Jan Staschulat

Instruction and Data Cache Timing Analysis in Fixed-Priority Preemptive Real-Time Systems

Cuvillier Verlag Göttingen

INSTRUCTION AND DATA CACHE TIMING ANALYSIS IN FIXED-PRIORITY PREEMPTIVE REAL-TIME SYSTEMS

JAN STASCHULAT
Institute of Computer and Communication Network Engineering
Department of Electrical Engineering and Information Technology
Technical University of Braunschweig
Braunschweig, Germany

Abstract

Embedded systems are prevalent in today's society and promise to be even more pervasive in the future. Applications vary from airplane jet or car controllers, communication devices like cellular phones to consumer electronics like set-top boxes. The steadily increasing number of functional requirements lead to a complex embedded hardware and software architecture. Often, applications not only have to compute correct results but have to achieve this within a given time period. Timing behavior is an important requirement if the application has to react to signals from the environment. To safely and tightly verify timing behavior is very challenging for today's complex embedded designs.

Caches are small memories close to the processor and they are needed to increase the processor performance but their influence on execution time is difficult to predict because of their complex behavior. Preemptive scheduling is popular in real-time systems to guarantee short response times and a high processor utilization. An additional cache-related preemption delay has to be considered when several tasks share the same cache and when preemptive task scheduling is used. Cache improvements can be strongly degraded by frequent replacements of cache blocks.

There are several approaches to make caches more predictable and efficient. Cache partitioning and cache locking strategies are used to make cache behavior partly orthogonal. These approaches require larger caches and main memories to become effective. However, caches are usually small in embedded systems because of their high cost. While these approaches are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the problem of cache behavior prediction if all tasks shared the cache.

This thesis makes several contributions to instruction and data cache timing behavior. First, we propose a novel schedulability analysis for fixed priority preemptive scheduling to consider timing effects for associative instruction caches at a context switch. The preemption delays are calculated by considering the preempted as well as the preempting task. The proposed schedulability analysis bounds *the number of preemptions more tightly by excluding infeasible cache interferences*. The analysis is conservative, e.g. determines a safe upper bound of the preemption delay, and has a low time complexity.

As a refinement, the cache interference by multiple task preemptions is analyzed. While previous approaches calculate the worst-case preemption point and assume that each preemption takes place at this preemption point, we *consider the preemption history* in the calculation of the total cost for multiple task preemptions. The advantage is that the bound of the total preemption delay for multiple task preemptions can consider the preemption history.

Execution time verification is often used on different levels of the system design. Less precise estimates are acceptable in early design stages while highly accurate ones are necessary for verification of hard real time constraints. Two approaches to bound the preemption delay have been proposed which both use data flow techniques but differ significantly in respect to time-complexity and analysis precision. In this thesis we combine these two approaches in a single scalable precision cache analysis to *scale the analysis precision and the time-complexity*.

In an automotive case study we found out that control intensive applications designed with ASCET-SD and Matlab/Simulink models contain only sequential code without loops. Caches cannot increase the performance for such applications because linear code significantly limits the spacial and temporal locality of memory accesses for which a cache is optimized. Existing timing analyses focus on a single task execution. However, embedded applications are activated very frequently if not regularly. Cache lines from a previous task activation might still be available in the cache and need not be loaded during a subsequent task execution. This effect of *multiple task execution* can result in a significantly reduced number of cache misses. In this thesis we estimate a conservative bound of the cache contents at the beginning of task activation and consider the effect in instruction cache timing behavior.

While previous analysis techniques focus on instruction caches, we also provide a novel timing analysis for data caches. Data cache behavior is more difficult to predict because it depends on control flow of the application but also on the input data. While instruction addresses are fixed, a single instruction can access different data memory addresses, for example operations on an array. In this thesis we propose a *static timing analysis for data caches* which considers input data dependency of memory accesses.

Finally, we integrate instruction and data cache timing analysis in a measurement-based WCET-analysis tool, which has been developed in previous work. Measuring the execution time requires insertion of instrumentation points which disturbs the temporal behavior of an application. In this thesis we present a novel *instrumentation methodology* that reduces the number of instrumentation points.

Kurzfassung

Eingebettete Systeme sind in unserem Alltag allgegenwärtig und werden in der Zukunft noch eine bedeutendere Rolle spielen. Anwendungen reichen vom Flugzeug und Mikro-Controller im Auto, über mobile Anwendungen wie dem Handy bis hin zu Multi-Media Anwendungen, wie der Settop-Box. Die immer steigendenen funktionalen Anforderungen an eingebettete Systeme schlagen sich im immer komplexeren Design der Software und Hardware nieder. Jedoch müssen Anwendungen nicht nur das korrekte Ergebnis liefern, sondern oft müssen diese Berechnungen innerhalb einer bestimmten Zeitdauer durchgeführt werden. Beispiele für Echtzeit (real-time)-Anforderungen reichen von harten (hard real-time) Bedingungen, wie etwa beim Airbag im Fahrzeug oder sicherheitskritischer Navigation im Flugzeug bis hin zu weichen Echtzeit-Bedingungen (soft real-time), die auch Quality-Of-Service genannt werden, wie etwa die Übertragung von Wort und Bild in der Telekommunikation. Die Vorhersage der Einhaltung dieser Zeitbedingungen ist eine große Herausforderung für heutige komplexe Mikroprozessoren.

Caches sind kleine Speicher in der Nähe des Prozessors, die notwendig sind, um die Performance zu erhöhen. Sie überbrücken den Geschwindigkeitsunterschied zwischen einem langsamen Speicherzugriff und der hohen Prozessor-Taktrate. Die Vorhersage des Zeitverhaltens einer Anwendung, oder auch Task genannt, wird dadurch um ein Vielfaches schwieriger, weil das Cache-Verhalten sehr dynamisch ist. Das unterbrechende Scheduling ist eine geeignete Methode der Ressourcenverwaltung in Echtzeitsystemen, weil sich kurze Antwortzeiten und eine hohe Auslastung erreichen lässt. Unterbrechendes Scheduling bedeutet, dass eine Anwendung unterbrochen wird um beispielsweise eine Anwendung mit einer höheren Priorität auszuführen.

Der unterbrechende und unterbrochene Task können die gleichen Cache Blöcke im Cache benutzen. Die durch den unterbrechenden Task ersetzten Cache Blöcke müssen dann ggf. vom unterbrochenen Task wieder nachgeladen werden. Die Dauer für das Nachladen von Cache Blöcken wird als Cache-abhängige Unterbrechungszeit (cache-related preemption delay) bezeichnet.

Als Alternative gibt es einige Verfahren um das Cache-Verhalten vorhersagbarer zu machen. Cache Partitionierung und Cache Locking sind zwei Techniken, um das Zeitverhalten des Caches unabhängig von anderen Tasks zu gestalten. Jedoch wer-

den größere Caches und größere Hintergrund Speicher benötigt, damit diese Techniken effektiv sind. In eingebetteten Systemen sind Caches jedoch aus ökonomischen Gründen meist klein. Diese Ansätze sind sicher sehr attraktiv, um das Zeitverhalten deterministischer zu machen, sie lösen jedoch nicht das Problem, wenn alle Anwendungen gemeinsam einen Cache benutzen.

Diese Dissertation beschäftigt sich mit der Analyse des Zeitverhaltens von Befehls- und Datencaches. Zunächst, wird eine innovative Schedulability-Analyse für unterbrechendes Scheduling mit festen Prioritäten vorgestellt, die das Zeitverhalten von assoziativen Befehlscaches beim Task-Wechsel berücksichtigt. Die Bestimmung der Unterbrechungskosten berücksichtigt das Cacheverhalten der unterbrochenen als auch der unterbrechenden Anwendung. Im Vergleich zu früheren Arbeiten, wird die Anzahl der Cache-Beeinflussung durch unterbrechende Anwendungen genauer jedoch konservativ abgeschätzt. Konservativ bedeutet, dass eine obere Schranke angegeben wird. Weiterhin ist die Zeitkomplexität diese Methode sehr gering.

Als weitere Verbesserung werden mehrere Unterbrechungen im Zusammenhang betrachtet. Während frühere Ansätze jede Unterbrechung an der teuersten (worst-case) Unterbrechungsstelle annehmen, wird in dieser Arbeit die teuerste Kombination für eine Menge von Unterbrechungskosten bestimmt. Der Vorteil liegt darin, dass die Abschätzung der Kosten von *allen* Unterbrechungen kleiner sein kann als die Summer der Kostenabschätzungen jeder einzelnen Unterbrechung.

Die Abschätzung der Unterbrechungskosten kann durch Datenflussanalysen bestimmt werden. In der Literatur gibt es zwei Ansätze, die jeweils Datenflussanalysen verwenden, die sich hinsichtlich ihrer Analysegenauigkeit und Zeitkomplexität jedoch stark unterscheiden. In dieser Dissertation werden beide Ansätze in einem Verfahren kombiniert, um die Genauigkeit und die Zeitkomplexität skalieren zu können.

In einer Studie zur Bestimmung des Cacheeinflusses in Kontroll-basierten Anwendungen im Fahrzeug hat sich gezeigt, dass der automatisch generierte C-Code linear ist und keine Schleifen enthält. Aufgrund dieser Struktur kann ein Cache die Ausführungzeit nicht verbessern weil weder zeitliche noch räumliche Lokalität der Speicherzugriffe vorliegt, für die Caches optimiert sind. In früheren Laufzeitanalysen wird von einer einzelnen Ausführung einer Anwendung ausgegangen. In einem eingebetteten System werden Anwendungen jedoch nicht nur einmal, sondern sehr häufig ausgeführt. Cache Blöcke, die von einer früheren Ausführung noch im Cache vorhanden sind, müssen in einer zweiten Ausführung nicht geladen werden und reduzieren so die Cachezugriffszeit. In dieser Arbeit wird ein Verfahren vorgeschlagen,

in dem mehrfache Ausführungen von Anwendungen in der Berechnung der Cacheverhaltens berücksichtigt werden.

Die oben genannten Analysen bezogen sich auf den Befehlscache. Nun schlagen wir eine innovative Analyse des Zeitverhaltens für Datencaches vor. Die Herausforderung bei Datencaches liegt in der weitaus stärkeren Abhängigkeit von Eingabedaten. Während bei Zugriffen im Befehlscaches die Speicheraddresse des Befehls konstant ist, kann die mehrfache Ausführung des gleichen Befehls auf unterschiedliche Datenworte zugreifen, beispielsweise Operationen auf einem Array. In dieser Arbeit wird ein Verfahren vorgestellt, um die Eingabendaten-Abhängigkeit von Speicherzugriffen zu analysieren. Diese Effekte werden dann in der Analyse des Zeitverhaltens von Datencaches berücksichtigt.

Abschließend integrieren wir die oben vorgestellen Cache-Analyse Verfahren in einem bestehende Tool zur Laufzeitanalyse. Das darin verwendete Methodik basiert auf eine Kombination aus Messung und statischer Analyse. Die Messung von Laufzeiten erfolgt auf realer Hardware und die eingefügten Messpunkte beeinflussen das zeitliche Verhalten der Anwendung. Um die Messungenauigkeit zu reduzieren, schlagen wir ein Verfahren vor, das die Anzahl der Messpunkte reduziert.

Um zusammenzufassen, in dieser Dissertation wird das Zeitverhalten von Befehls- und Datencaches in einbetteten Systemen mit unterbrechendem Scheduling genau analysiert. Damit steht ein umfangreiches Analyse-Framework zur Verfügung, um das komplexe Zeitverhalten von Caches für Echtzeitanwendungen zu berücksichtigen.

Acknowledgments

This thesis is the result of the research at the Institute of Computer and Communication Network Engineering (IDA) at the Technical University of Braunschweig, Germany.

I would like to express my deep gratitude to my advisor Professor Rolf Ernst for so many insightful discussions, continuous motivation and many reviews of my work. I would like to thank Professor Jan Madsen for agreeing to co-examine this work and Professor Lars Wolf for chairing the examination committee.

Many colleagues have contributed to this work through discussions, reviews, questions or other support. I would especially like to thank Jörn Christian Braam, Matthias Ivers, Simon Schliecker, Razvan Racu, Rafik Henia, Judita Kruse, Sven Heithecker, Amilcar Lucas, Bhavani Janarthanan, Clive Thomson, Peter Oruba, Steffen Stein, Sean Whitty, Arne Hamann, Boris Tolg, Wolfgang Bziuk, Peter Rüffer, Holger Dinse, Marek Jersak, Kai Richter, Fabian Wolf and many more members of the IDA institute. I would like to thank all students for providing many implementations and performing experiments. Thanks to the IDA staff for providing professional service, especially I would like to thank Bettina Böttger, Ina Niedermayer, Gaby Gorajski and Jan Pietrzyk.

I would like to thank researchers and practitioners in the real-time community for their support, especially Isabelle Puaut, Tulika Mitra, Guillem Bernat, Christian Ferdinand, Reinhard Wilhelm, Frank Mueller, Adam Betts, Raimund Kirner and Rainer Schlör. I would like to express my thanks to many anonymous reviewers for their comments and conference participants for fruitful discussions.

Thanks to all friends for having a joyful time: Kathleen Tschernatsch, Mathias Tauchert, Anja Andrea Kerber, Redouane Msaad, Anne and Grit Tölke, Susanne Fiedler, Alexandra Wilke, Steffen Dressler, Mathias Korn, Konstantin von Keitz, Christiane Börstinghaus, Jan Zänker, Robert Köhler, Ana Maria Racu, Alexandra

Schlagowski, Karin Lengfeld, Cecile Hamann, Kai Liebing, Jianying Ji, Sina Nitzkow, Jens Ewald, Iris Wetekam, and all other friends.

Great thanks to Tino Fettback for giving me full support and determination to finish the thesis. Finally, I would like to thank my parents, Renate and Karl-Heinz, grandmother Johanna, grand-father Heinrich, aunt Christiane, uncle Lutz and all family relatives for their love and support.

Contents

Chapter 1

INTRODUCTION

Embedded systems are prevalent in today's society and promise to be even more pervasive and found in many of the things we interact with in our daily lives in the near future [122]. Applications vary from today's airplane jet or car controllers, and communication devices like cellular phones to the future's autonomous kitchen appliances, and intelligent vehicles. The trend in semiconductor industry is that the Internet and e-commerce will change our lives and impact the semiconductor industry even further. A market forecast [29] of the structure of worldwide electronic production in 2010 is presented in figure 1.1. It shows that 60% of electronic production will be among embedded applications. One sector that stands out of the embedded market is communications (30%), while consumer (15%), industrial (12%) and automotive and defense (14%) have about the same percentage.

A second trend is that in the five key industrial sectors with a high share of microelectronics, software plays a more and more important role [54]. The prediction for the total growth from 2002 to 2015 is to 128% and more than doubles the total research and development (R&D) investment growth. Industries where software played a minor role in 2002 (automotive, medical equipment) will increase their effort to more than a third of their R&D volume and sectors which have already a high software rate (consumer electronics, telecom equipment) will raise the software R&D budget to over 60%. These figures show that the market share for embedded systems is growing and a high portion of industrial research and development activities will be dominated by embedded systems [65]. The complexity of embedded systems will inevitably increase to meet numerous demanding requirements.

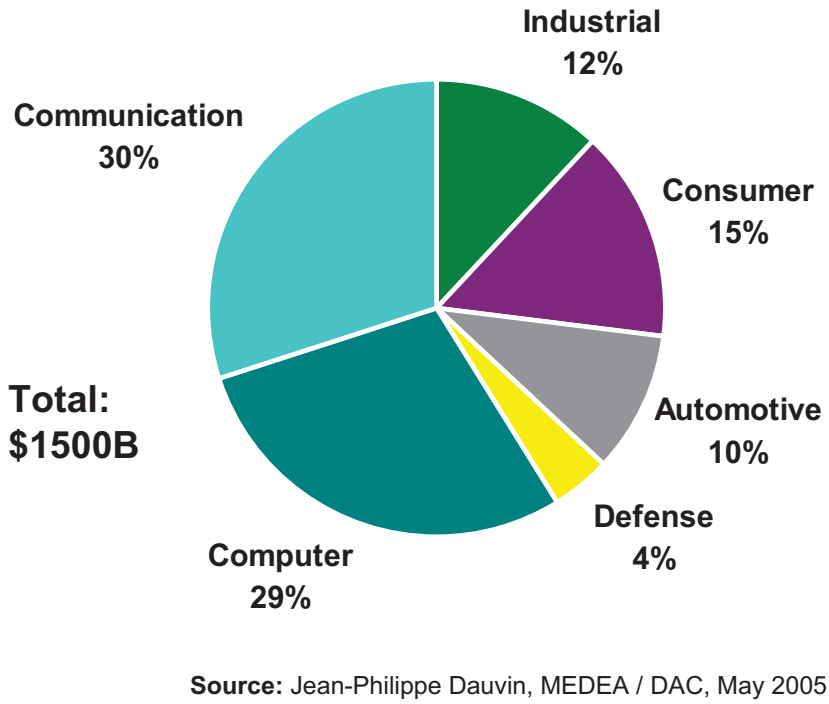

Figure 1.1. Structure of worldwide electronic production in 2010 [29].

1.1 Embedded system requirements

Embedded systems have to satisfy an increasing number of requirements. The *time-to-market* period is getting especially important as product life-cycles are constantly decreasing, such as in multimedia, telecommunication and consumer electronics. A rapid development is crucial for a successful product placement while functional as well as non-functional requirements are essential. The embedded system development process is studied from an economic viewpoint in [65].

Functional requirements

Correctness is a fundamental requirement to guarantee that an embedded system properly operates. Many hardware- and software-tests are carried out to verify functional behavior.

Flexibility is used in two different senses: configurability and re-configurability. The advantage of a configurable system is that the manufacturer can simplify the development for a variety of product lines. Re-configurability gives the customer the ability to use a device for different applications. For example, a firmware update is much less expensive than the exchange of hardware components in case software errors are detected.

Reliability varies strongly for different embedded systems. While a system crash of a cellular phone is acceptable once a year, a similar rate for safety-critical systems like in aerospace applications would be disastrous. Fault tolerance and quality of service (QoS) are terms used to describe the necessary robustness of an embedded system.

Non-functional requirements

Timing behavior denotes the time delay within a software task finishes its computation. For most embedded applications, a calculation has not only to be correct but has to finish before a specified time period. The term timing behavior covers a broad spectrum: from best effort strategies and quality of service, e.g. in networking domain, real-time constraints, e.g. for MPEG video processing, to safety-critical requirements, e.g. airbag in automotive.

The term *real-time* is often used when the embedded application reacts to signals from its environment. Such systems are further distinguished into *soft real-time* and *hard real-time* systems. In soft real-time systems, timing behavior is considered an important aspect yet is not essential to correct functional behavior. The quality can be reduced in case of timing bottlenecks and it is considered as correct functional behavior. As an alternative, the software task could be switched to a different processing mode, in which less accurate results are computed within a shorter time. In hard real-time systems, the application must finish before a pre-defined deadline. The term deadline denotes the longest acceptable time period before the computation has to finish. Examples are engine control software in automotive, flight control software in avionics systems and pacemakers.

Micro-systems with real-time behavior have been developed to make driving more secure. Micro-systems are embedded systems, in which electronic components are combined with micro-mechanical, micro-optical or micro-fluidic components. The automotive supplier Continental-Temic developed a lane-keeping system that automatically controls the car navigation. Another product by Ibeo is the Alasca XT laser, which observes the area in front of a vehicle from 30cm to 200 meters and can compute up to seven different functions simultaneously [33] [41].

A low *power consumption* is essential for mobile devices. A lower power dissipation allows lighter and smaller products as well as longer operation times. Other important product requirements are size, weight and design.

In order to meet these requirements, an appropriate hardware and software architecture has to be chosen. Often requirements have opposite goals: a short time-to-market

window versus an efficient hardware and software design. This thesis focuses on timing requirements and discusses the challenges of advanced hardware architectures.

1.2 Architecture and application properties

In this section we survey properties of modern embedded architectures including processors, memory hierarchy and operation systems. Then, we discuss their impact on the timing behavior of software applications.

Architectures

Embedded architectures consist of one or several processors with several memory devices and peripheral units. An example of an embedded architecture is shown in figure 1.2. The TriCore 1796 processor [51] is used in the automotive domain for engine control units (ECU)s. It consists of three cores, an instruction cache, scratch-pad memory, other memory units, several IP components, for example a CAN bus interface and several busses. The scratch-pad RAM, which is a SRAM memory, holds frequently executed instructions, e.g. of the operating system, to prevent cache replacements.

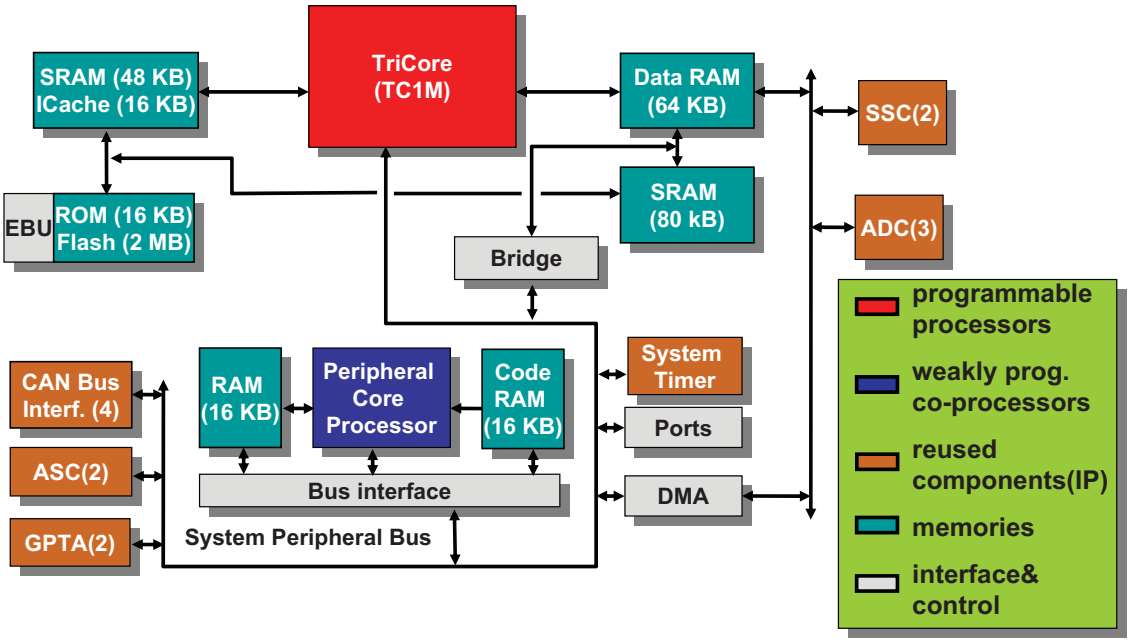

Figure 1.2. TriCore 1796 Architecture [51].

Such highly integrated systems are also called system-on-chip (SoC) because many components are integrated on a single chip.

Digital signal processors (DSP) are used for certain application domains, dedicated or weakly-configurable co-processors (ASICs respectively FPGAs).

The notion of a Network-on-Chip is been used to extend the classical bus-based interconnection, which is still the dominant interconnect structure for SoC's, into a dedicated, segmented and, possibly, packet-switched network fabric [12] [79].

Processor speed as well as fast memory access times are essential goals in processor design. As main memory is usually slow but cost-efficient, a hierarchy of memory devices are used.

Caches

Caches are small memory devices to bridge the gap between fast processor speed and slow main memory [45]. Frequently used memory blocks are stored in the cache while the capacity is significantly smaller than main memory. The time to access data in the cache from the processor is very fast, e.g. a single clock cycle, if the requested data is available in the cache. Otherwise, the cache controller has to request the data from main memory, which takes much longer, typically 20 - 100 or more clock cycles. This introduces a highly dynamic timing behavior because it is statically difficult to predict the cache contents. In a Harvard architecture, a cache is separated for instructions and data. Instruction caches hold only the program code and are only read from the processor while data caches hold the data, which are read and written from the processor. We give an overview of different cache architectures in section 2.1.

Operating systems

Embedded systems usually compute several tasks in parallel. For example, the following tasks have to be calculated at the same time in a mobile phone: estimating the current position, searching an address in the organizer, receiving an incoming phone call and responding to user interaction. An *operating system* schedules several tasks that request a resource at the same time. This process is also called scheduling. Three major classes of scheduling strategies can be distinguished. First, non-preemptive static execution scheduling. It is mainly used for highly regular digital signal processing (DSP) applications. Second, preemptive priority-driven scheduling. Preemptive scheduling means that task execution can be interrupted by another task, e.g. with a higher priority, which is then executed. When the higher priority task has finished, the preempted task resumes.

Preemptive priority-driven scheduling is often used in reactive systems operating in dynamic environments. Priorities can be assigned statically, for example where priority assignment follows rate-monotonic scheduling (RMS), or dynamically, for example where priority assignment follows earliest deadline first strategy.

A third class of scheduling strategies are preemptive time-slicing techniques. These techniques assert a fair distribution of the resource among all tasks but are less efficient, because unrequested time slots are wasted. Static *time-division media access* (TDMA) scheduling with fixed length time slots, can be distinguished from more dynamic *round robin* (RR) scheduling with variable-length time slots. While all scheduling strategies have their merits, we focus on RMS scheduling in this thesis. RMS is very popular in embedded systems because of its efficiency and its ability to guarantee short response times. For example, ERCOSek [31] is used in automotive and VxWorks [138] is used and telecommunications.

Real-time operating systems are operating systems which guarantee a pre-defined longest acceptable response time (deadline) for a task. Often preemptive scheduling policies are used to guarantee short response times. A scheduler interrupts the execution of the currently running task and then assigns the processor to a different task. This process is also called a context switch. It requires an additional time overhead, which is also called *preemption delay*, for saving the hardware state, e.g. the program counter and the register values, and for executing the scheduler itself. In section 2.3 we further discuss different scheduling policies.

Cache related preemption delay

If caches are used, the preemption delay is increased due to the cache interference of the preempting and preempted task. As an example, figure 1.3 shows a schedule where a task τ_1 is preempted by a task τ_2.

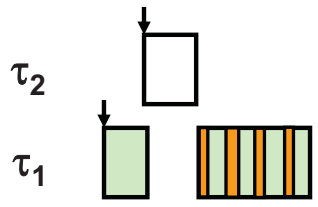

Figure 1.3. Task preemption and cache-related preemption delay (CRPD).

If the preempting task replaced cache blocks of the preempted task, then these cache blocks might have to be reloaded by the preempted task (solid blocks in the

figure). This delay is called cache-related preemption delay (CRPD). Cache blocks are not loaded at once after the preempted task τ_1 resumes but rather on demand when the processor accesses these cache blocks. In section 2.4.1 we describe in detail different techniques to bound this CRPD. This work will make several contributions to a tighter and more time-efficient calculation of the CRPD than previous approaches.

As an alternative to allow all tasks to share the entire cache, cache partition and cache locking strategies can be used to make the timing behavior partly orthogonal for different tasks. We discuss the applicability of these techniques in more detail in section 2.2.

Software development

Complex embedded software is typically developed by different vendors and is combined by a system-integrator. Model-based design tools, for example Matlab/ Simulink [82] and ASCET-SD [9] with automatic code generators, e.g. TargetLink [126], are used. Typically not all software components are known to the system integrator because of intellectual property restrictions. This limits also a full system level performance analysis which a priori assumes all necessary information.

Multiple task execution

An other property of embedded software is its regular execution frequency. For example, in an engine control unit in automotive domain [111], tasks are activated every 1ms, 10ms and 100ms periods and some tasks are activated synchronous to the engine rotation counter. Automatically generated source code, e.g. from Matlab/Simulink, consists of sequential code without loops. An instruction cache, such as found in the TriCore microprocessor, would only marginally increase the execution time, because cache blocks are not re-used. However, if some cache blocks remain in the cache from a previous task execution, than the cache accesses would result in frequent cache hits.

1.3 Challenges for real-time analysis

The demanding requirements for embedded systems and the complexity of the software and hardware architecture of embedded systems lead to challenging problems in real-time performance analysis. The requirement for short response times and a high utilization makes preemptive scheduling very attractive. Today's microprocessors use instruction and data caches to bridge the speed gap between slow memory access time and high processor frequency. The combination of preemptive scheduling techniques processors with caches are common in embedded systems.

State-of-the-art in industry to determine the timing behavior is to simulate an application on real hardware or on a processor simulator using stimuli data. The hope is to cover all relevant execution scenarios. While typical timing behavior can, of course, be obtained by software tracing and simulation, execution time guarantees cannot be made.

On the other hand, it would be possible to restrict cache usage or use non-preemptive scheduling. For example, cache locking and cache partitioning can be used to make cache behavior orthogonal to several tasks. However, these techniques need usually larger caches and main memory to become effective, but caches are typically small compared to main memory in embedded systems to reduce cost. Non-preemptive scheduling is rarely an option to highly reactive embedded systems, since short response time are required.

WCET- and schedulability analysis

As an alternative to simulation and testing, formal methods are being used to prove certain temporal properties of an embedded system [55]. To gain full flexibility of preemptive scheduling and speed-up by caches, performance analyses have been extended to consider cache timing effects. Such performance analyses are structured in a higher-level *schedulability* analysis and a lower-level *worst-case execution time* (WCET) analysis. While schedulability analysis considers the scheduling policies and system level parameters and abstracts the timing behavior of a single task, WCET-analysis focuses on a single task execution and simplifies global system level effects.

The following simplifications in WCET-analysis and schedulability analysis lead to pessimistic overestimations, when caches are used. On the one hand, WCET-analyses assume a single task execution and thus assume an empty cache at task activation [1] [8] [71] [127] [141]. Memory access times and cache miss penalties are considered to take always the worst case time. It is assumed that task is not interrupted by any other task. This means that only intrinsic cache effects are considered, e.g. only during a single task execution.

On the other hand, schedulability analyses assume the WCET to be available. For preemptive task scheduling on architectures with caches, cache-related preemption delays increase the worst-case execution time, and have to be considered as context switch times. But these delays depend on the cache behavior of tasks, which is the domain of WCET analyses. Existing cache-aware schedulability analyses [19] [67] [92] assume that each preemption takes place at the worst-case preemption point.

The challenge is to tightly bound the preemption delay and cache timing behavior and not just to deliver *a* bound. For this reason, the strict separation of schedulability analysis and WCET analysis has to be overcome. Valuable information in each analysis has to be made available for the other one. We identify the following challenges for schedulability and WCET-analysis.

Multiple task activation

As noted in the last section, we studied control intensive applications which contain only sequential code without loops. Caches cannot increase the performance for such applications, because linear code significantly limits the spacial and temporal locality of memory accesses for which a cache is optimized. However, a task is not executed only once. It is executed repeatedly. Previous cache blocks might be available from previous task activations and could reduce the prediction of cache misses in a WCET analysis.

Multiple preemptions

Second, if a task is preempted multiple times, it is unlikely that it will be preempted always at the worst case preemption point. In the approach by [66] the most expensive preemption points have been determined but only the preempted task was considered. Even this modeling leads to an overestimation, because the preemption history is important. For example, a cache block that is replaced at a preemption point cannot be replaced a second time, if the preempted task did not reload this cache block between the first and second preemption point.

Indirect preemptions

In schedulability analysis it is often assumed that each task preemption causes a cache interference [67] [19]. However this is not necessarily always the case. When a task τ is interrupted by a task τ_k, then τ is suspended. If the task τ_k is interrupted itself several times by an even higher priority task τ_i, only the first task interruption by τ_i might remove some cache blocks of τ. Since the task τ is suspended no cache blocks are reloaded between the preemptions of the task τ_k. And thus further preemptions by task τ_i have no negative cache effect for task τ.

Time complexity of schedulability analyses

Another challenge for real-time analyses are their time complexity. The time complexity of a formal analysis technique is crucial when designing larger embedded

systems. There have been analysis approaches for cache-related preemption delay analysis which are very time-consuming but accurate [89] and less accurate ones with a lower time-complexity [66] []. From the design perspective, different levels of analysis precision are desirable.

Data caches

Data caches have been a great challenge in static timing analysis. Their behavior is more difficult to predict, because it depends on control flow of the application but also on the input data. While instruction addresses are fixed, a single instruction can access different data memory addresses, for example operations on an array.

Measurement-based WCET analysis

Measurement-based WCET analysis approaches, such as [14] [91] [139] [135], are attractive to meet industrial strength requirements, as they are easy to re-target and cost-efficient. The principle is to measure the execution time of an application on real hardware by inserting intrusive instrumentation probes. The main drawback in using instrumentation probes is that they disturb the temporal behavior of the application, i.e. the execution time of the program differs when the probes are removed. Furthermore, the initial hardware state is difficult to assert during the measurements. The main challenge is to minimize the measurement overhead.

1.4 Contributions

This dissertation makes the following contributions:

- First, we propose a novel schedulability analysis for fixed priority preemptive scheduling to consider timing effects for associative instruction caches at a context switch. The preemption delays are calculated by considering the preempted as well as the preempting task. The proposed schedulability analysis bounds the number of preemptions more tightly by excluding infeasible cache interferences by considering indirect preemptions. The analysis is conservative, e.g. determines a safe upper bound of the preemption delay, and has a low time complexity.

- As a refinement, the cache interference by multiple task preemptions is analyzed. While previous approaches calculate the worst-case preemption point and assume that each preemption takes place at this preemption point, we consider the preemption history in the calculation of the total cost for multiple task preemptions. The advantage is that that the bound of the total preemption delay of a multiple

task preemptions can be smaller than the sum of the preemption delay bounds of each preemption.

- Execution time verification is often used in different levels in system design. Less precise estimates are acceptable in early design stages while highly accurate ones are necessary for verification of hard real time constraints. Two approaches to bound the preemption delay have been proposed which both use data flow techniques but differ significantly in respect to time-complexity and analysis precision.

- Existing timing analyses focus on a single task execution. However, embedded applications are activated very frequently if not regularly. Cache lines from a previous task activation might still be available in the cache and need not be loaded during a subsequent task execution. This effect of *multiple task execution* can result in a significantly reduced number of cache misses. In this thesis we estimate a conservative bound of the cache contents at the beginning of task activation and consider the effect in instruction cache timing behavior.

- While the previous analysis techniques focus on instruction caches, we also provide a novel timing analysis for data caches. Data cache behavior is more difficult to predict, because it depends on control flow of the application but also on the input data. While instruction addresses are fixed, a single instruction can access different data memory addresses, for example operations on an array. In this thesis we propose a static timing analysis for data caches which considers input data dependency of memory accesses.

- Finally, we integrate instruction and data cache timing analysis in a measurement-based WCET-analysis tool, which has been developed in previous work. Measuring the execution time requires insertion of instrumentation points which disturbs the temporal behavior of an application. In this thesis we present a novel instrumentation methodology that reduces the number of instrumentation points.

