma_{3,2}^{\text{ecb-u}} &= \max_{\forall k \in \{3\}} \left\{ \mathrm{CRPD}^{\text{UCB\&ECB}} \left(\mathrm{UCB}_i, \left(\bigcup_{h \in \mathrm{hep}(j)} \mathrm{ECB}_h \right) \right) \right\} \\
&= \max\{\mathrm{CRPD}^{\text{UCB\&ECB}} \left(\mathrm{UCB}_3, \mathrm{ECB}_1 \cup \mathrm{ECB}_2\right)\} \\
&= \max\{4\} = 4
\end{aligned}
$$

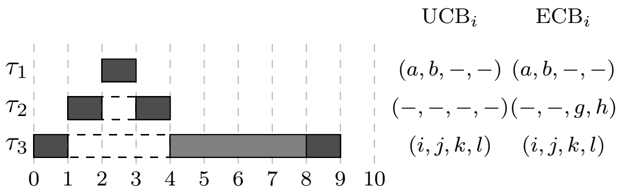

UCB$_i$
ECB$_i$
τ_1
τ_2
τ_3
$(a, b, -, -)$ $(a, b, -, -)$
$(-, -, -, -)$ $(-, -, g, h)$
(i, j, k, l) (i, j, k, l)
0
1
2
3
4
5
6
7
8
9
10

Figure 8.5.: Pessimism of the ECB-Union approach: The task set consists of 3 tasks $\{\tau_1, \tau_2, \tau_3\}$ with $C_1 = 1$, $C_2 = C_3 = 2$ and CRT $= 1$. Equation (8.18) computes for R_3 a total preemption cost of 6, whereas the actual cost is only 4.

Resilience Analysis

Resilience analysis as presented in Chapter 7 fits directly into the computation of $\gamma^{\text{ecb-u}}$. We just need to replace the former CRPD$^{\text{UCB\&ECB}}$ bound by CRPD$^{\text{RES}}$ (Equation (7.9)):

$$
\gamma_{i,j}^{res} = \max_{k \in \mathrm{aff}(i,j)} \left\{ \mathrm{CRPD}^{\text{RES}} \left(\mathrm{UCB}_i, res_i, \left(\bigcup_{h \in \mathrm{hep}(j)} \mathrm{ECB}_h \right) \right) \right\} \tag{8.19}
$$

In Chapter 7, we have discussed problems of the resilience analysis in case of multiple preemptions by different tasks. Fortunately, Equation (8.19) always considers nested preemption by all tasks with higher or equal priorities. Hence, the computed CRPD bounds based on resilience are safe upper bounds on the actual preemption cost.

Note that the resilience analysis is not applicable to the UCB-Union approach where we have to assume that one ECB suffices to displace all useful cache blocks.

8.3. Multiset Approaches

As discussed before, assigning the complete preemption cost to a task instead of to a job of the task reduces further pessimism. In addition, we now have two approaches, ECB-Union and UCB-Union, at hand that can correctly and precisely consider both sides, preempting and preempted task. Thus, we extend ECB-Union and UCB-Union to the corresponding multiset approaches.

Let $\text{Cost}_{i,j}$ denote the cost for a job of task τ_j preempting task τ_i. This cost may increase the response time of task τ_i up to $E_j(R_k)E_k(R_i)$ times (for each $k \in \text{aff}(i,j)$). The multiset M represents the cost for all possible preemptions of τ_j that may increase the response time of task τ_i

$$M = \bigcup_{k \in \text{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} \text{Cost}_{k,j} \right) \tag{8.20}$$

$\gamma_{i,j}^{\text{multiset}}$ is then given by the $E_j(R_i)$ largest values in M.

$$\gamma_{i,j}^{\text{ecb-m}} = \text{CRT} \cdot \sum_{l=1}^{E_j(R_i)} |M^l| \tag{8.21}$$

where M^l is the l-th largest value in M.

The multiset approaches then just differ by how $\text{Cost}_{k,j}$ is computed. Note that by construction, any multiset approach dominates its basic counterpart.

ECB-Union Multiset

$$\text{Cost}_{k,j}^{\text{ecb-m}} = \text{CRPD}^{\text{UCB\&ECB}} \left(\text{UCB}_k, \left(\bigcup_{h \in \text{hep}(j)} \text{ECB}_h \right) \right) \tag{8.22}$$

ECB-Union Multiset with Resilience

$$\text{Cost}_{k,j}^{\text{ecb-m-res}} = \text{CRPD}^{\text{RES}} \left(\text{UCB}_k, res_k, \left(\bigcup_{h \in \text{hep}(j)} \text{ECB}_h \right) \right) \tag{8.23}$$

UCB-Union Multiset

$$\text{Cost}_{k,j}^{\text{ucb-m}} = \text{CRPD}^{\text{ucb-u}} \left(\left(\bigcup_{h \in \text{aff}(k,j)} \text{UCB}_h^{\cup} \right), \text{ECB}_j \right) \tag{8.24}$$

Figure 8.6 depicts a task set for which the multiset approaches (ECB-Union Multiset and UCB-Union Multiset) outperform their basic counterparts. Assume that task τ_2 is executed at most once during the response time R_3, i.e., $E_2(R_3) = 1$; task τ_1 at most once during the response time R_2, i.e., $E_1(R_2) = 1$; but τ_1 up to 3 times during the response time R_3, i.e., $E_1(R_3) = 3$. UCB-Union and ECB-Union derive the correct upper bound of 2 for the preemption cost of a job of task τ_3. As $E_1(R_3) = 3$, this value is assumed to contribute three times to the response time R_3 by Equation (8.2) resulting in total preemption overhead of 6. As this upper bound 2 is added at most once to the multiset M, the multiset approaches derive precise overall preemption cost of 2

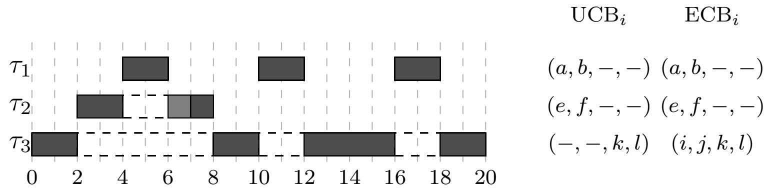


Figure 8.6.: Pessimism of the ECB-Union/UCB-Union approaches.

8.4. Resource Access Protocols and Preemption Cost

We now extend preemption cost aware response time equations to handle *mutually exclusive* accesses to *shared resources*. We assume these accesses to be scheduled according the *stack resource protocol (SRP)* (see [9] or Section 3.1.3). Response time analysis is extended by a blocking value B_i denoting the *maximal blocking* task τ_i is subject to. Blocking time B_i is given by the maximum execution time for which any task $k \in \mathrm{lp}(i)$ holds a resource shared with a task $j \in \mathrm{hep}(i)$ with priority higher than or equal to i (See Equation (3.19) in Section 3.1). Equation (8.25) includes blocking factor in the preemption cost aware response time analysis.

$$R_i = C_i + B_i + \sum_{\forall j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil (C_j + \gamma_{i,j}) \tag{8.25}$$

This equation (or the multiset equivalent) has been proposed in previous work [18, 50, 81, 85]. Except for the ECB-Only approach [18], Equation (8.25) may be optimistic. Consider the example depicted in Figure 8.7. Tasks τ_2 and τ_3 share a common resource x. Task τ_3 starts to execute and enters its critical section at 1. When task τ_2 is released at time 2, its execution is blocked by τ_3. Task τ_1 is released and executed at time 3, preempts τ_3 during its access to the shared resource, and evicts UCB i and j. Hence, not only the non-preemptive execution time of the access to the shared resource, but also the additional preemption delay of task τ_1 preempting the resource access of τ_3 delays the finishing time of τ_2. Note that the basic *priority ceiling protocol (PCP)* allows a task to execute until it tries to access the shared resource and then blocks the task. The *stack resource protocol (SRP)* directly prevents task activation until all of its resources are available. We have based the example on the latter one as it exhibit less preemptions and thus, is more apt to be used in practise. Yet, explanation and formulas are valid for both protocols.

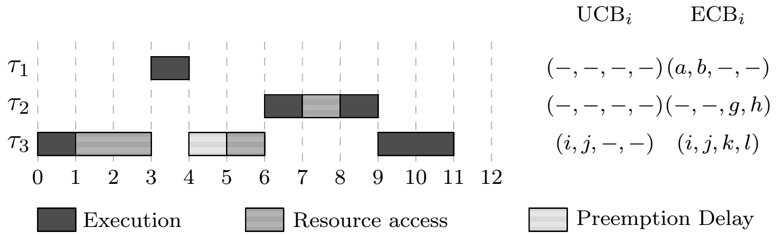


Figure 8.7.: Ganttchart demonstrating the optimism of the naive blocking time aware response time analyses: Preemption delay occurs during resource access of task τ_3. Finishing time of τ_2 is thus not only delayed by the B_i but also by the additional preemption delay τ_1 preempting τ_3.

Adding blocking factor B_i is only sufficient if the preemption costs are computed by the ECB-Only approach which assumes that each ECB evicts the worst-case number of UCBs—no matter which task is preempted. All other approaches also consider the preempted task. Thus, we have to extend the set $\text{aff}(i,j)$ of tasks that can affect the response of task τ_i and may be preempted by task τ_j. Without accesses to a shared resource, $\text{aff}(i,j)$ is defined as the set of tasks with priority lower than τ_j and higher than or equal to τ_i: $\text{aff}(i,j) = \text{lp}(j) \cap \text{hep}(i)$. In case we allow accesses to shared resources, we also have to consider tasks with priority lower than τ_i that share a resource *res* guarded by a *semaphore* S_{res} with ceiling priority $C(S_{res})$ higher than or equal to τ_i but lower than τ_j (as we are only interested in tasks that may be preempted by τ_j). The set of these tasks is denoted by $b(i,j)$ and defined as follows:

$$b(i,j) = \{l \mid \exists cs_{l,res} : l \in \text{lp}(i) \land C(S_{res}) \in \text{aff}(i,j)\} \tag{8.26}$$

where res is a resource, $cs_{l,res}$ a critical section of task τ_l accessing res and S_{res} the guarding semaphore. Using Equation (8.26), we can define the extended set of affecting tasks $\mathrm{aff}^{\mathrm{b}}(i,j)$ that includes the preemption cost during task blocking:

$$\mathrm{aff}^{\mathrm{b}}(i,j) = (\mathrm{lp}(j) \cap \mathrm{hep}(i)) \cup b(i,j) \tag{8.27}$$

Regarding the example from Figure 8.7, we see that response time analyses using UCB-Only (8.5), UCB-Union (8.11), and ECB-Union (8.18) now consider also the preemption cost of τ_1 preempting τ_3 for the computation of R_2.