1.5 Overview

This thesis is structured as follows. Background information on cache memories, predictable cache usage, previous cache-aware schedulability analyses as well as preemption delay analyses is presented in chapter 2. The novel cache-ware response time analysis for instruction caches considering indirect preemptions, multiple preemptions and multiple task activations is presented in chapter 3. The scalable precision cache analysis is presented in chapter 4 for direct mapped as well as for associative instruction caches. Data cache analysis is presented in chapter 5 and in chapter 6 we

present the novel instrumentation point selection method before we summarize and conclude in chapter 7. In Appendix A we describe the prototype SymTA/P for WCET calculation and a case study.

Chapter 2

CACHE BEHAVIOR IN EMBEDDED SYSTEMS

In this chapter, we provide background information about cache behavior and performance analysis for embedded systems. The architecture and functionality of caches are presented in section 2.1. Predictable cache usage techniques are described in 2.2. Then, we review scheduling techniques and schedulability analyses in general in section 2.3 and describe cache-aware schedulability analyses in section 2.4.

The last part of this chapter, section 2.5, explains the technical intricacies of data flow algorithms. These algorithms are employed in the thesis in various parts: in section 3.3.3 to calculate the delay for preemption scenarios for instruction caches, in section 4.1 to motivate the scalable precision cache analysis for instruction caches, in section 5.4.2 to compute persistent cache blocks for data cache analysis, and finally in appendix A.1.3 as part of the SymTA/P analysis framework.

2.1 Cache memories

Caches are small memories close to the processor. They increase the performance because they bridge the gap between fast processor speed and long memory access time [45]. In this section we discuss the basics of semiconductor memories in section 2.1.1, common cache architectures in section 2.1.2, and cache access mechanisms in section 2.1.3. In section 2.1.4 we summarize which cache effects are considered in this thesis.

2.1.1 Semiconductor Memories

The ideal memory would be low cost, high performance, high density, with low power dissipation, random access, non-volatile, easy to test, and highly reliable [95].

Unfortunately, a single memory having all these characteristics has not yet been developed, however, a system combining dynamic RAM and static RAM memories achieves many of these characteristics.

A DRAM (dynamic random access memory) is a MOS memory (Metal Oxide Semiconductor) which stores a bit of information as a charge on a capacitor [95]. Since this charge decays away in a finite length of time a periodic refresh is needed to restore the charge. The advantages of a DRAM is its small size, since a basic memory cell consists of a single transistor and a capacitor and thus a low cost per bit. The disadvantage of a DRAM is that it is volatile. The memory cells do need to be refreshed.

A static RAM (SRAM) cell consists of a basic bistable flipflop circuit containing four transistors. No periodic refresh is required. The only disadvantage compared to DRAMs is that of size and, as a result, of cost. The advantage is a faster memory access time than a DRAM.

The processor performance and the memory bandwidth demand is constantly increasing because of higher processor frequency and super scalar execution. The advances of DRAM technology to increase memory access time are by no means sufficient to cope with the increased bandwidth demand. With a relatively small, but very fast SRAM memory or a memory hierarchy, it can be achieved, that a system with slow DRAM main memory operates as fast as a system with fast SRAM as main memory [11]. Such a small memory device is called a cache.

2.1.2 Cache architecture and functionality

Principle of locality

A cache is a small fast memory close to the processor. The principle of locality has driven its architecture design. The principle of locality comes in two flavors: temporal and spacial locality. First, *temporal locality* means that a cache block is used several times during a short time period, e.g. in loops. Second, *spacial locality* means that if a memory word is requested than it is likely that other memory words near by will be accesses too.

Architecture

A cache consists of several *cache blocks*, also called cache lines, which represent several memory words. Typically, the size of a cache block ranges is 4, 8 or 16 memory words in embedded caches. When a memory block is requested from the processor, then the entire cache block is loaded to the cache. Then, an access to

memory blocks within this cache block can be serviced by the cache and no main memory traffic is necessary.

A memory block can be placed in the cache in different cache blocks. A cache can be direct mapped, *n*-way associative or full-associative. For an *n*-way associative cache, *n* cache blocks are organized in a cache set, as shown in figure 2.1. In a direct mapped cache each cache set has exactly one cache block, e.g. the associativity is one. A memory block can be placed in any block in the cache set it is mapped to.

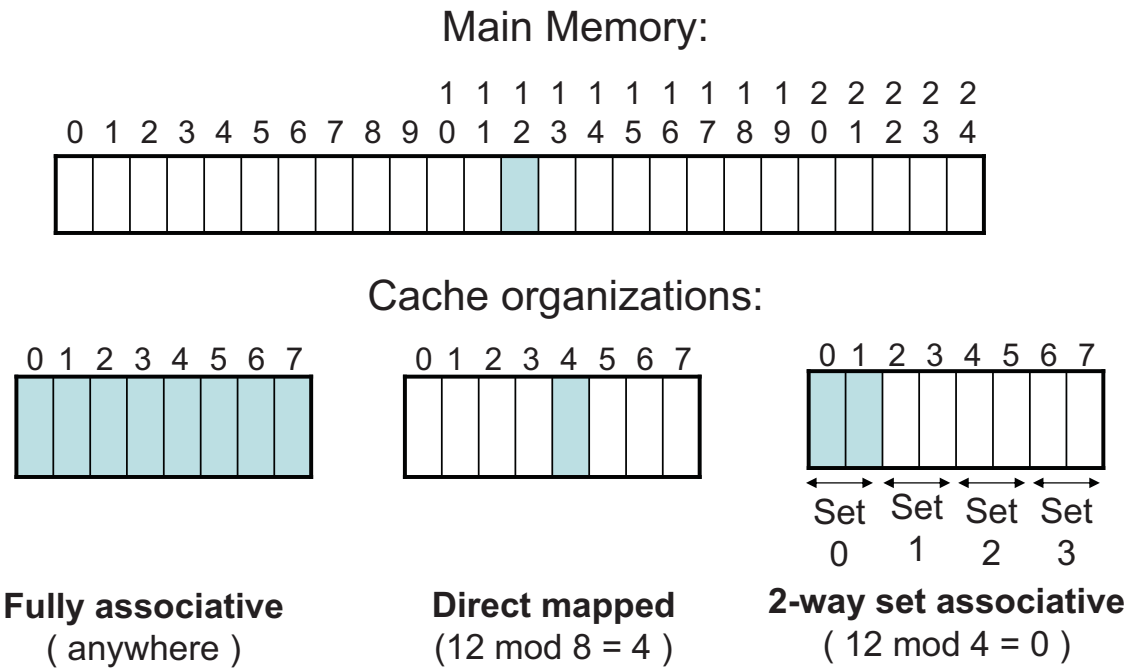

Figure 2.1. Cache architectures.

The address in a cache is split in block address and block offset, in which the block address is further divided into the tag field and index field. The block offset field selects the desired data from the block, the index field selects the set, and the tag field is used to compare the address of the requested memory block.

While instructions can only be read, data can be read and written. Thus, a cache is often separated into a data and an instruction cache. This architecture is called a *Harvard* architecture. If the cache is unified, it is called a *von Neumann* architecture.

Functionality

When the address matches the tag, the cache access is called a *cache hit*, otherwise a *cache miss*. In this case, the entire cache block is requested from main memory. The time-delay to access main memory is called *cache miss penalty*. When a miss occurs, the cache controller must select a block to be replaced with the desired data.

For direct mapped caches, there is only a single choice. In associative caches, a replacement strategy decides which block is discarded. Common strategies are least recently used (LRU), random or round robin (as used in Intels 80200 Xscale processor [53]).

When a data is written to the cache, write policies often distinguish cache designs. There are two basic options when writing to the cache: A *write through* (or store through) cache writes the information to both the block in the cache and to the block in main memory. A *Write back* (also called copy back or store in) cache writes the information only to the block in the cache. The modified cache block is written to main memory only when it is replaced. A *dirty bit* is commonly used to reduce the frequency of writing back blocks on replacement. To reduce writing stall time, write buffers are used, which allows the processor to continue as soon as the data is written to the buffer, thereby overlapping processor execution with main memory updating. For example, a write buffer of eight 64-bit entries is used in ARM1020E [47] in the 32kB data cache. The write policy is configurable write through or write back. The Trimedia [132] has a 16kB 8-way associative write allocate dual ported data cache supporting two parallel accesses (load/store).

Since a memory block is not needed on a write, there are two common options on a write miss: First, *Write allocate* (also called fetch on write) where the block is loaded on a write miss, followed by the write-hit actions, and second, *no-write allocate* (also called write around) where the block is modified in the main memory and not loaded into the cache.

Generally, write-back caches use write allocate (hoping that subsequent writes to that block will be captured by the cache) and write-through caches often use no-write allocate (since subsequent writes to the block will still have to go to memory).

2.1.3 Improving cache performance

Much research has focused on improving cache performance. The techniques can be organized in three categories: reducing cache miss rate, reducing the miss penalty and reducing the hit penalty. We briefly explain some techniques. A more detailed description can be found in [45].

Reducing cache miss rate

The cache miss rate can be reduced by a larger block size, higher associativity, a victim cache, a pseudo-associative cache, hardware prefetching, compiler controlled prefetching, or compiler optimizations. Larger blocks take advantage of spatial lo-

cality. At the same time, larger blocks increase the miss penalty, thus there is an optimum for a given architecture depending on the cache miss penalty. The reasons for a miss rate improvement by higher associativity can be summarized as follows: First, an eight-way associative cache is for practical purposes as effective in reducing misses as a fully associative. The second observation, called *2:1 cache rule of thumb*, is that a direct-mapped cache of size N has about the same miss rate as a 2-way set-associative cache of size N/2. However, greater associativity can come at the cost of increased hit time [46].

Victim cache

A third miss rate reduction technique are victim caches. The idea is to add a small, fully associative cache between a cache and its refill path. This *victim cache* contains only blocks that are discarded from a cache because of a miss (victims) and are checked on a miss to see if they have the desired data before going to main memory. If it is found there, the victim block and the cache block are swapped. Jouppi [58] found that victim caches of one to five entries are effective at reducing conflict misses.

Pseudo-associativity

Another approach to get the miss rate of set-associative caches and the hit speed of direct mapped caches is a *pseudo-associative* (or column associative) cache. A cache access proceeds just as in a direct-mapped cache. On a cache miss, another cache entry is checked to see if it matches there. A simple way is to invert the most significant bit of the index field to find the other block in the 'pseudo-set' [2].

Prefetching and other techniques

Another approach to reduce the miss rate without affecting the processor clock rate is to prefetch items before they are requested by the processor. For example the Alpha AXP 21064 fetches two cache blocks on a miss: the requested block and the next consecutive block. Prefetching relies on utilizing memory bandwidth that otherwise would be unused, and can actually decrease the performance if it interferes with demand misses. Compiler controlled prefetching can reduce useless prefetching and, finally, compiler optimizations can reduce miss rate without any hardware changes by code reordering including merging of arrays, interchanging nested loops or fusing loops.

Reducing miss penalty

To reduce memory traffic, only *sub-blocks* can be replaced. A valid bit is added to units smaller than the full block, called sub-blocks. Only a single sub-block has to be read on a miss. The valid bits specify some parts of the block as valid and some parts as invalid, so a match on the tag does not mean the word is necessarily in the cache, as the valid bit for that word must also be on. For example, the instruction cache of TriCore 1775 micro-processor [50] uses sub-block placement at the size of double-words.

While sub-block placement requires extra hardware, the following two techniques are based on the observation that just one word of the cache block is needed by the processor at a time. *Early restart*: As soon as the requested word of the block arrives, it is sent to the processor and the processor can continue execution. *Critical word first*: The missed word is requested first from memory and is sent to the processor as soon as it arrives; the processor can continue execution while filling the rest of the words in the block.

For example, the embedded processors TriCore [52] [145], e.g. 1775 [50] and 1796 [51], Trimedia [132], ARM920 [7] ARM1020E [47] and Intel 80200 (Xscale), [53], support critical word first.

Another way to reduce the miss penalty is the hierarchical design of caches. By adding another level of cache between the original cache and main memory, the first-level cache can be small enough to match the clock cycle time of the fast processor, while the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty. For example, the PowerPC 750GX includes 1 MB of internal L2 cache, which is 4-way set-associative, running at core frequency, which can be locked at the granularity of ways in all cache sets, and additional L1 and L2 cache buffers allowing up to four data miss operations [49].

Reducing hit penalty

Hit time is crucial because it affects the clock rate of the processor; on many machines the cache access time limits the clock cycle rate, even for machines that take multiple clock cycles to access the cache. A time-consuming portion of a cache hit is using the index portion of th address to read the tag memory and then compare it to the address. A guideline is to keep the cache simple, such as using direct mapped cache, especially for data caches. A main benefit of direct mapped caches is that the designer can overlap the tag check with the transmission of the data.

2.1.4 Considered cache effects

This thesis assumes the following cache architectures and execution models:

- A Hardward architecture with separate instruction and data caches.
- Associative instruction caches with LRU replacement strategy. We assume a constant cache hit and cache miss penalty.
- Direct mapped data caches with write through, no-write allocate write miss strategy and a constant cache hit and cache miss penalty.
- Hardware prefetching, write-buffers and multi-level caches are not considered.
- Replacement of the entire cache lines is assumed. The advanced features critical-word first and sub-block placement are not considered.
- The entire cache is shared by all applications.

While, critical-word first, sub-block placement, and pseudo-LRU replacement strategies are certainly very useful to improve cache performance, they need not be considered in the principles in the proposed cache analysis as these features are deterministic. The proposed cache analysis uses local cache simulation, and thus an extension of the cache simulation module would be sufficient. How our analysis could be extended to more complex hardware and software cache optimizations is discussed in section 7.

The proposed analysis framework considers the following cache effects: First, the cache effects at context switches are analyzed in a cache-aware response time analysis in section 3.2. In section 3.3 multiple preemptions are more tightly bounded than assuming for each preemption the worst case preemption delay.

Then, the effects due to multiple task execution are considered in section 3.4: Cache lines from a previous task execution might be available when the task is activated and thus a non-empty cache state can be considered in the worst-case execution time analysis.

In chapter 4 we take a closer took at the computation complexity to analyze the cache effects for preemption delay analysis.

While the above cache effects considered instruction caches, we further analyze data cache timing behavior during a single task execution in chapter 5. We focus on input-data dependency, because this is the main reason for the dynamic cache behavior.

2.2 Predictable cache usage

There are several approaches to make caches more predictable and efficient. They can be classified in cache partitioning, cache locking and the usage of scratch-pad RAM. A detailed overview of different techniques can be found in [107].

2.2.1 Cache partitioning

The advantage of cache partitioning techniques is that cache blocks do not have to be reloaded after interrupts and between consecutive executions of the same task. Also, cache behavior becomes (partly) orthogonal for tasks and, therefore, more predictable. Cache partitioning can be controlled by hardware or software.

Hardware based partitioning

Kirk [61] describes a hardware partitioning technique dividing a fully associative instruction cache memory into a static part that is pre-loaded at a context switch and a regular LRU-part. The idea is to guarantee a certain hit-ratio because some cache blocks are fixed resulting in a certain amount of cache hits. Whether a cache block is placed in the static or the LRU-part is decided either statically at compile-time or dynamically at run-time by maintaining frequency counters. In [62], a partitioning scheme called SMART (strategic memory allocation for real-time) is proposed with a shared segment and a number of private segments for each task. A private segment will eliminate extrinsic cache interference and reduces cache related preemption delay at a context switch. In [63] Kirk and Strosnider describe an implementation of SMART on a MIPS R3000 CPU with a 16kB direct mapped instruction cache. The extra hardware needed to handle the partitioning renders a 15% performance loss on a 25MHz processor.

Another hardware-oriented approach is to use a modified hash function for a direct mapped data cache [88]. The partitioning of a cache by additional hardware has the drawback that specially designed cache hardware is necessary. Data coherency must also be maintained since data structures can be duplicated in more than one partition.

Software based partitioning

As an alternative, a cache can be partitioned by software. Wolfe suggests in [144] [143] to partition instructions to memory addresses. Mueller [86] automates Wolfe's ideas by letting the compiler and linker assign tasks to addresses. Instruction partitioning is solved by non-linear control-flow transformations and the data partitioning uses transformations of data references. Large tasks may not entirely fit into their

partition, so the code must be located in memory such that it will only be mapped to its cache partition. Separated code sections are connected with unconditional jumps, which are inserted by the method. The advantage of this approach is that a task cannot suffer from cache related preemption delays. Nevertheless, the task will suffer from intrinsic cache misses: the more tasks having their own partition, the smaller the size of their partition, and consequently the higher the number of intrinsic cache misses.

A hybrid instruction cache partitioning technique is presented by Busquets-Mataix and Wellings in [17]. It describes a technique, which partially partitions a cache. It either provides a task with a *private* partition or assigns several tasks to a *shared* partition. For the shared partition the preemption delay of these tasks in accounted for in a cache-aware schedulability analysis. The assignment of cache partitions to tasks in an optimization problem which is exponential to the number of tasks.

Optimizing task layout

Task layout techniques are suggested which aim at minimizing the inter-task interference in the instruction cache. Task sets which may not be schedulable when the layout of tasks in main memory is arbitrarily chosen, might become schedulable provided that the layout minimizes the CRPD. From this viewpoint, task layout can be considered as an advanced form of (software) partitioning.

In [28] a technique is proposed to find out the optimal task layout which minimizes the worst case response time of *one* task and satisfies the deadline of all the other tasks. The method uses ILP (integer linear programming) formulations and the experimental results show that the method renders better performance than arbitrarily chosen layouts. The method is only feasible on direct mapped instruction caches with task sets containing periodic tasks.

Instead of allowing all tasks to share the cache, Lin and Liou propose in [74] to disable the cache to all tasks but the most privileged or frequently used ones. Intrinsic cache misses as well as CRPD are eliminated. The authors claim that if a task that does not fit into the cache, it will yield about the same WCET as in a system without a cache. However, in [8] is is shown that this approach is too pessimistic and that the worst case performance is better for a cached system compared to a non-cached system. Also a dynamic cache partitioning technique has been proposed in [120].

2.2.2 Cache locking

Another approach for predictable cache behavior is to lock frequently used cache blocks. The ability to lock cache blocks is available in most common processors

(ARM9 [7], ARM1020E [47], Intel's XScale [53], PowerPC 7448 [37], PowerPC 440 families [94] modern G2,G3, G4 CPUs [102], Motorola Cold Fire MCF5249, MIPS32 and others). Each processor implements cache locking in several ways, allowing static locking (the cache is loaded or locked at system start) and dynamic locking (the state of the cache is allowed to change during the system execution). Examples how a cache can be locked is given in the following. In Intel's 80200 XScale processor 32-way instruction and data caches, each of the 32 sets, 0-28 ways can be locked, unlocking ways are replaceable via a round robin policy. The PowerPC 750G allows locking the 4-way associative cache by way and the ARM1020E's 32kB 64-way associative instruction and data-caches support locking by line. Therefore, the cache contents is known and the time required for a memory access is predictable.

Campoy et al. propose in [25] a genetic algorithm to choose which instructions to lock in the instruction cache. They represent each memory block by one bit, which flips between zero and one (in/out of the cache). Static cache locking is not a suitable solution for data caches because it is difficult to evaluate the data cache timing behavior which is much more difficult than instruction cache behavior [134]. The use of the genetic algorithm causes a long computation time.

Two low-complexity algorithms for static cache locking have been presented in [97] [96]. In contrast to [25], the proposed algorithms select the contents of the locked cache in a non-blind manner, by using the tasks memory access patterns of the instruction flow. Both algorithms optimize the worst-case behavior of a task set: the first one aims at minimizing the CPU utilization of a task set, and does not prescribe any particular schedulability analysis method, whereas the second one, designed for fixed-priority schedulers, aims at reducing the interferences between tasks due to preemptions.

The algorithms lock *all* cache blocks to some program lines. Typically the program is much larger than the cache and thus only a small part of the program will *always* be in the cache. All other program lines will never be in the cache and require a time-consuming main memory access.

Both approaches [97] and [25] have been compared regarding analysis precision and time-complexity in [24]. Both algorithms deliver about the same analysis precision whereas the algorithm by [97] is much faster than the genetic algorithm of [25].

2.2.3 Scratch-pad RAM

Cache partitioning and cache locking approaches increase area and power cost as they require greater caches and main memory to become effective. Therefore, het-

erogeneous memory architectures with caches and scratch-pad SRAM have been introduced, for example TriCore [52] as shown in figure 1.2. A scratch-pad can hold frequently used cache blocks. Compiler techniques for such architectures have been proposed e.g. by Panda, Dutt and Nicolau [90], Udayakumaran and Baruah [131] and Angiolini, Benini and Caprara [5].

2.2.4 Discussion

While cache partition and cache locking strategies are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the general cache interference problem which is critical for large task sets. Embedded caches are small and thus partitioning or locking would increase intrinsic cache interference. Processor schedulability analyses have been extended to analyze the cache behavior where all tasks share the cache.

2.3 Scheduling and schedulability analysis

When multiple tasks share a single resource, then two or more task may request the resource at the same time. Scheduling resolves these conflicting requests. To properly design an embedded real-time system it is important to understand the effect that scheduling has on overall performance [20] [56] [101].

2.3.1 Scheduling

When multiple software tasks execute on a single processor, a scheduling policy decides how often and how long a task is executed. A deadline is defined for each task in real-time systems that denotes the longest execution time under which the functionality of the application can be guaranteed.

A software task is activated due to an activation event. Activating events include expiration of a timer, external or internal interrupt, and task chaining. The purpose of a scheduler is to arbitrate between multiple tasks that want to simultaneously use the same resource. A scheduler selects a task to which it grants the resource out of the set of active tasks according to some scheduling policy [21]. Other active tasks have to wait. The most complex schedulers are part of an operating system that schedule software tasks on embedded processors. But schedulers are also implemented in hardware to control access to other types of resources. For example communication via a shared bus requires bus access scheduling. Memory bandwidth can be considerably increased for certain types of memories, e.g. SDRAM, if memory-accesses are scheduled smartly [43]. Three major classes of scheduling strategies can be distinguished [21]:

1 Non-preemptive static execution order scheduling. It is mainly used for highly regular digital signal processing (DSP) applications.

2 Priority-driven scheduling. It is often used in highly reactive systems operating in dynamic environments. Here we further distinguish scheduling with fixed priorities and scheduling with dynamic priorities. In the fixed-priority case, priority assignments often follow a *rate-monotonic scheduling* (RMS) or *deadline-monotonic scheduling* (DMS) strategy, in the dynamic-priority case, priority assignments often follow an *earliest deadline first* (EDF) strategy.

3 Preemptive time-slicing techniques. These are used for a fair distribution of the resource among all tasks. Here we further distinguish static *time-division media access* (TDMA) scheduling with fixed length time slots, and more dynamic *round robin* (RR) scheduling with variable-length time slots

A major advantage of static execution order scheduling is that it requires no run-time scheduler and hence incurs no scheduling overhead. However, the flexibility that is needed in modern embedded systems, as well as data-dependent task execution times and increased interaction with the system environment more and more necessitate dynamic scheduling. Static scheduling is thus typically limited to highly regular applications. An example is ERCOSEK [31], in which tasks with the same period are statically ordered and can be preempted by other tasks with a different period.

TDMA scheduling essentially separates a real resource into multiple independent virtual resources, each with a guaranteed fraction of the available processing time or communication bandwidth. This is considered attractive for real-time applications in domains such as automotive, in particular because prediction of response times is easy, and because faults in one virtual subsystem cannot impact in a different virtual subsystem. However, TDMA scheduling is inefficient in the presence of dynamically changing loads, since a time-slice large enough to handle a worst-case load remains partially unused in all other cases. TDMA is also poorly suited to achieve short response times. Unless a system is globally synchronized, which is problematic , an activation may wait for a full turn (the sequence of all time slots) in the worst-case until it is granted a TDMA resource. It will also be interrupted for additional turns if it does not complete within one time slot. Round Robin (RR) scheduling [125] resolves the problem of unused TDMA resources , but short response times still cannot be achieved. Additionally, the calculation of worst- and best-case response times is complex. RR scheduling is rarely used in hard real-time applications.

Reactive, priority-based scheduling does not have the inefficiencies of TDMA. In particular, it is well suited for multi-processor systems, where a task is activated by the arrival of data from a different task. High priority tasks can immediately react to data arrival, thus ensuring short response times. The tasks also do not waste resources if they are idle, instead leaving a resource full to lower-priority task. While EDF-scheduling has been proven to be theoretically optimal in the sense that if EDF cannot guarantee that all deadlines are met, no other strategy can [22], it produces considerable scheduler overhead, since priorities have to be frequently re-calculated. Static priority scheduling is theoretically less efficient, but scheduler implementation is simple, and the smaller scheduler overhead often compensates for the theoretical inefficiencies [22].

2.3.2 Schedulability analysis

Schedulability analysis calculates the worst case task response times (WCRT). The WCRT is the time between task activation and task completion, for all tasks sharing a resource under the control of a scheduler. Schedulability analysis guarantees that all observable response times will fall into the calculated interval. We therefore say that schedulability analysis is conservative.

Worst case response times can then be compared against deadlines. If all deadlines are met in the worst case, then they will be met in all cases, thus guaranteeing that a system will satisfy all deadline constraints.

In this thesis we focus on fixed priority preemptive scheduling. Several schedulability analysis techniques have been proposed for fixed-priority preemptive scheduling [76] [57] [69] [129]. Liu and Layland [76] show that the rate monotonic priority assignment where a task with a shorter period is given a higher priority is optimal when task deadlines are equal to their periods. They also give the following sufficient condition for schedulability for a task set consisting of n periodic tasks $\tau_1, \cdots, \tau_n$:

$$U = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq n(2^{1/n} - 1) \tag{2.1}$$

where C_i is the worst case execution time (WCET) of τ_i and P_i its period.This condition states that if the total utilization U of the task set is lower than the given utilization bound ($n(2^{1/n} - 1)$), the task set is guaranteed to be schedulable under the rate monotonic priority assignment. Later Lehoczky et al. develop a necessary and sufficient condition for schedulability based on utilization bounds [69].

Due to the maximum utilization bound of 0.69 for a task set with infinite number of tasks, [16] [57] have developed an iterative response time analysis. The worst

case response time occurs when all tasks are released at the same time point (critical instant). The approach allocates in a time window R_i the task τ_i's WCET C_i, the tasks blocking time B_i and the interference produced by the execution of higher priority tasks. The blocking time is the maximum time that a task can be delayed by lower priority tasks due to resource contention.

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil \cdot C_j \tag{2.2}$$

The term $hp(i)$ denotes the set of tasks with a higher priority than task τ_i. The process is iterative because in every step the interference is added to the current window R_i^n, resulting in a longer time window R_i^{n+1} that might include greater interference in the next step. The process is finished when the window stops growing ($R_i^{n+1} = R_i^n$). If the resulted response time for any task is greater than its deadline ($R_i^{n+1} = R_i^n = R_i > D_i$), the task-set is not schedulable.

2.4 Cache effects in schedulability analysis

To consider cache effects when all tasks share the cache, schedulability analyses have been extended by cache-related preemption delays.

2.4.1 Cache related preemption delay analysis

In an embedded system with cache the context switch time depends on the contents of the cache. Measurement-based approaches have been proposed by [84] [27] [108] but safe bounds can not be guaranteed in general, because simulation depends on completeness of input data. Simulation has also been used to identify preferred preemption points [109]. During program simulation an upper bound on the number of active cache blocks is computed and preemption points with minimum reload times are identified. Preemption is then only allowed at these preemption points. However, simulation relies on test coverage. For full path coverage, all execution paths have to be covered, which is very time-consuming and costly. If not every path were tested, a critical scenario might be omitted which could then lead to an underestimated response time.

As an alternative to simulation-based approaches, time delays resulting from a preemption have been integrated into schedulability analysis. Basumallick and Nilsen [10] extend Liu and Layland's [76] schedulability condition by an additional term for the cache interference. One drawback of such a technique is that it suffers from the pessimistic utilization bound 0.69 for larger task sets. The actual time delay for cache

interference is not further analyzed. Many task sets that have a total utilization higher than this bound can be successfully scheduled [69].

In [19] five possible ways to determine the CRPD are given: 1.) The time to refill the entire cache. 2.) The time to refill the cache blocks displaced by the preempting task. 3.) The time to refill the cache blocks used by the preempted task. 4.) The time to refill the maximum number of useful cache blocks that the preempted task may hold in the cache when a preemption occurs. Useful blocks are those cache blocks that are could to be used again [73]. 5.) The time to refill the intersection of cache blocks of the preempting and preempted task [67].

The approach in [19] considers the penalty according to number 1 and 2 of this list. The response time approach of [129] is extended by an additional term for the cache related preemption delay.

To compare several cache-aware response time analyses we introduce the following notation for the response time equation:

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^n}{P_j} \right\rceil \cdot C_j + crpd(j,i,R_i^n) \right) \tag{2.3}$$

where

R_i denotes the response time of task τ_i

B_i denotes the blocking times of task τ_i

C_i denotes the worst case execution time of task τ_i

P_j denotes the period of task τ_j

$hp(i)$ denotes the set of tasks whose priorities are greater than the priority of τ_i.

The term $crpd(j,i,R_i^n)$ denotes the preemption delay when task τ_j preempts (the lower priority tasks $\{\tau_{j-1},\cdot,\tau_i\}$) during the time window R_i^n. It has been defined differently in the related work, as will be explained in equation 2.4 as in [19], equation 2.5 as in [67], equation 2.6 as in [92],. We will give two definitions in this thesis, one simplified estimation in equation 3.3 in section 3.2.1 and one more accurate calculation in equation 3.7 in section 3.2.2.

In the approach by [19] the term $crpd(i,j)$ is bounded by the maximum number of *used cache blocks of the preempting task* τ_j (denoted by γ_j):

$$crpd(j,i,R_i^n) = \left\lceil \frac{R_i^n}{P_j} \right\rceil \cdot \gamma_j \cdot t_{miss} \tag{2.4}$$

The time to reload a cache block is denoted with t_{miss} (cache miss penalty). This estimation is based on a pessimistic assumption that each cache block used by τ_j replaces a memory block that is needed by a preempted task.

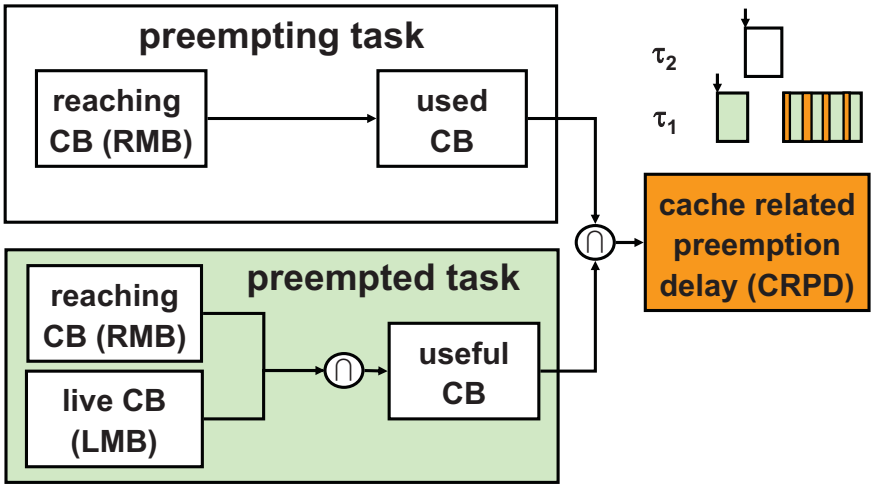

Figure 2.2. Schematic overview of cache-related preemption delay (CRPD) calculation.

Lee et al. [66] [68] introduced the term *useful* cache block to consider the preempted task. A useful cache block is defined as a cache block that is available before a preemption occurs and may be referenced again before being replaced by another memory block. As a consequence, the intersection of useful cache blocks of the preempted task and the used cache blocks of the preempting task bound the $crpd(j,i,R_i^n)$ [67].

$$crpd(j,i,R_i^n) = \left\lceil \frac{R_i^n}{P_j} \right\rceil \cdot \delta_{j,i} \cdot t_{miss} \tag{2.5}$$

This calculation is schematically shown in figure 2.2. Data flow analysis algorithms are set up to bound the preemption delay $\delta_{j,i}$ when a task τ_j preempts a task τ_i. A reaching memory block (RMB) analysis applied to the last node of the control flow graph(CFG) calculates the maximum set of used cache blocks. The set of useful cache blocks can is computed by the intersection of reaching memory blocks and live cache blocks (LMB). In section 2.5.1 and section 2.5.2 we describe two different data flow techniques to calculate the RMB and LMB.

The schedulability analysis by Lee is based on [19] and integer linear programming (ILP) formulation is used to bound the number of preemptions more tightly by considering task phasing. Task phasing means to consider the relationship between task activations of preempting tasks during the response time of the preempted task.

Instead of assuming all activations of higher priority tasks replacing cache blocks, *sets of tasks* executing during the preemption are considered.

This refines equation 2.5 in the sense that not every task activation is considered as cache interference. If for example two tasks, which use the *same n* cache blocks, preempt another lower priority task, than the replaced cache blocks of the preempting tasks are *first* merged and then intersected with the useful cache blocks of the preempted task. This results to a preemption delay of n cache blocks (provided that these are useful). In the simplified version (equation 2.5 a maximum preemption delay of $2n$ would be the result (provided that all cache blocks are useful). Please refer to [67] for the details of computing the term $crpd(j,i,R_i^n)$ in this case.

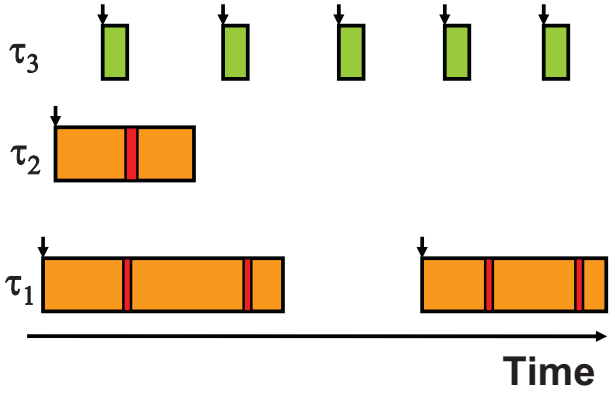

Figure 2.3. Example task schedule.

An example schedule is shown in figure 2.3. We consider only the first activation of τ_1. This task is preempted once by τ_2 and twice by τ_3. Instead of calculating each preemption pair, task τ_2 and τ_3 are considered as a set. This reduces the overestimation, if these tasks share common cache blocks.

To address this problem, a technique is presented, that takes into account the relationship between a preempted task and the set of tasks that execute during the preemption. For this purpose, preemptions of a task are categorized into a number of disjoint groups according to which tasks execute during a preemption. The number of such disjoint groups is $2^k - 1$ where k denotes the number of higher priority tasks. Then, the preemption cost for different preemption scenarios is calculated. More advanced constraints are set up to eliminate infeasible preemption scenarios.

However, some of the invocations of a higher priority task cannot be involved in any preemption of a lower priority task even when we assume the worst-case response time of the lower priority task. This is also shown in figure 2.3. The third task

invocation of τ_3 does not preempt any other task. To consider such effects the ILP is extended by advanced constraints using the hyper-period formed by τ_3 and τ_1.