For the multiset approaches (8.24), (8.22) and (8.23), it remains to discuss how often jobs of the blocking task may contribute to the response time. Let τ_h be the task blocking task τ_i. The priority ceiling protocol as well as the stack resource protocol ensure that each task is blocked at most once by a task of lower priority. Hence, at most one job of τ_h needs to be considered. Furthermore, we can bound the number of jobs of τ_j during blocking access of τ_h by $E_j(R_i)$.

$$M^{\mathrm{b}} = \bigcup_{k \in \mathrm{aff}(i,j)} \left(\bigcup_{E_j(R_k)E_k(R_i)} \mathrm{Cost}_{k,j} \right) \cup \bigcup_{k \in b(i,j)} \left(\bigcup_{E_j(R_i)} \mathrm{Cost}_{k,j} \right) \tag{8.28}$$

Chapter 9

Evaluation

> It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong.
> *Richard P. Feynman (1918 - 1988)*

In this chapter we evaluate the precision of the methods and analyses presented in this thesis. We compare the DC-UCBs analysis against the original concept, the resilience analysis against the simple combination of UCBs and ECBs (and against the unsound CRPD bound of [86]) and also the different CRPD-aware response time analyses against each other.

We start with a discussion of the target architecture (**ARM7**) and the set of benchmarks (Mälardalen Benchmark suite, Papabench).

9.1. Target Architecture

We use an **ARM7** processor[1]. The **ARM7** is a simple, low cost 32-bit RISC architecture. It is used in a wide area of embedded systems such as cell phones, game consoles, pocket calculators but also in environments with hard real-time constraints such as in the automotive industry. It features a three stage pipeline (fetch, decode and execute).

[1] `http://www.arm.com/products/CPUs/families/ARM7Family.html`

Note that **ARM7** does not refer to one specific processor but to a family of processor designs (of which ARM7-TDMI is probably the most important) that can be licensed by semiconductor companies. **ARM** processor thus exist in different variation and with different caches. We assume a frequency of 100 MHz and memory latency of $8\mu s$. We have selected three cache configurations as depicted in Table 9.1. As only benchmarks of limited sizes are available, we have also selected small caches of 4kB and 8kB size. For the sake of simplicity, we consider instruction caches only and assume perfect data caches, i.e., each data access is served in 1 cycle. Although this assumption is unrealistic, it allows to take precise information into account. For instance, the set of ECBs is determined by the code size and does not require any further analysis and the effective address of each memory access is static. We furthermore assume that the remaining *context switch costs*, i.e., pipeline and scheduler related costs are subsumed in the execution time bound of each task. Due to its simple structure, the **ARM7** is considered timing anomaly free [98]. Cache-miss penalty is thus determined solely by the memory latency.

Table 9.1.: Selected cache configurations

	Cache Type	Sets	Line Size	Total Size
Config 1:	direct-mapped	256	16 Byte	4kB
Config 2:	4-way LRU	64	16 Byte	4kB
Config 3:	8-way LRU	64	16 Byte	8kB

9.2. Benchmarks

The Mälardalen Benchmark Suite [34] is a set of 32 benchmarks dedicated to the evaluation of WCET tools. It can be considered the standard benchmark suite for timing analyses. It consists of various test programs and covers a wide range of different programs especially designed for embedded systems. Table 9.2 presents a subset of the Mälardalen benchmark suite with a short description of the tasks (according to [34]), code sizes and execution time bounds. Note that the timing bounds are valid for each cache configuration. Due to the limited sizes of the benchmarks, no intra-task cache eviction occurs. In fact all but one benchmark completely fit into a cache of size $4kB$. These programs however do not form a meaningful task set. We therefore also included PapaBench [65], a set

Table 9.2.: Mälardalen Benchmark Suite

Task	Description	Code Size	WCET
minmax	Derives min/max of set of integers	$608B$	$298\mu s$
insertsort	Insertion sort on array of size 10.	$384B$	$223\mu s$
fibcall	Iterative Fibonacci calculation).	$256B$	$117\mu s$
fac	Non-recursive Faculty Computation	$256B$	$67\mu s$
bs	Binary search (array of size 15).	$320B$	$1.46ms$
bsort100	Bubblesort program.	$544B$	$11.3ms$
ns	Test for deeply nested loops	$576B$	$637\mu s$
matmult	Matrix multiplication (20x20).	$864B$	$8.5ms$
fir	Finite impulse response filter.	$928B$	$465\mu s$
crc	Cyclic redundancy check .	$1216B$	$2.7ms$
select	Selects Nth largest number.	$1280B$	$757\mu s$
qsort-exam	Non-recursive quick sort algorithm.	$1440B$	$800\mu s$
sqrt	Square root by Taylor series.	$3680B$	$2.1ms$
qurt	Computation of quadr. equations.	$4160B$	$2.9ms$

of benchmarks based on a real-time application, the control software of an unmanned aircraft vehicle. It consists of a set of tasks distributed statically over two processors (MCU0 and MCU1) and executed either in *manual* or *automatic* flight control mode. For the evaluation, we have selected the largest of the possible configurations of PapaBench benchmarks, i.e., tasks in manual mode on MCU0. Table 9.3 presents a short description of these tasks, the periods of the tasks (according to [65]), code size and execution time bound. Again, timing bounds are valid for each cache configuration (due to the limited sizes of the benchmarks). All benchmarks have been compiled using the gcc arm cross-compiler[2]. The execution time bounds have been derived by the **aiT timing analyzer**[3] for **ARM7**. The **ARM**-Processor has no floating point unit, floating point arithmetic is instead implemented in software. Tasks that rely on such computations, *squrt* for instance, require library functions and thus have an increased code size.

9.3. DC-UCB Analysis

We start the evaluation with the analysis of the preempting task and identify the improvement of the DC-UCB analysis compared to the original definition of useful cache blocks [49]. The results of the DC-UCB analysis strongly depend on the precision of the *cache analysis*. If no

[2] http://www.gnuarm.com/

[3] http://www.absint.com/

Table 9.3.: Papabench Benchmark Suite (Processor *MCU0*, Automatic Mode)

Task	Description	Period	Priority	Code Size	WCET
I5	interrupt_spi_1	$50ms$	1	$304B$	$129\mu s$
I6	interrupt_spi_2	$50ms$	2	$144B$	$68\mu s$
T12	stabilization	$50ms$	3	$2976B$	$3.2ms$
I4	interrupt_modem	$100ms$	4	$288B$	$148\mu s$
T11	reporting_task	$100ms$	5	$5360B$	$5.9ms$
T10	receive_gps_data	$250ms$	6	$3008B$	$3ms$
T7	link_fbw_send	$250ms$	7	$192B$	$105\mu s$
T6	climb_control	$250ms$	8	$3296B$	$3.4ms$
T5	altitude_control	$250ms$	9	$1232B$	$826\mu s$

memory accesses can be classified as *always hit*, the set of DC-UCBs will be empty. To avoid such an unfair comparison, we have employed *virtual inlining* and *loop unrolling* to improve the precision of the cache analysis. Figure 9.1 and Figure 9.2 show the maximal number of UCBs/DC-UCBs at some program point of each task. Note that results are the same for all cache configurations as no intra-task cache eviction occurs.

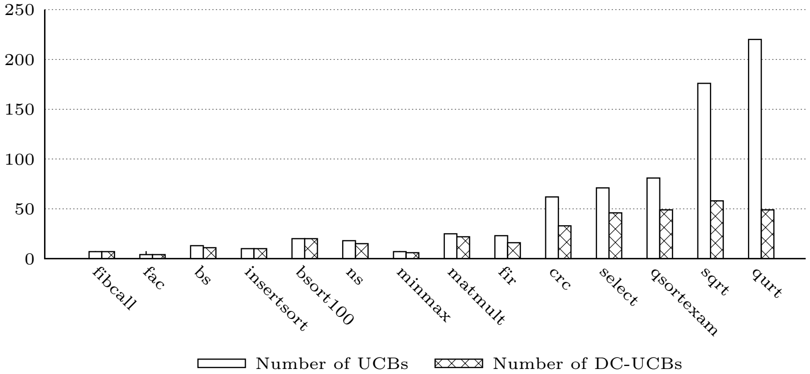


Figure 9.1.: Maximal number of UCBs/DC-UCBs; Mälardalen Benchmarks

In general, the number of UCBs and DC-UCBs strongly depend on the effectiveness of the *temporal locality* of a cache and thus, on the structure of the tasks: multiple function invocations and loop structures result in a high number of UCBs. Straight-line code on which each instruction is executed at most once results in a low number of UCBs. While nearly all tasks from Mälardalen Benchmark suite contain loops, only 3 of the

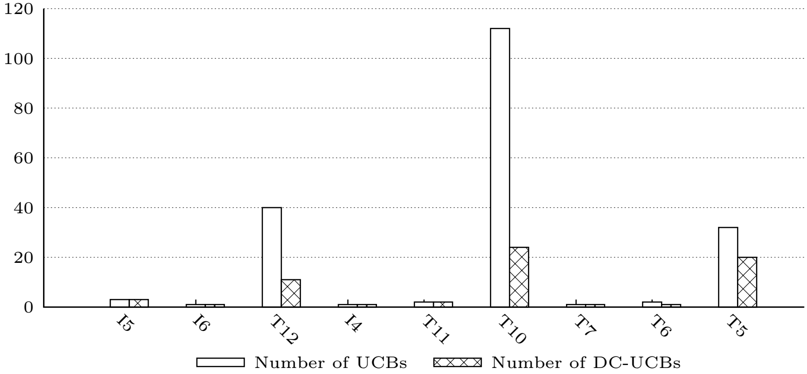


Figure 9.2.: Maximal number of UCBs/DC-UCBs; PapaBench

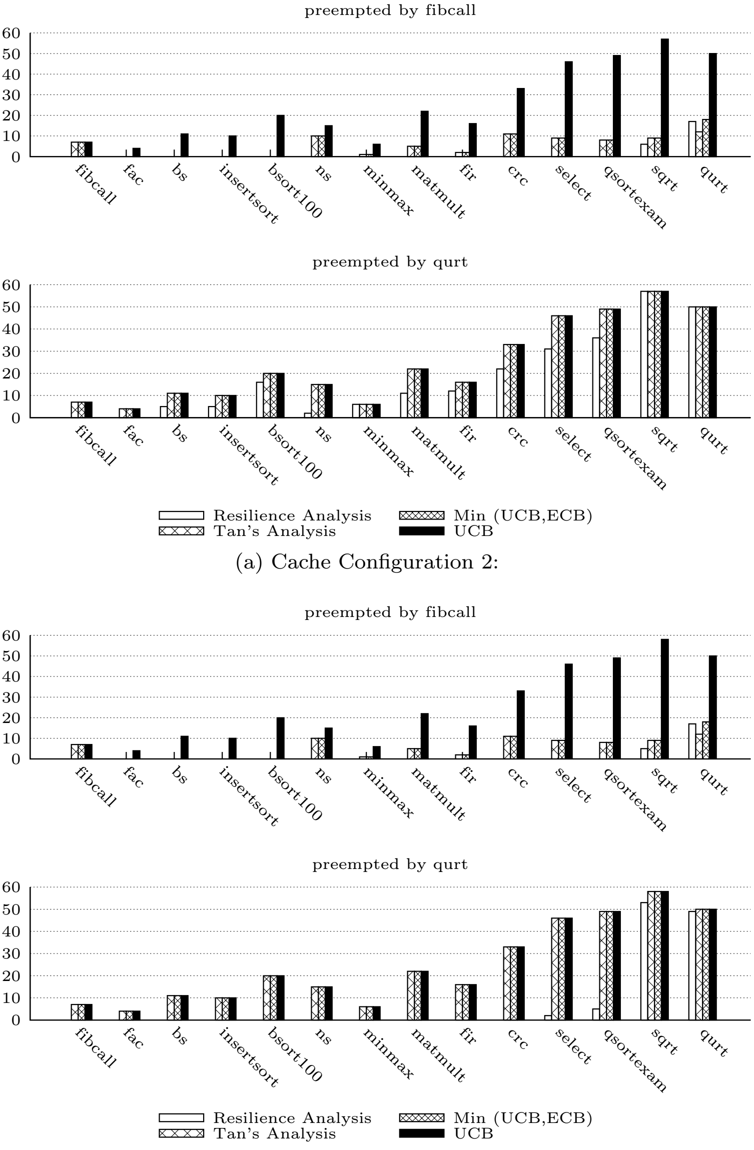
9 tasks from PapaBench do. This explains the stronger variation of the results for PapaBench and the low number of UCBs for some tasks.