The main drawbacks of this approach are that it scales exponentially with number of tasks due to the categorization in disjoint groups of preempting tasks and that best case response times (BCRT) are assumed to be known. However, the BCRT analysis is a complicated problem where only approximative solutions have been proposed for the general case [44][40] and the BCRT determination is not described by the authors, instead the best-case execution time (BCET) is used. BCRT can, of course, always be set to zero but then task phasing is not effective.

Tan and Mooney [123] [124] propose a path analysis on the preempted and preempting task, which the authors call intra- and inter-task cache analysis, to estimate the single preemption delay. The schedulability analysis is based on [19]. One drawback of this approach is that strictly all program paths of the preempted and preempting task have to be considered.

2.4.2 CRPD analysis refinements

Cache analysis approaches have been proposed which address only certain aspects in cache-aware response time analysis. First, [130] and [89] propose an approach for single preemption delay analysis without considering schedulability analysis. Tomiyama et al. [130] consider cache effects of the preempting task and ignore the preempted task. They determine the program execution path which uses the maximum number of cache blocks using an ILP technique. Mitra et al. [89] extend the technique of Lee et al. [67] to bound the maximum number of useful cache blocks more accurately by using a state-based approach for the preempting and preempted task. This approach is described in more detail in section 2.5.2. Both approaches target at direct mapped instruction caches.

Second, schedulability analyses by [19] [92] have been proposed with simplified assumptions for single preemption delay. The aforementioned approach by Busquets-Mataix et al. [19] considers as single preemption delay the time to refill the cache blocks used by the preempting task.

Petters presents in [92] an approach that considers the useful cache blocks of preempted task. The response time of a task is computed by substituting in the general equation 2.3:

$$crpd(j,i,R_i^n) = \Delta_{i,j}^k(R_i^n) \tag{2.6}$$

when the $N_{i,j}^k$ preemptions have been considered. Let

- $N_{i,j}^k$ be the number of preemption points still to be considered at step (k) of the iterative procedure; Initially $N_{i,j}^0 = \lceil \frac{R_i^n}{P_j} \rceil$.

- $S_{i,j}^k$ be the set of tasks which potentially suffer from a preemption by task τ_j instead of τ_i. Initially, $S_{i,j}^0 = \tau_i \cup H_i \cap L_j$, with L_j the set tasks with lower priority than τ_j and H_i the set of tasks with higher priority than τ_i.

- $\Delta_{i,j}^k(R_i^n)$ the preemption delay at iteration k, initially 0.

Then $\Delta_{i,j}^k(R_i^n)$ is computed iteratively:

$$\begin{aligned} m: \quad & \text{useful}_m = \max(useful_l | \tau_l \in S_{i,j}^k) \\ S_{i,j}^{k+1} &= S_{i,j}^k - \{\tau_m\} \\ N_{i,j}^{k+1} &= N_{i,j}^k - \left\lceil \frac{R_m}{P_j} \right\rceil \cdot \left\lceil \frac{R_i^n}{P_m} \right\rceil \\ \Delta_{i,j}^{k+1}(R_i^n) &= \Delta_{i,j}^k(R_i^n) + \min\left(N_{i,j}^k, \left\lceil \frac{R_m}{P_j} \right\rceil \cdot \left\lceil \frac{R_i^n}{P_m} \right\rceil \right) \cdot useful_m \cdot t_{miss} \end{aligned}$$

First, the total number of preemption delays is bounded by the number of activations of τ_j during R_i^n. Second, the number of useful cache blocks of all lower priority tasks is computed. The number of useful cache blocks is approximated by a constant percentage of all used cache blocks. And finally, the maximum number of useful cache blocks among the lower priority tasks is assumed to be replaced. The drawback of this approach is that for larger caches where the preempting and the preempted tasks share only small parts of the cache, this modeling can lead to a pessimistic overestimation.

Unfortunately this technique cannot be applied when the CRPD is calculated as the intersection of used cache blocks of the preempted and useful cache blocks of the preempting task. This can be explained by a simple schedule, as shown in figure 2.3. Consider the first preemption by τ_3 on τ_2 and τ_3 and the preemption by τ_2 on task τ_1. Assuming that the set of useful cache blocks USE^{τ_2} of τ_2 are greater than USE^{τ_2} of task τ_2, we get for the *total delay* by τ_3 and τ_2: $USE^{\tau_1} + USE^{\tau_1}$. If we denote, for the moment, the intersection of used cache blocks of τ_j and useful cache blocks of τ_i by $\delta_{j,i}$ we would get according to Petters: $\delta_{3,2} + \delta_{2,3}$ assuming $\delta_{3,2} > \delta_{3,1}$. This is an underestimation because the indirect preemption of τ_3 on τ_1 is not considered. The correct delay would be $\delta_{3,2} + \delta_{3,1} + \delta_{2,3}$. This insight motivates our proposed cache-aware schedulability analysis in chapter 3.

2.4.3 Intrinsic and extrinsic cache analysis

The reviewed approaches analyze the extrinsic cache behavior, by calculating the additional CRPD. On the other hand, intrinsic effects are those cache effects that take place during a single task execution. Intrinsic cache effects have been studied intensively, e.g. in [1] [8] [71] [127] [141]. All these approaches assume an empty cache at task activation or an equivalent worst case initial cache state (e.g. for data cache).

To overcome the pessimistic empty cache assumption, [59] propose a static pre-runtime scheduling technique considering available cache blocks from a previous task execution. However, the important class of preemptive scheduling is not supported.

In [81] a technique is presented to combine intrinsic and extrinsic cache analysis. One drawback is that only the preempted task is considered for extrinsic cache analysis and an empty cache at task activation is assumed for intrinsic cache analysis.

A scheduling technique based on simulated annealing and program path analysis integrating intrinsic and extrinsic cache effects is presented by [77]. The preemption delay is computed by considering all used cache blocks of the preempting and preempted task. A drawback of this approach is that all program paths in the CFG have to be considered, which leads to exponential time complexity.

2.4.4 Discussion

In summary, three main sub-problems have to be addressed for cache-aware schedulability analysis:

1) accurate analysis of single preemption delay

2) time-efficient integration of CRPD analysis into schedulability analysis

3) integration of intrinsic and extrinsic cache analysis.

Concerning the first problem, the approaches of [67] and [89] provide the most accurate analysis by considering the preempted as well as the preempting task. Concerning the second problem, the time complexity of the extended schedulability analysis by [67] grows exponentially with the number of tasks and the technique described by [92] is not extensible to a preemption delay analysis considering both the preempted and preempting task. The path analysis by [123] grows with the number of program paths and is, therefore, limited in the size of applications. The approaches by [92] [67] [123] assume a fixed preemption delay for each preemption. Finally, for the third problem, the approaches that integrate intrinsic and extrinsic cache effects suffer from

unprecise single preemption delay analysis and the empty cache assumption [81] and from a high time complexity [77].

The proposed cache-aware schedulability analysis in chapter 3 makes the following contributions:

- First, the analysis identifies those preemptions that do not cause a cache interference. If a task is suspended and is preempted several times, only the first preemption can replace cache blocks.
- Second, preemption delay analysis is integrated into the schedulability analysis in low time complexity.
- Third, the preemption delay considers both: the preempting and preempted task.

These contributions are presented in section 3.2. In the previous analysis techniques [67] [92] [19], each preemption is assumed at the worst case preemption point. In section 3.3 a methodology is described to bound the worst case preemption delay for multiple preemptions by searching for the worst case preemption scenario. A novel inter- and intra-cache analysis is presented in section 3.4 which considers multiple task activations.

All previous analysis approaches target instruction caches. The main difficulty in data cache analysis is the input data dependency of memory accesses. Related work about data cache analysis is reviewed in section 5.1.1.

The preemption delay analyses by Lee [66] and Mitra [89] use data flow techniques which differ in their time-complexity and analysis precision. In the next section we describe both approaches in detail.

2.5 Data flow analysis techniques

In this section we review cache analysis approaches that use data flow analysis techniques. Data flow analysis is a technique used in optimizing compilers to statically analyze source code properties for exploration of common sub-expressions or dead-code elimination [3]. It has been applied to CRPD analysis by [67] and [89] based on the CFG of the task and to single task analysis by [141].

First, we review the set-based preemption delay analysis by Lee et al.

[66], [67] in section 2.5.1. Second, we present in section 2.5.2, the approach by Mitra et al. [89] which is based on cache states. Both approaches lay the foundation for the new schedulability analysis in chapter 3 and the scalable precision cache analysis in chapter 4. Finally, we review the single task cache analysis by [141] in section 2.5.3 which will be used in section 3.4 to consider multiple task activations.

2.5.1 Set-based preemption delay analysis

The technique for estimating the number of useful and used cache blocks is based on data flow analysis [3] over the task's program expressed in a flow graph. In a flow graph, each node represents a basic block. A basic block is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end [3]. The edges of the graph represent potential control flow between basic blocks.

We explain the data flow analysis on the example in figure 2.4. It shows a loop-statement with two `if-then-else` statements. A node B_i lists the memory blocks that correspond to the assembly instructions of basic block B_i. For example, the memory blocks m_1, m_2 and m_3 are loaded to the cache during execution of basic block B_2. We assume a direct mapped cache with 4 cache sets. The mapping of memory blocks to cache sets is given on the right side in figure 2.4.

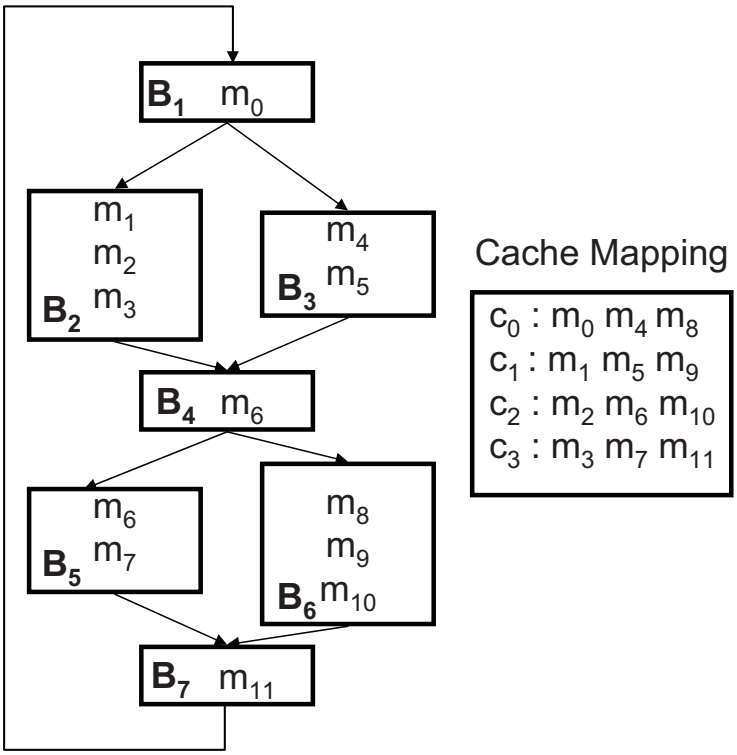

Figure 2.4. Control flow graph with memory blocks and cache mapping for direct mapped instruction cache with 4 blocks.

As described in the last section, cache lines might have to be reloaded after a context switch. The preemption delay depends on the used cache blocks of the preempt-

ing as well as the useful cache blocks of the preempted task. This approach has been shown in figure 2.2.

The definition of *useful* cache blocks is motivated by an example. Consider basic block B_4 in figure 2.4. At basic block B_4 the memory block m_0 (from B_1) and m_3 (from B_3) and other memory blocks could be available in cache set c_0 and c_3, respectively, because they are loaded via some incoming path of B_4. Second, on a possible outgoing paths of B_4, memory block m_0 may be alive in the next iteration of the loop at B_1 without being replaced. This is the case when the task continues on path B_5, B_7, B_1 where no cache allocations to cache set c_o occur. On the other hand, memory block m_3 cannot be used again because via any outgoing path because it is replaced by m_{11} at basic block B_7. Therefore, m_0 is *useful* at B_4 but m_3 is not.

For a more formal description, *reaching memory blocks (RMB)* and *live memory blocks (LMB)* are defined for each cache set. These terms are similar to reaching definitions and live variables as used in traditional data flow analysis [3]. The set of reaching memory blocks of cache set c at point B, denoted by $RMB^c[B]$, contains all possible contents of cache set c at point B, where a possible contents corresponds to a memory block that may reside in the cache set at the point. For a memory block to reside in cache set c, first, it has to be mapped to c. Second, it has to be the last reference to the cache set in some execution path reaching B.

The set of live memory blocks of cache set c at point B, denoted by $LMB^c[B]$, is defined similarly and is the set of memory blocks that may be the first reference to cache set c after B. With these definitions, a cache block is useful, denoted as $USE^c_{lee}[B]$ at point B if it is an element of $RMB^c[B]$ and $LMB^c[B]$.

To formulate the problem of computing RMB as a data flow problem, a set of generated memory blocks, $gen^c[B]$, is defined as either empty or containing a single memory block. It is empty if basic block B does not have any reference to memory blocks mapped to cache set c. On the other hand, if the basic block B has at least one reference to a memory block mapped to c, $gen^c[B]$ contains as its unique element the memory block that is the last reference to the cache set c at the end of the basic block B.

As an example, consider basic block B_3 in figure 2.4. Assuming that the instruction cache is direct mapped and has four sets

$$gen^{c_0}[B_3] = \{m_4\}, \quad gen^{c_1}[B_3] = \{m_5\}$$

and the other *gen*-sets are empty. In the following description we abbreviate the *gen*-sets with a vector for all cache sets and, thus:

$$gen[B_3] = [\{4\},\{5\},\{\},\{\}]$$

All *gen*-sets for figure 2.4 are summarized in table 2.1.

B_i	$gen[B_i]$	B_i	$gen[B_i]$
B_1	$[\{0\},\{\},\{\},\{\}]$	B_5	$[\{\},\{\},\{6\},\{7\}]$
B_2	$[\{\},\{1\},\{2\},\{3\}]$	B_6	$[\{8\},\{9\},\{10\},\{\}]$
B_3	$[\{4\},\{5\},\{\},\{\}]$	B_7	$[\{\},\{\},\{\},\{11\}]$
B_4	$[\{\},\{\},\{6\},\{\}]$		

Table 2.1. Gen-sets for the flow graph in figure 2.4.

Calculation of reaching memory blocks (RMB)

With $gen^c[B]$ defined in this manner, the RMB of c just before the beginning of B and just after the end of B, which are denoted by $RMB^c_{in}[B]$ and $RMB^c_{out}[B]$, respectively, can be computed by the following two equations:

$$RMB^c_{in}[B] = \bigcup_{\text{p is a pred of B}} RMB_{out}[p] \tag{2.7}$$

$$RMB^c_{out}[B] = \begin{cases} gen^c[B] & if \quad gen^c[B] \neq \emptyset \\ RMB^c_{in}[B] & otherwise \end{cases} \tag{2.8}$$

The first equation states the memory blocks that reach the beginning of a basic block B are those those that reach a predecessor of B. The second equation states that $RMB^c_{out}[B]$ is equal to $gen^c[B]$ if $gen^c[B]$ is not empty and $RMB^c_{in}[B]$ otherwise. Equation 2.8 can be rewritten as

$$RMB^c_{out}[B] = gen^c[B] \cup (RMB^c_{in}[B] - kill^c[B]) \tag{2.9}$$

which is commonly used in traditional data flow analysis. The set $kill^c[B]$ is the set of memory blocks of cache set c which are replaced in basic block B. The $kill^c[B]$-set is calculated as follows. It is empty if $gen^c[B]$ is empty, or it is $M_c - gen^c[B]$ if $gen^c[B]$ is not empty. The term M_c denotes the set of all memory blocks mapped to cache set c of a program.

To show that equation 2.9 is equal to equation 2.8 we distinguish whether $gen^c[B] = \emptyset$ or not. In case $gen^c[B] = \emptyset$ then $RMB^c_{out}[B] = RMB^c_{in}[B]$. In the other case, where

$gen^c[B] \neq \emptyset$, equation 2.9 can be rewritten ($RMB^c_{in}[B] \subset M_c$, $gen^c[B] \subset M_c$)

$$\begin{aligned} RMB^c_{out}[B] &= gen^c[B] \cup (RMB^c_{in}[B] - (M_c - gen^c[B])) \\ &= gen^c[B] \cup (RMB^c_{in}[B] \cap gen^c[B]) = gen^c[B] \end{aligned}$$

These data flow equations can be solved using a well-known iterative approach [3]. It starts with

$$RMB^c_{in}[B] = \emptyset \quad \text{and} \quad RMB^c_{out}[B] = gen^c[B]$$

for all basic blocks B and cache sets c. Then, equations 2.7 and 2.8 are repeatedly applied until the values of $RMB^c_{in}[B]$ and $RMB^c_{out}[B]$ converge. The iterative process is described by the iterative algorithm in figure 2.5, assuming that the set $gen^c[B]$ has been computed for each basic block B and cache set c.

```
(1)   for each basic block B do
(2)       for each cache set c do begin
(3)           RMB^c_in[B] = ∅; RMB^c_out[B] := gen^c[B];
(4)       end
(5)   change := true;
(6)   while change do begin
(7)       change := false;
(8)       for each basic block B do
(9)           for each cache set c do begin
(10)              RMB^c_in[B] := ⋃ (P a pred of B) RMB^c_out[P];
(11)              oldout := RMB^c_out[B];
(12)              if (gen^c[B] ≠ ∅) then RMB^c_out[B] := gen^c[B];
(13)                                 else RMB^c_out[B] := RMB^c_in[B];
(14)              if RMB^c_out[B] ≠ oldout then change := true;
(15)          end
(16)  end
```

Figure 2.5. Iterative data flow algorithm for *RMB* calculation.

The algorithm performs for each iteration of the while loop $O(|E||C|)$ union operations, where E is the set of (directed) edges of the flow graph and C is the set of cache blocks used in the program. The maximum number of iterations of the while loop is equal to the length of the longest acyclic path of the flow graph, or $O(|V|)$, where

V is the set of nodes (basic blocks) of the flow graph [3]. this gives the total time complexity of $O(|V||E||C|)$. However, it is shown in [42] that, when the flow graph is reducible, and the basic blocks are processed in the reverse postorder of a depth-first spanning tree of the flow graph, the maximum number of iterations of the while loop is bounded by $d+2$, where d is the number of back edges in the depth-first spanning tree, which is usually introduced because of loops and recursions.

The data flow algorithm in figure 2.5 is demonstrated on the example flow graph of figure 2.4. Initially all $RMB^c_{in}[B]$ are empty and $RMB^c_{out}[B]$ are initialized with the $gen^c[B]$ sets. The results are summarized in table 2.2. The RMB sets of the third iteration are the same as in the second iteration, therefore, they are omitted. The results are graphically represented in figure 2.6 for the second iteration.

B_i	$RMB_{out}(B_i)$ 1st iteration	$RMB_{out}(B_i)$ 2nd iteration
B_1	$[\{0\},\emptyset,\emptyset,\emptyset]$	$[\{0\},\{1,5,9\},\{6,10\},\{11\}]$
B_2	$[\{0\},\{1\},\{2\},\{3\}]$	$[\{0\},\{1\},\{2\},\{3\}]$
B_3	$[\{4\},\{5\},\emptyset,\emptyset]$	$[\{4\},\{5\},\{6,10\},\{11\}]$
B_4	$[\{0,4\},\{1,5\},\{6\},\{3\}]$	$[\{\mathbf{0},\mathbf{4}\},\{\mathbf{1},\mathbf{5}\},\{\mathbf{6}\},\{\mathbf{3},\mathbf{11}\}]$
B_5	$[\{0,4\},\{1,5\},\{6\},\{7\}]$	$[\{0,4\},\{1,5\},\{6\},\{7\}]$
B_6	$[\{8\},\{9\},\{10\},\{3\}]$	$[\{8\},\{9\},\{10\},\{3,11\}]$
B_7	$[\{0,4,8\},\{1,5,9\},\{6,10\},\{11\}]$	$[\{0,4,8\},\{1,5,9\},\{6,10\},\{11\}]$

Table 2.2. Reaching cache blocks $RMB_{out}[B_i]$ for set-based approach by Lee.

We demonstrate one execution of the while loop for basic block B_4 in the second iteration. In this case the RMB_{out} sets of B_2 and B_3 are merged and cache set c_2 is replaced with $gen^2[B_4]$:

$$\begin{aligned} RMB_{in}[B_4] &= [\{0\},\{1\},\{2\},\{3\}] \cup [\{4\},\{5\},\{6,10\},\{11\}] \\ &= [\{0,4\},\{1,5\},\{2,6,10\},\{3,11\}] \\ RMB_{out}[B_4] &= [\{0,4\},\{1,5\},\{\mathbf{6}\},\{3,11\}] \end{aligned}$$

Note, that the $RMB_{out}[B_4]$ shows that m_0 and m_3 are reached at B_4 as claimed in the motivating example.

Calculation of live memory blocks (LMB)

The computation of live memory blocks $LMB^c[B]$ is similarly to the calculation of $RMB^c[B]$. The difference is that the *LMB* are solved by a *backward* data flow analysis in that the *in* sets ($LMB^c_{in}[B]$), are computed from the *out* sets ($LMB^c_{out}[B]$), whereas the RMB problem is a *forward* data flow problem, in that the out-sets are computed

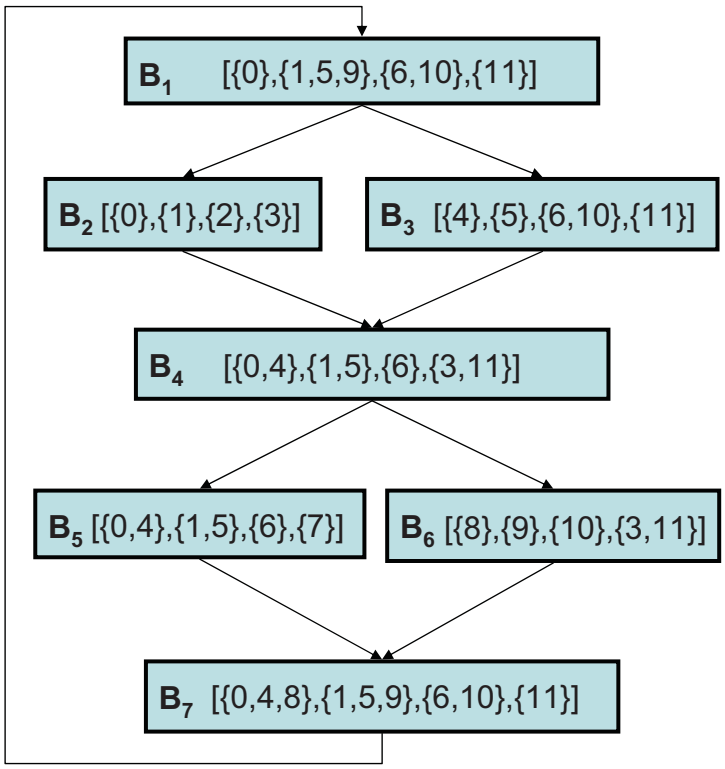

Figure 2.6. *RMB* calculation by Lee.

from the in-sets. In the LMB problem, the $gen^c[B]$-set is either a set with only one element corresponding o the memory block whose reference is the *first* reference to cache set c in basic block B, or empty if none of the references of B are mapped to c. Using $gen^c[B]$-sets defined in this way, the following two equations relate the $LMB_{in}[B]$ and $LMB_{out}[B]$:

$$LMB^c_{out}[B] = \bigcup_{\text{s is a succ of B}} LMB^c_{in}[s] \tag{2.10}$$

$$LMB^c_{in}[B] = \begin{cases} gen^c[B] & if \quad gen^c[B] \neq \emptyset \\ LMB^c_{out}[B] & otherwise \end{cases} \tag{2.11}$$

An iterative algorithm similar to the one for computing *RMB* can be used to solve the backward data flow problem. The difference is that the algorithm starts with $LMB^c_{out}[B] = \emptyset$ and $LMB^c_{in}[B] = gen^c[B]$ for all B and c and uses the above two equations. This algorithm has the same time complexity as the one for computing *RMB*.

The backward flow algorithm for equations 2.10 and 2.11 is demonstrated on the example flow graph of figure 2.4. The values of the sets $LMB_{in}[B_i]$ and $LMB_{out}[B_i]$ are shown in table 2.3. The $LMB_{out}[B_i]$ are shown because they are necessary for the

calculation of the useful cache blocks. Note, that m_0 is a live memory block in B_4 but m_3 is not.

B_i	$LMB_{in}[B_i]$ 1st iter.	$LMB_{in}[B_i]$ 2nd iter.	$LMB_{out}[B_i]$ 2nd iter
B_1	$[\{0\},\{1,5\},\{2,6\},\{3,7,11\}]$	$[\{0\},\{1,5\},\{2,6\},\{3,7,11\}]$	$[\{0,4,8\},\{1,5\},\{2,6\},\{3,7,11\}]$
B_2	$[\{8\},\{1\},\{2\},\{3\}]$	$[\{0,4,8\},\{1\},\{2\},\{3\}]$	$[\{0,8\},\{1,5,9\},\{6\},\{7,11\}]$
B_3	$[\{4\},\{5\},\{6\},\{7,11\}]$	$[\{4\},\{5\},\{6\},\{7,11\}]$	$[\{0,8\},\{1,5,9\},\{6\},\{7,11\}]$
B_4	$[\{8\},\{9\},\{6\},\{7,11\}]$	$[\{0,8\},\{1,5,9\},\{6\},\{7,11\}]$	$[\{0,8\},\{1,5,9\},\{6,10\},\{7,11\}]$
B_5	$[\emptyset,\emptyset,\{6\},\{7\}$	$[\{0\},\{1,5\},\{6\},\{7\}$	$[\{0\},\{1,5\},\{2,6\},\{11\}]$
B_6	$[\{8\},\{9\},\{10\},\{11\}]$	$[\{8\},\{9\},\{10\},\{11\}]$	$[\{0\},\{1,5\},\{2,6\},\{11\}]$
B_7	$[\emptyset,\emptyset,\emptyset,\{11\}]$	$[\{0\},\{1,5\},\{2,6\},\{11\}]$	$[\{0\},\{1,5\},\{2,6\},\{3,7,11\}]$

Table 2.3. Live cache blocks $LMB_{in}[B_i]$ and $LMB_{out}[B_i]$ for approach by Lee.

Calculation of useful cache blocks

It is shown in [66] that the number of useful cache blocks is the same at all execution points in a basic block. Once the *RMB* and *LMB* are computed, the useful cache blocks $USE^{\tau}_{lee}[p]$ at each execution point p can be determined by taking the intersection of $RMB[B]$ and $LMB[B]$ of task τ.

$$USE^{\tau}_{lee}[B] = RMB_{out}[B] \cap LMB_{out}[B] \tag{2.12}$$

Throughout this thesis we will use the notation $USE^{\tau}[p]$ or $USE[p]$ when we are concerned about the useful cache blocks in general (without specifying which data flow analysis technique is used) or when it is clear (or not important) which task is preempted.

The total number of useful cache blocks is given by $|USE^{\tau}_{lee}[B]| + 1$. The one additional block is required because if a task is preempted within a memory block, the cache block mapped by the memory block is definitely useful at the preemption point.

The useful cache blocks for the example in figure 2.4 are shown in table 2.4. The values for $RMB_{out}[B]$ and $LMB_{out}[B]$ are given in table 2.2 and table 2.3.

A preemption at basic block B_1, B_4 or B_7 would require 4 additional cache blocks assuming the preempting task removes all cache blocks. If the preemption would occur at B_6 the CRPD would be bounded by one cache block. Preferred preemption points have been analyzed in [109] to minimize the total preemption delay. We elaborately described the example because we will revise this analysis in section 4.1.

B_i	$USE_{lee}(B_i)$	$\lvert USE_{lee}(B_i)\rvert$
B_1	$[\{0\},\{1,5\},\{6\},\{11\}]$	4
B_2	$[\{0\},\{1\},\emptyset,\emptyset$	2
B_3	$[\emptyset,\{5\},\{6\},\{11\}]$	3
B_4	$[\{0\},\{1,5\},\{6\},\{11\}]$	4
B_5	$[\{0\},\{1,5\},\{6\},\emptyset]$	3
B_6	$[\emptyset,\emptyset,\emptyset,\{11\}$	1
B_7	$[\{0\},\{1,5\},\{6\},\{11\}]$	4

Table 2.4. Useful cache blocks $USE^{\tau}_{lee}[B_i]$ for approach by Lee.

Calculation of used cache blocks

If a preemption occurs, only those useful cache blocks are reloaded that are used by the preempting task. The reaching cache block algorithm is applied to the preempting task τ' and the $RMB^{\tau'}[end]$ of the last basic block of the program, denoted by *end*, represents an upper bound of all used cache blocks.

Calculation of preemption delay

Finally, the set of useful cache blocks $USE_{\tau}[p]$ at each execution point p is intersected with the set of used cache blocks of the preempting task τ'. This is schematically shown in figure 2.2:

$$crpd_{lee}(\tau,\tau') = max\{ \quad |USE^{\tau}_{lee}[B] \cap RMB^{\tau'}[end]| \quad | \quad B \in BB(\tau)\} \tag{2.13}$$

in which $BB(\tau)$ is the set of all basic blocks of task τ. The total CRPD is bounded by multiplying the number of cache blocks $crpd_{lee}(\tau,\tau')$ with the (constant) cache miss penalty.

2.5.2 State-based preemption delay analysis

The motivation of the work by [89] is to consider execution path information in the calculation of the CRPD. Supposing that a cache has two cache sets and the possible states of the cache when a task τ is preempted are as $\{[m_0,m_1],[m_2,m_3]\}$, memory blocks m_0,m_2 map to cache set c_0 and m_1,m_3 map to cache set c_1 assuming a direct mapped cache. Lee's approach would result a possible cache content of cache set $c_0 = \{m_0,m_2\}$ and $c_1 = \{m_1,m_3\}$. This representation corresponds to the following cache states: $\{[m_0,m_1]$, $[m_0,m_3]$, $[m_2,m_1]$, $[m_2,m_3]$ which can lead to an over-approximation

because only two of them are possible. We demonstrate the difference in CRPD calculation later in an extended example.

Definitions

A cache state is defined to denote the contents of all cache blocks. Mitra et al. provide an analysis for direct mapped instruction caches. For a direct mapped cache with S blocks, a cache state is a vector of S elements $c[0,\cdots,S-1]$ in which $c[i]=m$ if cache block i holds memory block m. Otherwise , if the ith cache block does not hold any memory block we denote this as $c[i]=\perp$. Thus, a cache state is a vector of length S where each element of the vector belongs to $M\cup\{\perp\}$. M is the set of all memory blocks. In the extension of notation, we assume that any operation $\odot$ over $M\cup\{\perp\}$ can be applied to cache states by applying the operation pointwise to its elements. For example, if $\odot$ denotes a binary operation over $M\cup\{\perp\}$ and c,c' are cache states then $c''=c\odot c'$ where $c''[i]=c[i]\odot c'[i]$.

Similar to Lee, a *reaching cache state* is defined at a basic block B of a program denoted as $RCS[B]$, is the set of possible cache states when B is reached via any incoming program path. Given a program, the *Live cache states* at a basic block B, denoted as $LCS[B]$, is defined as the possible first memory references to cache blocks via any outgoing program path from B.

Calculation of reaching cache states (RCS)

A preemption is considered at the end of each basic block as a possible preemption point, as in Lee et al. [66]. The calculation of *RCS* and *LCS* involves some modifications of the work of [66] [67], because the notion of cache states is more elaborate. To compute $RCS[B]$, the quantities $RCS_{in}[B]$ and $RCS_{out}[B]$ are computed as a least fixed point. Once the fixed point is reached, $RCS[B]=RCS_{out}[B]$. Initially $RCS_{in}[B]=\emptyset$ and $RCS_{out}[B]=gen[B]$. For each basic block B, the gen-set is defined as $gen[B]=[m_0,\cdots,m_{S-1}]$ where $m_i=m$ if m is the *last* memory block in B that maps to cache block i and $\perp$ if no memory block in B maps to cache block i. Thus, $gen[B]$ represents all the memory blocks loaded to the cache by basic block B. The iterative equations are as follows:

$$RCS_{in}[B] = \bigcup_{p\in pred(B)} RCS_{out}[p] \tag{2.14}$$

$$RCS_{out}[B] = \{r\odot gen[B] | r\in RCS_{in}[B]\} \tag{2.15}$$

$$c\odot c' = \begin{cases} c' & if \quad c'\neq\perp \\ c & otherwise \end{cases} \tag{2.16}$$

The $\odot$ operation is defined in equation 2.16 over memory blocks m and m'. It is extended to cache states by applying the operation to each cache set. The data flow algorithm of figure 2.5 can be used to calculate the fixed point.

Calculation of live cache states (LCS)

The calculation of $LCS[B]$ is similar. The same fixed point algorithms are used to compute $LCS_{out}[B]$ and $LCS_{in}[B]$. Once the fixed point is reached, $LCS[B] = LCS_{out}[B]$. Initially, $LCS_{out}[B] = \emptyset$ and $LCS_{in}[B] = gen[B]$. For each basic block B, the gen-set is defined as $gen[B] = [m_0, \cdots, m_{n-1}]$ where $m_i = m$ if m is the *first* memory block in B that maps to cache block i and $\perp$ if no memory block in B maps to cache block i. The iterative equations are as follows:

$$LCS_{out}[B] = \bigcup_{s \in succ(B)} LCS_{in}[s] \tag{2.17}$$

$$LCS_{in}[B] = \{l \odot gen[B] | l \in LCS_{out}[B]\} \tag{2.18}$$

The operation $\odot$ is defined as in the computation of RCS_B.

Calculation of useful cache blocks

If a task resumes after a preemption, those cache blocks are reloaded which are useful. The useful cache blocks are computed, as in Lee et al., by the intersection of LCS and RCS. To formally capture this notion for the representation of cache states, a cache utility vector $CUV[B]$ defined for basic block B.

$$CUV[B] = \{l \approx r | l \in LCS[B], r \in RCS[B]\} \tag{2.19}$$

The operator $\approx$ is the equality predicate over memory blocks, e.g. can be applied to the elements of $LCS[B]$ and $RCS[B]$, that is $m \approx m' = 1$ iff $m = m'$. It is extended to cache states by applying the operator $\approx$ pointwise to its elements. This results in a boolean vector of length n, where n denotes the number of cache blocks in a cache. The set CUV contains $|LCS[B]| \cdot |RCS[B]|$ boolean vectors, since each cache state in $LCS[B]$ is compared to each cache state in $RCS[B]$. Each boolean vector in $CUV[B]$ represents the useful cache blocks at basic block B:

$$USE_{mitra}[B] = max\{cnt\_ones(c) \quad | \quad c \in CUV[B]\} \tag{2.20}$$

where $cnt\_ones(d)$ returns the number of 1s in bit vector c. One cache block is added to account for the currently used cache block as in the approach by Lee et al..