The original concept of useful cache blocks is highly pessimistic, especially for larger tasks that spent considerable execution time in loops (*squrt*, *qurt* and *T10*, *T12*). The concept of definitely-cached useful cache blocks reduces the bounds by up to 80%. Improvement decreases as the overall number of useful cache blocks decrease. Note that due to spatial locality, i.e., cache line size larger than instruction size, nearly each memory block is useful at some point in the task.

9.4. Resilience Analysis

Figure 9.1 and Figure 9.2 show the preemption costs only considering the preempted task. We now combine these bounds with the effect of the preempting task and evaluate the precision of the different methods. We therefore compare the *resilience* analysis to the bounds on the cache-related preemption delay based on i) the set of UCB (Equation (5.2)), ii) the simple combination of UCBs and ECBs (Equation (7.2)) and iii) the unsound combination by Tan and Mooney (Equation (5.9)). Note that all results are based on the set of definitely cached UCBs.

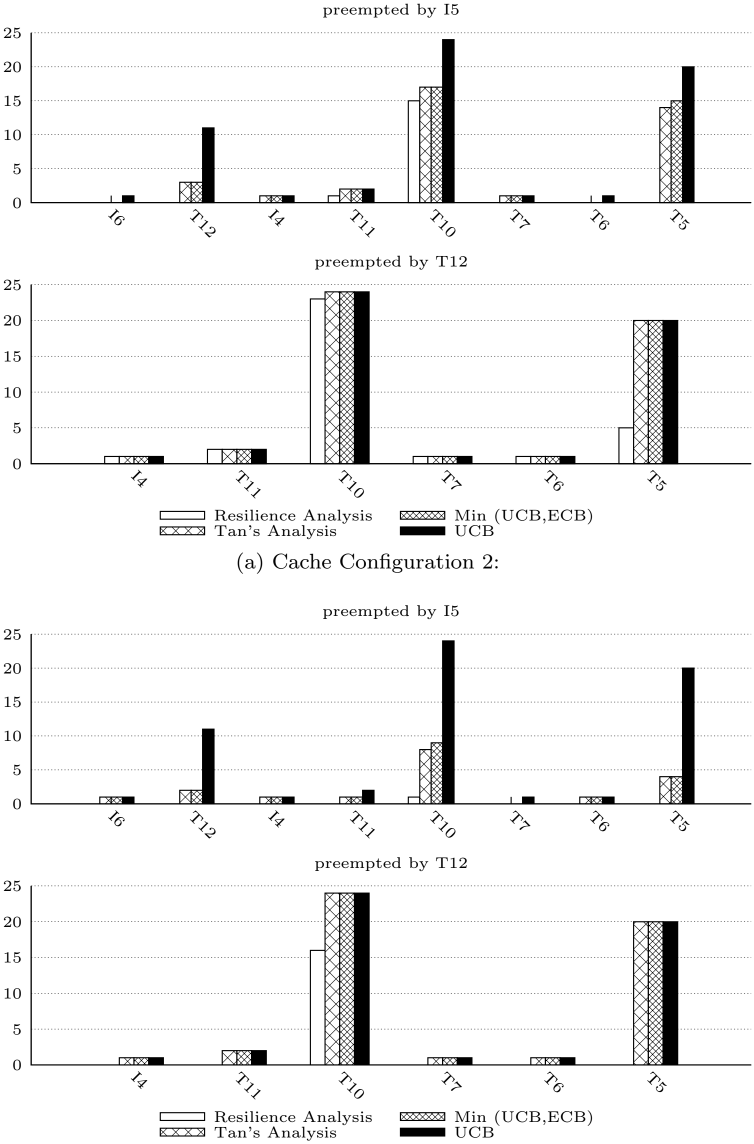
In our first setting, we use benchmarks from the Mälardalen benchmark suite to evaluate the precision of the different methods in case of a single preemption. We have selected *fibcall* (smallest number of ECBs) and qurt (highest number of ECBs) as the preempting tasks. Figure 9.3 shows the results. If tasks are preempted by *fibcall*, resilience analysis



(a) Cache Configuration 2:

(b) Cache Configuration 3:

Figure 9.3.: Bound on the number of additional misses; Mälardalen Benchmarks



(a) Cache Configuration 2:

(b) Cache Configuration 3:

Figure 9.4.: Bound on the number of additional misses; PapaBench

is able to prove in all but two cases (*sqrt* and *qurt*) that no additional misses occur. All other analyses are more pessimistic. Only for *qurt*, Tan's analysis derives a smaller but possibly optimistic bound than the bound derived by the resilience analysis. In this scenario, the results are independent of the associativity, i.e., results do not vary when changing the associativity from 4 to 8. Tasks *qurt* evicts more useful cache block and thus, preemption by task *qurt* results in higher values. Only the resilience analysis benefits from a higher associativity (see for instance *select* and *qsortexam*). Results of the other analyses do not change, while the bound on the number of additional misses based on the resilience analysis drops from 31 to 2 (*select*) and from 36 to 5 (*qsortexam*).

In the second setting, we assume that $I5$, resp. $T12$, preempts any task with lower priority. Figure 9.3 shows the result. We can draw similar conclusion from this scenario. Resilience analysis substantially improves the bounds on the number of additional misses. For instance, resilience analysis derives a bound of 10 additional misses for task T5 and cache configuration 2. All other analyses assume that all UCBs (20) are evicted. Results improve even further for cache configuration 3; resilience analysis is able to prove that no additional misses occur whereas bounds of the other analysis do not change.

9.5. CRPD Aware Scheduling Analysis

As the Mälardalen benchmarks do not form a meaningful task set, we evaluate the precision of the various CRPD-aware response time analyses for the PapaBench benchmark suite and for randomly generated task sets. We have derived the response times for each task and each cache configuration based on the different analyses presented in Chapter 8. Bounds on the response time of task T10 and T5 for the different approaches and different cache configurations are shown in Figure 9.5. Due to the low processor utilization of the task set, only very few distinct preemption scenarios can occur and each task preempts a lower priority task at most once. Hence, the multiset approaches result in the same response time as their basic counterparts. We have therefore omitted theses results in the graph. Despite the low processor utilization and the limited number of UCBs (compared to Mälardalen Benchmarks), preemption cost increase the response time by up to 10% (compared to response time ignoring preemption cost). In all cases, ECB-Union performs best and, if applicable, resilience analysis improves the results even further. The resilience analysis is also the only analysis that benefits from an increased

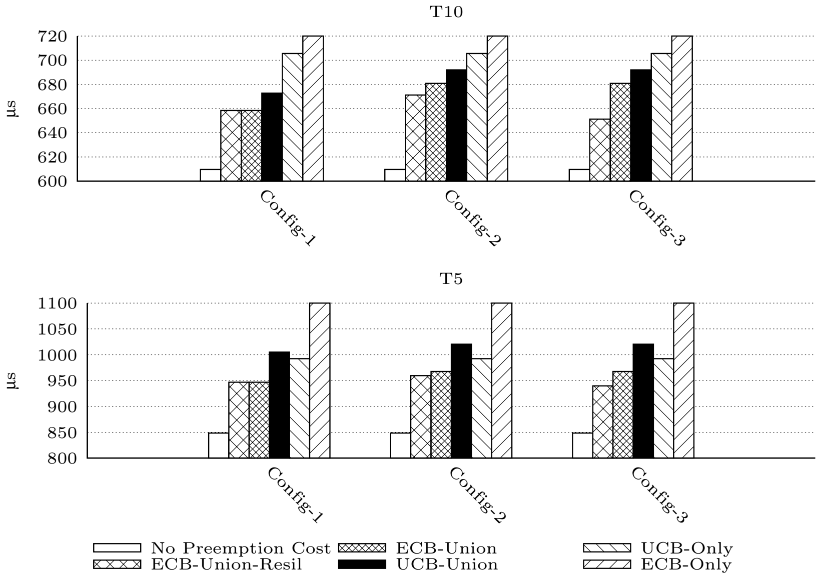


Figure 9.5.: Bounds on the response time of T5 and T10 (in μs)

associativity.

The lowest bounds are given for cache configuration 1: in case of direct-mapped caches, one ECB evicts at most one useful cache block whereas in case of an k-associative cache, up to k additional misses may occur.

9.5.1. Randomly Generated Task Sets

In addition to the evaluation based on the concrete set, we also evaluated the precision of the different scheduling analyses based on a large number of randomly generated task sets with varying cache configurations and varying task-set parameters. The task sets were generated using the following setting:

- Default number of tasks was 10.
- Task utilizations were randomly generated (according to a uniform distribution using the UUnifast [16] algorithm).
- Task periods were generated according to a log-uniform distribution (minimal possible period of 5ms and maximal possible period of

500ms as common in automotive and aerospace hard real-time applications).

- Task execution times were set according to the generated utilization and period: $C_i = U_i \cdot T_i$.
- Implicit task deadlines, i.e., $D_i = T_i$. Note that results for constrained deadlines, i.e., $D_i \in [2C_i; T_I]$ are similar (with fewer task sets deemed schedulable by all approaches).
- Priorities were assigned in *deadline monotonic* order.

We have selected cache configuration 1, i.e., direct-mapped cache with 256 cache sets. Preemption costs were generated using the following parameters (default values given in parentheses):

- The *cache-miss penalty* (CRT $= 8\mu s$).
- Cache usage (of each task in isolation), i.e., the number of ECBs, was generated using the UUnifast [16] algorithm (assuming a total cache utilization $CU = 10$). In such a case, UUnifast may produce values larger than 1 which means a task fills the whole cache. We assumed the ECBs of each task to be consecutively arranged starting at a random cache set $s \in [0; S-1]$, i.e., from s to $s + |\text{ECB}| \mod S$ (where S is the number of cache sets).
- Number of UCBs for each task was generated according to a uniform distribution ranging from 0 to the number of ECBs times a reuse factor: $[0, RF \cdot |ECB|]$. The factor RF (= 30) can be adapted to resemble different types of applications (for instance from data oriented applications usually with little reuse to control-based applications with heavy cache reuse).

Staschulat's approach relies on the assumption that for the i-th preemption only the i-th highest number of UCBs has to be considered. To mimic such a reduction, we assumed that the number of UCBs decrease by one per preemption. The UCB computation for the benchmarks and other measurements [14] indicate that only for very high number of preemptions, a reduced number of UCBs can be assumed. Hence, reducing the number of UCBs by one per preemption must be considered optimistic and in favor of Staschulat's approach (Equation (8.17)).

In each experiment, we have varied the task-set utilization (ignoring preemption cost) from 0.025 to 0.975 in steps of 0.025 and generated 1000 task sets for each utilization value. Schedulability of those task set was then determined using the different CRPD aware response time analyses.

All graphs show the results of each approach except for the resilience-based analyses. In this section, we are only interested in the improvement due to improved response time analyses, not due to different CRPD analyses (therefore also cache configuration 1). We also added a necessary schedulability test by means of simulation. We simulated execution of tasks starting from near simultaneous release, i.e., task were released in order of priority with lowest priority first to increase the number of preemptions. The task set was deemed unschedulable if at least one task misses its deadline. This results in an upper bound on the task set schedulability including preemption cost.

Base Configurations

The results for the base configurations is depicted in Figure 9.6. For low utilizations, all methods except ECB-Only and Staschulat's approach deem most task sets schedulable whereas for high utilizations, nearly all task sets are deemed unschedulable by all approaches. For utilizations from 0.4 to 0.8, ECB-Union and the new multiset approaches strongly improve upon previous methods by about 20%. Note that the task utilizations do not include preemption cost.

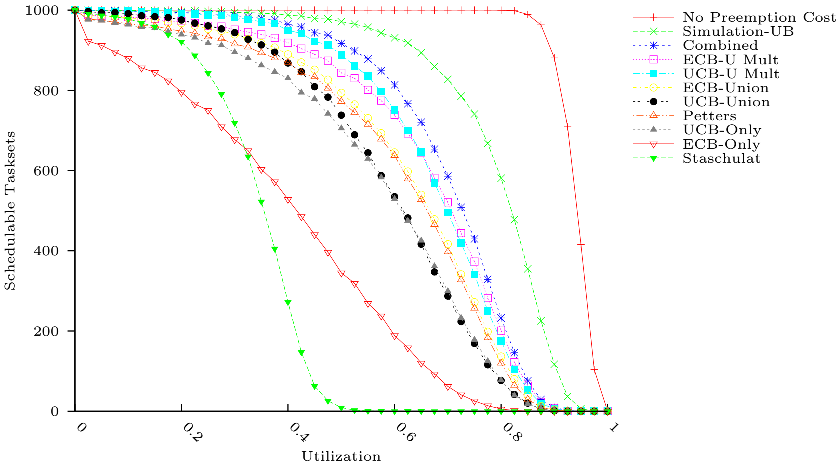


Figure 9.6.: Evaluation of the base configuration. Number of task sets deemed schedulable at the different total utilizations.

Varying Parameter

This section evaluates the sensitivity of the response time analyses to a variation of the parameters. We have fixed all parameters (to the values of the base configuration) except one and varied the remaining parameter. We used the weighted schedulability measure $W_y(p)$ [10] to show the variation of the results. The weighted schedulability measure $W_y(p)$ [10] for schedulability test y is a function of a parameter p. It combines the data for all task sets generated for a specific value of p weighted by the utilization of each task set. Let $S_y(\tau, p)$ be the binary result (1 or 0) of schedulability test y for a task set τ and parameter value p then:

$$W_y(p) = (\sum_{\forall \tau} u(\tau) \cdot S_y(\tau, p)) / \sum_{\forall \tau} u(\tau) \tag{9.1}$$

where $u(\tau)$ is the utilization of task set τ. We present results for varying *cache utilization*, *cache reuse* and *number of tasks* as those are the predominant parameters.

Cache Utilization has the strongest impact on the total cache-related preemption delay. The two extremes are i) all tasks fit into the cache, i.e., cache utilization is less than one and ii) each tasks completely fills the cache. In the first case, no additional misses due to preemption occur, and in the second case, the overall preemption delay solely depends on the number of UCBs, i.e., the cache reuse factor. Figure 9.7 shows the weighted schedulability measure for each approach as a function of the cache utilization. At a low cache utilization, only very few UCBs are evicted as the set of ECBs of each task is low. We observe that the UCB-Union (that computes an upper bound on the number of UCBs) is less pessimistic than the ECB-Union approach. With increasing cache utilization, results of the ECB-Union approach improve and finally outperform the UCB-Union approach: As each task uses a larger proportion of the cache, it proves beneficial to consider the precise set of UCB and bounding the set of ECBs (as done by ECB-Union), instead of using the precise number of ECBs and bound the number of possibly evicted UCBs (UCB-Union). The Multiset approaches perform significantly better than the basic counterparts (except for Staschulat's approach), but show a similar dependency with respect to a change of the cache utilization. Results of ECB-Union Multiset improve (and finally outperform) UCB-Union Multiset approach as the cache-utlization increases.

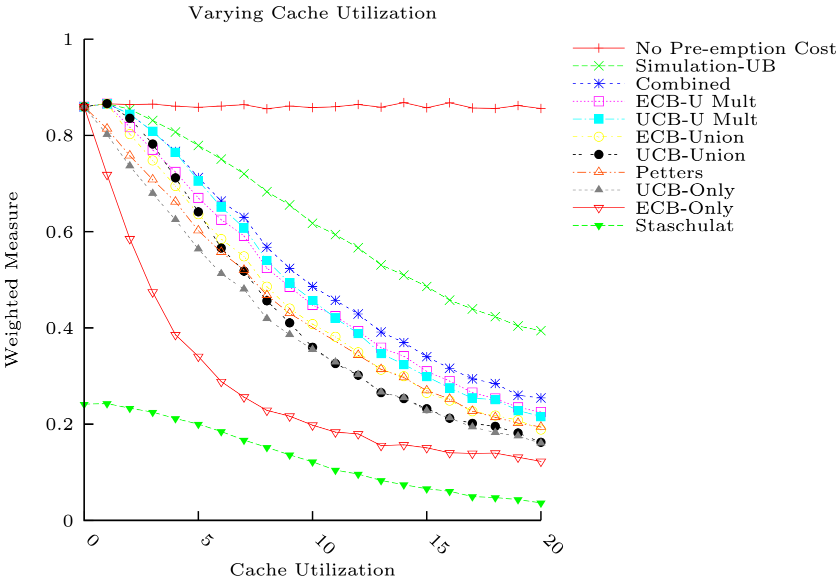


Figure 9.7.: Weighted schedulability measure; varying cache utilization from 0 to 20, in steps of 2.

Cache-Reuse influences the overall preemption delay in a similar way as the cache-utlization. If no blocks are reused, i.e., the set of UCB is empty for each task, no additional misses due to preemption occur. If all blocks are reused, the preemption delay solely depends on the number of ECBs. Figure 9.8 shows the weighted schedulability measure for each approach as a function of the reuse factor. At a low reuse factor, only very few cache blocks are useful. Considering the precise number of useful cache blocks proves beneficial in this situation and hence, ECB-Union outperforms UCB-Union. The opposite applies with an increased cache utilization. Again, the Multiset counterparts show the same behavior than the basic approaches. As the cache-reuse only affects the number of UCBs, the performance of the ECB-only approach is constant.

Number of Tasks also influences the overall schedulability. As the number of tasks increase, also the number of preemptions and so, the overall preemption cost increase. See Figure 9.9 for the weighted schedulability. Especially Staschulat's approach shows a very strong performance degradation as more jobs need to be considered. Note that the improvement

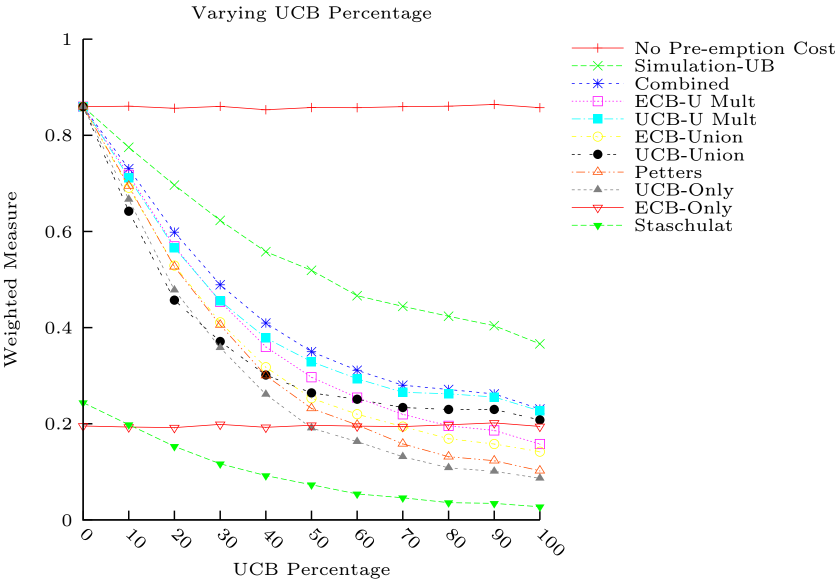


Figure 9.8.: Weighted schedulability measure; varying reuse factor from 0% to 100%, in steps of 10%.

of UCB-Union with respect to ECB-Union can be explained by the cache utilization. We have seen that UCB-Union performs best in case of low cache utilization (See Figure 9.7). As we have fixed the cache-utilization but increased the number of tasks, it becomes less likely that ECBs map to the UCBs of the preempting tasks. This resembles a situation with lower cache utilization.