Calculation of used cache blocks

The useful cache blocks determine how many cache blocks can potentially be reloaded after the preempted task resumes. However, only those blocks are replaced which are actually used by the preempting task. A final usage vector $FUV_{\tau'}$ is defined to represent the cache states with all used cache blocks. We can use the same fixed point algorithms described above to calculate the reaching cache blocks RCS_{end} at the last basic block *end* of the preempting task.

$$FUV_{\tau'} = \{used(r) | r \in RCS_{end}\} \tag{2.21}$$

The function $used()$ is defined over all cache states as $used(r) = r'$ where r' is the following bit-vector:

$$r'[i] = \begin{cases} 1 & if \quad r[i] \neq \perp \\ 0 & otherwise \end{cases} \tag{2.22}$$

Calculation of preemption delay

Given the useful cache blocks $CUV[B]$ at each basic block of the preempted task τ and the used cache blocks $FUV_{\tau'}$ of the preempting task τ' , the preemption delay is computed by the intersection at all possible preemption points:

$$delay(\tau, \tau') = \bigcup_{B \in BB(\tau)} \{c \wedge f | c \in CUV[B], f \in FUV_{\tau'}\} \tag{2.23}$$

The $\wedge$ operator is the AND operator and is applied to boolean vectors and $BB(\tau)$ is the set of all basic blocks of τ. Thus, if a cache block contains a useful memory block of τ and it is used by task τ' then it can cause an additional cache miss:

$$crpd_{mitra}(\tau, \tau') = max\{cnt\_ones(d) | d \in delay(\tau, \tau')\} \tag{2.24}$$

where $cnt\_ones(d)$ returns the number of 1s in bit vector d. The number of cache blocks $crpd_{mitra}(\tau, \tau')$ is multiplied by the (constant) cache miss penalty. An example for this calculation of these bit-vectors can be found in [89]. In this section we focus on the calculation of the data flow properties *RCS* and *LCS*.

Example

We explain the reaching cache block, live cache block and useful cache block calculation for the CFG in figure 2.4. The reaching cache blocks $RCS_{out}[B]$ are summarized in table 2.5. The fixed point is also reached in the third iteration. These results correspond with the second column in table 2.5 and are represented graphically in figure 2.7.

B_i	$RCS_{out}(B_i)$ 1st iteration	$RCS_{out}(B_i)$ 2nd iteration
B_1	$[0,\perp,\perp,\perp]$	$[0,1,6,11]$,$[0,9,10,11]$,$[0,5,6,11]$
B_2	$[0,1,2,3]$	$[0,1,2,3]$
B_3	$[4,5,\perp,\perp]$	$[4,5,6,11]$,$[4,5,10,11]$
B_4	$[0,1,6,3]$,$[4,5,6,\perp]$	$[\mathbf{0},\mathbf{1},\mathbf{6},\mathbf{3}]$, $[\mathbf{4},\mathbf{5},\mathbf{6},\mathbf{11}]$
B_5	$[0,1,6,7]$,$[4,5,6,7]$	$[0,1,6,7]$,$[4,5,6,7]$
B_6	$[8,9,10,3]$	$[8,9,10,3]$,$[8,9,10,11]$
B_7	$[0,1,6,11]$,$[4,5,6,11]$,$[8,9,10,11]$	$[0,1,6,11]$,$[4,5,6,11]$,$[8,9,10,11]$

Table 2.5. Reaching cache states $RCS_{out}[B_i]$ for state-based approach by Mitra [89].

We explain the calculation of *RCS* of equations 2.14- 2.16 for the second iteration at basic block B_4:

$$\begin{aligned} RCS_{in}[B_4] &= \{[0,1,2,3],[4,5,6,11],[4,5,10,11]\} \\ RMB_{out}[B_4] &= \{[0,1,\mathbf{6},3],[4,5,\mathbf{6},11]\} \end{aligned}$$

Memory block m_6 is mapped to cache set c_2. Note, that the number of cache states reduces because the duplicated cache states are removed ($[4,5,6,11]$).

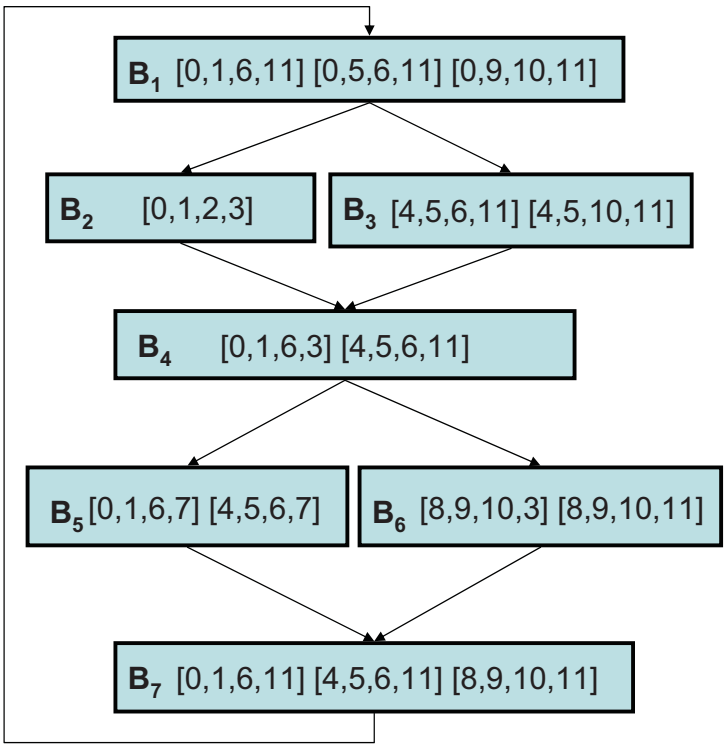

Figure 2.7. RCS calculation by Mitra.

The results for the backward data flow analysis for the live cache states $LCS_{in}[B_i]$ are summarized in table 2.6. The results for the live cache states $LCS_{out}[B_i]$ and the

B_i	$LCS_{in}[B_i]$ 1st iteration	$LCS_{in}[B_i]$ 2nd iteration
B_1	$[0,1,2,3],[0,5,6,7],[0,5,6,11]$	$[0,1,2,3],[0,5,6,7],[0,5,6,11]$
B_2	$[\bot,1,2,3],[8,1,2,3]$	$[0,1,2,3],[8,1,2,3]$
B_3	$[4,5,6,7],[4,5,6,11]$	$[4,5,6,7],[4,5,6,11]$
B_4	$[\bot,\bot,6,7],[8,9,6,11]$	$[0,1,6,7],[0,5,6,7],[8,9,6,11]$
B_5	$[\bot,\bot,6,7]$	$[0,1,6,7],[0,5,6,7]$
B_6	$[8,9,10,11]$	$[8,9,10,11]$
B_7	$[\bot,\bot,\bot,11]$	$[0,1,2,11],[0,5,6,11]$

Table 2.6. Live cache states $LCS_{in}[B_i]$ for state-based approach by Mitra et al. [89].

cache utility vector are shown in table 2.7.

B_i	$LCS_{out}[B_i]$	USE_{mitra}	USE_{lee}
B_1	$[4,5,6,7],[4,5,6,11],[0,1,2,3],[8,1,2,3]$	**3**	4
B_2	$[0,1,6,7],[0,5,6,7],[8,9,6,11]$	2	2
B_3	$[0,1,6,7],[0,5,6,7],[8,9,6,11]$	**2**	3
B_4	$[8,9,10,11]$, $[0,1,6,7],[0,5,6,7]$	**3**	4
B_5	$[0,1,2,11],[0,5,6,11]$	**2**	3
B_6	$[0,1,2,11],[0,5,6,11]$	1	1
B_7	$[0,1,2,3],[0,5,6,7],[0,5,6,11]$	**3**	4

Table 2.7. Live cache states $LCS_{out}[B_i]$, number of useful cache blocks USE_{mitra} of state-based approach by Mitra and USE_{lee} of set-based approach by Lee.

A comparison of useful cache blocks by Mitra's approach USE_{mitra} with the result by Lee's approach in table 2.4 shows that in most cases Mitra's approach yields a more accurate bound than Lee's approach. It is crucial to understand this difference: While in Lee's approach all memory blocks that map to the same cache set are merged, the state-based approach uses a separate cache state to distinguish this case. This obviously leads to a higher time- and space-complexity. In chapter 4 we will develop a scalable precision cache analysis where the number of cache states at each node is bounded and thus the time- and space-complexity as well. As a key result, we demonstrate in section 4.3.5 that a high analysis precision can be accomplished with already a small number of cache states.

2.5.3 Intrinsic cache analysis

Intrinsic cache analysis consideres the timing behavior of the instruction or data cache during a single execution of a task. It is assumed that the cache is in its worst case hardware state regarding timing behavior when program execution starts. In the last decade, there have been many approaches published to determine an upper bound of the execution time for processors with caches, e.g. [1] [8] [10] [34] [70] [36] [136] [127] [87], [107].

In previous work [139] [141] [140] [142], a static timing and cache analysis framework has been developed at our institute by Ye, Wolf, Ernst and Staschulat, which will be overviewed now. It uses a similar data flow algorithms as in preemption delay analysis for *reaching cache blocks* RMB, as presented in section 2.5.1.

A cache definition is a content modification of a cache set. The $gen_s[PrS]$-sets are the sets containing the last definitions for each cache set when executing the program segment (PrS). A program segment is a more general notation for basic blocks. The $gen_s[PrS]$ is computed by local cache simulation.

The $kill_s[PrS]$-set are the sets containing the destroyed definitions for the cache sets of the disjoint program segment. A definition from a disjoint PrS is destroyed when the same cache set is defined by the current PrS and the line block from the disjoint PrS is removed from the set as it was selected by the replacement strategy. The $kill_s[PrS]$-sets can be computed by local cache simulation. The $in_s[PrS]$-sets are the sets containing the cache set definitions of reaching the program segment PrS. The $out_s[PrS]$-sets are the sets containing the cache set definitions leaving the PrS. The cache prediction approach consists of three steps.

In a first step, cache behavior is simulated for every PrS starting with a first miss scenario $in_s[PrS] =$. Local cache hits and misses for the address sequence of the PrS are found by local simulation and the $gen_s[PrS]$-sets and $kill_s[PrS]$-sets are computed by this address sequence.

In a second step, local data flow analysis defines the $out_s[PrS]$ from $gen_s[PrS]$ and $kill_s[PrS]$ and $in_s[PrS]$ as follows:

$$in_s[PrS] = \bigcap_{\text{p is a pred of PrS}} out_s[p] \tag{2.25}$$

$$out_s[PrS] = gen_s[PrS] \cup (in_s[PrS] - kill_s[PrS] \tag{2.26}$$

For the worst case analysis, only those definitions occurring on all previous PrS are propagated and, therefore, the intersection operator $\cap$ is used. These equations are solved by forward data flow algorithms [3]. The basic algorithm has been shown in figure 2.5 in section 2.5.1 to compute the set of reaching cache blocks. For this

problem, the corresponding lines are replaced with equations 2.25 and 2.26. The algorithm terminates when the contents of the out-sets converges. When data flow analysis is finished, the $in_s[PrS]$-sets contain a subset of the propagated definitions for worst case analysis.

In a third step, worst case analysis finds the cache definitions in the $in_s[PrS]$ that PrS-simulation has classified as misses for the first reference due to the conservative cache state, namely empty $in_s[PrS]$ at the beginning. These cache definitions are classified as cache hit.

The execution time c_i for the cache behavior for each program segment i is given by:

$$c_i = c_i^{hit} \cdot t_{hit} + c_i^{misses} \cdot t_{miss} \tag{2.27}$$

in which t_{hit} denotes the cache-hit and t_{miss} the cache-miss access time. c_i^{hit} and c_i^{miss} denote the number of cache hits and misses for the worst case that have been calculated by the above data flow analysis. Then, worst case path of the program is found by implicit path enumeration:

$$\max \sum_{i}^{n} c_i x_i \quad \sum_{s \in successor(PrS)} e(PrS_i, s) = x_i = \sum_{p \in predecessor(PrS)} e(p, PrS_i) \tag{2.28}$$

where x_i is the execution count of program segment PrS_i and n the total number of program segments. The (structural) constrains are given by the fact that the incoming flow and the outgoing flow have to be equal to the execution count x_i of each program segment PrS. These equations can be solved by integer linear programming [13].

Chapter 3

CACHE-AWARE RESPONSE TIME ANALYSIS

3.1 Background and Motivation

Accurate timing analysis is key to efficient embedded system synthesis and integration. Caches are needed to increase the processor performance but they are hard to predict because of their complex behavior especially for preemptive scheduling. Current approaches use simplified assumptions to bound the cache related preemption delay or propose a scheduling analysis which scales exponentially with the number of tasks. In this chapter we make three contributions to cache-aware response time analysis:

- First, we propose a novel schedulability analysis for fixed priority preemptive scheduling to consider timing effects for associative instruction caches at a context switch. The preemption delays are calculated by considering the preempted as well as the preempting task. The proposed schedulability analysis bounds *the number of preemptions more tightly by excluding infeasible cache interferences*. The analysis is conservative, e.g. determines a safe upper bound of the preemption delay, and has a low time complexity.

- As a refinement, the cache interference by multiple task preemptions is analyzed. While previous approaches calculate the worst-case preemption point and assume that each preemption takes place at this preemption point, we *consider the preemption history* in the calculation of the total cost for multiple task preemptions. The key observation is that the bound of the total preemption delay for multiple task preemptions can be smaller than the sum of the preemption delay bounds for each preemption.

- In an automotive case study we found out that control intensive applications designed with ASCET-SD and Matlab/Simulink models contain only sequential code without loops. Caches cannot increase the performance for such applications, because linear code significantly limits the spacial and temporal locality of memory accesses for which a cache is optimized. Existing timing analyses focus on a single task execution. However, embedded applications are activated very frequently if not regularly. Cache lines from a previous task activation might still be available in the cache and need not be loaded during a subsequent task execution. This effect of *multiple task execution* can result in a significantly reduced number of cache misses. In this thesis we estimate a conservative bound of the cache contents at the beginning of task activation and consider the effect in instruction cache timing behavior.

Scope and Limitations

We assume fixed priority preemptive task scheduling, constant time delay for cache miss and cache hit penalty and LRU replacement strategy for associative instruction caches.

Overview

This chapter is structured as follows. A novel response time analysis which considers indirect preemptions for cache interference is presented in section 3.2. Preemption delay analysis for multiple preemptions is proposed in section 3.3 to bound the total delay more tightly. Multiple task activations are analyzed to calculate the worst case initial cache contents in section 3.4. Results from experiments are shown in section 3.5, and finally we give concluding remarks in section 3.6.

3.2 Cache-aware schedulability analysis

Existing schedulability analyses either give a simplified account on preemption delay calculation, like Busquets et al. [19] (assuming only the preempting task) and Petters et al. [92] (assuming only the preempted task), or integrate preemption delay analysis (considering both: the preempted as well as the preempted) but with high time complexity, like Lee et al. [67]. In this section we present a cache-aware schedulability analysis that considers the preempted as well as the preempting task in the preemption delay calculation and which more tightly bounds the number of cache interferences in a low time complex algorithm.

This section is structured as follows. First, we propose a conservative approximation in section 3.2.1. Then, we present our novel schedulability analysis considering indirect task preemptions in section 3.2.2.

3.2.1 Simplified approach

Motivational example

An example schedule with four tasks under fixed priority and fixed periodic preemptive scheduling policy is shown in figure 3.1. Task τ_4 has the highest and task τ_1 the lowest priority. An arrow signals task activation and preemption delays are denoted by a solid block. For simplicity, the preemption delay is drawn as a single time block even though cache blocks are reloaded separately after task resumption and the time during suspension is not distinguished.

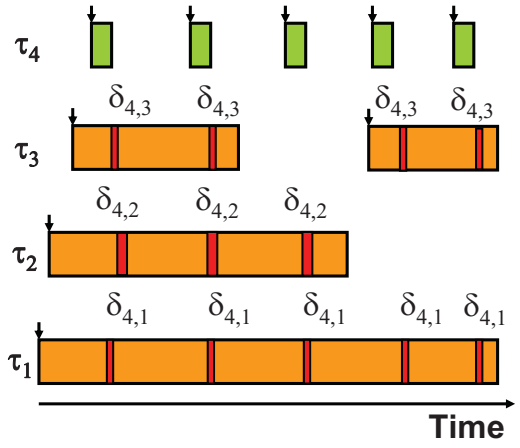

Figure 3.1. Example schedule simplified approach.

$E_{4,1} = 5$	$E_{3,1} = 2$
$E_{4,2} = 3$	$E_{3,2} = 1$
$E_{4,3} = 2$	$E_{2,1} = 1$
	$E_{1,1} = 1$

Figure 3.2. Task activations $E_{j,i}$.

A simplified conservative calculation is to consider *every task preemption as a cache interference*. We will use the notation of $\delta_{j,i}$ for an upper bound of the preemption delay when task τ_j preempts a task τ_i exactly one time. These delays can be calculated for examples by approaches of Busquets et al. [19], Lee et. al [67], and Mitra et al. [89]. In chapter 4, we will present a new method to compute these preemption delays.

For the given schedule in figure 3.1 we only have to count how many task preemptions occur:

$$\delta_{4,1} \cdot 5 + \delta_{4,2} \cdot 3 + \delta_{4,3} \cdot 2 \cdot 2 \tag{3.1}$$

Task τ_4 is activated 5 times during R_1, and τ_4 is activated 3 times during R_2, therefore $\delta_{4,1}$ is multiplied 5 times and $\delta_{4,2}$ 3 times. Task τ_3 is activated twice during R_1 and can be preempted twice by τ_4. We will formulate now the general case.

Considering all task preemptions

The total response time for fixed priority preemptive scheduling can be calculated using iterative fixed point algorithms. Recall the general response time equation 2.3 on page 27:

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{R_i^n}{P_j} \right\rceil \cdot C_j + crpd(j,i,R_i^n) \right) \tag{3.2}$$

The same notations as in equation 2.3 are being used. The worst case execution time C_i could be obtained by a WCET-analysis, e.g. by SymTA/P (SYMbolic Timing Analysis for Processes) [141] as described in appendix A. The term $crpd(j,i,R_i^n)$ is computed in equation 3.3.

$$crpd(j,i,R_i^n) = \Delta_{j,i}^s(R_i^n) = \sum_{k=i}^{j-1} \delta_{j,k} \cdot E_{j,k}^n \cdot E_{k,i}^n \cdot t_{miss} \tag{3.3}$$

We use the following new notations:

- $\delta_{j,i}$. The maximum time delay due to a cache interference, if task τ_j preempts τ_i once.
- $E_{j,i}^n$ denotes the maximum number of activations of task τ_j during the interval R_i^n. For fixed priority, periodic scheduling this term is equal to $E_{j,i}^n = \left\lceil \frac{R_i^n}{P_j} \right\rceil$. If the term is computed for the time window of some task τ_k, which iteration has finished, we define $E_{j,k}^n = \left\lceil \frac{R_k}{P_j} \right\rceil$. The main reason for the super-script n is to consider the correct value of R_i^n in the iterative response time equation.
- $\Delta_{j,i}^s(R_i^n)$ denotes an upper bound of the total cache-related preemption delay for cache interferences when task τ_j preempts all lower priority tasks $\{\tau_{j-1}, \cdots, \tau_i\}$ during the time window R_i^n.

Proof. Task τ_j can possibly preempt a lower priority task τ_k at most $E_{j,k}^n$ times during its response time R_k. During the response time R_i^n, τ_k is activated at most $E_{k,i}^n$ times and, therefore, the preemption delay $\delta_{j,k}$ is multiplied by $E_{j,k}^n \cdot E_{k,i}^n$. It represents the conservative assumption that the number of preemptions is equal to the number of cache interferences.

The proof the iterative response time equation in equation 3.3 is based on the critical instant, e.g. that the longest response time is given when all tasks are activated at the *same time*. This would not be conservative when cache interference is considered. The worst case occurs when the lowest priority task is activated just before the next highest priority task and so on. This has been shown in [99] and is sketched in figure 3.1: first task τ_1, then τ_2, τ_3 and finally τ_4 are activated. Nevertheless, the argument of the proof is correct when the preemption delay is *always* considered, especially when all tasks are activated at the same time.

Motivational example continued

We demonstrate this simplified approach for the example. For the schedule in figure 3.1, the equation 3.3 is applied for the preemption delay terms $\Delta^s_{4,1}(R_1)$, $\Delta^s_{3,1}(R_1)$ and $\Delta^s_{2,1}(R_1)$. Note, that the preemption delay in line 3.4 is equal to the earlier computed result in equation 3.1. The values of $E^n_{j,i}$ are summarized in figure 3.2. In this example we omit the superscript n in R^n_i and assume that the fixed point has been reached We use only R_1 as the response time of task τ_1.

$$\begin{aligned}
\Delta^s_{4,1}(R_1) &= \sum_{k=1}^{3} \delta_{4,k} \cdot E_{4,k} \cdot E_{k,1} = \delta_{4,1} \cdot E_{4,1} \cdot E_{1,1} + \delta_{4,2} \cdot E_{4,2} \cdot E_{2,1} \\
&\quad + \delta_{4,3} \cdot E_{4,3} \cdot E_{3,1} = \delta_{4,1} \cdot 5 + \delta_{4,2} \cdot 3 + \delta_{4,3} \cdot 4 \qquad (3.4) \\
\Delta^s_{3,1}(R_1) &= \sum_{k=1}^{2} \delta_{3,k} \cdot E_{3,k} \cdot E_{k,1} = \delta_{3,1} \cdot E_{3,1} \cdot E_{1,1} + \delta_{3,2} \cdot E_{3,2} \cdot E_{2,1} \\
&= \delta_{3,1} \cdot 2 + \delta_{3,2} \cdot 1 \\
\Delta^s_{2,1}(R_1) &= \delta_{2,1} \cdot E_{2,1} \cdot E_{1,1} = \delta_{2,1}
\end{aligned}$$

3.2.2 Indirect preemptions

Subsection 3.2.1 described the underlying response time analysis assuming that each preemption causes a cache interference. However, if a task is suspended, multiple preemptions can only replace the cache contents once. This reasoning assumes an upper bound for all used cache blocks of the preempting task.

Schedule A in figure 3.3 shows an example of indirect preemptions: The second, third and fifth task activation of τ_4 cannot replace cache blocks of τ_1 and the second activation of τ_4 cannot replace cache blocks of τ_2 because the preempted task is suspended.

The goal is to distinguish between the number of cache interferences and the number of preemptions. In figure 3.3 there are 12 task preemptions but only 8 cache interferences. Now consider schedule B in figure 3.4. It shows the same schedule

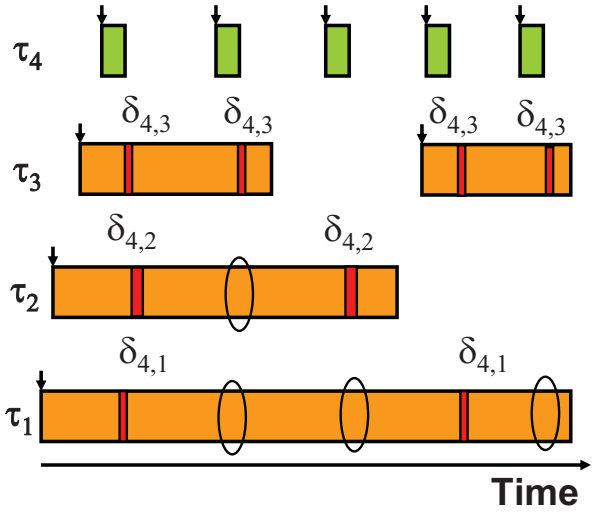

Figure 3.3. Schedule A.

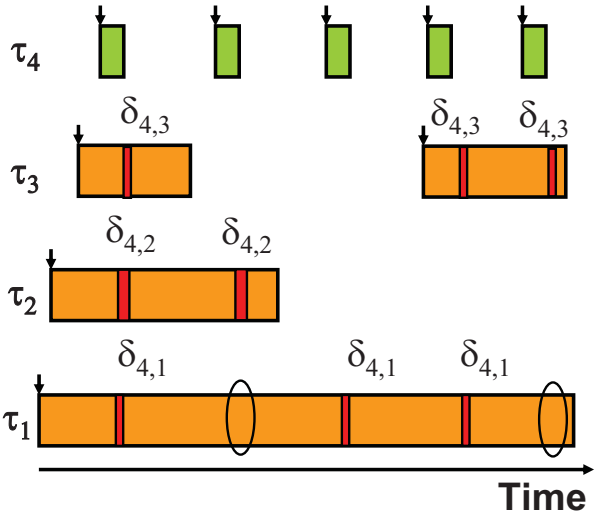

Figure 3.4. Schedule B.

where task τ_3 and τ_2 finish earlier than their WCET. Here, the second and the fifth activation of τ_4 are not interfering with task τ_1. The total number of task preemptions caused by τ_4 is 10 and the maximum number cache interferences is 8. In both examples the number of cache interferences is the same, however, the value of the preemption delay is different:

$$\Delta_4(R_1) = \delta_{4,1} \cdot 2 + \delta_{4,2} \cdot 2 + \delta_{4,3} \cdot 4 \qquad \text{for Figure 3.3} \tag{3.5}$$

$$\Delta_4(R_1) = \delta_{4,1} \cdot 3 + \delta_{4,2} \cdot 2 + \delta_{4,3} \cdot 3 \qquad \text{for Figure 3.4} \tag{3.6}$$

We show how the number of cache interferences can be bounded in the following Lemma. Based on the general schedulability equation 2.3 on page 27, we define a new schedulability analysis by considering indirect preemptions. We will substitute the term $\left\lceil \frac{R_i^n}{P_j} \right\rceil$ by $E_{j,i}^n$ and

$$crpd(j,i,R_i^n) = \Delta_{j,i}(R_i^n) \cdot t_{miss} \tag{3.7}$$

Lemma 3.1. Given a set of tasks T scheduled by a fixed priority preemptive scheduling policy. A task τ_i is schedulable and the response time is given by the following equations, if the response time $R_i^n = R_i^{n+1}$ converges for some n and is smaller or equal to its deadline D_i. Otherwise, the calculation terminates and no statement about schedulability is made. If all task of the set are schedulable, the task set is said to be schedulable.

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} (E_{j,i}^n \cdot C_j + \Delta_{j,i}(R_i^n) \cdot t_{miss}) \tag{3.8}$$

$\Delta_{j,i}(R_i^n)$ is given by the following equations:

$$\Delta_{j,i}(R_i^n) = \sum_{k=1}^{X_{j,i}(R_i^n)} max^k D_{j,i}(R_i^n) \tag{3.9}$$

$$X_{j,i}(R_i^n) \leq \sum_{k \in H_i \cap L_j \cup \{\tau_j\}} E_{k,i}^n \tag{3.10}$$

$$D_{j,i}(R_i^n) = \bigcup_{k \in H_i \cap L_j \cup \{\tau_i\}} \left\{ \delta_{j,k}^{E_{j,k}^n \cdot E_{k,i}^n} \right\} \tag{3.11}$$

$$H_i = \{\tau \in T | p(\tau) > p(\tau_i)\} \tag{3.12}$$

$$L_j = \{\tau \in T | p(\tau) < p(\tau_i)\} \tag{3.13}$$

$$E_{j,k}^n = \left\lceil \frac{R_k}{P_j} \right\rceil \quad R_k = R_k^n \text{ if } \tau_k \text{ is being iterated} \tag{3.14}$$

The above notations have the following meaning:

C_i denotes the core execution time of task τ_i including the instruction cache effect during execution.

B_i denote potential blocking times for task τ_i.

$\Delta_{j,i}(R_i^n)$ is an upper bound of the worst case cache interference caused by task τ_j regarding all lower priority tasks than τ_j and higher or equal to task τ_i: $\{\tau_{j-1}, \cdots, \tau_i\}$ during response time R_i^n.

$X_{j,i}(R_i^n)$ denotes an upper bound of the total number of cache interferences when τ_j preempts any tasks of the set $\{\tau_{j-1}, \cdots, \tau_i\}$ during response time R_i^n. Note that the task τ_j is explicitly included in the sum index k.

$\sum_k max^k M$ denotes the sum of the k largest elements of a set M. For example: $\sum_{k=1}^{3} max^k \{1,2,3,4,5,6\}$ evaluates to 15.

$D_{j,i}(R_i^n)$ denotes the set of preemption delays. It contains all possible preemption delays when τ_j preempts the tasks $\{\tau_{j-1}, \cdots, \tau_i\}$ at maximum frequency during response time R_i^n. We use the power notation for multiple occurrences of an element in a set. For example, $\{\delta_{j,k}, \delta_{j,k}, \delta_{j,k}\}$ is abbreviated as $\{\delta_{j,k}^3\}$. Note that the task τ_i is explicitly included in the sum index k.

H_i denotes the set of tasks with a strictly higher priority then task τ_i. The term $p(\tau_i)$ denotes the priority of task τ_i.

L_i denotes the set of tasks with a strictly lower priority then task τ_i. The term $p(\tau_i)$ denotes the priority of task τ_i.

$E^n_{j,k}$ denotes the number of task activations of task τ_j during the response time of task τ_k. P_j denotes the period of taskτ_j. The definition of $E_{j,k}$ distinguishes between R_k and R^n_k. If task τ_k is being iterated in equations 3.9-3.11 (e.g. $k = i$) then the most recently computed response time R^n_k is taken. Otherwise the response time of a (higher priority) task τ_k has *already* been calculated and the fixed-point has been reached. In this case, no index n can be given since n denotes the latest response time of task $\tau_i \neq \tau_k$.

Example

The lemma is applied to an example before the proof is given. Assume the following task set as specified in table 3.1. To simplify the calculations of this example, we assume one time unit as cache miss penalty, hence the reloading time for e.g. three cache blocks is three time units.

Task τ_i	τ_1	τ_2	τ_3
Priority p	1	2	3
Period P_i	100	500	1500
Execution time C_i	20	50	100
Preemption delays $\delta_{j,i}$	$\delta_{1,2} = 2$	$\delta_{2,3} = 12$	
$\delta_{j,i}$	$\delta_{1,3} = 10$		

Table 3.1. Example task set configuration.

The response time for task τ_1 is 20 time units, since it is the highest priority task and no preemptions can occur. The response time for task τ_2 is computed as follows:

$$
\begin{aligned}
R^0_2 &= 50 \\
R^1_2 &= 50 + E^0_{1,2} \cdot C_1 + \Delta_{1,2}(R^0_2) = 50 + 1 \cdot 20 + 2 = 72 \\
\Delta_{1,2}(R^0_2) &= \sum_{k}^{X_{1,2}(R^0_2)} max^k D_{12}(R^0_2) = max^1\{2\} = 2 \\
X_{1,2}(R^0_2) &= \sum_{k \in \{\tau_1\}} E^0_{k,2} = E^0_{1,2} = \left\lceil \frac{R^0_2}{P_1} \right\rceil = \left\lceil \frac{50}{100} \right\rceil = 1 \\
D_{1,2}(R^0_2) &= \bigcup_{\tau_k \in \{\tau_2\}} \delta^{E^0_{1,k} \cdot E^0_{k,2}}_{1,k} = \{\delta^{1 \cdot 1}_{1,2}\} = \{2\}
\end{aligned}
$$

The next iteration for R_2 converges with a total response time of 72 time units for task τ_2:

$$\begin{aligned}
R_2^2 &= 50 + E_{1,2}^1 \cdot C_1 + \Delta_{1,2}(R_2^1) = 50 + 1 \cdot 20 + 2 = 72 \\
\Delta_{1,2}(R_2^1) &= \sum_{k}^{X_{1,2}(R_2^1)} max^k D_{1,2}(R_2^1) = max^1\{2\} = 2 \\
X_{1,2}(R_2^1) &= \sum_{k \in \{\tau_1\}} E_{k,2}^1 = E_{1,2}^1 = \left\lceil \frac{R_2^1}{P_1} \right\rceil = \left\lceil \frac{72}{100} \right\rceil = 1 \\
D_{1,2}(R_2^1) &= \bigcup_{\tau_k \in \{\tau_2\}} \delta_{1,k}^{E_{1,k}^1 \cdot E_{k,2}^1} = \{\delta_{1,2}^{1 \cdot 1}\} = \{2\}
\end{aligned}$$

For task τ_3 four iterations are necessary. The first iteration initializes with the core execution time C_3 while the second is shown in the following:

$$\begin{aligned}
R_3^0 &= 100 \\
R_3^1 &= C_3 + E_{1,3}^0 \cdot C_1 + \Delta_{1,3}(R_3^0) + E_{2,3}^0 \cdot C_2 + \Delta_{2,3}(R_3^0) \\
&= 100 + 1 \cdot 20 + 12 + 1 \cdot 50 + 12 = 194 \\
E_{1,3}^0 &= \left\lceil \frac{R_3^0}{P_1} \right\rceil = \left\lceil \frac{100}{100} \right\rceil = 1 \quad E_{2,3}^0 = \left\lceil \frac{R_3^0}{P_2} \right\rceil = \left\lceil \frac{100}{500} \right\rceil = 1 \\
E_{1,2}^0 &= \left\lceil \frac{R_2}{P_1} \right\rceil = \left\lceil \frac{72}{100} \right\rceil = 1 \quad E_{3,3}^0 = \left\lceil \frac{R_3^0}{P_3} \right\rceil = \left\lceil \frac{100}{1500} \right\rceil = 1 \\
\Delta_{1,3}(R_3^0) &= \sum_{k}^{X_{1,3}(R_3^0)} max^k D_{1,3}(R_3^0) = 2 + 10 = 12 \\
&\quad X_{1,3}(R_3^0) = \sum_{k \in \{\tau_1 \tau_2\}} E_{k,3}^0 = E_{1,3}^0 + E_{2,3}^0 = 1 + 1 = 2 \\
&\quad D_{1,3}(R_3^0) = \bigcup_{\tau_k \in \{\tau_2, \tau_3\}} \delta_{1,k}^{E_{1,k}^0 \cdot E_{k,3}^0} = \{\delta_{1,2}^{1 \cdot 1}, \delta_{1,3}^{1 \cdot 1}\} = \{2, 10\} \\
\Delta_{2,3}(R_3^0) &= \sum_{k}^{X_{2,3}(R_3^0)} max^k D_{2,3}(R_3^0) = 12 \\
&\quad X_{2,3}(R_3^0) = \sum_{k \in \{\tau_2\}} E_{k,3}^0 = E_{2,3}^0 = 1 \\
&\quad D_{2,3}(R_3^0) = \bigcup_{\tau_k \in \{\tau_3\}} \delta_{2,k}^{E_{2,k}^0 \cdot E_{k,3}^0} = \{\delta_{2,3}^{1 \cdot 1}\} = \{12\}
\end{aligned}$$

Note in the calculation of $E_{1,2}^0$ that the calculation of the response time of task τ_2 has been finished, thus just R_2 is used (not R_2^0!), The index n is only valid for the task

whose response time is being computed. For the third and fourth iteration we only sketch the calculation. The interested reader is asked to verify these values.

$$\begin{aligned} R_3^2 &= C_3 + E_{1,3}^1 \cdot C_1 + \Delta_{1,3}(R_3^1) + E_{2,3}^1 \cdot C_2 + \Delta_{2,3}(R_3^1) \\ &= 100 + 2 \cdot 20 + 22 + 1 \cdot 50 + 12 = 224 \\ R_3^3 &= C_3 + E_{1,3}^2 \cdot C_1 + \Delta_{1,3}(R_3^2) + E_{2,3}^2 \cdot C_2 + \Delta_{2,3}(R_3^2) \\ &= 100 + 3 \cdot 20 + 32 + 1 \cdot 50 + 12 = 254 \end{aligned}$$

As a result, the response time for task τ_3 including all delays due to cache interference is 254 time units.