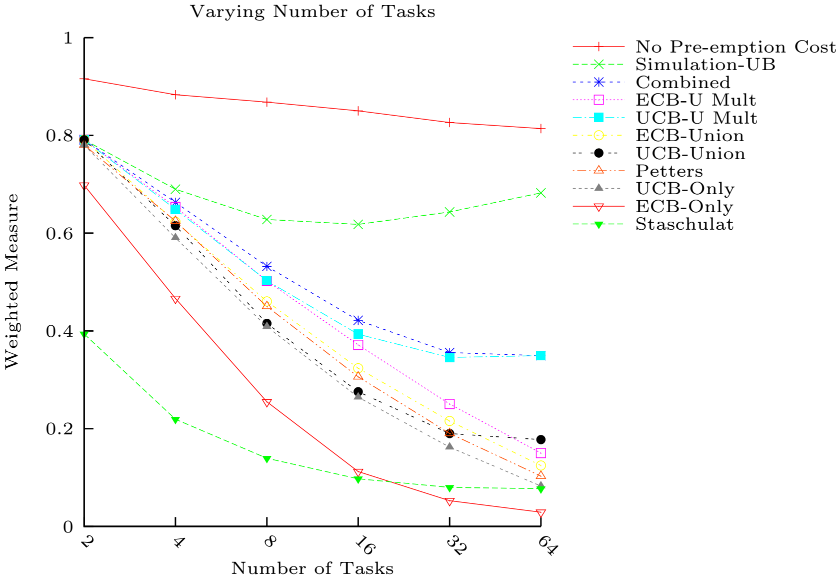


Figure 9.9.: Weighted schedulability measure; varying number of tasks from $2 = 2^1$ to $2^6 = 64$.

Chapter 10

Conclusions

Science is like sex: sometimes something useful comes out, but that is not the reason we are doing it.
Richard P. Feynman (1918 - 1988)

In preemptively scheduled systems with caches, the traditional interface between *timing analysis* and *scheduling analysis* must be considered outdated. The cache-related preemption delay may substantially prolong a task's execution time and influence the system's performance. Ignoring the CRPD is not an option, subsuming CRPD within the execution time bound is imprecise.

10.1. Summary of Contributions

This thesis presents a formal definition of the *cache-related preemption delay*, (including an analysis of the limitations of a separate CRPD computation), analyses to bound the CRPD (for direct-mapped caches, for LRU caches, based on preempted task and preempting task), and shows how to correctly account for the CRPD in schedulability analysis for *fixed-priority preemptive systems*. Evaluation has shown that the new methods strongly improve upon former approaches.

The *cache miss penalty* is unbounded for processors with *timing anomalies* and *domino effects*, the number of additional misses due to a single preemption is unbounded in case of FIFO and PLRU caches. Hence, a separate CRPD computation is only applicable to *timing composable architecture* or *compositional architectures with constant-bounded effect* and *direct-mapped* or *LRU caches.*

The concept of *definitely-cached useful cache blocks* removes substantial pessimism caused by double-counting of potential cache misses. Instead of deriving absolute bounds on the preemption cost, it suffices to consider the over-approximation of a preceding timing analysis and to derive a bound on the number of additional misses with respect to the worst-case execution time bound.

To improve the CRPD bounds, the effect of the preempting task can be taken into account. This, however, contains several pitfalls: the number of cache sets used by the preempting task is not a valid bound on the number of additional misses due to preemption in case of set-associative caches. We have corrected prior optimistic bounds and introduced in this thesis the concept of *resilience* of a useful cache block. The resilience is a measure for the disturbance of a preempting task a useful cache block of the preempted task may survive.

CRPD aware response time analysis needs to correctly account for nested preemption. This is comparably simple if one focuses on one side only, i.e., only on the preempting or alternatively only on the preempted task. To incorporate both sides and thus to include precise CRPD bounds, one can either consider the precise effect of the preempting task and upper bound the impact to all preempted tasks (UCB-Union [86]) or consider the precise impact on a preempting task and upper bound the effect of all possibly preempting tasks (ECB-Union). These analyses can be improved even further by considering the total effect of a preempting task on the response time of another task and not just by assigning each job of the preempting task a preemption delay (multiset approaches). Last but not least, we have shown how to extend the CRPD-aware response time analyses to systems with *mutual exclusive access* to *shared resources.*

Each of the new methods presented in this thesis strongly improve upon prior analysis. Furthermore, we have proven the correctness of the CRPD bounds based on definitively-cached useful cache blocks and resilience.

10.2. Future Work

Despite the undeniable impact of the cache-related preemption delay to the overall performance of preemptively scheduled systems, most research still abstracts from this low-level behavior and assumes a single execution time bound. It remains to determine how pessimistic this assumption is and how results may change when considering preemption delays. Some scheduling schemes such as deferred preemption, preemption thresholds or FP-FIFO scheduling may not only allow for a precise integration, but also for an minimization of the preemption cost. An optimal selection of preemption points or priority levels can reduce the preemption overhead and thus increase the schedulability of such systems.

Cache partitioning, cache locking or scratchpads can be alternatives to standard caches as assumed in this thesis. A comparison of these methods may give a system designer valuable information about which implementation to use in case of preemptive scheduling. Also different cache layout techniques influence the preemption cost. Finding an optimal or at least near-optimal layout promises to improve the schedulability.

10.3. Conclusions

For hard real-time systems, preemptive scheduling is not only an optional design choice but often unavoidable. For instance, tasks may miss deadlines if scheduled non-preemptively or system interrupts are required and can not be disabled. A correct and sound computation of the context switch cost and the cache-related preemption delay is thus a prerequisite for static timing analysis of preemptively scheduled systems with caches. For any practical use, however, CRPD bounds must not only be sound but also precise. Strongly underutilized systems are a waste of resource and may lead to unacceptable hardware costs. As we have accomplished to determine provably sound and precise CRPD bounds, the applicability of static timing analysis has been extended to preemptively scheduled systems.

Bibliography

[1] Bryan Ackland, Alex Anesko, Douglas Brinthaupt, Steven J. Daubert, Asawaree Kalavade, Jürgen Knobloch, Ed Micca, Mallik Moturi, Christopher J. Nicol, Jay H. O'Neill, Joe Othmer, Eduard Säckinger, Kanwar J. Singh, J. Sweet, Christopher J. Terman, and Joseph Williams. A single-chip, 1.6-billion, 16-b MAC/s multiprocessor DSP. *IEEE Journal of Solid-state Circuits*, 35:412–424, 2000.

[2] Sebastian Altmeyer and Claire Burguière. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, (ECRTS '09), pages 109–118, July 2009.

[3] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium*, (RTSS '11), pages 261–271, December 2011.

[4] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. *Real-Time Systems*, (to appear), 2012.

[5] Sebastian Altmeyer and Gernot Gebhard. WCET analysis for preemptive systems. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 105–112, July 2008.

[6] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. Resilience analysis: Tightening the crpd bound for set-associative caches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, (LCTES '10), pages 153–162, April 2010.

[7] Sebastian Altmeyer and Claire Maiza Burguière. Cache-related preemption delay via useful cache blocks: Survey and redefinition. *J. Syst. Archit.*, 57:707–719, August 2011.

[8] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8:284–292, 1993.

[9] Theodor P. Baker. Stack-based scheduling for realtime processes. *Real-Time Systems*, 3:67–99, April 1991.

[10] Andrea Bastoni, Björn Brandenburg, and James Anderson. Cache-related preemption and migration delays: Empirical approximation and impact on schedulability. In *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, (OSPERT '10), pages 33–44, July 2010.

[11] Swagato Basumallick and Kelvin Nilsen. Cache issues in real-time systems. In *Proceedings of the First ACM SIGPLAN Workshop Languages, Compilers, and Tools for Real-Time Systems*, (LCTES '94), June 1994.

[12] Christoph Berg. PLRU cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[13] Marko Bertogna, Giorgio Buttazzo, Mauro Marinoni, Gang Yao, Francesco Esposito, and Marco Caccamo. Preemption points placement for sporadic task sets. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, (ECRTS '10), pages 251–260, 2010.

[14] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio Buttazzo. Optimal selection of preemption points to minimize preemption overhead. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, (ECRTS '11), pages 217–227, July 2011.

[15] Enrico Bini, Giorgio Buttazzo, and Giuseppe Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, (ECRTS '01), pages 59–69, July 2001.

[16] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30:129–154, 2005.

[17] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-related preemption delay computation for set-associative caches - pitfalls and solutions. In *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2009.

[18] Jose Busquets-Mataix, Juan Serrano, Rafael Ors, Pedro Gil, and Andy Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, (RTAS '96), pages 204–214, April 1996.

[19] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series).* Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[20] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, April 1976.

[21] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (POPL '77), pages 238–252, 1977.

[22] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (POPL '78), pages 84–96, 1978.

[23] Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order (2. ed.).* Cambridge University Press, 2002.

[24] Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on RealTime Systems*, pages 43–50, June 2000.

[25] Robert Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Trans. Comput.*, 57:1261–1276, September 2008.

[26] Peter J. Denning. The locality principle. *Commun. ACM*, 48:19–24, July 2005.

[27] Radu Dobrin and Gerhard Fohler. Reducing the number of preemptions in fixed priority scheduling. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, pages 144–152, July 2004.

[28] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th annual Design Automation Conference*, (DAC '07), pages 264–265, 2007.

[29] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis.* PhD thesis, Universitatis Upsaliensis, 2003.

[30] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Rheinhard Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Embedded Software Workshop*, volume 2211, pages 469–485, 2001.

[31] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, 1999.

[32] Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, (ECRTS '10), pages 155–164, July 2010.

[33] Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In Björn Lisper, editor, *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 28–39, July 2010.

[34] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks - past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, pages 137–147, July 2010.

[35] Reinhold Heckmann, Christian Ferdinand, Absint Angewandte, and Informatik Gmbh. Worst-case execution time prediction by static program analysis. In *18th International Parallel and Distributed Processing Symposium*, (IPDPS '04), pages 26–30, April 2004.

[36] Reinhold Heckmann, Marc Langebach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[37] John L. Hennessy and David A. Patterson. *Computer architecture (2nd ed.): a quantitative approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[38] Marc D. Hill and Alan J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38:1612–1630, December 1989.

[39] Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET 2008 – Report from the Tool Challenge 2008. In *Proceedings of the 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 149–171, July 2008.

[40] Kevin Jeffay and Donald L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14rd IEEE Real-Time Systems Symposium*, (RTSS '93), pages 212–221, December 1993.

[41] Mathai Joseph and Paritosh Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986.

[42] Martin Kaiser. Bounding task switch costs by cache analysis. Master's thesis, University of Saarland, 2009.

[43] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[44] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software*

and compilers for embedded systems, (LCTES/SCOPES '02), pages 46–55, June 2002.

[45] Ukur Keskin, Reinder J. Bril, and Johan J. Lukkien. Exact response-time analysis for fixed-priority preemption-threshold scheduling. In *Work-in-Progress Session Emerging Technologies and Factory Automation*, pages 1–4, September 2010.

[46] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, (POPL '73), pages 194–206, 1973.

[47] Sung-Kwan Kim, Sang Lyul Min, and Rhan Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, (RTAS '96), pages 230–240, April 1996.

[48] David B. Kirk and Jay K. Strosnider. Smart (strategic memory allocation for real-time) cache design. In *Proceedings of the 20st IEEE Real-Time Systems Symposium*, (RTSS '90), pages 322–330, December 1990.

[49] Chang-Gun Lee, Joosun Hahn, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, (RTSS '96), pages 264–274, December 1996.

[50] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[51] John P. Lehoczky, Lui Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the 11th IEEE Real-time Systems Symposium*, (RTSS '89), pages 166–171, December 1989.

[52] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, 1982.

[53] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[54] Tiantian Liu, Minming Li, and Chun Xue. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Systems*, 48:166–197, 2012.

[55] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, (RTSS '99), pages 12–22, December 1999.

[56] Florian Martin. PAG—an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[57] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In Kai Koskimies, editor, *CC*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998.

[58] Florian Martin Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In *Proceedings of the 7th International Conference on Compiler Construction*, (CC 98), pages 80–94, August 1998.

[59] Steven Martin, Pascale Minet, and Laurant George. Non preemptive fixed priority scheduling with fifo arbitration: uniprocessor and distributed cases. Technical report, INRIA Rocquencourt, December 2007.

[60] Patrick Meumeu Yomsi and Yves Sorel. Extending rate monotonic analysis with exact cost of preemptions for hard real-time systems. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, (ECRTS '07), pages 280–290, July 2007.

[61] Frank Mueller. Compiler support for software-based cache partitioning. *SIGPLAN Not.*, 30(11):125–133, 1995.

[62] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18:209–239, 2000.

[63] Frank Mueller and David B. Whalley Marion Harmon. Predicting instruction cache behavior. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, 1994.

[64] Hemendra Singh Negi, Tulika Mitra, and Abhik Roychoudhury. Accurate estimation of cache-related preemption delay. In *Proceedings of the 1st ACM international conference on Hardware/software codesign and system synthesis*, (CODES+ISSS '03), pages 201–206, October 2003.

[65] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench: a free real-time benchmark. In *Proceedings of the 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[66] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[67] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European conference on Design and Test*, (EDTC '97), pages 7–17, 1997.

[68] Stefan M. Petters and Georg Farber. Scheduling analysis with respect to hardware related preemption delay. In *Proceedings of the Workshop on Real-Time Embedded Systems*, December 2001.

[69] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. Wcet-aware software based cache partitioning for multi-task real-time systems. In *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[70] Isabelle Puaut and David Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, (RTSS '02), pages 114–124, December 2002.

[71] John Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, (RTSS '02), pages 315–325, December 2002.

[72] Jan Reineke. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, Saarbrücken, November 2008.

[73] Jan Reineke and Daniel Grund. Sensitivity of cache replacement policies. Reports of SFB/TR 14 AVACS 36, SFB/TR 14 AVACS, March 2008. ISSN: 1860-9821, http://www.avacs.org.

[74] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *Proceedings of the 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2006.

[75] David A. Schmidt. Trace-based abstract interpretation of operational semantics. *Lisp Symb. Comput.*, 10:237–271, May 1998.

[76] Jörn Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, (RTSS '00), pages 195–204, December 2000.

[77] L. Sha, Raj Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39:1175–1185, September 1990.

[78] Johan Stärner and Lars Asplund. Measuring the cache interference cost in preemptive real-time systems. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, (LCTES '04), pages 146–154, June 2004.

[79] Jan Staschulat and Rolf Ernst. Multiple process execution in cache related preemption delay analysis. In *Proceedings of the 4th ACM international conference on Embedded software*, (EMSOFT '04), pages 278–286, September 2004.

[80] Jan Staschulat and Rolf Ernst. Scalable precision cache analysis for real-time software. *Trans. on Embedded Computing Sys.*, 6(4):25, 2007.

[81] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling analysis of real-time systems with precise modeling of cache related preemption delay. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, (ECRTS '05), pages 41–48, July 2005.

[82] Stefan Steinke, Lars Wehmeyer, Bo-Sik Lee, and Peter Marwedel. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*, (DATE '02), pages 409–419, March 2002.

[83] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad allocation for concurrent embedded software. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, (CODES+ISSS '08), pages 37–42, October 2008.

[84] Yudong Tan and Vincent Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *Proceedings of the 8th International Workshop SCOPES 2004*, pages 182–199, 2004.

[85] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multi-tasking real-time systems with caches. *Trans. on Embedded Computing Sys.*, 6(1):7, 2007.

[86] Yudong Tan and Vincent J. Mooney. Timing analysis for preemptive multi-tasking real-time systems with caches. In *Proceedings of the conference on Design, automation and test in Europe*, (DATE '04), pages 1034–1039, February 2004.

[87] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[88] Henrik Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, (RTCSA '00), pages 23–33, December 2000.

[89] Henrik Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, October 2002.

[90] Hiroyuki Tomiyama and Nikil D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the 8th ACM international workshop on Hardware/software codesign*, (CODES '00), pages 67–71, September 2000.

[91] Yau tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. In *ACM Trans. on Design Automation of Electronic Systems*, pages 380–387, 1995.

[92] Xavier Vera, Björn Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, (RTSS '03), pages 154–164, December 2003.

[93] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Transactions on Embedded Computing Systems*, 7(1):4:1–4:38, December 2007.

[94] Yun Wang and Manas Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications*, (RTCSA '99), pages 328–338, 1999.

[95] Randall T. White, Christopher A. Healy, David B. Whalley, Frank Mueller, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, (RTAS '97), pages 192–202, June 1997.

[96] Reinhard Wilhelm. Prompt design principles for predictable multicore architectures. In *Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, (SCOPES '09), pages 31–32, April 2009.

[97] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7:36:1–36:53, May 2008.

[98] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28:966–978, July 2009.

[99] Andrew Wolfe. Software-based cache partitioning for real-time applications. *J. Comput. Softw. Eng.*, 2(3):315–327, 1994.

[100] Areej Zuhily and Alan Burns. Optimality of (d-j)-monotonic priority assignment. *Information Processing Letters*, 103:247–250, 2007.

[101] Richard Zurawski. *Embedded Systems Handbook, Second Edition: Embedded Systems Design and Verification.* CRC Press, Inc., Boca Raton, FL, USA, 2nd edition, 2009.

Index

List of Figures

List of Tables

Appendix A

UCB Analysis

The set of UCBs at program point p is the set of memory blocks cached at p and reused at a program point p' later than p. To compute this set, we split all paths through p in two sets: a set of paths starting in p and a set of paths emanating from p.

Concrete Semantics

The first set is given by the path-based forwards collecting semantics:

$$\mathrm{Coll}_{\Pi}^{\rightarrow} : V \rightarrow 2^{\Pi}$$

$$\mathrm{Coll}_{\Pi}^{\rightarrow}(p) = \{\pi \mid \pi \in \Pi \wedge \pi = [p_s, \ldots, p]\} \tag{A.1}$$

while the latter set is given by the path-based backwards collecting semantics:

$$\mathrm{Coll}_{\Pi}^{\leftarrow} : V \rightarrow 2^{\Pi}$$

$$\mathrm{Coll}_{\Pi}^{\leftarrow}(p) = \{\pi \mid \pi \in \Pi \wedge \pi = [p, \ldots, p_n]\} \tag{A.2}$$

with the concrete transformer $tf_C^{\leftarrow/\rightarrow}$:

$$tf_C^{\leftarrow/\rightarrow} : V \rightarrow (2^{\Pi} \rightarrow 2^{\Pi})$$

$$tf_C^{\rightarrow}(p)(S) := \{\pi \cdot p \mid \pi \in S\} \tag{A.3}$$

and

$$tf_C^{\leftarrow}(p)(S) := \{p \cdot \pi \mid \pi \in S\} \tag{A.4}$$

that appends/prepends a program point to all path of the incoming flow information. To extract the set of UCBs from the collecting path-based semantics, we need two auxiliary functions.

$$\widehat{\text{age}}^{\leftarrow/\rightarrow} : \Pi \rightarrow \text{Age}$$

The function $\widehat{\text{age}}^{\leftarrow}$ counts all different elements on path π backwards (i.e., starting at the last program point in p) to the last access to m. In case no such access exists, it returns ∞:

$$\widehat{\text{age}}^{\leftarrow}(\pi)(m) = \begin{cases} |\bigcup_{j<i} \sharp(p_j)| & \text{if } \pi = [p_1, \ldots, p_i] \cdot \pi' \wedge \sharp(p_i) = m \\ & \qquad \wedge \forall j \in \{1, \ldots, i-1\} : \sharp(p_j) \neq m \\ \infty & \pi = [p_1, \ldots, p_n] \\ & \qquad \wedge \forall j \in \{1, \ldots, n\} : \sharp(p_j) \neq m \end{cases} \tag{A.5}$$

Conversely, $\widehat{\text{age}}^{\rightarrow}$ counts all memory blocks on path π forwards to the first next access to m and returns ∞ if no such access exists. Note that this implementation implicitly assumes an initially empty cache.

$$\widehat{\text{age}}^{\rightarrow}(\pi)(m) = \begin{cases} |\bigcup_{j>i} \sharp(p_j)| & \text{if } \pi = \pi \cdot [p_i, \ldots, p_n] \cdot \pi' \wedge \sharp(p_i) = m \\ & \qquad \wedge \forall j \in \{1, \ldots, i-1\} : \sharp(p_j) \neq m \\ \infty & \pi = [p_1, \ldots, p_n] \\ & \qquad \wedge \forall j \in \{1, \ldots, n\} : \sharp(p_j) \neq m \end{cases} \tag{A.6}$$

We refer to $\widehat{\text{age}}(\pi)(m)$ as the concrete age of m on path π.

The set of UCBs at program point p is then bounded by the set of all blocks for which the number of accesses to distinct blocks at least on one path to p and at least on one path from p is less than K:

$$\begin{aligned} \text{UCB}(p) \subseteq \{m \mid \exists \pi_1 \in \text{Coll}_{\Pi}^{\leftarrow} : \widehat{\text{age}}(\pi_1)(m)^{\leftarrow} < K \\ \wedge \exists \pi_2 \in \text{Coll}_{\Pi}^{\rightarrow} : \widehat{\text{age}}(\pi_2)(m)^{\rightarrow}\} \end{aligned} \tag{A.7}$$

Note that this is a true superset of the set of UCBs at p as Equation (A.7) may also subsume spurious paths.

Abstract Domain

In the following, we omit the direction of the analysis, as forward and backward analyses are equivalent.