Proof of Lemma 3.1

The proof is organized in two parts. In the first part, it is shown that $\Delta_{j,i}(R_i^n)$ is an upper bound of the cache interference caused by τ_j during R_i. In the second part, it is shown that the total preemption delay caused by tasks $\tau_j, \cdots \tau_{i+1}$ during R_i is given by $\Delta_{j,i}(R_i^n) + \cdots + \Delta_{i+1,i}(R_i^n)$. In the following, we will assume that the task-id is chosen corresponding to its priority, e.g. task τ_1 has the lowest priority in the task system (e.g. the task with id 1 has the lowest priority) and we assume that task τ_j has a higher priority than task τ_i (lower task-id means lower priority). These notations are used in the proof to simplify the description.

Part 1. The first part is organized as a proof by induction. The equations 3.9-3.11 are first translated into an iterative version, shown in equations 3.15-3.17.

$$\Delta_{j,i}(R_i^n) = \sum_{k=1}^{X_{j,i}^{j-1}(R_i^n)} max^k D_{j,i}^{j-1}(R_i^n) \tag{3.15}$$

$$X_{j,i}^k(R_i^n) \leq X_{j,i}^{k-1}(R_i^n) + E_{k,i}^n \qquad i < k \leq j-1 \tag{3.16}$$

$$D_{j,i}^k(R_i^n) = D_{j,i}^{k-1}(R_i^n) \cup \{\delta_{j,k}^{E_{j,k}^n \cdot E_{k,i}^n}\} \qquad i < k \leq j-1 \tag{3.17}$$

The term $X_{j,i}^k(R_i^n)$ denotes the number of cache interferences where τ_j preempts the lower priority tasks $\{\tau_i, \cdots, \tau_k\}$ during response time R_i^n. It is initialized with the number of preemptions when task τ_j preempts only task τ_i: $X_{j,i}^i(R_i^n) = E_{j,i}^n$ which is equivalent to the number of task activations of task τ_j during the response time R_i^n ($E_{j,i}^n$).

The term $D_{j,i}^k(R_i^n)$ denotes the set of all possible preemption delays if τ_j preempts the tasks $\{\tau_i, \cdots, \tau_k\}$ at maximum frequency during response time R_i^n. It is initial-

ized with $D^i_{j,i}(R^n_i) = \{\delta^{E^n_{j,i}}_{j,i}\}$ which corresponds to the set of preemption delays if τ_j preempts only task τ_i at maximum frequency during response time R^n_i.

The proof starts by considering the cache interference of τ_j and the lower priority task τ_i. Then each iteration considers the next intermediate task τ_k between τ_i and τ_j in a bottom-up fashion. The calculation of the number of cache interferences, $X^k_{j,i}(R^n_i)$ and the set of preemption delays, $D_{j,i}(R^n_i)^k$, are now computed.

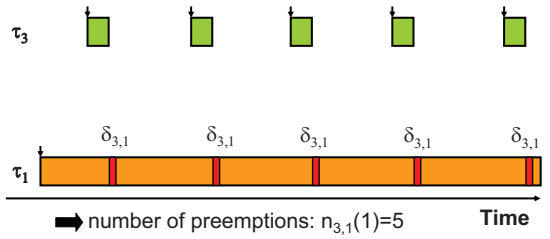

Figure 3.5. Induction start.

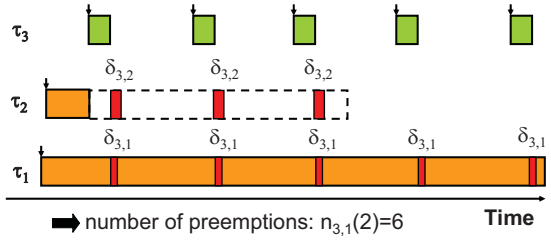

Figure 3.6. Case 1 - zero preemptions.

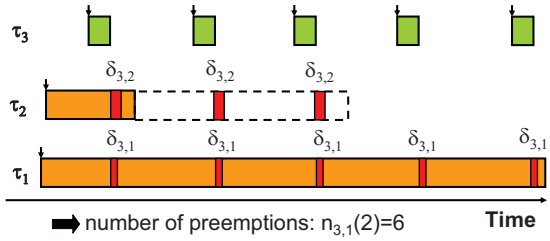

Figure 3.7. Case 2 - single preemption.

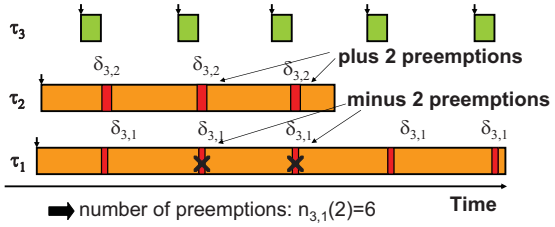

Figure 3.8. Case 3 - multiple preemptions.

Induction start. We consider the preemption delay when τ_j only preempts the lowest priority task τ_i of the task set. This scenario is shown in figure 3.5. The induction start is given by $X_{j,i}(R^n_i) = E^n_{j,i}$ and $D_{j,i}(R^n_i) = \{\delta^{E^n_{j,i}}_{j,i}\}$. The number of cache interferences is given by the number of task activations of τ_j during R^n_i. In the schedule, we use the abbreviated notation $n_{3,1}(2) = 5$ instead of $X^2_{3,1}(R^n_1)$. Then term $D_{j,i}(R^n_i)$ contains $E_{j,i}$ times the single preemption delay $\delta_{j,i}$: $D_{j,i}(R^n_i)$. The equation 3.15 results to the sum of all these terms, which is correct.

Induction step for $\tau_{k-1} \rightarrow \tau_k$. We assume that the tasks $\{\tau_i, \cdots, \tau_{k-1}\}$ have been considered and the terms $X^{k-1}_{j,i}(R^n_i)$ and $D^{k-1}_{j,i}(R^n_i)$ have been calculated. First we proof equation 3.16 by showing, that the number of cache interferences increases by *at most one* for each task activation of τ_k during R^n_i. Then, $E^n_{k,i} + X^{k-1}_{j,i}(R^n_i)$ represents all task activations of $\{\tau_k, \tau_{k-1}, \cdots, \tau_i\}$ during R^n_i. Three possible schedules have to be analyzed:

Case 1: Task τ_k executes and finishes before a preemption by τ_j occurs. This is shown in figure 3.6. In this case, no additional preemption delays occur. However,

incrementing the number of cache interferences by one is safe ($n_{3,1}(2) = 6$). This is the reason for the in-equality operator in equation 3.16 and 3.10.

Case 2: Task τ_k executes just as long as such that exactly one activation by τ_j interferes. This is shown in figure 3.7. The number of cache interferences is incremented by one ($n_{3,1}(2) = 6$). All remaining preemptions by τ_j affect lower priority tasks and, consequently, have already been considered in $X_{j,i}^{k-1}(R_i^n)$.

Case 3: Task τ_k executes for some time such that τ_j preempts τ_k p-times, where p is in the range of $2 \leq p \leq E_{j,k}^n$. This is shown in figure 3.8 (where $p = E_{3,2} = 3$). For the first preemption, the number of cache interferences is increased by one, as in case 2. For all remaining $p-1$ preemptions, the number of cache interferences would be increased by $p-1$. However, these preemptions cannot affect any lower priority tasks because these tasks are suspended. Also, the preemption delay refers the the worst case execution path of τ_j, hence a second activation of τ_j can be considered as using the same (upper bound of) cache blocks as in its first execution. Thus, τ_j cannot replace any additional cache blocks of lower priority tasks. Therefore, the preemption delay $\delta_{j,i}$ has to include all paths in the calculation to consider *all* used cache blocks of the preempting task. A path-based cache analysis, as described in [89], does not meet this requirement. Thus, $p-1$ would be subtracted from the number of cache interferences. None of the remaining $E_{j,i}^n - p$ preemptions require additional cache reloads for τ_k because τ_k is preempted exactly p times and the delay for a lower priority task has been considered in $X_{j,i}^{k-1}(R_i^n)$ according to the induction assumption. In summary, the total number of *additional* cache interferences increases by one for each activation of task τ_k ($n_{3,1}(2) = 6$).

We have shown that for each activation of τ_k, the number of cache interferences increases by at most one. Thus, the in-equation 3.16 is correct.

Set of preemption delays. The set $D_{j,i}^k(R_i^n)$ contains all possible preemption delays. The equation 3.17 represents the worst case scenario, that τ_k is preempted by τ_j at maximum frequency. This frequency is given by the product of the number of activations of τ_j during a single execution of τ_k ($E_{j,k}^n$) and the number of activations of τ_k during R_i^n ($E_{k,i}^n$). Therefore, the preemption delay ($\delta_{j,k}$) is inserted into the set $D_{j,i}^k(R_i^n)$ $E_{j,k}^n \cdot E_{k,i}^n$-times.

Maximum operation. The iterative process is finished for $k = j-1$ and equation 3.15 is calculated. The term $X_{j,i}j-1(R_i^n)$ denotes the maximum number of preemption delays and $D_{j,i}^{j-1}(R_i^n)$ all possible preemption delay costs that are caused by task τ_j regarding the tasks $\{\tau_{j-1}, \cdots, \tau_i\}$ during R_i^n. The maximum delay is given by

the sum of the $X_{j,i}^{j-1}(R_i^n)$ greatest elements of the set $D_{j,i}^{j-1}(R_i^n)$. Note, that this set includes the case when a task completes *before* its worst-case execution time.

Part 2. The equation 3.8 requires to calculate $\Delta_{j,i}(R_i^n)$ for each task τ_j in $i < j \leq h$, where τ_h shall denote the highest priority task. The calculation of $\Delta_{j,i}(R_i^n)$ is carried out independently. This might be a cause for overestimation, because the maximum-operation in equation 3.9 (3.15 respectively) chooses the maximum preemption delays only considering τ_j and, therefore, assumes implicitly a specific schedule. This might lead to an overestimation because the implied schedules in the calculation of $\Delta_{j_1,i}(R_i^n)$ and $\Delta_{j_2,i}(R_i^n)$ might be excluding. However, in order to solve this problem, all (implicitly assumed) schedules would had to be tested which is an NP-complete optimization problem and exponential in number of tasks. The author decided to accept the overestimation in favor of a reduced analysis complexity. □

3.2.3 Time complexity

In this section we evaluate the time complexity of the simplified approach of section 3.2.1 and the advanced approach of section 3.2.2. For timing complexity considerations we use the simplified notation of $E_{j,i}$ instead of $E_{j,i}$ and R_i instead of R_i^n. We will use n for the number of tasks in the system.

The time complexity for calculating $\Delta_{j,i}^s(R_i)$ is $O(n)$. The calculation was given in equation 3.3 which consists of a sum of $j-1$ summands. Each summand involves two multiplications. The time to add as well as to multiply is considered as constant. Thus we have

$$O(\Delta_{j,i}^s(R_i)) = O((2 \cdot 1) \cdot n + n \cdot 1) = O(n) \tag{3.18}$$

The time-complexity for calculating the preemption delay $\Delta_{j,i}(R_i)$ of the schedulability analysis in section 3.2.2 is

$$O(\Delta_{j,i}(R_i)) = O(n \cdot E_{j,i} \cdot \log(E_{j,i} \cdot n) \tag{3.19}$$

where n denotes the number of tasks and $E_{j,i}$ the number of task activations of task τ_j during the response time of task τ_i.

We evaluate the complexity of the equations 3.9, 3.10, and 3.11 in bottom up fashion. The time complexity of equation 3.11 depends on inserting the preemption costs. The preemption costs $\delta_{j,k}$ are inserted $E_{j,k} \cdot E_{k,i}$ times in the set $C_{j,i}$. An upper bound of this product is $E_{j,i}$, the total number of task activations of a higher priority task j. The equation is evaluated $j-1-i$ times, which can be bounded by n. The time complexity for this operation is $O(n \cdot E_{j,i} \cdot t_{insert})$, in which t_{insert} is the time to insert

an element. In our implementation, we use a single-linked list to represent a set, in which elements are appended at the end. This takes constant time: $O(n \cdot E_{j,i} \cdot 1)$.

The time complexity of equation 3.10 depends on adding $j-1-i \leq n$ elements. It can be assumed that adding two integers takes constant time. Thus, the time complexity is bounded by $O(n \cdot 1)$.

The time complexity of equation 3.9 depends on repeatedly choosing the maximum element of a set and adding them. The number of elements in the set $C_{j,i}(n-1)$ is bounded by $n \cdot E_{j,i}$ (see above). Taking the maximum involves sorting the set. These operations can be done in in $O(M \log M)$ where M denotes the number of elements in the set. In this case, we have: $O(n \cdot E_{j,i} \cdot \log(n \cdot E_{j,i}))$. In summary, we get:

$$\begin{aligned} O(\Delta_{j,i}(R_i)) &= O(n \cdot E_{j,i}) + O(n) + O((n \cdot E_{j,i}) \cdot \log(n \cdot E_{j,i})) \\ &= O(n \cdot E_{j,i} + n + n \cdot E_{j,i} \cdot \log(n \cdot E_{j,i})) \\ &= O(n \cdot E_{j,i} \cdot \log(n \cdot E_{j,i})) \end{aligned}$$

In this section we have presented a cache-aware response time analysis for fixed priority preemptive scheduling. For this response time analysis, the cost for each preemption has to be calculated. This is described in the next section.

3.3 Delay for multiple preemptions

In previous cache analysis approaches, each preemption has been assumed at the worst case preemption point. In early work, an entire cache flush is assumed [19], in more recent work, the intersection of used cache blocks of the preempting task and useful cache blocks of the preempted task has been calculated [67] [89] [111]. Lee et al. analyze in [66] the preemption cost of different preemption points, but take only the preempted task into account.

However, cache behavior is highly dynamic and multiple preemptions cannot always remove the worst case number of cache blocks. The bound of the total delay of two preemptions can be smaller than the sum of the bounds of the delay of each preemption. Further, all preemptions might not occur at the same worst case preemption point in general.

Until we better understand the dynamic cache behavior at a context switch, we will not understand the larger question of accurate timing analysis for instruction caches which is necessary for a precise schedulability analysis. If cache behavior is unpredictable, caches would be switched off for real-time applications leading to inefficient designs.

Preemption delay for multiple preemptions depends on the execution history of a task as well as its *preemption history*. Cache blocks that are replaced by a preemption cannot be replaced a second time thus reducing the total delay for multiple preemptions.

In this section we make the following contribution. The preemption delay for instruction caches is calculated for multiple preemptions. We use the term of a *preemption scenario* in this context. Section 3.3.1 describes the system model and provides a motivating example of the proposed three-step approach. First, an iterative optimization algorithm which determines the worst case preemption scenario is described in section 3.3.2. Second, a data flow analysis which calculates an upper bound of the preemption delay for a specific preemption scenario is presented in section 3.3.3. Finally, the schedulability analysis is revisited to consider multiple preemption costs in section 3.3.5.

3.3.1 System model and motivating example

The analysis is based on the following assumptions. The preempted task τ_i as well as the preempting task τ_j are represented by its control flow graph. Each basic block contains the memory addresses of the corresponding assembly instructions, which can be extracted from a memory map file, after compiling and linking the source code of each task. Disassembling tools like *fromelf* of the RealView from ARM [6] have been used for this purpose. Like in the previous section, the only instruction caches are considered. An analysis for data caches is presented in chapter 5. Further, we assume that the total number of preemptions is given by a schedulability analysis. As described in section 3.2, only information necessary is the total number of task preemptions.

In previous approaches [67] [92] [123] [19], each preemption has been assumed to take place at the worst case preemption point with a maximum delay of $\delta_{j,i}$. This conservative approximation was also used in the simplified cache-aware schedulability analysis of section 3.2.1 in equation 3.3 as well as in the cache-aware schedulability analysis of section 3.2.2 which considers indirect preemptions in equation 3.11.

Thus the *total* delay for n preemptions, when task τ_j preempts task τ_i is given by:

$$n \cdot \delta_{j,i} \tag{3.20}$$

There are two reasons why this upper bound can be improved: First, all preemptions might not take place at the worst case preemption point. For example, not every preemption can occur within a loop, where the preemptions costs are usually high.

Second, preemption delays are not additive. In Lee et al. [66] the preemption delay for several preemption points has been computed and sorted descendently. Then, those preemption points where chosen (e.g. in possibly different loops) with the highest costs. If two preemptions occur shortly after each other and each preemption replaces the entire cache for simplicity, then the total delay for these two preemptions would be estimated as the time for two cache flushes with equation 3.20. However, if the task does not reload sufficiently many cache lines between these two preemptions, the second preemption cannot replace the entire cache again and the total time delay for two preemptions is much less than the time for two cache flushes. Therefore, we say that preemption delays are not additive but the total time delay for multiple preemptions has to be computed.

Example

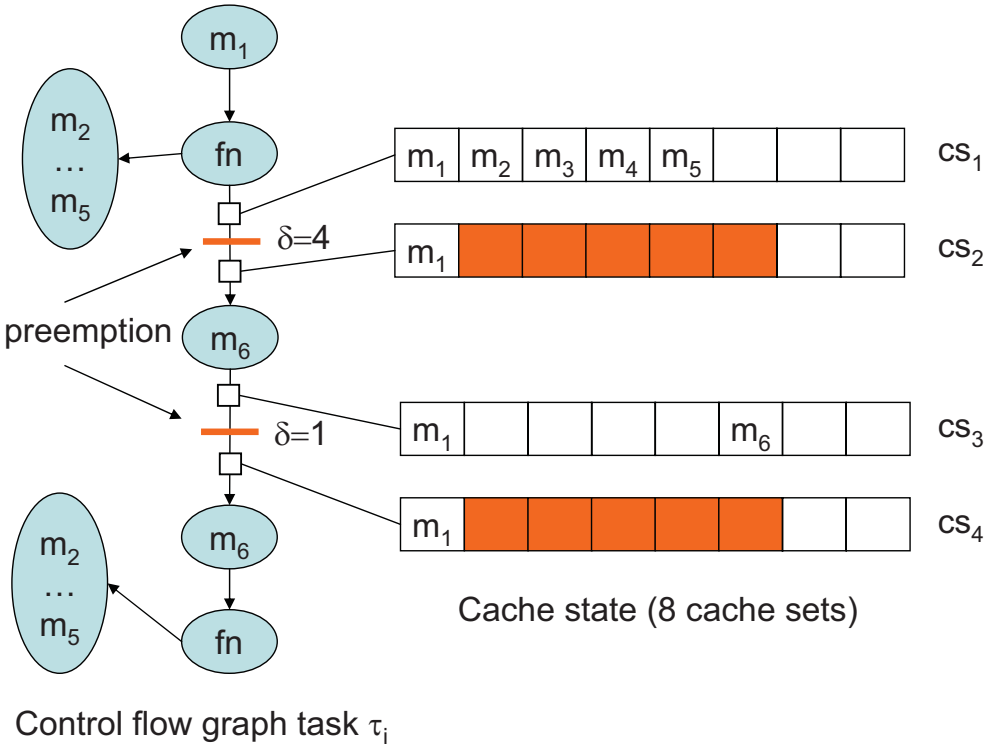

Figure 3.9. Corelation of preemption delay of multiple preemptions.

Figure 3.9 shows a control flow graph and the corresponding cache state of a direct mapped cache with eight sets for some program execution points. A function call is abbreviated by fn. The useful memory blocks are shown for each basic block as well as the cache state at some points of the program execution. After executing basic block 1 and 2, cache state cs_1 contains memory blocks $m_1, \cdots, m_5$. We assume that a

preemption removes 5 memory blocks $m_2, \cdots, m_6$, resulting to a preemption delay of 4 cache blocks ($m_2, \cdots, m_5$). The resulting cache state is shown in cs_2. Then, basic block 3 is executed. The cache contents is shown in cs_3. If a second preemption occurs after basic block 3, it causes a delay of only *one* cache block (m_6), because the cache blocks of the function call have already been removed.

Notations

We define the following terms.

- A preemption scenario, $S^n_{j,i}$ is defined it as:

$$S^n_{j,i} = (\tau_j, \tau_i, \{p_1, p_2, \cdots, p_n\}) \tag{3.21}$$

  where task τ_j preempts τ_i n times. The actual preemption points p_k correspond to some basic blocks in the preempted task τ_i.

- The total preemption delay for a preemption scenario $S^n_{j,i}$ is denoted by $\delta(S^n_{j,i})$.

- The term $\delta_{j,i}(n)$ denotes the n-th marginal preemption cost. It corresponds to the preemption delay at the nth preemption point in the worst case preemption scenario $\delta_{p_n}((\tau_j, \tau_i p_1, \cdots, p_n))$ while considering all preemptions at the preemption nodes $\{p_1, \cdots p_{n-1}\}$. This delay will be calculated in equation 3.24.

- The variable $n_{j,i}$ denotes the frequency how often a task τ_j preempts τ_i in a single execution of task τ_i.

For the example, the preemption scenario of figure 3.9 is given by $S^2_{j,i} = (\tau_j, \tau_i, \{B_2, B_3\})$. and $\delta(S^2_{j,i}) = 5$ is the total preemption delay for the scenario.

Overview of methodology

With these definitions, we can formulate the problem. Given the number of task preemptions $n_{j,i}$ when task τ_j preempts τ_i, we want to find the preemption scenario $S^{n_{j,i}}_{j,i}$ with the maximum preemption delay. This addresses the two issues for the overestimation in equation 3.20. First, we determine the worst case preemption points, and second, we consider the preemption history in the calculation of the preemption delay for each preemption point.

In the approach by Lee [66] the n-th greatest preemption costs are calculated in isolation and then used in the schedulability analysis but this calculation considers only the useful cache blocks of the preempted task. In a more recent paper [67], Lee

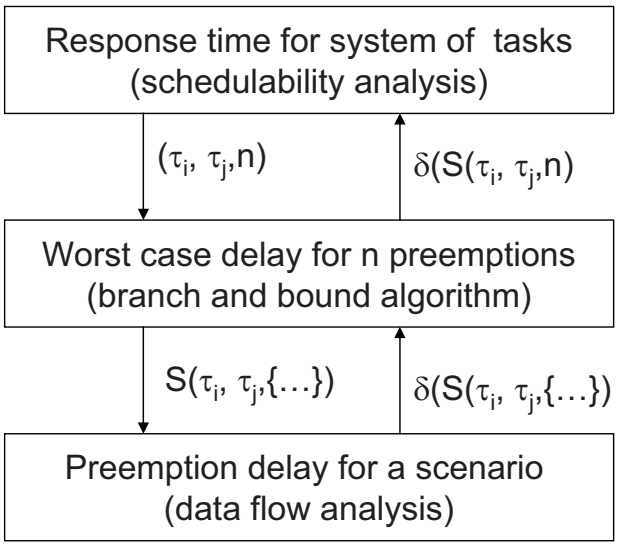

Figure 3.10. Overview of multiple preemption delay analysis.

consider both, the preempted as well as the preempting task, but assume the worst case preemption point for each preemption again.

Figure 3.10 shows an overview of the analysis. First, schedulability analysis requests for the preemption delay, when task τ_j preempts task τ_i for n times. Second, the worst case preemption points are identified using a branch and bound algorithm. This is described in section 3.3.2. Third, the branch and bound algorithm requests the preemption delay for each preemption scenario. The calculation for each preemption scenario is presented in section 3.3.3 that considers the preemption history. The revised schedulability analysis for nth marginal preemption delays is presented in in section 3.3.5.

3.3.2 Preemption scenarios

This section describes an algorithm to find the worst case preemption scenario.

A preemption scenario $S^1_{j,i}$ with a single preemption at node p_k and the cache-related preemption delay δ is denoted by:

$$S^1_{j,i} = (\tau_j, \tau_i, \{p_k\})) \quad \delta((\tau_j, \tau_i, \{p_k\}))$$

The time delay for a preemption scenario with multiple preemptions was defined as $\delta((\tau_j, \tau_i, \{p_1, \cdots, p_n\}))$ in equation 3.21. The calculation of these time delays is presented in section 3.3.3. In section, we propose to use the branch and bound algorithm to determine which preemption points p_k to choose.

If all possibilities were evaluated, this would be too time-consuming: Given a control flow graph with m nodes there are $\binom{m}{n}$ possible combinations of preemption points for n preemptions, assuming one preemption at each node. Such a problem can be solved by an optimization algorithm which finds the optimum value.

A branch and bound algorithm is such an algorithm. It consists of two components, the branching and bounding step. The branching step decomposes the problem into sub-problems by creating a tree-structure. A solution is represented by a path from the root node to a leaf node. If a solution is found, it is saved as a lower bound. Then, the algorithm continues to backtrack in a different sub-tree. The bounding step compares this lower bound with an approximation of the upper bound for the current sub-tree. If the upper bound is smaller than the lower bound of an existing solution, then this sub-tree can be discarded and the search continues in a different sub-tree. The hope is to bound as many sub-trees as possible very early during the search. In the worst case, the algorithm has to evaluate every path in the subtree, which takes exponential time, however, if the branching and bounding criteria are well designed, the optimization algorithm converges very fast.

The branch and bound algorithm starts with the computation of the single preemption delay $\delta((\tau_j, \tau_i, \{p_k\}))$ for each preemption point p_k of the control flow graph. Then, the nodes are sorted descendently according to their preemption delays in a node list L. Every node of a loop l_i is inserted LB_i times in L. The maximum number of loop iterations LB_i has to be specified manually. Then, a tree is constructed, where nodes represent preemption points p_i and the depth represents the number of preemptions. Nodes on the same level k are possible candidates for the kth preemption. A path from the start-node to a leaf node represents a preemption scenario. A node is a leaf node, if its level is equal to the number of preemptions $n_{j,i}$.

Initially, the node p_k with the maximum preemption cost is inserted at the root node. Branching is decided by the single preemption cost $C(n_k)$. As the next preemption point, the node is chosen, with the highest cost among the available nodes of node list L. This can be done in $O(1)$ because the node with the largest preemption delay is the first element of L. The bounding step is controlled by comparing a lower bound of an existing solution with an upper bound of a possible solution. A sub-tree at preemption point p_k is bounded (and discarded), if the estimated cost for k preemptions plus the upper bound of possible $n-k$ preemptions is smaller than the lower bound lb of an existing solution. In this case the search backtracks at the $k-1$th level.

Example

Figure 3.11 illustrates an example of finding the worst case preemption delay for three preemptions. The control flow graph is shown on the left side and the decision diagram is shown on the right side of the figure. We assume the single preemption delays of 10 cache blocks for nodes 6,7,8, this is reasonable when node 6 and 9 contains a function call, like in figure 3.9. A preemption at node 3 causes two cache misses, for example because the same cache block is used in node 2 and 3. All other nodes have a single preemption delay of one cache block. First, the root node is

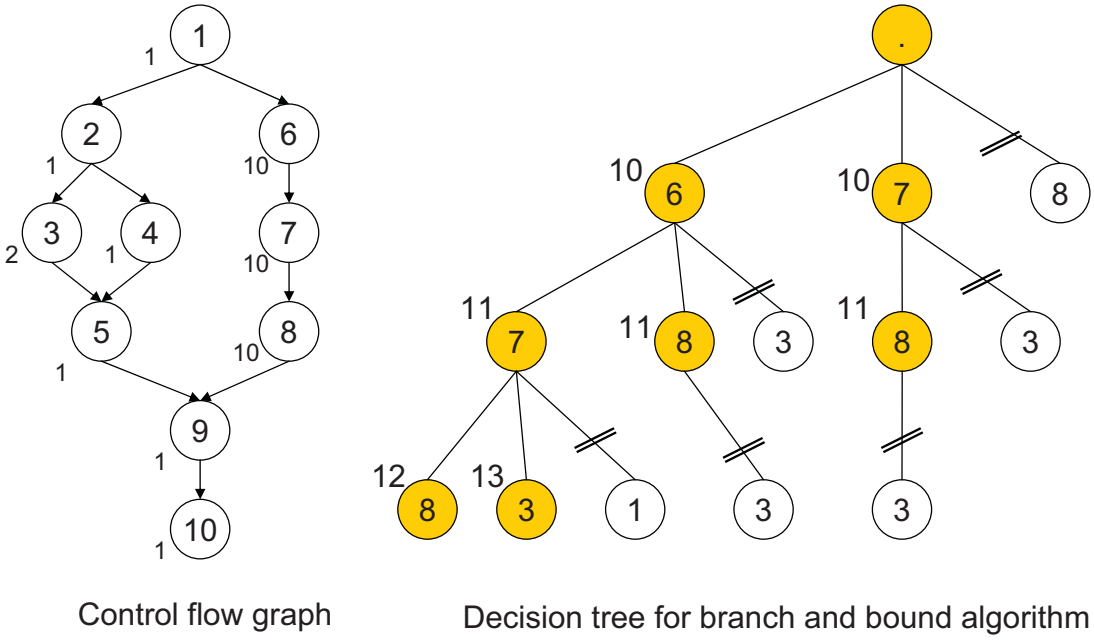

Figure 3.11. Example for branch and bound algorithm.

created. The list L of possible nodes is maintained during the algorithm. Refer to table 3.2 for a complete listing of each step. Initially, $L = (6,7,8,3,2,4,5,9,10)$, according to the assumed preemption delays and the lower bound lb is zero.

The branching step chooses always the first element of node list L, in this case node 6. Then, the bounding step evaluates if the current cost (0) plus an upper bound of expected costs is lower than the lower bound lb. The upper bound is given by the sum of the largest single preemption delays of the list L. The term $0 + (10 + 10 + 10) = 30 \leq 0 = lb$ is evaluated to false, thus this node is not bounded and node 6 is inserted. The preemption delay $\delta(\tau_j, \tau_i, \{6\}) = 10$ is computed as described in section 3.3.3 (compare to Step 1 in table 3.2). Then, the node list is updated:

Step	L	Upper bound	Action	S	$\delta(S)$	lb
1	$(6,7,8,3,1,\cdots)$	$0+(10+10+10)$	insert 6	$\{6\}$	10	0
2	$(7,8,3,1,\cdots)$	$10+(10+10)$	insert 7	$\{6,7\}$	11	0
3	$(8,3,1,\cdots)$	$11+(10)$	insert 8, bt 7	$\{6,7,8\}$	12	12
4	$(3,1,\cdots)$	$11+(2)$	insert 3, bt 7	$\{6,7,3\}$	13	13
5	$(1,\cdots)$	$11+(2)$	bound 1, bt 6			13
6	$(8,3,1,\cdots)$	$10+(10+2)$	insert 8,	$\{6,8\}$	11	13
7	$(3,1,\cdots)$	$11+(2)$	bound 3, bt 6			13
8	$(3,1,\cdots)$	$10+(2+1)$	bound 3, bt root			13
9	$(7,8,3,1,\cdots)$	$0+(10+10+2)$	insert 7	$\{7\}$	10	13
10	$(8,3,1,\cdots)$	$10+(10+2)$	insert 8	$\{7,8\}$	11	13
11	$(3,1,\cdots)$	$11+(2)$	bound 3, bt 7			13
12	$(3,1,\cdots)$	$10+(2+1)$	bound 3, bt root			13
13	$(8,3,1,\cdots)$	$0+(10+2+1)$	bound 8,			13

Table 3.2. Branch and bound algorithm. Node list L, term for upper bound, action, preemption scenario (S), total preemption delay $\delta(S)$ of scenario S, and lower bound (lb).

$L=(7,8,3,1,\cdots)$. For the presentation of the example, we will use this reduced list to save space. The remaining nodes $2,4,5,9,10$ have the same cost and are not important for the demonstration of the algorithm. Node 7 is chosen in step 2, and the bounding step evaluates $10+(10+10)\leq 0$ to false, thus inserting node 7 with the total preemption cost $\delta(\tau_j,\tau_i,\{6,7\})=11$. The cost is not 20, because cache blocks at not 6 are removed once and cannot be removed a second time. Here we assume an additional cost of one cache block for node 7. In step 3, the node list contains: $L=(8,3,1,\cdots)$. The branching step chooses node 8 and the bounding step evaluates $11+(10)=21\leq 0$ to false, thus node 8 is inserted and the total preemption delay is assumed to be $\delta(\tau_j,\tau_i,\{6,7,8\})=12$. As the leaf node is reached (level=3), the lower bound is set to 12 and the search backtracks at node 7. In step 4, the available nodes are $L=(3,1,\cdots)$ and node 3 is inserted with new lower bound of 13 cache blocks. Then, the algorithm backtracks at node 7. In step 5, the available nodes are $L=(1,\cdots)$ and the bounding step results true for the comparison $(13\leq 13)$, thus node 1 is bounded and the search backtracks at node 6. The data for all steps of the analysis is shown in table 3.2.

In summary, 13 computations of the upper bound (bounding step) and 7 computation of the preemption delay for a scenario were necessary compared to $\binom{10}{3}=120$ possible combinations. This significant reduction was possible because many subtrees were bounded very early.

The algorithm does not test whether the preemption scenario is feasible. For this example, the scenario $(\tau_j,\tau_i,\{6,7,3\})$ is not feasible, because node 3 is located in

a different branch. Such an additional check could be provided by a reachability algorithm, but would increase the complexity. Currently, such a feasibility check has not been implemented in the branch and bound algorithm.

3.3.3 Delay for preemption scenarios

Methods to calculate the preemption delay which consider the preempted as well as the preempting task have been proposed by Lee et al. [67] and Mitra et al. [89], see also section 2.5.1 and section 2.5.2. Each preemption is assumed to take place at the worst case preemption point.

Even though, these methods provide safe upper bounds, they are not tight. As motivated in section 3.3.1, the total bound of the delay for multiple preemptions can be smaller than the sum of each bound of the preemption delay. If the preemption history is not considered in cache-related delay analysis, the response time for a task could be overestimated which consequently would lead to an inefficient system design.

In this section we present a method to consider the preemption history. Instead of an isolated calculation of the preemption delay, we determine the delay for preemption scenarios.

Insertion of preemption nodes

Possible preemption points are specified by a preemption scenario. Each node p_k in a preemption scenario $S = (\tau_j, \tau_i, \{p_1, \cdots p_n\})$ represents a preemption point of the preempted task τ_i by the preempting task τ_j. We insert a node containing all used cache blocks of the preempting task τ_j after each preemption point and re-calculate the properties of useful cache blocks of the preempted task.

Example

Figure 3.12 shows an example for the preemption scenario of figure 3.11. The modified graph is shown for the worst case preemption scenario $S = (\tau_j, \tau_i, \{3,6,7\})$. In this case, preemption nodes were inserted after B_3, B_6 and B_7.

Revised data flow analysis

The data flow analysis for calculation the useful cache blocks has to be modified. If it were not, the set of useful cache blocks could contain some cache blocks of the *preempting* task. This would increase the preemption delay, because cache blocks of the preempting task would be subject to a preemption itself, which is incorrect.