Instead of computing all paths to/from p and then extracting the set of UCBs, we directly bound for each memory block m the number of different memory blocks accessed since last access/until next access to

m. As soon as an element reaches an age K or larger, we do not to keep track of its precise age anymore. In fact, the UCB analysis can be seen as a forward and backward may-cache analysis (see Chapter 3.3):

$$tf : V \rightarrow (\text{Age} \rightarrow \text{Age})$$

$$tf(p)(\text{age}) := \\ \lambda m. \begin{cases} 0 & m = \sharp(p) \\ \text{age}(m) & \text{age}(m) > \text{age}(\sharp(p)) \\ \text{age}(m) + 1 & \text{age}(m) \leq \text{age}(\sharp(p)) \wedge \text{age}(m) < K - 1 \\ \infty & \textit{otherwise} \end{cases} \tag{A.8}$$

Also $\sqsubseteq$ and $\bigsqcup$ are defined according to the may-cache analysis:

$$\text{age}_1 \sqsubseteq \text{age} \Leftrightarrow \forall m \in M : \text{age}_1(m) \geq \text{age}_2(m) \tag{A.9}$$

$$\text{age} \bigsqcup \text{age} = \lambda m. \min(\text{age}_1(m), \text{age}_2(m)) \tag{A.10}$$

$$\begin{aligned} \text{UCB}(p) \subseteq \{m \mid \exists \pi_1 \in \text{Coll}_{\Pi}^{\leftarrow} : \widehat{\text{age}}(\pi_1)(m)^{\leftarrow} < K \\ \wedge \exists \pi_2 \in \text{Coll}_{\Pi}^{\rightarrow} : \widehat{\text{age}}(\pi_2)(m)^{\rightarrow}\} \\ \subseteq \{m \mid \text{age}_p^{\leftarrow}(m) < K \wedge \text{age}_p^{\rightarrow}(m)\} \end{aligned} \tag{A.11}$$

where $\text{age}_p^{\leftarrow}(m)$ denotes the computed forward and $\text{age}_p^{\rightarrow}(m)$ the backward age of m at program point p.

Concretization/Abstraction

The set of paths represented by an abstract state is the set of paths that *respect* the age bounds for each memory block. As we are aiming for an over-approximation of the set of UCBs, we compute lower bounds on the ages:

$$\gamma(\text{age}) = \{\pi \mid \forall m \in M : \text{age}(m) \leq \widehat{\text{age}}(\pi)(m)\} \tag{A.12}$$

Conversely, the abstract age-bound of a memory block is given by the least concrete age of this block on any path:

$$\alpha(S) = \lambda m. \min\{|\widehat{\text{age}}(\pi)(m)| \mid \pi \in S\} \tag{A.13}$$

Theorems and Proofs

Theorem A.1 (Monotonicity)
The abstract transformer of the UCB analysis tf is monotone, i.e.,

$$\forall p \in V : \forall age_1, age_2 \in Age : age_1 \sqsubseteq age_2 \Rightarrow tf(p)(age_1) \sqsubseteq tf(p)(age_2)$$

Proof (Monotonicity (Theorem A.1))
Let $p \in V$ and $m \in M$ be arbitrary and m' be the accessed element at program point p. We know that for each memory block, the age bound $age_1(m)$ is at the least age bound $age_2(m)$, i.e.

$$\forall m \in M : age_1(m) \geq age_2(m)$$

We prove Theorem A.1 using case distinction on $age_1(m)$.

$m = m'$ *In this case, both age bounds are set to zero, hence:*

$$tf(p)(age_1) = 0 = tf(p)(age_2)$$

$age_1(m) = \infty$ *Since $m \neq m'$, we can conclude that $tf(p)(age_1) = \infty$. Hence,*

$$tf(p)(age_1) = \infty \geq tf(p)(age_2)$$

$age_1(m) > age_2(m)$ *We know that m is not accessed by p and $age_1(m)$ is finite. As $age_2(m)$ is also finite, we can conclude that the abstract transformer increases the age bound at most by one:*

$$tf(p)(age_2) \leq age_2(m) + 1 = tf(p)(age_1)$$

$age_1(m) = age_2(m)$ *If both age bounds are the same, we use a case distinction on the age bound of m' (given that $age_1(m') \geq age_2(m')$:*

$age_1(m') = age_2(m')$ *As the age bounds age_1 and age_2 are equal in both cases m and m', the abstract transformer also results in equal age bounds for m:*

$$tf(p)(age_1) = tf(p)(age_2)$$

$age_1(m') > age_2(m')$ *In this case, we can conclude that the abstract transformer only increases the age bound age_2, if also age_1 is increased. I.e. if $age_2(m) \geq age_2(m')$ holds, so does $age_1(m) \geq age_1(m')$, and hence:*

$$tf(p)(age_1) \geq tf(p)(age_2)$$

□

Theorem A.2 (Local Consistency)
The abstract transformer tf and the concrete transformer tf_C are locally consistent, i.e.,

$$\forall age \in Age : \forall p \in V : (tf_C(p))(\gamma(age)) \subseteq \gamma((tf(p))(age))$$

Proof (Local Consistency (Theorem A.2))
We prove Theorem A.2 by contradiction. Note that we only prove the forward direction. The proof for the backward analysis is equivalent.

Assume

$$\exists age \in Age : \exists p \in V : (tf_C(p))(\gamma(S)) \nsubseteq \gamma((tf(p))(S))$$

Let π be a path in $tf_C(p))(\gamma(S))$ not contained in $\gamma((tf(p))(S))$ and m' be the element accessed at p: As $tf_C(p)$ only append p to all paths, we can conclude that path π can be written as $\pi = \hat{\pi} \cdot p$ with $\hat{\pi} \in \gamma(S)$. Furthermore, we know by the construction of γ:

$$\forall m \in M : \widehat{age}(\hat{\pi})(m) \geq age(m)$$

As $\pi \notin \gamma((tf(p))(age))$, we know that exists a memory block m, such that $\widehat{age}(\pi)(m) < (tf(p)(age))(m)$. We use a case distinction on m and its concrete age:

$m = m'$ *By construction of the abstract transformer, we can conclude that*

$$\widehat{age}(\pi)(m) = 0 = (tf(p)(age))(m)$$

$\widehat{age}(\pi)(m) = \infty$ *As the concrete age is ∞, the abstract age bound can not be larger, and so:*

$$\widehat{age}(\pi)(m) \geq (tf(p)(age))(m)$$

$\widehat{age}(\hat{\pi})(m) \leq \widehat{age}(\hat{\pi}, m')$ *In this case, the concrete age of m on path π is increased by one compared to the concrete age on $\hat{\pi}$. Since $\widehat{age}(\hat{\pi})(m) \geq age(m)$ and also the abstract age is increased by at most one, we know:*

$$\widehat{age}(\pi)(m) \geq (tf(p)(age))(m)$$

$\widehat{age}(\hat{\pi})(m) > \widehat{age}(\hat{\pi}, m')$ *The abstract transformer increases the age bound of* m*, only if* $age(m) \leq age(m')$ *holds. This, however, can only hold if* $\widehat{age}(\hat{\pi})(m) > age(m)$*. Hence,*

$$\widehat{age}(\pi)(m) \geq (tf(p)(age))(m)$$

□

Theorem A.3 (Soundness of the Abstraction)

$$\forall S \in 2^{\Pi} : S \sqsubseteq \gamma(\alpha(S))$$

Proof (Soundness of the Abstraction (Theorem A.3))
We prove Theorem A.3 by contradiction. Note that we only prove the forward direction. The proof for the backward analysis is equivalent. Assume there exists a path $\pi \in S$ *such that* $\pi \notin \gamma(\alpha(S))$*. Hence, there must a memory block* m *such that the concrete age* $\widehat{age}(\pi)(m)$ *is less then* $(\alpha(S))(m)$*. This, however, is not possible as* $\pi \in S$ *and* $(\alpha(S))(m) = \min\{\widehat{age}(\hat{\pi})(m) \mid \hat{\pi} \in S\}$*.* □

Appendix B

Proofs

B.1. DC-UCB Analysis

Proof (Monotonicity (Theorem 6.3))
We prove Theorem 6.3 using case distinction on the classification of the accessed element. We know that $a \subseteq b$:

Case 1 ($\sharp(p) = \bot$)

$$\begin{aligned} tf(p)(a) &= a \\ &\subseteq b \\ &= tf(p)(b) \end{aligned}$$

Case 2 (Classify($\sharp(p), p) = ah$)

$$\begin{aligned} tf(p)(a) &= a \cup \{\sharp(p)\} \\ &\subseteq b \cup \{\sharp(p)\} \\ &= tf(p)(b) \end{aligned}$$

Case 3 (Classify($\sharp(p), p) \neq ah$)

$$\begin{aligned} tf(p)(a) &= a \setminus \{\sharp(p)\} \\ &\subseteq b \setminus \{\sharp(p)\} \\ &= tf(p)(b) \end{aligned}$$

□

Proof (Local Consistency (Theorem 6.4))
We prove Theorem 6.4 by contradiction. Assume for a set $S \in 2^M$ and a program point $p \in V$

$$(tf_C(p))(\gamma(S)) \not\subseteq \gamma((tf(p))(S))$$

Let $\pi \in \Pi$ be a path with

$$\pi \in (tf_C(p))(\gamma(S))$$

and

$$\pi \notin \gamma((tf(p))(S))$$

We now perform a case distinction on the memory reference of p.

Case 1 ($\sharp(p) = \bot$) As p has no memory reference, a path π satisfies Els(π, m) for a memory block m if and only if it also satisfies Els$(p \cdot \pi, m)$ with p prepended. Hence,

$$(tf_C(p))(\gamma(S)) = \gamma(S)$$

Abstract transformer only updates the control flow information in case of a memory reference. Hence

$$(tf(p))(S) = S \wedge \gamma((tf(p))(S)) = \gamma(S)$$

and thus,

$$(tf_C(p))(\gamma(S)) \subseteq \gamma((tf(p))(S))$$

Case 2 (Classify$(\sharp(p), p) = ah$) Let m be the memory reference at p, i.e., $m = \sharp(p)$. We know that

$$tf(p)(S) = S \cup \{m\}$$

and Els now holds for memory block m on each path π with p prepended, i.e.,

$$\forall \pi : \textit{Els}(p \cdot \pi, m)$$

while Els retains its value for all other memory blocks

$$\forall n \neq m : \textit{Els}(p \cdot \pi, n) = \textit{Els}(\pi, n)$$

Assume there is a path

$$\hat{\pi} \in (tf_C(p))(\gamma(S)) \wedge \hat{\pi} \notin \gamma((tf(p))(S))$$

$$\begin{aligned}
&\hat{\pi} \notin \gamma((tf(p))(S)) \\
\Rightarrow &\hat{\pi} \notin \gamma(S \cup \{m\}) \\
\Rightarrow &\exists n \in M \setminus (S \cup \{m\}) : \; \mathit{Els}(\hat{\pi}, n) \\
\Rightarrow &\exists n \in M \setminus (S \cup \{m\}) : \; \mathit{Els}(p \cdot \pi, n) \\
\Rightarrow &\exists n \in M \setminus (S \cup \{m\}) : \; \mathit{Els}(\pi, n)
\end{aligned}$$

This contradicts the assumption as

$$\pi \in \gamma(S)$$

and

$$\forall n \neq m : \mathit{Els}(p \cdot \pi, n) = \mathit{Els}(\pi, n)$$

Case 3 (Classify($\sharp(p), p) \neq ah$) Let m be the memory reference at p, i.e., $m = \sharp(p)$. We know that

$$tf(p)(S) = S \setminus \{m\}$$

and Els now does not hold for memory block m on any path π with p prepended, i.e.,

$$\forall \pi : \neg \mathit{Els}(p \cdot \pi, m)$$

while Els retains its value for all other memory blocks

$$\forall n \neq m : \mathit{Els}(p \cdot \pi, n) = \mathit{Els}(\pi, n)$$

Assume there is a path

$$\hat{\pi} \in (tf_C(p))(\gamma(S)) \wedge \hat{\pi} \notin \gamma((tf(p))(S))$$

$$\begin{aligned}
&\hat{\pi} \notin \gamma((tf(p))(S)) \\
\Rightarrow &\hat{\pi} \notin \gamma(S \setminus \{m\}) \\
\Rightarrow &\exists n \in M \setminus (S \setminus \{m\}) : \; \mathit{Els}(\hat{\pi}, n) \\
\Rightarrow &\exists n \in M \setminus (S \setminus \{m\}) : \; \mathit{Els}(p \cdot \pi, n) \\
\Rightarrow &\exists n \in M \setminus (S \setminus \{m\}) : \; \mathit{Els}(\pi, n)
\end{aligned}$$

This contradicts the assumption since $\pi \in \gamma(S)$, $\forall n \neq m : \mathit{Els}(p \cdot \pi, n) = \mathit{Els}(\pi, n)$ and $\forall \pi : \neg \mathit{Els}(p \cdot \pi, m)$.