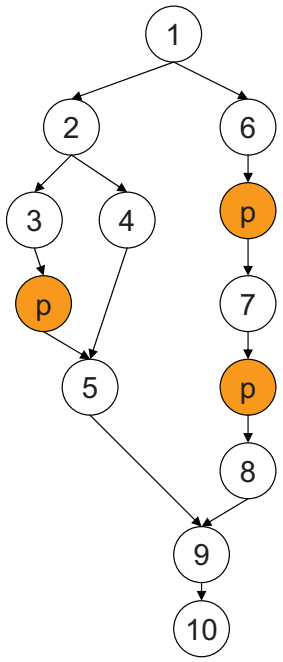

Figure 3.12. Control flow graph with preemption scenario $(\tau_j, \tau_i, \{3,6,7\})$.

In the following, we use the data flow analysis technique by Lee et al., as presented in section 2.5.1, to compute the set of useful cache blocks. The technique by Mitra et al. could have been used as well. In chapter 4 we propose a scalable precision cache analysis which combines these two methods to trade off analysis time complexity and analysis precision.

The calculation of useful cache blocks involves the calculation of reaching cache blocks (RMB) and live cache blocks (LMB) (figure 2.2). The $gen^c[P]$-set of each preemption node P is defined by the used cache blocks of the preempting task $RMB^{\tau_j}[end]$, in which *end* is the last node of the preempting task τ_j. The *RMB* calculation for all preemption nodes P is modified by replacing equation 2.9 (shown here again):

$$RMB^c_{out}[B] = gen^c[B] \cup (RMB^c_{in}[B] - kill^c[B])$$

by equation 3.22, in which P denotes a preemption node.

$$RMB^c_{out}[P] = RMB^c_{in}[P] - kill^c[P] \tag{3.22}$$

Note, that equation 3.22 is *only* used for preemption nodes, for all other nodes the equation 2.9 is used. The new equation 3.22 reflects the fact that cache blocks loaded by the preempting task, e.g. the $gen^c[P]$, are not inserted in the $RMB^c_{out}[P]$ set, however, the $gen^c[P]$ as part of the $kill^c[P]$-set can remove cache blocks. Analogously the calculation for *LMB* is modified.

Finally, the useful cache blocks of the preempted task are computed by the intersection of reaching cache blocks (RMB) and live cache blocks (LMB), according to section 2.5.1.

The preemption delay for a preemption scenario S with a set of preemption points $\{p_1, \cdots, p_n\}$ is then calculated by the sum of preemption delays at each preemption node:

$$\delta((\tau_j, \tau_i p_1, \cdots, p_n)) = \sum_{i=1}^{n} USE^{\tau_i}[p_i] \cap RMB^{\tau_j}[end] \tag{3.23}$$

In which *end* is the last node of the preempting task τ_j. We use the general notation USE^{τ_i}. It can be substituted by USE_{lee} (equation 2.12 on page 40), USE_{mitra} (equation 2.20 on page 43) or by the scalable analysis USE_{scale} (equation 4.18 on page 4.18).

Equation 3.23 represents the entire delay for all n preemptions. In the branch-and bound, and later in the schedulability analysis, the nth marginal preemption cost is of interest. In addition to equation 3.23 we define the preemption cost at node p_n, assuming that a preemption took place at preemption nodes $p_1, \cdots, p_{n-1}$, by

$$\delta_{p_n}((\tau_j, \tau_i p_1, \cdot, p_n)) = USE^{\tau_i}[p_n] \cap RMB^{\tau_j}[end] \tag{3.24}$$

It represents the preemption cost at node p_n, while preemption nodes have been inserted at $p_1, \cdots, p_{n-1}$ and the data flow analysis properties have been re-calculated.

Iteration space in loops

Basic blocks can be executed more than once for example in loops. The control flow graph represents only the structure of a program and different iteration spaces in loops cannot be distinguished. The insertion method of preemption nodes in the CFG, as described in the last section, has to be revised.

We explain the problem with an example, in which a task has a loop with node n_1 and n_2 and a loop bound of 2 iterations. Assuming that the task is preempted 2 times. For example, consider a preemption scenario $S_1 = (\tau_j, \tau_i, \{n_1^1, n_2^1\})$, in which the task τ_i is preempted by τ_j once at node n_1 and once at node n_2. We further assume that the single preemption delay at node n_1 and n_2 is 5 cache blocks while the delay at n_2 is only one cache block, if there was a preemption at n_1. Figure 3.13 shows the CFG and the iteration space for the first and second loop iteration.

According to the insertion method, a preemption node is inserted after n_1 and after n_2. The preemption delay is calculated as $1 \cdot 5 + 1 \cdot 1 = 6$ cache blocks. This implicitly assumes that both preemptions occurred in the *same* loop iteration (upper part in figure 3.13. If the preemptions would take place in different iterations, e.g. a preemption

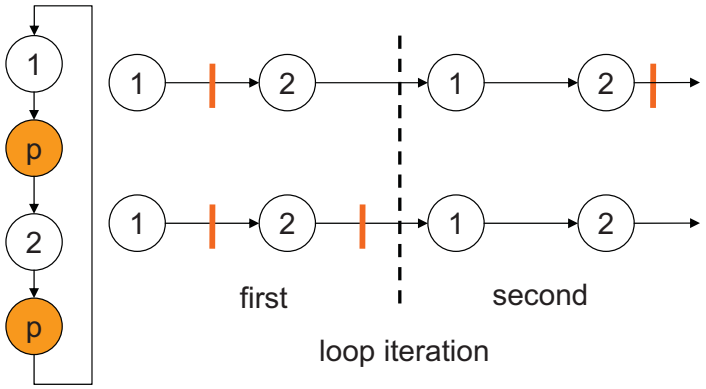

Figure 3.13. CFG with preemption points and un-rolled loop.

at n_1 in the first and the preemption at n_2 in the second, as shown in the lower part in figure 3.13 the preemption delay would be: $1 \cdot 5 + 1 \cdot 5 = 10$. Thus, the insertion method is not conservative in this case.

A solution to this problem is calculate the preemption delay of multiple preemptions in loops separately if the number of the same preemption point is less then the iteration bound. The DFA provides a conservative result, if the different preemption points are placed in the same iteration.

We distinguish two cases:

1.) The number of each preemption point p_l in the preemption scenario appears as often as the iteration bound lb. Then, a preemption scenario can be written as $S_{j,i} = (\tau_j, \tau_i, \{p_{l_1}^{lb}, \cdots, p_{l_n}^{lb}, \})$ where p_{l_1} to p_{l_n} are loop nodes.

2.) There exist some preemption points whose number is smaller than the loop bound. A preemption scenario can be written as

$$S_{j,i} = (\tau_j, \tau_i, \{p_{l_1}^{x_1}, \cdots, p_{l_n}^{x_n}, p_{k_1}, \cdots, p_{k_m}\})$$

in which at least one x_i for a preemption point p_{l_i} is *smaller* than the iteration bound lb and in which the preemption points $p_{k_1}, \cdots, p_{k_m}$ are located outside the loop.

In the first case, the insertion method is correct. In the second case, we calculate the preemption delay as follows:

$$\begin{aligned}\delta((\tau_j,\tau_i,\{p_{l_1}^{x_1},\cdots,p_{l_n}^{x_n},p_{k_1}\cdots p_{k_m}\})) &= \delta((\tau_j,\tau_i,\{p_{l_1}\}))\cdot x_1 \\ &+ \cdots \\ &+ \delta((\tau_j,\tau_i,\{p_{l_n}\}))\cdot x_n \\ &+ \delta((\tau_j,\tau_i,\{p_{k_1}\cdots p_{k_m}\})) \qquad (3.25)\end{aligned}$$

Equation 3.25 states that for each preemption point within the loop, $(p_{l_1}\cdots p_{l_n})$, the preemption delay is calculated in isolation *without* considering the preemption history. The remaining preemption points, $p_{k_1}\cdots p_{k_m}$, outside the loop can be considered as a preemption scenario again.

This simplification, of course, leads potentially to an overestimation. The experiments in 3.5.3 will show that the analysis gain for analyzing loops is very small. Often loop bounds are much larger than the number of preemptions. Already for two small nested loops with each 100 loop iterations, there need to be more than 10000 preemptions before the scenario sensitive analysis would yield tighter results. It is unlikely that such task which are as frequently preempted, are scheduled for real-time.

The author believes that an analysis considering preemption scenarios in loops would require much more effort, especially if nested loops should also be addressed, while the potential analysis gain is very limited. It would involve to unroll all loops and to analyze preemption scenarios distinguishing preemption points for different loop iterations. This would require a time-consuming analysis, but could be implemented if analysis precision of the above described simplification is not considered as sufficiently accurate. As an alternative the preemption delay could be approximated. This is described in the next section.

3.3.4 Approximation of multiple preemption delays

In this section we provide an alternative way to calculate the z-th preemption cost. If the methodology for the branch-and-bound algorithm should be too time-consuming for larger tasks, this alternative method can be used. For the following considerations we do not consider function calls.

We propose a static algorithm to consider the zth greatest preemption delays. The key idea is to distinguish between loops and linear code. For linear code the maximum number of useful cache blocks is one, i.e. the current block.

Based on the data flow analysis of section 3.3.3 we calculate the zth greatest preemption delay $\delta_{j,i}(z)$ by sorting all preemption delays in decreasing order and choos-

ing the z-th greatest one. The total preemption delay for z preemptions $\delta_{j,i}^{multi}(z)$, as used in the experimental section, is given by $\delta_{j,i}^{multi}(z) = \sum_{k=1}^{z} \delta_{j,i}(z)$.

Sequential code

For each node within sequential code the number of useful cache blocks is at most one cache block, e.g. if the cache block was loaded in an earlier basic block and used later. This simplification does not hold for programs with sub-functions.

Loops

In section 3.3.3 we pointed out the difficulty of a static analysis of preemption delays in loops. Because the iteration space cannot be modeled with a simple control flow graph we provide a conservative approximation. If the number of preemptions is larger than the loop bound, then the total number of cache blocks that can be replaced by n preemptions within a loop l_i is given by

$$(lb_i - 1) \cdot USE[B_i] + (n - lb_i + 1) \tag{3.26}$$

The maximum number of loop iterations is given by lb_i and the node with the maximum number of useful cache blocks $USE[B_i]$ is denoted by B_i. It states, that if in each iteration a preemption occurs then the cost of any *further* preemption can be considered as one, because the path between B_i from the last to the current iteration is sequential code. In sequential code, the maximum number of useful cache blocks is one (see above).

3.3.5 Revised schedulability analysis

The schedulability analysis in section 3.2.1 and 3.2.2 assumed that a preemption of a task τ_i by a task τ_j occurs at the worst case preemption point. In the last sections, we have developed an analysis to compute a bound for the total preemption delay for multiple preemptions by considering the preemption history. Refer also to figure 3.10. In this section, we extend the schedulability analyses to consider the n-th marginal preemption delay.

As shown in the previous sections, the preemption delay is given for a preemption scenario $S_{j,i}(n)$ for n preemptions. Then, we can calculate the nth marginal preemption delay $\delta_{j,i}(n)$ by using the alternative calculation of the preemption delays in equation 3.24, which results the delay for the nth preemption point.

Simplified schedulability analysis

In section 3.2.1 we have presented a simplified schedulability analysis, in which all preemptions cause a cache interference. In order to consider multiple preemption delays, the equation 3.3 is replaced by equation 3.27:

$$\Delta^{s}_{j,i}(R_i) = \sum_{k=1}^{j-1} \left(\left(\sum_{n=1}^{E_{j,k}} \delta_{j,k}(n) \right) \cdot E_{k,i} \right) \tag{3.27}$$

Instead of multiplying the preemption delay $\delta_{j,k}$ with $E_{j,k}$ times the n-th marginal preemption delay is used. The total preemption delay is tighter but still conservative, because the maximum preemption delay is always taken.

Cache interference of Indirect preemptions

In section 3.2.2 the scheduling approach had been revisited to distinguish between cache interferences and task preemptions. In order to consider multiple preemption delays, the equation 3.11 is replaced by equation 3.28 which is initialized by $C_{j,i}(i) := \{\delta_{j,i}(1), \cdots, \delta_{j,i}(E_{j,i})\}$.

$$C_{j,i}(k) = C_{j,i}(k-1) \cup \{\delta_{j,k}(1), \cdots, \delta_{j,k}(E_{j,k} \cdot E_{k,i})\} \qquad \text{for k. } i < k \leq j-1 \tag{3.28}$$

The difference is that the same worst-case preemption delay $\delta_{j,k}$ is not inserted multiple times (e.g. $E_{j,k} \cdot E_{k,i}$ times), but the nth greatest preemption delays are inserted. This modification results in a tighter but still conservative bound of the preemption delay.

3.4 Multiple task activation

This subsection focuses on multiple executions of a single task with the goal to overcome the pessimistic empty cache assumption for single task cache analysis. Intrinsic cache effects have been intensively studied, for example [8] [71] [127] [141] but always with the empty cache assumption at task activation. Current approaches for CRPD analysis (extrinsic cache analysis), for example [19], [67], [89], assume an empty cache at task activation as well. In the following we extend single task cache analysis and CRPD analysis to consider a warm cache at task activation.

3.4.1 Cache content propagation

The number of available cache blocks depends on cache behavior of a previous task activation and the cache usage of the tasks that execute between two task activations. In preliminary work [115] [111] we have partly addressed this issue but not integrated

the results in the cache-aware response time analysis. We assume the conservative approximation that *all* higher and lower priority tasks of a system execute. We model the execution of intermediate tasks $\tau_{j_1}, \cdots, \tau_{j_k}$ as a sequence of task executions because the order and frequency of task executions does not change the available cache blocks at the second task activation. This is shown in section 3.4.2.

The available cache blocks at the end of the execution of task τ_i are given by the maximum number of reaching cache blocks, denoted by $RMB^{\tau_i}[end]$ of each task. The term $RMB^{\tau_i}[end]$ is computed in section 3.3.3. The set of cache states $RMB^{\tau_i}[end]$ of task τ_i is inserted in the first node of task τ_{j_1}. In general the cache content $RMB^{\tau_{j_k}}[end]$ of task τ_{j_k} is inserted in the first node of task $\tau_{j_{k+1}}$. The last task in this chain is τ_i. Figure 3.14 shows two activations of τ_2 and an intermediate execution of a task τ_3.

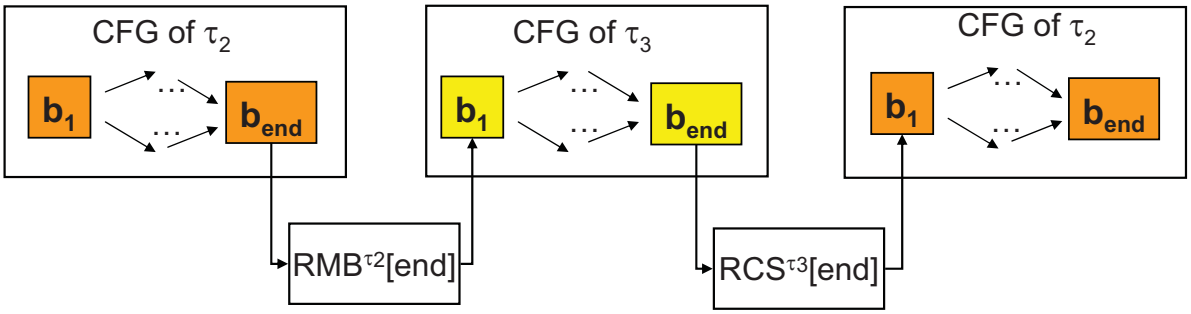

Figure 3.14. Cache content propagation for multiple task activation.

The cache state of $RMB^{\tau_2}[end]$ at the end of task τ_2 is propagated to the first node of the τ_3. Then, global data flow analysis determines the RCS states for all nodes of task τ_3. The cache state at the last node $RMB^{\tau_3}[end]$ is then propagated to the first node of τ_2. This procedure is repeated for each intermediate task.

3.4.2 Cache usage of intermediate tasks

To apply the cache content propagation we show the following lemma.

Lemma 3.2. The order and the frequency of intermediate task executions does not change the cache access behavior for the second task execution.

Proof of Lemma 3.2. We show the independence for n-way associative instruction caches in two steps.

$$CB_s^{\tau_i}(\cdots, \tau_k, \tau_k, \cdots) = CB_s^{\tau_i}(\cdots, \tau_k, \cdots) \quad (3.29)$$

$$CB_s^{\tau_i}(\cdots, \tau_k, \tau_l, \cdots) = CB_s^{\tau_i}(\cdots, \tau_l, \tau_k, \cdots) \quad (3.30)$$

Where $CB_s^{\tau_i}(\tau_{k_1}, \cdots, \tau_{k_n})$ denotes a unique upper bound of the worst case cache contents of τ_i in cache set s after the execution of tasks $\tau_i, \tau_{k_1}, \cdots, \tau_{k_n}$. Note, this set

contains *only the cache blocks of* τ_i, but not the cache blocks of $\{\tau_{k_1}, \cdots, \tau_{k_n}\}$. Equation 3.29 states that multiple executions of τ_k can be reduced to a single execution of τ_k. Equation 3.30 states that the remaining cache blocks of τ_i are independent of the execution order of intermediate tasks. Thus, any execution sequence of arbitrary order and frequency can be transformed to a canonical sequence of task executions without repetitions.

We assume an associative instruction cache with LRU replacement strategy. It suffices to show both equations for a cache set because cache sets are controlled independently. At this point we exclude complex cache architectures that install a set dependency, such as victim caches or pseudo-associative caches. A conservative data flow analysis, as presented in section 2.5.3, can be used to calculate the worst case cache behavior. Using that data flow analysis, it is not necessary to evaluate every execution path which would be exponential in time complexity.

When a cache block B_i is loaded to a full cache set, the LRU strategy replaces the least recently used cache block with B_i. If cache block B_i is already in the cache set and requested again by the CPU, then only the order of all cache blocks is changed. Suppose that a cache set holds n cache blocks. Further let K, L, and I denote the number of cache blocks of task τ_k, τ_l, and τ_i respectively. Let M be the total number of new cache blocks that are mapped to a cache set. First, τ_k then the intermediate tasks τ_k and τ_l are executed, and finally τ_i is executed again.

We consider three cases:

1 $M \geq n$. The number of new cache blocks is greater than the associativity. This means particularly that all I cache blocks of τ_i are replaced.

2 $M \leq n - I$. All new cache blocks fit in the cache set and none of the I cache blocks of task τ_i are replaced.

3 $n - I < M < n$. The number of new cache blocks is larger then the vacant positions of the cache set, but smaller than the total number of cache blocks. In this case, $I + M - n$ cache blocks of τ_i are replaced.

First, we show that equation 3.29 holds. To show that executing task τ_k once has the same effect as executing τ_k twice. We assume that in the cache are K cache blocks, e.e. those cache blocks of task τ_k. Hence we set $M = K$. For case 1. the number of remaining cache blocks of task τ_i is zero, if task τ_k is executed once or twice. For case 2. no cache blocks of τ_i are replaced after one execution of τ_k. Also during the

second execution of τ_k no cache blocks are replaced. For case 3 we have to show

$$\forall m' \in CB_s^{\tau_i}(\tau_k, \tau_k) \rightarrow m' \in CB_s^{\tau_i}(\tau_k) \tag{3.31}$$

$$\forall m' \in CB_s^{\tau_i}(\tau_k) \rightarrow m' \in CB_s^{\tau_i}(\tau_k, \tau_k) \tag{3.32}$$

Suppose $\exists m' \in CB_s^{\tau_i}(\tau_k, \tau_k)$ and $m' \notin CB_s^{\tau_i}(\tau_k)$. During execution of τ_k no cache blocks of τ_i are loaded, consequently m' cannot be in the cache set. So $m' \notin CB_s^{\tau_i}(\tau_k, \tau_k)$ which is a contradiction. For equation 3.32 suppose $m' \in CB_s^{\tau_i}(\tau_k)$ arbitrarily, which means after the execution of τ_k m' is still in the cache set. During the second execution of τ_k the access to all its cache blocks $m_{k_1}, \cdots, m_{k_K}$ are all cache hits, because $M < n$. So only a reordering according to LRU strategy can occur. Particularly no cache block of τ_i is replaced, so $m' \in CB_s^{\tau_i}(\tau_k, \tau_k)$. We have now shown that for all cases equation 3.29 holds.

Equation 3.30 holds for case 1 and 2: In this case we consider the case, that task τ_k and τ_l have loaded their cache blocks to the cache. Using the above general notation of M, we set $M = K + L$. In case 1 all cache blocks of τ_i and in case 2 no cache blocks are replaced. For case 3 assume $m' \in CB_s^{\tau_i}(\tau_k, \tau_l)$. So the least $K + L - n$ cache blocks are replaced. No cache blocks of τ_k can be replaced by τ_l since $K + L < n$. Consequently, all $K + L - n$ cache blocks are replaced from τ_i. Similarly it follows from the LRU strategy that the $L + K - n$ cache blocks being replaced during execution of τ_l and τ_k must be from τ_i. The order of the I cache blocks of τ_i cannot change during execution of τ_k and τ_l and execution of τ_l and τ_k because the I cache blocks are loaded before the execution of τ_l and τ_k. This proves equation 3.30 for case 3 and completes the proof that equations 3.29 and 3.30 are true. □

3.4.3 Revisited single task cache analysis

The effect of a multiple task execution is not only interesting for preemption delay calculation. In traditional WCET-analysis for cache behavior, an empty cache is assumed. For example the intrinsic cache analysis by [141], as shortly reviewed in section 2.5.3.

As the framework to consider cache blocks from a previous task activation is available, this intrinsic cache analysis can be modified as well. In order to consider a non-empty cache content, the first node in the control flow graph is modified by inserting the cache lines, which are available from the previous task activation. In the next section we provide experimental results, that this improvement yields to tighter analysis results.

3.5 Experiments

This section provides experiments for the presented cache-aware response time analysis framework. First, we describe the experimental setup in section 3.5.1. Then, we provide results for preemption delays for single preemptions for different cache configurations and benchmarks in section 3.5.2. Preemption delays for multiple preemptions is provided in section 3.5.3. Results for the analysis of multiple task activations is presented in section 3.5.4 and the experiments for the cache-aware response time analysis are presented in section 3.5.5.

3.5.1 Setup

Id	Mem	C-Ln	WCET	Type	Description
τ_1	316	83	0.432	sequential	linear
τ_2	2864	260	7.561	sequential	statemate
τ_3	888	180	115.7	loop	fast Fourier transform
τ_4	296	275	489.6	loop	packet receiver
τ_5	1023	286	1078	loop	whetstone

Table 3.3. Benchmark Description with Memory Usage[B], c-lines and WCET[$10^3 clk$] for 1KB direct mapped instruction cache and code type.

Table 3.3 states for each benchmark the memory usage, number of c lines, the worst case execution time (WCET) for a 1KB direct mapped instruction cache and whether the task contains loops or not (e.g. sequential code). The benchmarks are mainly taken from [23] [67]. The WCET was determined by a cycle accurate ARM9 processor simulator of the ARM developer suite (ADS) [6] [141]. Cache access time was determined by cache analysis with Symta/P assuming a 20 cycles cache miss penalty, a cache block size of 8 byte and a fixed instruction length of 32 bit. For the CRPD analysis we generated the control flow graph from source code with Symta/P and mapped the memory lines of assembly instructions to the corresponding nodes by disassembling the binary with `fromelf`, a tool provided by ADS.

The results of the experiments can be generalized using a cache footprint, which describes the relation of memory usage of an application and the cache size. The cache footprint is defined as the average number of tasks that use a single cache block. Table 3.4 presents footprint for several preemption scenarios and for all tasks of table 3.3. For example, both tasks τ_3 and τ_6 occupy the entire 256B cache, hence, the footprint is 2.

PrS	256B	1 kB	4 kB	16 kB
τ_3 τ_6	2.00	1.98	0.73	0.18
τ_6 τ_5	2.00	2.0	0.98	0.25
τ_5 τ_3	2.00	1.97	1.14	0.28
τ_4 τ_5	1.53	1.13	0.73	0.18
all	4.5	2.69	0.83	0.26

Table 3.4. Cache footprint of preemption scenarios (PrS) and of all tasks in table 3.3 for direct mapped instruction cache.

3.5.2 Single preemption delays

The preemption delay for different tasks are given in table 3.5 for 256B, 1kB, and 4kB direct mapped instruction cache. A preemption scenario (PrS) is a pair of the preempting and preempted task. For each PrS (example: τ_3 τ_6) and cache size, the number of used cache blocks (US) of the preempting task (τ_3), the number of useful cache blocks (UF) of the preempted task (τ_6) and the cache related preemption delay when τ_3 preempts τ_6 (D). The PrS are organized by the code structure type (loop(l), sequential (s)).

PrS	Type	256B			1 kB			4 kB		
		US	UF	PD	US	UF	PD	US	UF	PD
τ_3 τ_6	l-l	32	32	32	125	128	125	228	129	1
τ_6 τ_7	l-l	32	17	17	128	66	66	147	66	1
τ_6 τ_4	l - s	32	1	1	128	1	1	147	1	1
τ_6 τ_5	l - s	32	1	1	128	1	1	147	1	1
τ_5 τ_6	s - l	32	32	32	128	128	128	356	129	129
τ_5 τ_3	s - l	32	32	32	128	79	79	356	79	61
τ_4 τ_5	s - s	17	1	1	17	1	1	17	1	1

Table 3.5. Preemption delay for several cache sizes for direct mapped cache, used cache blocks of preempting task (US), useful cache blocks of preempted task (UF), and preemption delay (PD)

With increasing cache size the number of used and useful cache blocks increases until the cache is large enough to hold all cache blocks. Then, the cache usage is stable. For smaller caches the CRPD is the minimum of used and useful cache blocks. If a linear code is preempted the preemption delay is always one. For larger caches the CRPD is much smaller than used and useful number of cache blocks which reflects

the fact that the preempting and preempted task share only a small part of the cache which is also represented by the cache footprint (refer to table 3.4).

The influence of the associativity on CRPD is shown in table 3.6. Only the PrS with loops are considered, because a preemption delay for linear tasks is at most one. With increasing associativity the number of used and useful cache blocks increases. The reason is that fewer cache sets that are available and a conservative analysis has to consider the worst case execution path.

PrS	1-way			4-way			16-way		
	US	UF	PD	US	UF	PD	US	UF	PD
τ_3 τ_6	228	129	1	228	138	126	512	512	512
τ_6 τ_7	147	66	1	147	66	66	512	66	66
τ_5 τ_6	356	129	129	356	131	130	356	512	356
τ_5 τ_3	356	79	61	356	138	138	356	507	356

Table 3.6. Preemption delay for associative caches with 4 kB. used cache blocks of preempting task (US), useful cache blocks of preempted task (UF), and preemption delay (PD)

3.5.3 Multiple preemption delays

The effect of multiple preemptions as described in section 3.3 is presented in table 3.7. Again we consider only loop-tasks, because for linear tasks the preemption delay is at most one cache block. The table shows the total delay for different numbers of preemptions for a 1kB and 4kB direct mapped instruction cache as the ratio $\frac{\delta_{j,i}^{multi}}{n \cdot \gamma j,i^{max}}$, where $\delta_{j,i}$ is defined in section 3.3. A ratio of 1.00 states that the consideration of multiple preemption delay computed the same result. The ratio of 1.00 for the 4kB cache in PrS (τ_3 τ_6) and (τ_6 τ_7) is because the total preemption delay is in these cases one cache block, which is already optimal. The results show that if a task is preempted 100 or 1000 times the proposed consideration of different preemption delays shows a significant improvement (for PrS (τ_5 τ_3) the improvement is 97%). The proposed analysis can reduce the preemption delay by an order of magnitude for tasks that are preempted very frequently.

3.5.4 Multiple task activation for single task execution

The effect of multiple task executions, as described in section 3.4.3 on single task cache behavior is shown in table 3.8. We consider three assumptions: an empty cache at task activation (*empty*), all tasks of the table 3.3 execute between two activations (*warm*), no task executes between two task activations (*lock*). The term *lock* denotes

PrS	1 kB				4 kB			
	1	10	100	1000	1	10	100	1000
τ_3 τ_6	1.00	1.00	0.70	0.08	1.00	1.00	1.00	1.00
τ_6 τ_7	1.00	1.00	1.00	0.56	1.00	1.00	1.00	1.00
τ_5 τ_6	1.00	1.00	0.70	0.08	1.00	1.00	0.70	0.08
τ_5 τ_3	1.00	1.00	0.93	0.91	1.00	1.00	0.17	0.03

Table 3.7. Multiple preemption delay ratio for 1kB and 4kB direct mapped instruction cache.

the fact that all cache blocks available in cache at the end of the first activation are not removed by other tasks. The term *lock* is motivated by locking the cache lines.

Task	linear			statemate			fft		
	empty	warm	lock	empty	warm	lock	empty	warm	lock
256B	332	332	47	6861	6861	6861	173529	173529	173529
1kB	332	332	47	6861	6861	6861	65742	65742	65343
4kB	332	332	47	6861	5056	648	65742	65248	62949
16kB	332	47	47	6861	3441	648	65742	64127	62949

Table 3.8. Cache access time[clk] for multiple task activations for an empty cache at task activation (empty), assuming that all tasks of the system execute between two activations (warm), and assuming that no task executes between two task activations (lock)

The results show a significant reduction of cache access time to 14% and 9% for locking cache blocks for `linear` and `statemate`, respectively, which are benchmarks with sequential code. MATLAB/Simulink generated code which is used in automotive control applications often has this property. For the conservative assumption that all tasks of the system execute between two task activations the cache access time for 16kB cache is reduced to 14% and 50% respectively. For the loop-task `fft` control structures within loops lead to several paths within a loop. For a worst case analysis we cannot assume that the worst case path during the second activation had been executed during the first activation too. For tasks consisting of loops with common code (e.g. without branches) the improvement of analysis precision would be significant. All memory blocks that are on a common path will be detected by the analysis. For example, if a loop contains a long sequential section followed by some if-then-else statement, then all memory blocks from the sequential section will be detected and forwarded to the next task activation.

3.5.5 Cache-aware schedulability analysis

Finally, we integrate the CRPD analysis into response time analysis, as described in section 3.2. We compare the response time to the approaches by [92] and [18]. In

Petter's approach only the preempted task is considered, however, the actual number of useful cache blocks is not analyzed but assumed to a constant proportion of the total memory usage (e.g. 30%). For a fair comparison of the response time results of Petter's algorithm and our algorithm, we apply our data flow analysis to compute the useful cache blocks. The approach by Busquets is based on the number of used cache blocks of the preempting task. However, in the experiments they assume that the entire cache is flushed. For a comparison we apply Busquets' algorithm with the actual number of used cache blocks which was determined by data flow analysis. A comparison to the more sophisticated scheduling analysis of [67] was out of scope, because the algorithm needs best case execution times which were not available. Figure 3.15 shows the total response time of the task system specified in table 3.3

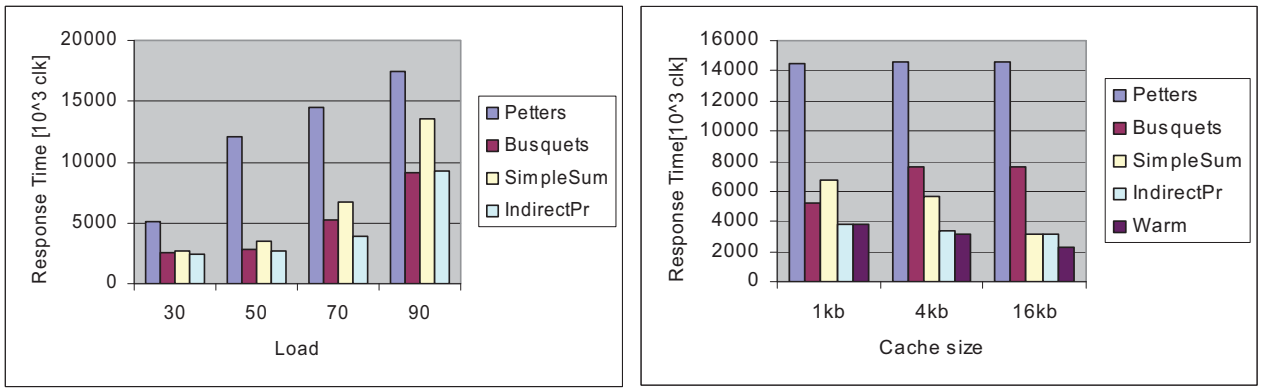

Figure 3.15. Response time of lowest priority task (τ_5) for system setup of table 3.3.

for *Petter*'s approach, *Busquets* approach, the simple summation model *SimpleSum* of section 3.2.1 and the analysis of indirect preemptions *IndirectPr* of section 3.2.2. The periods have been chosen to create different system loads for a 1kB direct mapped instruction cache. The approach *SimpleSum* delivers much shorter response times than *Petters* but longer response times than *Busquets* because many separate CRPD terms are considered. Compared to *Busquets*, the response time by the approach that considers indirect preemptions (*IndirectPr*), is about 30% smaller for 70% system load and about the same for the other system loads.

In a second experiment, shown in figure 3.15, we consider a warm cache at task activation for several cache sizes assuming that all tasks of the system execute between two task activations with a system load of 70%. The ratio of *Warm* and *Petters* ranges from 0.27 till 0.16 and the ratio of *Warm* and *Busquets* from 0.73 till 0.30 for the 1kB and 16kB cache, respectively. These results show a significant reduction of response time by 70% compared to the best previous approach for 16kB warm cache.

The analysis time of the cache-aware schedulability algorithm is evaluated with a larger set of hypothetical tasks. These tasks are characterized by the period P_i , the worst-case execution time C_i, number of useful and used cache blocks, shown in table 3.9.

Task ID	P_i	C_i
1,2,3,4	15	0.1
5,6,7,8	100	4
9,10,11,12	1000	20
13,14,15,16	10000	200
17,18,19,20	100000	2000

Table 3.9. Task setup. Period P_i, WCET C_i in 10^3 clk.

We assume that 20% of the WCET is due to core execution time and 80% is due to memory accesses. With these assumptions we set the parameters $used(\tau_i) = \frac{4}{5}\frac{C_i}{CMP}$ and $useful(\tau_i) = 0.3 \cdot used(\tau_i)$. The total response time is partitioned into response time t_{resp} and total preemption delay t_{crpd}. The core execution time of a task includes the worst-case execution time C_i and the execution time of higher priority tasks.

Table 3.10 shows the response time and analysis time for a set of 5, 10 and 20 tasks for a 8KB instruction cache with 512 cache blocks and a cache miss penalty (CMP) of 10 clock cycles (clk).