As each case of the (exhaustive) case-distinction leads to a contradiction to the assumption, we have proven the assumption wrong. □

Proof (Soundness of the Abstraction(Theorem 6.5))
We prove the theorem by contradiction. Assume there is an arbitrary path $\hat{\pi} \in S$ with $\hat{\pi} \notin \gamma(\alpha(S))$. As $\hat{\pi}$ is not in $\gamma(\alpha(S))$, we can conclude that there exists a memory block m for which $Els(\hat{\pi}, m)$ holds (i.e., m is a DC-UCB on path $\hat{\pi}$) that is not contained in the abstraction $\alpha(S)$:

$$\exists m \notin \alpha(S) : Els(\pi, m)$$

By the definition of α, however, we know that m is contained in $\alpha(S)$ if there is a path for which $Els(\pi, m)$ holds. Since $\hat{\pi} \in S$ and $Els(\hat{\pi}, m)$, we can conclude that the assumption was wrong. □

B.2. Resilience

Proof (Monotonicity (Theorem 7.3))
As the transformer of resilience analysis tf_{res} using the transformer on the constrained age tf_{ca} and on the unconstrained ages tf_{ua}, we will prove the monotonicity of tf_{ca} and tf_{ua} separately.

Unconstrained Age *Let $p \in V$ and $m \in M$ be arbitrary and m' be the accessed element at program point p. We know that for each memory block, the age bound $ua_1(m)$ is at the least age bound $ua_2(m)$, i.e.,*

$$\forall m \in M : ua_1(m) \geq ua_2(m)$$

We prove the monotonicity of tf_ua using case distinction on $ua_1(m)$.

$m = m'$ *In this case, both age bounds are set to zero, hence:*

$$tf_{ua}(p)(ua_1) = 0 = tf_{ua}(p)(ua_2)$$

$ua_1(m) = \infty$ *Since $m \neq m'$, we can conclude that $tf_{ua}(p)(ua_1) = \infty$. Hence,*

$$tf_{ua}(p)(ua_1) = \infty \geq tf_{ua}(p)(ua_2)$$

$ua_1(m) > ua_2(m)$ *We know that m is not accessed by p and $ua_1(m)$ is finite. As $ua_2(m)$ is also finite, we can conclude that the abstract transformer increases the age bound at most by one:*

$$tf_{ua}(p)(ua_2) \leq ua_2(m) + 1 = tf_{ua}(p)(ua_1)$$

$ua_1(m) = ua_2(m)$ *If both age bounds are the same, we use a case distinction on the age bound of m' (given that $ua_1(m') \geq ua_2(m')$):*

$ua_1(m') = ua_2(m')$ *As the age bounds ua_1 and ua_2 are equal in both cases m and m', also the abstract transformer results in equal age bounds for m:*

$$tf_{ua}(p)(ua_1) = tf_{ua}(p)(ua_2)$$

$ua_1(m') > ua_2(m')$ *In this case, we can conclude that the abstract transformer only increases the age bound ua_2, if also ua_1 is increased. I.e., if $ua_2(m) \geq ua_2(m')$ holds, so does $ua_1(m) \geq ua_1(m')$, and hence:*

$$tf_{ua}(p)(ua_1) \geq tf_{ua}(p)(ua_2)$$

Constrained Age *Let $p \in V$ and $m \in M$ be arbitrary and m' be the accessed element at program point p. We know that for each memory block, the unconstrained age bound $ua_1(m)$ is at the least unconstrained age bound $ua_2(m)$, i.e.,*

$$\forall m \in M : ua_1(m) \geq ua_2(m)$$

and the constrained age bound $ca_1(m)$ is at the least unconstrained age bound $ca_2(m)$, i.e.,

$$\forall m \in M : ca_1(m) \geq ca_2(m)$$

We prove the monotonicity of $tf_c a$ using case distinction on $ca_1(m)$.

$m = m' \vee m' \notin UCB(p)$ *In this case, both age bounds are set to zero, hence:*

$$tf_{ca}(p)(ua_1) = 0 = tf_{ca}(p)(ua_2)$$

$ca_1(m) > ca_2(m)$ *We know that m is not accessed by p and $ua_1(m)$ is finite. As $ca_2(m)$ is also finite, we can conclude that the abstract transformer increases the age bound at most by one:*

$$tf_{ca}(p)(ca_2, ua_2) \leq ca_2(m) + 1 = tf_{ca}(p)(ca_1, ua_1)$$

$ca_1(m) = ca_2(m)$ *If both age bounds are the same, we use a case distinction on the unconstrained age bound of m' (given that $ua_1(m') \geq ua_2(m')$:*

$ua_1(m') = ua_2(m')$ *As the age bounds ua_1 and ua_2 are equal in both cases m and m', also the abstract transformer results in equal age bounds for m:*

$$tf_{ca}(p)(ca_1, ua_1) = tf_{ca}(p)(ca_2, ua_2)$$

$ua_1(m') > ua_2(m')$ *In this case, we can conclude that the abstract transformer only increases the age bound ca_2, if also ca_1 is increased. I.e., if $ca_2(m) \geq ua_2(m')$ holds, so does $ca_1(m) \geq ua_1(m')$, and hence:*

$$tf_{ca}(p)(ca_1, ua_1) \geq tf_{ca}(p)(ca_2, ua_2)$$

□

Proof (Local Consistency (Theorem 7.4))
We prove local consistency of tf_{res} by contradiction. Note that we only prove the forward direction. The proof for the backward analysis is equivalent. Assume a pair of ages (ua, ca) and a program point $p \in V$ such that

$$(tf_C(p))(\gamma(S)) \not\subseteq \gamma((tf_{res}(p))(ua, ca))$$

Let π be a path such that

$$\pi \in tf_C(p))(\gamma(ua, ca)) \wedge \pi \notin \gamma((tf_{res}(p))(ua, ca))$$

and m' be the element accessed at p:

As $tf_C(p)$ only append p to all paths, we can conclude that path π can be written as $\pi = \hat{\pi} \cdot p$ with $\hat{\pi} \in \gamma(S)$. Furthermore, we know by the construction of γ for the unconstrained ages

$$\forall m \in M : \widehat{age}(\hat{\pi})(m) \geq ua(m)$$

and for the constrained ages:

$$\forall m \in M : m \in \mathit{UCB}(p) \Rightarrow \widehat{age}(\hat{\pi})(m) \geq ca(m)$$

As $\pi \notin \gamma((tf_{res}(p))(ua, ca))$, we know that exists a memory block m, such that

$$\begin{aligned}&\widehat{age}(\pi)(m) > (tf_{ua}(p)(ua))(m)\\ &\qquad \vee (\widehat{age}(\pi)(m) > (tf_{ca}(p)(ca, ua))(m) \wedge m \in \mathit{UCB}(p))\end{aligned} \tag{B.1}$$

We use a case distinction on m and its concrete age:

$m = m'$ *By construction of the abstract transformer, we can conclude that*

$$\widehat{age}(\pi)(m) = 0 = (tf_{ua}(p)(ua))(m) = (tf_{ca}(p)(ca, ua))(m)$$

$\widehat{age}(\hat{\pi})(m) < \widehat{age}(\hat{\pi})(m')$ *In this case, the concrete age of m on path π is increased by one. The transformer tf_{ua} retains the bound of m, only if $ua(m) \leq ua(m')$ holds. This, however, can only hold if $\widehat{age}(\hat{\pi})(m) < ua(m)$. Hence, if the unconstrained bound is not increased by one it was definitely larger than the concrete age. And so:*

$$\widehat{age}(\pi)(m) \leq (tf(p)(age))(m)$$

If $m \in UCB(\pi)$ then also $m \in UCB(\hat{\pi})$ (as we know that $m \neq m'$). Here, we can apply the same argumentation as for the unconstrained age and conclude:

$$m \in UCB(p) \Rightarrow \widehat{age}(\pi)(m) \leq (tf_{ca}(p)(ca, ua))(m)$$

$\widehat{age}(\hat{\pi})(m) > \widehat{age}(\hat{\pi})(m')$ *In this case, the concrete age of m on path π is not increased. Hence,*

$$\widehat{age}(\pi)(m) = \widehat{age}(\hat{\pi})(m) \leq (tf(p)(age))(m)$$

If $m \in UCB(\pi)$ then also $m \in UCB(\hat{\pi})$ (as we know that $m \neq m'$) and so:

$$m \in UCB(p) \Rightarrow \widehat{age}(\pi)(m) \leq (tf_{ca}(p)(ca, ua))(m)$$

Note that the case distinction is exhaustive as

$$\widehat{age}(\hat{\pi})(m) = \widehat{age}(\hat{\pi})(m') \Leftrightarrow m = m'$$

This finishes the proof. □

Proof (Soundness of the Abstraction (Theorem 7.5))
We prove Theorem 7.5 by contradiction. Note that we only prove the forward direction. The proof for the backward analysis is analog. Let $\alpha(S) = (ua, ca)$. Assume there exists a path $\pi \in S$ such that $\pi \notin \gamma(\alpha(S))$. Hence, there must a memory block m such that the concrete age $\widehat{age}(\pi)(m)$ is larger then the unconstrained age $ua(m)$ or m is useful on π and the concrete age $\widehat{age}(\pi)(m)$ is larger then the constrained age $ca(m)$ This, however, is not possible as $\pi \in S$ and the unconstrained age $ua(m) = \max\{\widehat{age}(\hat{\pi})(m) \mid \hat{\pi} \in S\}$ is defined as the maximal concrete age on any path while the constrained age $\lambda m. \max\{\widehat{age}(\pi)(m) \mid (\pi \cdot p) \in S \wedge m \in UCB(p))\}$ is defined as the maximal concrete age on any path on which m is useful. □