Approach	$n = 5$			$n = 10$			$n = 20$		
	t_{resp}	t_{crpd}	t_{ana}	t_{resp}	t_{crpd}	t_{ana}	t_{resp}	t_{crpd}	t_{ana}
Petters	2390	327	0.14	5382	1690	0.50	24495	22269	5.61
Busquets	2377	114	0.12	5172	502	0.28	14268	3008	1.02
SimpleSum	2376	40	0.11	5153	326	0.12	14857	4672	0.94
IndirectPr	2376	38	0.14	5144	244	0.38	14040	2449	5.76

Table 3.10. Response time t_{resp}, preemption delay t_{crpd} in 10^3 clk and analysis time t_{ana} in seconds The number of tasks is given by n.

The results show, that the analysis time is very small (all schedulability analyses finish within seconds), while the precision of preemption delay is improved by an order of magnitude compared to Petters and about 20% compared to Busquets approach for the set of 20 tasks.

3.6 Conclusion

In this section we have proposed a novel cache-aware response time analysis including cache related preemption delay for fixed priority preemptive scheduling for

direct mapped and associative instruction caches. The time-complexity is $O(n \cdot E_{j,i} \cdot \log(n \cdot E_{j,i})$, in which in denotes the number of tasks and $E_{j,i}$ the number of task activation of preempting task τ_j during the response time of task τ_i. Then, we have proposed an analysis for multiple preemptions. The preemption history was considered by determining the worst case preemption scenario with a branch and bound algorithm and computing the preemption delay for multiple preemptions with a modified data flow analysis. Finally, we have considered cache blocks from a previous task activation. We have shown that the number and order of intermediate tasks is irrelevant to the cache behavior at the second task activation. In several experiments we have demonstrated the applicability of theses new analysis algorithms which show a significant improvement over previous conservative analysis approaches.

Chapter 4

SCALABLE PRECISION CACHE ANALYSIS

During the design process of an embedded system, engineers focus on different levels of abstraction and accuracy of overall system performance. In the design space exploration phase, rough estimates of the system performance are sufficient, while more accurate estimates are necessary in the last steps of system integration. One part of the response time of a task is the cache-related preemption delay which had been considered in schedulability analysis in chapter 3.

The approach by Lee et al. [66] and the approach by Mitra et al. [89] are two recent approaches to calculate the cache-related preemption delay which consider both: the preempted as well as the preempting task. While Lee et al.'s approach is very time-efficient but overestimates the preemption delay, the approach by Mitra et al. is more precise but requires a higher time-complexity. because (possibly) exponential number of cache states are stored at each node. Both approaches have been discussed in section 2.5.1 and 2.5.2 in detail.

Even in the last phase of system integration, an analysis approach with a high time-complexity might not be acceptable because of an ever decreasing time-to-market window. Then, the only alternative would be the less precise Lee et al.'s approach or approaches that consider only one task, such as [130] [92] [18]. The resulting overestimated response times could lead to an over-dimensioned and more expensive system design.

In this section we provide a scalable precision cache analysis where the precision scales with the time-complexity. The data flow technique uses the number of cache states that are stored at each node as scaling parameter. Results from experiments

shows a minimum loss of precision at significant computation time reduction. Parts of this chapter has been previously published in [116] and [117].

This section is structured as follows. A motivating example is given in section 4.1. Then, we describe a new cache model in section 4.2. We present the scalable precision cache analysis first for direct mapped instruction caches in section 4.3, and then for associative instruction caches in section 4.4. In section 4.5, the overall framework for real-time analysis is described. Experiments are presented in section 4.6 before we summarize and conclude in section 4.7.

4.1 Motivational example

We motivate our scalable precision analysis with a comparison of Lee's and Mitra's approach for the *RCS* calculation.

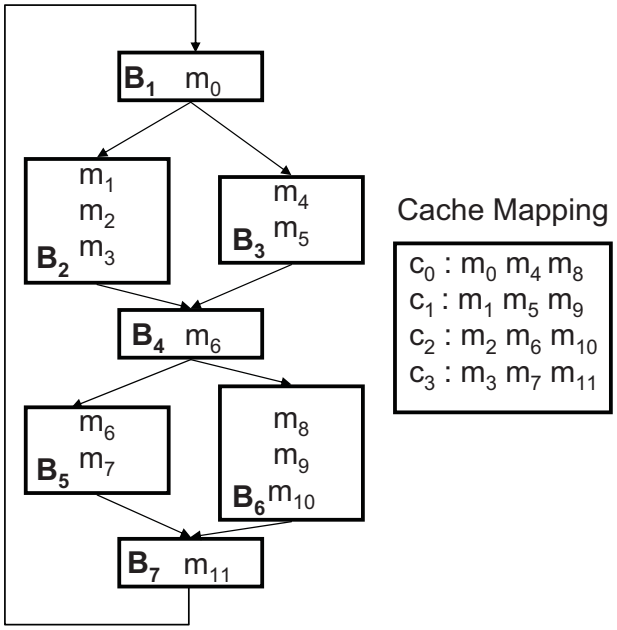

Figure 4.1. Control flow graph with memory blocks and cache mapping for direct mapped instruction cache with 4 cache blocks.

A task is represented by its control flow graph (CFG) where nodes represent basic blocks and edges specify the control flow between basic blocks. An example control flow graph is shown in figure 4.1. It shows a a loop-statement with two `if-then-else` statements. A node B_i lists the memory blocks that correspond to the assembly instructions of basic block B_i. For example, the memory blocks m_1, m_2 and m_3 are loaded to the cache during execution of basic block B_2. For this example, we assume a direct mapped cache with 4 cache sets. The mapping of memory blocks to cache sets is also given in figure 4.1.

4.1.1 Set-based approach by Lee

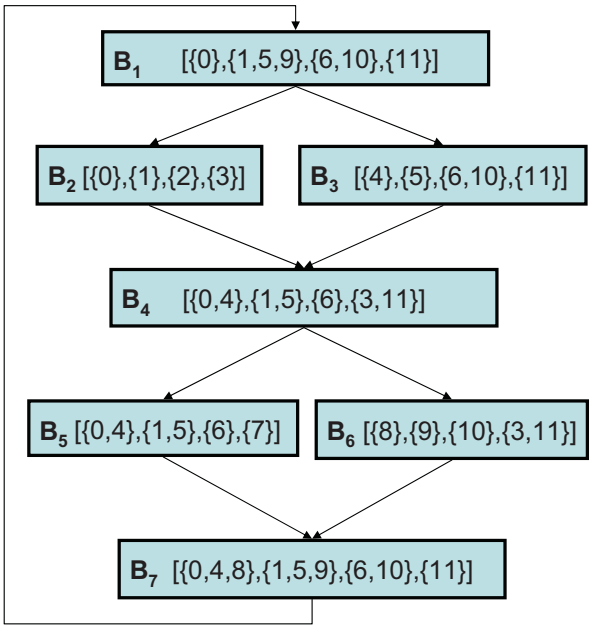

Figure 4.2. RMB calculation by Lee.

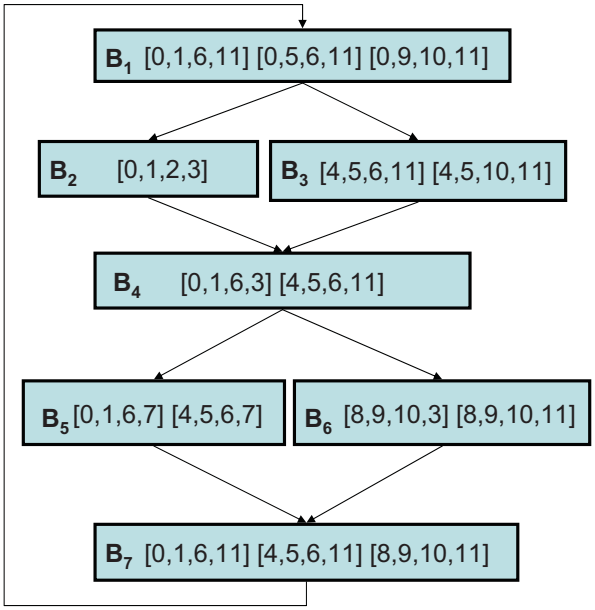

Figure 4.3. RCS calculation by Mitra.

The approach by Lee uses a *set of memory blocks* to store multiple memory blocks in a cache set. Figure 4.2 shows the *RMB* sets after the data flow analysis has converged. Table 2.1 summarized the *gen* sets for the example. To abbreviate the notation we use only the index of memory blocks. The notation of the cache state at B_4 $[\{0,4\},\{1,5\}\{6\},\{3,11\}]$ represents a cache in which m_0 and m_4 are available in cache set c_0, memory blocks m_1 and m_5 are available in cache set c_1, m_6 is available in cache set c_2, and m_3 and m_{11} are available in c_3. In the data flow analysis, the contents of each cache set is propagated via the edges and is merged for each cache set.

The data flow algorithm is demonstrated on the example flow graph of figure 4.1. Initially all $RCS^c_{in}[B]$ are empty and $RCS^c_{out}[B]$ are initialized with the $gen^c[B]$ sets. The results are summarized in table 2.2 for the first and second iteration. Since the RMB sets of the third iteration are the same as in the second iteration, they are omitted.

We explain the calculation of *RCS* of equations 2.14- 2.16 for the second iteration at basic block B_4. In this case the RCS_{out} of B_2 and B_3 are merged and cache set c_2 is replaced with $gen^{c_2}[B_4]$:

$$\begin{aligned} RCS_{in}[B_4] &= [\{0\},\{1\},\{2\},\{3\}] \cup [\{4\},\{5\},\{6,10\},\{11\}] \\ &= [\{0,4\},\{1,5\},\{2,6,10\},\{3,11\}] \\ RCS_{out}[B_4] &= [\{0,4\},\{1,5\},\{\mathbf{6}\},\{3,11\}] \end{aligned}$$

4.1.2 State-based approach by Mitra

As an alternative, the approach by Mitra [89] uses several *cache states* when more memory blocks are available in a cache set. The output for the *RCS* calculation is shown in table 2.5 and is graphically represented for the second iteration in figure 4.3.

A cache state consists of cache sets, that are either empty ($\perp$) or contain a single memory block m_i. The data flow analysis is, again, described at basic block B_4: The contents for all cache sets is given in the following:

$$\begin{aligned} RCS_{in}[B_4] &= \{[0,1,2,3],[4,5,6,11],[4,5,10,11]\} \\ RCS_{out}[B_4] &= \{[0,1,\mathbf{6},3],[4,5,\mathbf{6},11]\} \end{aligned}$$

Memory block m_6 is mapped to cache set c_2. Note, that the number of cache states reduces because the duplicated cache states are removed ($[4,5,6,11]$). Since $gen[B_4] = [\{\},\{\},\{6\},\{\}]$, the third cache set, containing m_2, m_6, m_{10} is replaced by m_6. In this approach, the cache states are duplicated if some cache sets are not equal resulting in an increased number of cache states. For example, node B_5 and B_6 contain two cache states, but basic block B_7 has three cache states. This increase of cache states scales exponentially with the number of branches in a task. This higher time-complexity comes with the gain of a higher analysis precision.

4.1.3 Comparison and discussion

To directly compare the approaches we calculate the useful cache blocks $USE[B_4]$ for basic block B_4. These results were taken from the tables 2.2, and 2.5. The computation of LCS has not been shown because of space requirements, but the computation is analogous to the RCS calculation. The set of useful cache blocks calculated by Lee's approach $USE_{lee}[B_4]$ is given by:

$$\begin{aligned} RMB_{out}[B_4] &= [\{0,4\},\{1,5\},\{6\},\{3,11\}] \\ LMB_{out}[B_4] &= [\{0,8\},\{1,5,9\}\{6,10\},\{7,11\}] \\ USE_{lee}[B_4] &= RMB_{out}[B_4] \cap LMB_{out}[B_4] = [\{0\},\{1,5\},\{6\},\{11\}] \end{aligned} \quad (4.1)$$

Assuming that all useful cache blocks are used by the preempting task, the total preemption delay would be 4 cache blocks. The set of useful cache blocks calculated by

Mitra's approach $USE_{mitra}[B_4]$ is given by:

$$\begin{aligned} RCS_{out}[B_4] &= \{[0,1,6,3],[4,5,6,11]\} \\ LCS_{out}[B_4] &= \{[8,9,10,11][0,1,6,7][0,5,6,7]\} \\ USE_{lee}[B_4] &= \max(RMB_{out}[B_4] \cap LMB_{out}[B_4]) \\ &= [0,1,6,3] \cap [0,1,6,7] = [0,1,6,\perp] \end{aligned} \tag{4.2}$$

Assuming that all useful cache blocks are removed by the preempting task, the total cache-related preemption delay would be three cache blocks. This is one cache block less, or 25% in relative terms, than in Lee's approach. The number of useful cache blocks are shown in table 4.1 for all basic blocks. In most cases the number of useful cache blocks computed by state-based approach is smaller than the set-based approach. The reason for a higher precision is the greater number of cache states that captures execution path information.

Basic block	USE_{mitra}	USE_{lee}
B_1	**3**	4
B_2	2	2
B_3	**2**	3
B_4	**3**	4
B_5	**2**	3
B_6	1	1
B_7	**3**	4

Table 4.1. Comparison of useful cache blocks of state-based approach by Mitra USE_{mitra} and of set-based approach by Lee USE_{lee}.

To conclude, Mitra performs the analysis over a richer domain of cache states, while Lee uses a set-based representation. The analysis precision of Mitra can be higher than the precision by Lee, while the computation time is higher compared to Lee. The question arises, whether the analysis precision of the state-based approach could be accomplished with a fewer number of states? If not the same analysis precision can be reached, then how does the precision scale with the time-complexity? How many cache states would be necessary for sufficiently accurate results?

To answer these questions, we propose an scalable precision cache analysis that limits the number of cache states at each node. Whenever the number of cache states is larger then a given bound, cache states are merged. This technique bounds the time-complexity because the number of cache states is bounded while possibly reducing

the analysis precision. A new cache model which is necessary to provide a sufficiently general data structure is described in the next section.

4.2 Scalable precision cache model

In the following we define a cache state c and a cache set c_i that are used in the scalable precision cache analysis for direct mapped and associative instruction caches.

Definition. A cache state c is defined as a vector of cache sets c_i: $c = [c_1, \cdots, c_S]$ where S denotes the total number of cache sets.

Definition. A cache set c_i is defined as a vector of sets of memory blocks M: $c_i = [M, \cdots, M]$ The length of the vector is given by the associativity n of the cache. The cache set $c_i[1]$ for a direct mapped cache at execution point p contains the memory blocks $m_1, \cdots, m_k$ if these memory blocks may have been mapped to c_i at execution point p: $c_i[1] = \{m_1, \cdots m_k\}$, otherwise $c_i = \emptyset$. Analogously the nth element of cache set $c_i[n]$ for associative caches is defined as containing the nth most recently used cache blocks.

For a direct mapped cache, the vector c_i has exactly one element, because there is only one cache set to which a memory block can be mapped: $c_i = [M]$. For example, a cache state of a direct mapped cache with four sets is defined as:

$$c = [[M], [M], [M], [M]] \tag{4.3}$$

As a second example, a cache state for a 2-way associative cache with four sets is defined as

$$c = [[M,M], [M,M], [M,M], [M,M]] \tag{4.4}$$

The usage of all memory blocks M for each cache set can be refined for complexity considerations because not every memory block can be mapped to every cache set. Cache set c_i contains only those memory blocks that map to this cache set, e.g. $c_i = [M_i, M_i, M_i, M_i]$ in which M_i denotes the set of memory blocks that map to cache set c_i. This definition allows us to tighten time- and space complexity considerations because the number of elements of M_i is given by $|M_i| = \frac{|M|}{S}$ (S denotes the total number of cache sets).

4.3 Preemption delay analysis for direct mapped caches

Data flow techniques to bound the total preemption delay have been proposed by the state-based approach by Mitra et al. [89] and the set-based approach by Lee et al. [66]. While the approach by Lee simplifies the notion of a cache contents by

using sets which leads to a low time-complexity; the approach by Mitra distinguishes between different execution paths by using cache states which potentially leads to a large number of cache states at each basic block. Our approach combines the strengths of both approaches by limiting the number of cache states at each basic block, thereby scaling the time-complexity as well as the analysis precision. Whenever the threshold of the maximum number of cache states is exceeded, cache states are merged reducing the time-complexity but also reducing the accuracy of cache content prediction. We start by describing the scalable data flow analysis.

4.3.1 Scalable data flow analysis

Based on the general data flow analysis, we present the new scalable precision cache analysis.

Computing reaching cache states

We adopt the presentation of the iterative data flow algorithms from Mitra For the RCS property, we define $RCS_{in}[B]$ and $RCS_{out}[B]$ as the cache state of reaching cache blocks just before and just after the execution of basic block B. After the fixed point is reached we set $RCS[B] = RCS_{out}[B]$. Initially

$$RCS_{in}[B] = \emptyset \quad RCS_{out}[B] = gen[B] \tag{4.5}$$

For each basic block B, the $gen[B]$ is defined as a vector with n elements $gen[B] = [[M_0], \cdots, [M_{n-1}]]$ where $M_i = \{m\}$ if m is the *last* memory block in B that maps to cache block i and $\emptyset$ if no memory block in B maps to cache block i. The number of cache blocks which is equal to the number of cache sets for direct mapped caches, is given by n. Thus, $gen[B]$ represents all the memory blocks that are available in the cache at the end of the execution of basic block B. The iterative equation 2.14-2.16 are modified as follows:

$$RCS_{in}[B] = bound_Z \left(\bigcup_{p \in pred(B)} RCS_{out}[p] \right) \tag{4.6}$$

$$RCS_{out}[B] = \{r \odot gen[B] | r \in RCS_{in}[B]\} \tag{4.7}$$

$$c \odot c' = \begin{cases} c' & if \quad c' \neq \emptyset \\ c & otherwise \end{cases} \tag{4.8}$$

For the proposed cache model, $c \odot c'$ denotes a binary operation on memory blocks M and is applied for each cache set. The cache set cs_i is represented by a vector with a single element: $cs_i = [M]$ thus $c = cs_i[1]$ and $c' = cs'_i[1]$. If the $gen[B]$ set is not empty,

then the result is $gen[B]$, otherwise the set of memory blocks is replaced by $RCS_{in}[B]$. The function $bound_Z(C)$ reduces the number of total cache states of C to Z elements, where C is a set of cache states. Its implementation is described in section 4.3.2

Computing live cache states

Similarly the LCS property is computed by an iterative fixed point algorithm, the only difference is that the $LCS_{out}[B]$ is defined in terms of $LCS_{in}[B]$ of all successors of basic block B: Initially,

$$LCS_{out}[B] = \emptyset \quad LCS_{in}[B] = gen[B] \tag{4.9}$$

For each basic block B, $gen[B]$ is defined as $gen[B] = [[M_0], \cdots, [M_{n-1}]]$ where $M_i = \{m\}$ if m is the *first* memory block in B that maps to cache block i and $\emptyset$ if no memory block in B maps to cache block i. The iterative equations are:

$$LCS_{out}[B] = bound_Z\left(\bigcup_{s \in succ(B)} LCS_{in}[s]\right) \tag{4.10}$$

$$LCS_{in}[B] = \{l \odot gen[B] | l \in LCS_{out}[B]\} \tag{4.11}$$

The operation $\odot$ is defined as in the computation of RCS_B.

4.3.2 Bounding number of cache states

The number of cache states is bounded with a function $bound_Z$. The function $bound_Z(C)$ reduces the number of states of set C to Z elements, if $|C| > Z$ otherwise $bound_Z(C) = C$. The idea is to merge those cache states which are almost equal. We formalize this idea by using a distance metric over cache states, that characterizes the number of different elements of two cache states. Then, we repeatedly choose two elements with minimum distance and merge them until the total number of elements in C is equal to Z. The objective is to reduce the number of cache states while remaining a many different states as possible.

The implementation of the $bound_Z$ algorithm is shown in figure 4.4. In line 2 two elements $c_i, c_j \in C$ with the minimum distance $min\{d(c_k, c_l) | c_k, c_l \in C\}$, are chosen. In line 3 these elements are removed from C. In line 4 the merged cache state $c_i \cup c_j$ is inserted to C. Therefore, the number of elements of C decreases by one in each iteration and, thus, the algorithm always terminates.

```
// input: set of cache states C
// output: set of cache states C with |C| ≤ Z
void bound(Z, C) {
1     while (|C| > Z) do {
2         choose (c_i, c_j) with d(c_i, c_j) minimal, c_k ∈ C
3         C = C \ {c_i} \ {c_j}
4         C = C ∪ {c_i ⊙ c_j}
5     }
6 }
```

Figure 4.4. $bound_Z(C)$ algorithm.

Distance metric

The function $d(a,b)$ of two cache states a,b is defined as a metric that delivers the difference of two cache states. Several metrics are possible. We present a simple metric d_1 and a more complex metric d_2. The simple metric d_1 only counts the number of different cache sets ignoring how many elements in each set are different. This is shown in equation 4.12 where S denotes the total number of cache sets.

$$d_1(a,b) = \sum_{k=1}^{S} \begin{cases} 1 & if \quad a_k[1] \neq b_k[1] \\ 0 & otherwise \end{cases} \tag{4.12}$$

The variable $a_k[1]$ denotes the set of memory addresses of cache set c_k. The scalable precision cache model for a direct mapped cache represents each cache set as a vector with one set (refer to section 4.2), thus we have to use the first element of the vector. The time-complexity of the calculation of d_1 is

$$O(d_1) = O(S \cdot X) \tag{4.13}$$

where X denotes the maximum number of elements of the set $a_k[1]$. Equality test of two sets can be computed in linear time. We use the O notation to express the time-complexity [26]. A bound for the maximum number of memory blocks X is discussed in section 4.3.4.

As an alternative, a more sophisticated metric d_2 is proposed to count the number of different memory blocks of each cache set with the symmetric difference. The definition of d_2 is given in equation 4.14.

$$d_2(a,b) = \sum_{k=1}^{S} |(a_k[1] \cup b_k[1]) \setminus (a_k[1] \cap b_k[1])| \tag{4.14}$$

The complexity of d_2 is given by taking S times the union of two cache sets with X elements; by taking the intersection S times of two sets with at most X elements each and by taking the set difference of a set with $2X$ elements and X elements. We bound the time-complexity for the intersection of two sets with at most X elements by $O(X^2)$. Efficient implementations with $O(X \log X)$ also exist.

$$O(d_2) = O(S \cdot (t_{\cup}(X,X) + t_{\cap}(X,X) + t_{setdiff}(2X,X)) = O(S \cdot X^2) \tag{4.15}$$

Cache state selection

There are several selection strategies to choose candidates of cache states of C. One strategy, Sel_1, is to maintain only one merged set, such that the metric is computed between pure cache states (singleton sets) and one cache state that might contain sets with more than one element. This favors the idea to maintain as many pure cache states as possible. The complexity of the selection function Sel_1 is proportional to the number of cache states $O(Sel_1) = O(|C|)$.

A second selection strategy, Sel_2, is to choose from *all* cache states of C. This metric Sel_2 requires to compare all pairs of cache states leading to quadratic complexity $O(Sel_2) = O(|C|^2)$. The metric Sel_2 is expected to yield more accurate results than Sel_1 because more elements are considered. An example for applying these metrics is given in section 4.3.5.

Merging of cache states

The merge-operation $\odot$ for two cache states is used in the algorithm $bound_Z$ in figure 4.4 as well as in the iterative data flow algorithm for RCS and LCS calculation in equation 4.8.

We define the operation $\odot$ over M (M denotes the set of all memory blocks). It can also be used for cache states by applying it pointwise to its elements. Two cache states a and b with S cache sets can be merged by applying the union operator for sets to each element:

$$a \odot b = ((a_1[1] \cup b_1[1]), \cdots, (a_S[1] \cup a_S[1])) \tag{4.16}$$

The time complexity of the merge-operation $\odot$ scales with the number of elements in each cache set X and the number of cache sets S of the cache:

$$O(a \odot b) = O(S \cdot (t_{\cup}(X,X)) = O(S \cdot X^2) \tag{4.17}$$

Preemption delay calculation

The cache related preemption delay is computed by the intersection of useful cache blocks of the preempted task τ and the used cache blocks of the preempting task τ'. The same methodology of the approach by Mitra et al. can be applied. For a complete treatment we specify the set of useful cache blocks:

$$USE^{\tau}_{scale}[B_i] = RCS^{\tau}_{out}[B_i] \cap LCB^{\tau}_{out}[B_i] \tag{4.18}$$

Finally, the preemption delay $crpd^{\tau\tau'}_{scale}$ is given in equation 4.19 in which B_{end} denotes the last basic block of a task.

$$crpd^{\tau\tau'}_{scale} = \max\{USE^{\tau}_{scale}[B_i] \cap RCS^{\tau'}_{out}[B_{end}] \quad | \quad \forall B_i \in \text{task } \tau\} \tag{4.19}$$

4.3.3 Time complexity

The time complexity of the $bound_Z$ algorithm in figure 4.4 is determined by the following steps.

1 complexity for counting number of cache states (line 1)

2 complexity for choosing c_i, c_j with $d(c_i, c_j)$ minimal (line 2)

3 complexity for removing and inserting a new cache state to C (line 3,4)

4 complexity for merging two cache states c_i and c_j (line 4)

5 number of iterations of while-loop (line 1-5)

Complexity for each step

First, we describe the time-complexity for each step within the and a while-loop, then we calculate the maximum number of while-loop iterations. All values of the time-complexities are summarized in table 4.2.

Step 1 can be implemented in linear time relative to the number of elements of C. In the RCS (LCS) algorithm, as shown in equation 4.6 (equation 4.10), $|C|$ is bounded by the number of predecessor (successor) nodes. The maximum number of predecessor nodes can be bounded by 2, because `if-then-else` statements and loop-constructs like `for, while` create at most two branches. A `switch-case` statement, even though potentially many branches are created, can be represented by multiple `if-then-else` statements. Therefore, we can bound $|C| < 2Z$ because the RCS (LCS) each predecessor (successor) node has at most Z cache states.

Step 2 is determined by the time-complexity of the selection algorithm Sel_i multiplied with the time-complexity of the distance metric d_i plus finding the minimum

element. The time-complexity for each algorithm is summarized in table 4.2. Then, the smallest element can be found in $O(2Z)$ or $O((2Z)^2)$ depending on the selection method (Sel_1 or Sel_2).

Step	Operation	Complexity
1	count	$O(2Z)$
2	Sel_1	$O(2Z)$
2	Sel_2	$O((2Z)^2)$
2	d_1	$O(S \cdot X)$
2	d_2	$O(S \cdot X^2)$
3	insert	$O(2Z)$
3	remove	$O(2Z)$
4	union	$O(S \cdot X^2))$
5	loop bound	Z

Table 4.2. Complexity of analysis steps for $bound_Z$ algorithm for direct mapped instruction cache. Z denotes the maximum number of cache states at a node, X denotes the maximum number of memory blocks of a cache set, S denotes the number of cache sets in a cache.

Step 3 is determined by inserting and removing an element from a set. This can be done in linear time, assuming that there are at most $2Z$ elements in set C we get: $O(2Z)$. Step 4 is determined by the complexity for taking the union of each cache set with X elements: $O(S \cdot X^2))$.

Finally, we bound the maximum number of while-loop iterations. Since $|C| < 2Z$ there are at most Z loop iterations, because in each iteration the number of elements in C decreases by one element and the loop terminates when $|C| \leq Z$.

Overall time complexity

In summary, the complexity of the entire $bound_Z$ algorithm is given by multiplying the sum of counting number of elements, choosing the element with the minimum distance, removing and inserting an element and taking the union of two sets with the maximum number of loop iterations:

$$O(bound_Z) = Z \cdot O(2Z) + O(Sel_i) \cdot O(d_i) + O(Sel_i) + 2O(Z) + O(S \cdot X^2) \quad (4.20)$$

For distance metric d_1 and selection metric Sel_1 ,the complexity of the $bound_Z$ algorithm evaluates to:

$$O(bound_Z(d_1, Sel_1)) = O(Z^2 \cdot X + Z \cdot X^2)) \quad (4.21)$$

For the configuration of the distance and selection metric d_2 and Sel_2, the complexity evaluates to:

$$O(bound_Z(d_2, Sel_2)) = O(Z^3 \cdot X^2) \tag{4.22}$$

To conclude, the complexity scales quadratically with the number of cache states Z with the configuration d_1, Sel_1 and cubicly with Z for the configuration d_2, Sel_2 when we assume that X is a small constant. We will argue below that X is a small constant for most embedded applications.

4.3.4 Bounding number of memory blocks

To tightly bound the maximum number of memory blocks X in a cache set s_i is difficult. A naive upper bound would be the number of memory blocks that map to cache set s_i, which is bounded by the total number of memory blocks M divided by the number of cache sets S: $\frac{M}{S}$. But this is obviously an overestimation because a single access in a basic block to cache set s_i replaces all memory blocks of s_i by a single memory block, namely $gen[B]$. In irregular time periods, the number of cache blocks X will be reduced to one. Formally determining the length of this time period is complex.

We focus now on the question, at which basic blocks the number of memory blocks will increase. If the cache sets $s_i(p_1)$ and $s_i(p_2)$ in *both* predecessor nodes p_1 and p_2 of a basic block B are not empty and if no cache blocks are mapped to s_i during the execution of basic block B, then the number of memory blocks increases. The worst case number of memory blocks in s_i occurs *after* the deepest nested `if-then-else` structure, with a full tree (each `then` and `else` branch exists), and a memory block is accessed in *each* branch at the lowest nested level. Assuming that d denotes the highest level of nested branch statements, then there can be 2^d memory blocks in s_i at all successor nodes b_k (after the deepest nest level) that do not access s_i, e.g. all b_k with $gen_{b_k} = \emptyset$. Then, X is bounded by the in-equation 4.23:

$$X \leq \min(2^d, \frac{M}{S}) \tag{4.23}$$

For time-complexity considerations, we have to assume that the maximum number of elements occurs at each node. However, even if the maximum number of 2^d memory blocks is reached in a program, it will only be valid for a small number of nodes: e.g. the longest path where no accesses occur to cache set s_i. But, it is very difficult to formally determine how many basic blocks lie on that path.

As an alternative, we give some intuitive argument: The instruction cache size is significantly smaller than the number of all memory blocks (factor 10 to 100). There

will be frequent accesses to the same cache set s_i which reduces the total number of elements in a cache set to 1. It is likely that the average number of elements in a cache set is small.

4.3.5 Example

We apply the scalable precision cache analysis to the example control flow graph in figure 4.1 and bound the number of cache states to $Z = 2$. First, we calculate the sets of RCS and LCS and, then, we compare the number of useful cache blocks to the results in the approaches by Lee and Mitra.

Calculation of reaching cache states

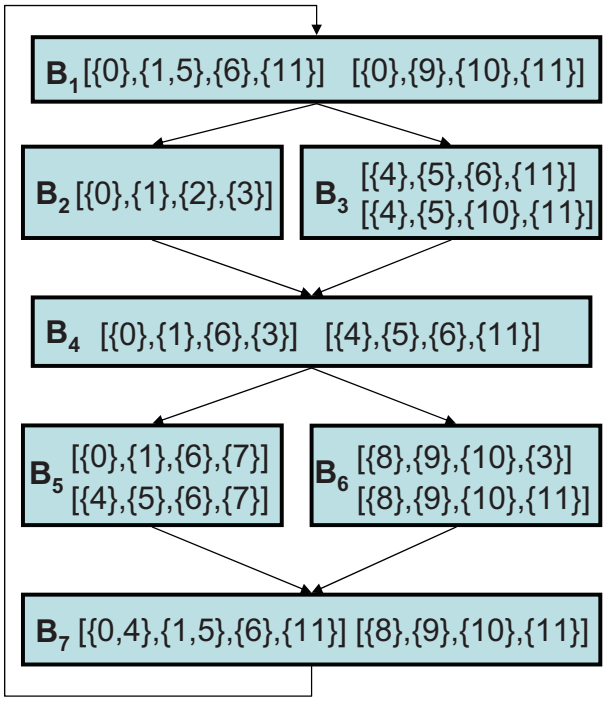

Figure 4.5. RCS calculation for scalable precision cache analysis.

Table 4.3 shows the result of the $RCS_{out}[B_i]$ for each node of the forward iterative data-flow analysis of equations 4.6-4.8. The result after the fixed point has reached is shown in figure 4.5.

Two iterations are sufficient until the algorithm converges in this example. Initially all $RCS_{in}[B_i] = \emptyset$. The $gen[B_i]$ for the control flow graph in figure 4.1 have been shown in figure 2.1.

The *RCS* of the second iteration are shown graphically in figure 4.5. We apply the scalable data flow analysis to this example starting at B_1 in the first iteration. Because

B_i	$RCS_{out}(B_i)$ 1st iter.	$RCS_{out}(B_i)$ 2nd iter.
B_1	$[\{0,\emptyset,\emptyset,\emptyset]$	$[\{0\},\{1,5\},\{6\},\{11\}]$ $[\{0\},\{9\},\{10\},\{11\}]$
B_2	$[\{0\},\{1\},\{2\},\{3\}]$	$[\{0\},\{1\},\{2\},\{3\}]$
B_3	$[\{4\},\{5\},\emptyset,\emptyset]$	$[\{4\},\{5\},\{6\},\{11\}]$ $[\{4\},\{5\},\{10\},\{11\}]$
B_4	$[\{0\},\{1\},\{6\},\{3\}]$ $[\{4\},\{5\},\{6\},\emptyset]$	$[\{0\},\{1\},\{\mathbf{6}\},\{3\}]$ $[\{4\},\{5\},\{\mathbf{6}\},\{11\}]$
B_5	$[\{0\},\{1\},\{6\},\{7\}]$ $[\{4\},\{5\},\{6\},\{7\}]$	$[\{0\},\{1\},\{6\},\{7\}]$ $[\{4\},\{5\},\{6\},\{7\}]$
B_6	$[\{8\},\{9\},\{10\},\{3\}]$	$[\{8\},\{9\},\{10\},\{3\}]$ $[\{8\},\{9\},\{10\},\{11\}]$
B_7	$[\{8\},\{9\},\{10\},\{\mathbf{11}\}]$ $[\{0,4\},\{1,5\},\{6\},\{\mathbf{11}\}]$	$[\{0,4\},\{1,5\},\{6\},\{11\}]$ $[\{8\},\{9\},\{10\},\{11\}]$

Table 4.3. Reaching cache states $RCS_{out}[B_i\}]$ for scalable precision cache analysis with $Z = 2$.

the *in* set is empty, $RCS_{out}[B_1] = gen[B_1]$. For basic block B_2, the incoming edge from B_0 is evaluated and equation 4.7 is applied. The result is shown in the second column in line 2 in table 4.3. Analogously, all other $RCS_{out}[B_i]$ are computed for B_3, B_4, B_5, B_6. The calculation at B_7 is different because via the two incoming edges from B_5 and B_6 there are 3 cache states but only $Z = 2$ states are allowed and, thus, the $bound_2$ algorithm is applied:

$$\begin{aligned} RCS_{in}[B_7] &= bound_2([c_1 = \{0\},\{1\},\{6\},\{7\}], c_2 = [\{4\},\{5\},\{6\},\{7\}], \\ &\qquad c_3 = [\{8\},\{9\},\{10\},\{3\}]) \\ RCS_{in}[B_7] &= \{[\{0,4\},\{1,5\},\{6\},\{7\}][\{8\},\{9\},\{10\},\{3\}]\} \end{aligned}$$

We use Sel_2 and d_1 as selection method and distance metric throughout this example:

$$d_1(c_1,c_2) = 2 \quad d_1(c_1,c_3) = 4, \quad d_1(c_2,c_3) = 4$$

Therefore, c_1 and c_2 are merged because their distance is the smallest (2). Note that the cache set $RCS_{in}[B_7][0]$ and $RCS_{in}[B_7][1]$ contain two memory blocks. Since the $gen[B_7]$ contains m_{11} in the last cache set, only the memory blocks in the last cache set are replaced. The $RCS_{out}[B_7]$ is shown in table 4.3 in the last line of the second column (in bold face).

The calculation in the second iteration of $RCS_{out}[B_i]$ is straight forward for B_1, B_2, B_3. At B_4 cache states have to be merged, because there are three states available via in-

coming edges from B_2 and B_3:

$$\begin{aligned} RCS_{in}[B_4] &= bound_2[c_1 = [\{0\},\{1\},\{2\},\{3\}]c_2 = [\{4\},\{5\},\{6\},\{11\}], \\ &\qquad c_3 = [\{4\},\{5\},\{10\},\{11\}] \\ RCS_{in}[B_4] &= \{[\{0\},\{1\},\{2\},\{3\}],[\{4\},\{5\},\{6,10\},\{11\}]\} \end{aligned} \tag{4.24}$$

Again we use Sel_2 and d_1: $d_1(c_1,c_2) = 4$, $d_1(c_1,c_3) = 4$ and $d_1(c_2,c_3) = 1$. The cache states c_2 and c_3 are merged because they result in minimum distance of 1. The result is shown in equation 4.24. The $RCS_{out}[B_4]$ is shown in table 4.3 in the third column. We point out the replacement by $gen[B_4] = \{6\}$-set in bold face. The analysis continues with B_5 and B_6 and, finally, the last merge-operation occurs at B_7. The four incoming states from B_5 and B_6 are evaluated using Sel_2 and d_1 metric. The result is shown in the last line in the second column in table 4.3.

Calculation of live cache states

B_i	$LCS_{in}[B_i]$	$LCS_{out}[B_i]$
B_1	$[\{0\},\{1\},\{2\},\{3\}]$ $[\{0\},\{5\},\{6\},\{7,11\}]$	$[\{0,8\},\{1\},\{2\},\{3\}]$ $[\{4\},\{5\},\{6\},\{7,11\}]$
B_2	$[\{0\},\{1\},\{2\},\{3\}]$ $[\{8\},\{1\},\{2\},\{3\}]$	$[\{0\},\{1,5\},\{6\},\{7\}]$ $[\{8\},\{9\},\{6\},\{11\}]$
B_3	$[\{4\},\{5\},\{6\},\{7\}]$ $[\{4\},\{5\},\{6\},\{11\}]$	$[\{0\},\{1,5\},\{6\},\{7\}]$ $[\{8\},\{9\},\{6\},\{11\}]$
B_4	$[\{0\},\{1,5\},\{6\},\{7\}]$ $[\{8\},\{9\},\{6\},\{11\}]$	$[\{0\},\{1,5\},\{6\},\{7\}]$ $[\{8\},\{9\},\{10\},\{11\}]$
B_5	$[\{0\},\{1\},\{6\},\{7\}]$ $[\{0\},\{5\},\{6\},\{7\}]$	$[\{0\},\{1\},\{2\},\{11\}]$ $[\{0\},\{5\},\{6\},\{11\}]$
B_6	$[\{8\},\{9\},\{10\},\{11\}]$	$[\{0\},\{1\},\{2\},\{11\}]$ $[\{0\},\{5\},\{6\},\{11\}]$
B_7	$[\{0\},\{1\},\{2\},\{11\}]$ $[\{0\},\{5\},\{6\},\{11\}]$	$[\{0\},\{1\},\{2\},\{3\}]$ $[\{0\},\{5\},\{6\},\{7,11\}]$

Table 4.4. Live cache states $LCS_{in}[B_i]$ and $LCS_{out}[B_i]$ for scalable data flow analysis with $Z = 2$.

The results for the LCS_{in} and LCS_{out} of the scalable data flow algorithm are shown in table 4.4. The LCS calculation uses a backward data flow analysis and has been given in the equations 4.10 and 4.11.

Merge operations were necessary at B_1 in the first iteration and at B_4 and B_1 in the second iteration. Note the difference between $LCS_{out}[B_i]$ and $LCS_{in}[B_i]$. The former

represents the live cache states *before* the memory blocks of B_i are loaded to the cache and the latter represents the live cache states *after* the memory blocks of B_i are loaded to the cache.

Calculation of useful cache blocks

B_i	USE_{scale}	USE_{mitra}	USE_{lee}
B_1	**3**	3	4
B_2	2	2	2
B_3	**2**	2	3
B_4	**3**	3	4
B_5	**2**	2	3
B_6	1	1	1
B_7	**4**	3	4

Table 4.5. Useful cache blocks $USE_{scale}$for scalable analysis with $Z = 2$. Comparison with state-based approach USE_{mitra} and set-based approach USE_{lee} from table 4.1.

The useful cache blocks $USE_{scale}[B_i]$ are computed by the intersection of reaching cache states $RCS_{out}[B_i]$ and live cache states $LCS_{out}[B_i]$. The total number of useful cache blocks according to equation 4.18 are summarized and compared to the results of Lee and Mitra's approach in table 4.5

The result is is in most cases equal to the tighter result of USE_{mitra} and in one case equal to the result of USE_{lee} (at basic block B_7).

4.4 Preemption delay analysis for associative caches

The general description in section 4.3 considered only direct mapped caches. This section extends the proposed analysis to set-associative caches. In a n-way associative cache a memory block can be placed into n cache blocks within its designated cache set. This set-associative cache organization requires a policy called the replacement policy that decides which block to replace when a new memory block is mapped to the cache set when all cache blocks are occupied. The least recently used (LRU) policy, which replaces the block that has not been referenced for the longest time, is a commonly used strategy.

Associative caches are common place in embedded architectures but have mainly be ignored in preemption delay analysis. In [89] [92] [18] target direct mapped instruction caches. Only in [66] associative caches have been considered. However, their description contains a flaw, which we point out in section 4.4.5. In the follow-

ing, we present the analysis for associative instruction caches with *LRU* replacement strategy based on the new scalable precision cache model.

4.4.1 Scalable data flow analysis

According to our definition in section 4.3, the RCS_B contains all possible cache blocks at basic block B. In the case of direct-mapped caches, a cache set can hold only one memory block. This model has to be extended. In the following we formulate the computation of reaching cache states *RCS*. The extension of *LCS* is analogous.

We define $RCS^c_{in}[B]$ and $RCS^c_{out}[B]$ as the sets of all possible cache states of cache set c at the beginning and the end of basic block B, respectively. The set $gen^c[B]$ represents a vector of n sets of memory blocks:

$$gen^c[B] = (gen^c_1[B], gen^c_2[B], \cdots, gen^c_n[B])$$

The $gen^c[B]$ set contains the *last* memory blocks that are accessed during the execution of basic block B. More formally, each $gen^c_i[B]$ is either empty or $gen^c_i[B] = \{m\}$ if m is the ith most recently accessed memory block in B. The set $gen^s_n[B]$ contains the most recently accessed cache block and $gen^s_1[B]$ the least recently one. With this definition of $gen^c[B]$, the sets $RCS^c_{in}[B]$ and $RCS^c_{out}[B]$, are related as follows:

$$RCS_{in}[B] = bound_Z(\bigcup_{p \in pred(B)} RCS_{out}[p])) \tag{4.25}$$

$$RCS^c_{out}[B] = \bigcup_{r \in RCS^c_{in}[B]} LRU_{gen^c_1}(\cdots((LRU_{gen_n}(r))) \quad \forall c.1 \leq c \leq S \tag{4.26}$$

The function $bound_Z(C)$ is the same as in figure 4.4. The replacement algorithm for the scalable precision cache model $LRU_m(c)$ for an n-way associative cache is presented in section 4.4.3. Note that the $RCS_{in}[B]$ is defined for each cache state and $RCS^c_{out}[B]$ is defined for each cache set c. However, this is only a matter of presentation.

The $LRU_m(r)$ function models the cache behavior for loading a single memory block m to the cache state r. In a n-way associative cache, there can be n elements mapped to the same cache set during the execution of basic block B. The function $LRU_m(r)$ is applied as often as there are non zero elements in gen^c_i-set.

4.4.2 Bound algorithm for associative caches

In the following we extend the distance metric, selection algorithm and merge operation for associative caches based on the presentation in section 4.3.2 for the $bound_Z$ algorithm.

Distance metric

The function $d(a,b)$ of two cache states a,b is defined as a metric that delivers the difference of two cache states. The metrics d_1 and d_2 are applied to n-way associative caches as follows:

$$d_1(a,b) = \sum_{s=1}^{S} \begin{cases} 1 & if \exists i. \quad a_s[i] \neq b_s[i] \quad \forall \leq i \leq m \\ 0 & otherwise \end{cases} \tag{4.27}$$

The complexity of metric d_1 is

$$O(d_1) = O(S \cdot m \cdot X) \tag{4.28}$$

where X denotes the maximum number of elements of the set $a_s[i]$. Equality test of two sets can be computed in linear time [26]. A bound for the maximum number of memory blocks X is discussed in section 4.3.4. The metric d_2 counts the number of different memory blocks of each cache set with the symmetric difference:

$$d_2(a,b) = \sum_{s=1}^{S} \sum_{nn=1}^{n} |(a_s[nn] \cup b_s[nn]) \setminus (a_s[nn] \cap b_s[nn])| \tag{4.29}$$

The complexity of d_2 is given by taking $S \cdot n$ times the following operations: 1) the union of two sets with totally $2X$ elements; 2) the intersection of two sets with at most $2X$ elements; and 3) the set difference of a set with $2X$ elements and X elements.

$$O(d_2) = O(S \cdot m \cdot (t_{\cup}(X,X) + t_{\cap}(X,X) + t_{setdiff}(2X,X)) = O(S \cdot m \cdot X^2)) \tag{4.30}$$

Cache state selection

The selection methods Sel_1 and Sel_2 are independent of the associativity. Therefore, the complexity is the same as in section 4.3.2: $O(Sel_1) = O(|C|)$ and $O(Sel_1) = O(|C|^2)$.

Merging of cache states

The merge-operation $\odot$ has already been defined for direct mapped caches in section 4.3.2. For an n-way associative cache the $\odot$ operation is applied pointwise to every vector element $a_i[1], \cdots, a_i[n]$ of a cache set a_i. The time complexity is given by:

$$O(a \odot b) = O(S \cdot n \cdot (t_{\cup}(X,X)) = O(S \cdot n \cdot X^2)) \tag{4.31}$$

4.4.3 LRU algorithm for scalable precision cache model

The replacement algorithm $LRU_{gen_i^c}(r)$ for cache set c of cache state r is defined in the following equation 4.32:

$$LRU_{gen_i^c}(r) = \begin{cases} r & if \quad gen_i^c = \emptyset \\ LRU_m(r) & if \quad gen_i^c = \{m\} \end{cases} \tag{4.32}$$

The function $LRU_m(r)$ models the LRU replacement strategy for the scalable precision cache model. The pseudo-code is shown in figure 4.6.

```
Input: cache set c, memory block m Output: cache set c'
0 function LRU_m(c)
1 Initialization ∀i.   c'[i] = ∅
2 if(m ∉ c)
3     c'[n] = {m}
4     c'[j] = c[j+1]   ∀j.   n > j ≥ 1
5 else
6     c'[n] = {m}
7     ∀i.m ∈ c[i] do
8       c'[j] = c'[j] ∪ c[j+1]   ∀j.   n > j ≥ i
9       c'[j] = c'[j] ∪ c[j]     ∀j.   i > j ≥ 1
10        if ( ∃m' ≠ m.   m' ∈ c[i])
11          c'[j] = c'[j] ∪ c[j+1]   ∀j.   n > j ≥ 1
12 remove memory block m from all c'[j].   ∀n > j ≥ 1
```

Figure 4.6. LRU algorithm for scalable cache model.

Lemma 4.1. The algorithm in figure 4.6 computes the $LRU_m(c)$ replacement strategy, when memory block m is mapped to cache set c. Provided that c is a vector of sets: $c = [c_1, c_2, \cdots, c_n]$, $c_i \subset M$, where M is the set of all memory blocks and c_1 denotes the least recently used and c_n the most recently used cache block of cache set c.

Example

We apply the algorithm to the control flow graph of figure 4.1 with an 4-way associative instruction cache with 2 cache sets. For demonstration we compute the reaching cache states (RCS) for each node. Figure 4.7 shows the same control flow graph with the possible cache states. A cache state consists of two cache sets c_0

and c_1 with each four positions. For example, in basic block B_3 memory block m_0 and m_4 are mapped to cache set c_0 and memory block m_5 to cache set c_1. To save space, m_i is abbreviated as i and an empty set is denoted with $-$. In order to demonstrate the cache state reduction, we restrict the number of cache states to $Z = 2$. In B_4 two cache states are reached and only the memory block m_6 is mapped to c_0 ($gen^{c_0}[B_4] = [\{\},\{\},\{\},\{m_6\}]$). Since $m_6 \notin c_0$ in both cache states the condition in line 2 is true and the lines 3-4 of LRU algorithm in figure 4.6 are executed. All memory blocks move one position to the left and m_6 is placed at the most recently used position. Cache set c_1 is not modified. These two cache states are propagated to B_5 and B_6, where m_6, m_7 and m_8, m_9, m_{10} are accessed respectively.

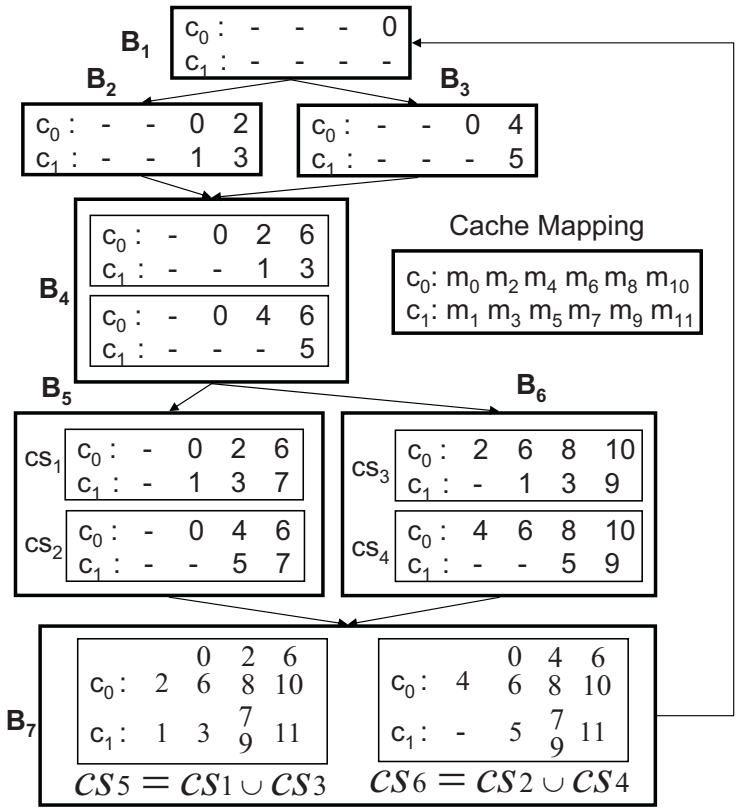

Figure 4.7. Reaching Cache States for 4-way associative instruction cache with two sets.

When the algorithm computes the $RCS_{in}[B_7]$, according to equation 4.25, four cache states $cs_1, cs_2\ cs_3, cs_4$ are available on incoming edges, but only two are allowed. Therefore, two of the cache states with the minimum distance, according to equation 4.14, will be merged. We assume for this example the selection method Sel_2, which compares all cache states, and distance metric d_2, which computes the

symmetric difference.

$$d_2(cs_1, cs_2) = 5 \quad d_2(cs_1, cs_3) = 5 \quad d_2(cs_2, cs_3) = 10$$
$$d_2(cs_1, cs_4) = 10 \quad d_2(cs_2, cs_4) = 5 \quad d_2(cs_3, cs_4) = 5$$

The cache states $cs_1 \cup cs_3$ and $cs_2 \cup cs_4$ are merged:

$$cs_5'' = cs_1 \cup cs_3 = \begin{cases} c_0 & [\{2\},\{0,6\},\{2,8\},\{6,10\}] \\ c_1 & [\emptyset,\{1\},\{3\},\{7,9\}] \end{cases}$$
$$cs_6'' = cs_2 \cup cs_4 = \begin{cases} c_0 & [\{4\},\{0,6\},\{4,8\},\{6,10\}] \\ c_1 & [\emptyset,\emptyset,\{5\},\{7,9\}] \end{cases}$$

In basic block B_7 memory block m_{11} is mapped to c_1. Note that $m_{11} \notin c_1$, such that all elements are only shifted one position to the left by the LRU operator $cs_5' = LRU_{m_{11}}(cs_5'')$ and $cs_6' = LRU_{m_{11}}(cs_6'')$. These cache states are shown in basic block B_7. To save space several elements within a set at a position are aligned vertically, such as m_6 and m_{10} in cache set c_0.

In the second iteration of the data flow analysis, we start again with basic block B_1. Now $m_0 \in c_0$ in both cache states c_5'' and c_6''. Therefore, the loop in line 7 is executed once, the contents of cache set $c_0[2]$ to $c_0[4]$ are moved one position to the left and m_0 is placed in the $c_0[4]$ slot. This results to the new cache states cs_5 and cs_6:

$$cs_5 = \mathrm{LRU}_{m_0}(cs_5') = \begin{cases} c_0 & [\{\mathbf{2},\mathbf{6}\},\{\mathbf{2},\mathbf{8}\},\{\mathbf{6},\mathbf{10}\},\{\mathbf{0}\}] \\ c_1 & [\{1\},\{3\},\{7,9\},\{11\}] \end{cases}$$
$$cs_6 = \mathrm{LRU}_{m_0}(cs_6') = \begin{cases} c_0 & [\{\mathbf{4},\mathbf{6}\},\{\mathbf{4},\mathbf{8}\},\{\mathbf{6},\mathbf{10}\}\{\mathbf{0}\}] \\ c_1 & [\emptyset,\emptyset,5,\{7,9\}] \end{cases}$$

As the last example of the algorithm, we show the RCS computation for the cache state cs_5 at basic block B_2 with $gen^{c_0}[B_2] = [\emptyset,\emptyset,\emptyset,\{m_2\}]$ and $gen^{c_1}[B_2] = [\emptyset,\emptyset,\{m_1\},\{m_3\}]$. Now the LRU algorithm is applied three times:

$$RCS_{out}[B_2] = LRU_{m_3}(LRU_{m_2}(LRU_{m_1}(cs_5)))$$

We show the computation in three steps and denote each intermediate cache state as cs', cs'' and cs''' respectively:

$$cs' = LRU_{m_1}(cs_5) = \begin{cases} c_0 & [\{2,6\},\{2,8\},\{6,10\},\{0\}] \\ c_1 & [\{\mathbf{3}\},\{\mathbf{7},\mathbf{9}\},\{\mathbf{11}\},\{\mathbf{1}\}] \end{cases} \tag{4.33}$$

$$cs'' = LRU_{m_2}(cs') = \begin{cases} c_0 & [\{\mathbf{6},\mathbf{8}\},\{\mathbf{6},\mathbf{10}\},\{\mathbf{0}\},\{\mathbf{2}\}] \\ c_1 & [\{3\},\{7,9\},\{11\},\{1\}] \end{cases} \tag{4.34}$$

$$cs''' = LRU_{m_3}(cs'') = \begin{cases} c_0 & [\{6,8\},\{6,10\},\{0\},\{2\}] \\ c_1 & [\{\mathbf{7},\mathbf{9}\},\{\mathbf{11}\},\{\mathbf{1}\},\{\mathbf{3}\}] \end{cases} \tag{4.35}$$

The access of $LRU_{m_1}(cs_5)$ shows a reordering in cache set c_1 where m_1 is placed in the last recently used position $c_1[3]$. The result is shown in equation 4.33.

The access of $LRU_{m_2}(cs')$ is shown in equation 4.34, which is somewhat more difficult to explain because $m_2 \in c_1[1]$ and $m_2 \in c_1[2]$. We explain the algorithm by expanding all possible cache states that are represented by cs', then we apply the LRU algorithm and finally we compare the result with the LRU algorithm for the scalable precision model. We leave out all in-valid states for cache set c_0: $[2,2,6,0]$, $[2,2,10,0]$, $[6,2,6,0]$ and $[6,8,6,0]$ because a memory block m_i occurs several times. If we apply the standard LRU algorithm to the remaining cache states, we get:

$$LRU_{m_2}[2,8,6,0] = [8,6,0,2] \quad LRU_{m_2}[2,8,10,0] = [8,10,0,2]$$

$$LRU_{m_2}[6,8,10,0] = [8,10,0,2] \quad LRU_{m_2}[6,2,10,0] = [6,10,0,2]$$

The union of the above resulting cache states is then the conservative cache state cs'' that is computed by the algorithm in figure 4.6 (lines 6-12).

Finally, $LRU_{m_3}(cs'')$ is applied in equation 4.35, which is a reordering for cache set c_1.

Note, that the set-based cache model may lead to an overestimation, because the model includes cache states that are invalid. However, the representation is conservative, such that the actual cache related preemption delay is always smaller than the estimated one.

Proof of Lemma 4.1

Proof of Lemma 4.1. The proof is presented over the structure of the cache set elements. We start with the restriction that all c_i contain only one element ($|c_i| \leq 1$) and extend this model step-wise to $c_i \subset M$.

Part I. We assume $\forall c_i. \quad |c_i| \leq 1$.

This case represents an ordinary cache state, with n sets for a n way set-associative

cache. We distinguish if there exists c_i with $m \in c_i$ or not.
(a) $\forall c_i. \quad m \notin c_i$. The cache block in c_1 will be replaced and the elements will be reordered, such that $c' = [c_2, \cdots, c_n, \{m\}]$. This is implemented in lines 2-4 in figure 4.6.
(b) $\exists c_i. \quad m \in c_i$. From assumption $|c_i| \leq 1$ follows that c_i is unique and the loop in line 7 will be executed exactly once. The memory block $m \in c_i$ is placed at the most recently used position c_n and all $c_{i+1} \cdots c_n$ elements shifted one position to the left (lines 6-8): $(c_1, \cdots, c_{i-1}, c_i, c_{i+1}, \cdots, c_n)m = (c_1, \cdots, c_{i-1}, c_{i+1}, \cdots, c_n, \{m\})$.

Note, that the contents of the cache does not change. All elements left from c_i do not change their position (line 9). The condition in line 10 will always evaluate to false, because $|c_i| \leq 1$. Finally, the memory block m is removed from the positions 1 to $n-1$ of the cache set vector (line 12). Thus, we have shown that if all $|c_i| \leq 1$ the $LRU_m(c)$ is correct.

Part II. Assumption $\forall c_i. \quad m \notin c_i : |c_i| \leq 1$.
All c_i that do not contain m are singleton sets, only those c_i with $m \in c_i$ may contain more elements.
(a) $m \notin c_i \forall c_i$ this has been shown in (I.)
(b) there exists a unique $c_i. \quad m \in c_i$. If $|c_i| \leq 1$, refer to I, otherwise the case $|c_i| = d \geq 1$ is detected in lines 10-11 in the algorithm. We have to distinguish two cases:

$$\text{(b1)} \quad (c_1, \cdots, c_{i-1}, \{m\}, c_{i+1}, \cdots, c_n)$$
$$\text{(b2)} \quad (c_1, \cdots, c_{i-1}, \{m_k\}, c_{i+1}, \cdots, c_n), \forall m_k \in c_i . m_k \neq m$$

For (b1) we have shown already in I. that $LRU_m(c)$ is correct. In the case of (b2) there are $d-1$ possible cache states, where $m \notin c$. This means that the least recently used memory block c_1 is replaced, the contents of $c_i, i = 2, \cdots, n$ move one position to the left, and $c_n = \{m\}$.
(c) There exist several $c_i . m \in c_i$. Note that in the set representation there may be several sets that contain m, but there cannot be an original cache state with $m \in c_i, m \in c_j, i \neq j$. Thus we can apply lines 8-11 to each cache set c_i that contains m separately and take the union of the resulting cache set c'. Let us apply the algorithm to some c_i, and there exist $c_{j_1}, \cdots, c_{j_k}$ other sets that contain m. We can formally construct the set of all possible cache states that are described by this cache set and apply the LRU strategy to each cache state, as in part I, and take the union of the resulting cache states. This is implemented in lines 8-9.

Part III. Induction step: All c_i may have more then one element.
In Part II we have shown that the $LRU_m(c)$ algorithm is correct when all c_i that do not contain m are singleton sets. If a c_i contains more than one element, we construct

every possible cache state and apply Part II for each state, then the union is taken (line 10-11).

This completes the proof. □

4.4.4 Time complexity

The time complexity for n-way associative caches is similar to the complexity discussion in section 4.3.3.

The complexity of the $bound_Z$ algorithm as shown in figure 4.4 depends on the same steps. The complexity of each step is summarized in table 4.6.

Step	Operation	complexity
1	count	$O(2Z)$
2	Sel_1	$O(2Z)$
2	Sel_2	$O((2Z)^2)$
2	d_1	$O(S \cdot n \cdot X)$
2	d_2	$O(S \cdot n \cdot X^2)$
3	insert	$O(Z)$
3	remove	$O(Z)$
4	union	$O(S \cdot n \cdot X^2))$
5	loop bound	Z

Table 4.6. Time-complexity of analysis steps for $bound_Z$ algorithm for n-way associative instruction caches. Z denotes the maximum number of cache states at a node, X denotes the maximum number of memory blocks of a cache set, S denotes the number of cache sets in a cache

The time complexity for the LRU algorithm is calculated easily examining each line of the algorithm in figure 4.6:

$$\begin{aligned} O(LRU_m(r)) &= \max(1+X\cdot n, \quad 1+2\cdot t_{\cup}(X,X)\cdot n+X+n\cdot t_{rem}(X)) \\ O(LRU_m(r)) &= 2O(X^2)\cdot n+O(X)(n+1)+1=(C^2\cdot n) \end{aligned} \tag{4.36}$$

The computation time for a union operator is denoted by $t_{\cup}(X,X)$ and for the remove-operator by $t_{rem}(X)$. To conclude, the LRU algorithm scales quadratically with the number of cache states C times the degree of associativity.

In summary, the complexity of the entire $bound_Z$ algorithm is given by

$$O(bound_Z) = Z\cdot(O(Sel_i)\cdot O(d_i)+O(Sel_i)+2O(2Z)+O(cs\cdot m\cdot X^2)) \tag{4.37}$$

For the distance and selection metric d_1 Sel_1 the complexity of the $bound_Z$ algorithm evaluates to:

$$O(bound_Z) = O(Z^2 \cdot X + Z \cdot X^2)) \tag{4.38}$$

and the complexity for the distance and selection metrics d_2 and Sel_2 evaluates to:

$$O(bound_Z) = O(Z^3 \cdot X^2) \tag{4.39}$$

To conclude, the complexity scales quadratically with the number of cache states with the configuration d_1, Sel_1 and cubicly with d_2, Sel_2 when we assume that X is a small constant. This is the same time-complexity as for direct mapped caches.

4.4.5 Lee's approach contains a flaw

The approach by Lee et al. [66] extends the data flow analysis to set-associative caches. The proposed solution is principally correct, but contains a flaw which we would like to point out. If a basic block accesses a memory block that already exists in the RMB_{in} set then this cache block can occur multiple times in the proposed algorithm by Lee [66] (section 7.1). The correct solution would be to reorder the memory blocks. In [87] the program line reordering for LRU has been considered for cache analysis for instruction caches. In the following we briefly review the algorithm by [66] and give an example.

The reaching cache states are denoted by $RMB^c_{in}[B]$ and $RMB^c_{out}[B]$ for the incoming and outgoing reaching cache blocks of cache set c at basic block B. A state of a cache set for an n-way set-associative cache is defined as a vector $(m_{i_1}, \cdots, m_{i_n})$, where m_{i_1} is the least recently referenced block and m_{i_n} the most recently referenced block. The $gen^c[B]$ contains the state of cache set c generated in basic block B. It is either empty when none of the memory blocks mapped to cache set c are referenced in basic block B or a singleton set whose only element is a vector: $(gen^c_1[B], \cdots, gen^c_n[B])$. In the vector $gen^c_n[B]$ is the memory block whose reference in basic block B is the last reference to the cache set c in B.

We give the an example for a 4-way associative cache with the following assumptions about $RMB^c_{in}[B]$ and $gen^c[B]$ at some basic block B in cache set c:

$$RMB^c_{in}[B] = (m_1, m_2, m_3, m_4) \quad gen^c[B] = (null, null, m_3, m_5)$$

Note, that m_3 is reached via some incoming path *and* is referenced in basic block B. The $RMB^c_{out}[B]$ is defined in Lee's algorithm depending on the contents of the *gen*-set. We give the data flow equations for RMB_{in} and RMB_{out} only for the case that applies

to this example:

$$
\begin{aligned}
RMB^{c}_{in}[B] &= \bigcup_{p \in pred(B)} RMB_{out}[p] \\
RMB_{out}[B] &= \begin{cases} \bigcup_{rmb \in RMB^{c}_{in}[B]} \{(rmb_{n-1}, rmb_n, gen^{c}_{n_1}[B], gen^{c}_{n}[B])\} \\ \quad \text{if } gen^{c}_{1}[B], \cdots gen^{c}_{n-2}[B] = \emptyset \\ \quad\quad \text{and } gen^{c}_{n-1}[B], gen^{c}_{n}[B] \neq emptyset \\ \cdots \text{ otherwise} \end{cases}
\end{aligned}
\tag{4.40}
$$

For the above example, the $RMB^{c}_{out}[B]$ is given by

$$RMB^{c}_{out}[B] = (rmb_3, rmb_4, gen^{c}_{3}[B], gen^{c}_{4}[B]) = (\mathbf{m_3}, m_4, m_3, m_5)$$

however, when the memory blocks $m_3 \rightarrow m_5$ are loaded to cache set c, the LRU replacement algorithm will result to:

$$(\mathbf{m_2}, m_4, m_3, m_5)$$

The mistake occurs for memory block m_3 which is reordered by the LRU algorithm. In the algorithm by Lee the *gen*-sets and RMB_{in} sets are not examined for equal elements. Therefore, m_3 occurs in $RMB^{c}_{out}[B]$ multiple times and m_2 is missing which might result in an underestimation of the CRPD.

4.5 Cache analysis framework for real-time verification

The verification of real-time behavior involves the computation of worst case response times. Typically, several embedded applications run on an embedded microcontroller with an real-time operating system, such as ERCOSEK [31] for automotive applications. The response time analysis has to be extended to consider the preemption delays due to cache reloads.

4.5.1 Cache-aware response time analysis

Preemption delay analysis alone does not solve the cache behavior problem. A cache-aware response time analysis has to calculate how often a task is activated and how often a task is preempted for a given set of tasks.

Several cache-aware response time approaches have been proposed[19] [92] [67]. The approach by [19] considers only the preempting task while the approach by [92]

consider only the preempted task. However, the advantage is that the time-complexity is about the same as the time complexity of for solving the fixed point of the underlying iterative response time equations. In the approach by [67] the preempted as well as the preempting task are considered, but the cache-aware response time analysis scales exponentially with the number of tasks in the system.

In previous work we have developed a cache-aware response time analysis [113] for fixed priority preemptive scheduling. It computes the total number of preemptions in a given schedule in a polynomial time-complexity. Therefore, it suits a framework with an overall low time-complexity. The preemption delay is computed by a state-based data flow analysis [111]. It has been turned out that the complexity of the state-based approach was too large for greater sets of software tasks.

Therefore, we replace the time-consuming preemption delay analysis of [111] by the scalable precision cache analysis as presented in this section. In this section we apply the cache-aware response time analysis in several experiments to the results for the scalable precision cache analysis.

4.5.2 Pseudo-LRU replacement strategy

Sometimes 4-way associative caches and above, processors implement pseudo LRU (section 2.1.3). As long as this replacement strategy is deterministic, the replacement algorithm in figure 4.6 can be adapted accordingly.

4.5.3 Guidance to choose scaling parameter

We give more guidance about how to choose the scaling parameter in the following. The maximum number of memory blocks within a cache set occurs in the deepest nested branch statement as described in section 4.3.4. This value 2^d, where d is the nest level, can be used as a reference. However, in reality, a much smaller number will suffice, because the nested branch is not a full tree and cache references do not occur to the same cache set in every leaf node of the branch tree.

4.6 Experiments

This section presents the experimental results for the described analysis method for five benchmarks, taken from [66], [89] and [23] and five preemption scenarios (PrS).

Table 4.7 summarizes the benchmark characteristics: main memory usage in Bytes [B], the number of C source code lines and the WCET in 10^3 clock cycles [clk] for a 4-way set associative 1KB instruction cache. The worst case execution time of each

Id	Mem	C-Ln	WCET	Description
τ_1	376	83	1.401	square root calculation
τ_2	144	34	39.23	exchangesort
τ_3	888	180	15.34	fast Fourier transform
τ_4	296	275	1.617	packet receiver
τ_5	1023	286	4051	whetstone

Table 4.7. Benchmark Description with Memory Usage[B], c-lines and WCET[$10^3 clk$].

task was determined with SymTA/P[1] using the cycle accurate ARM945 processor simulator[2] for the different instruction cache architectures with a 20 cycle cache miss penalty. Each instruction is four byte long and cache block size is fixed to eight byte, such that two instructions fit in a cache block. Since these benchmarks are rather small compared to a real application, the cache size has to be adjusted accordingly. However, if both the application size and cache size are considered using a cache footprint index, our results can be scaled to larger applications.

The cache footprint index determines how many tasks use on average a single cache block. Table 4.8 shows this index for the evaluated preemption scenarios for a direct mapped cache and varying cache sizes.

PrS	Preempting, preempted task	256B	512B	1024B	2048B
A	τ_1 τ_2	1.53	0.95	0.47	0.24
B	τ_1 τ_3	2.0	1.68	1.31	0.83
C	τ_1 τ_5	2.0	1.68	1.34	1.17
D	τ_4 τ_3	2.0	2.0	1.97	1.23
E	τ_4 τ_5	2.0	2.0	2.0	1.57

Table 4.8. Cache footprint index for a direct mapped cache for different preemption scenarios (PrS) and cache sizes.

For example in the PrS B, both tasks τ_1 and τ_2 fully utilize the 256B cache, hence the footprint index is 2. The footprint of 1.17 for PrS C and 2KB cache shows that on average a cache block is used by one task only. For PrS A and 2KB cache the footprint index of 0.24 represents a small application and a large instruction cache. The cache footprint index can be used to generalize the results for real-size applications, since

[1] www.ida.ing.tu-bs.de/research/projects/symta
[2] RealView development suite. www.arm.com

the cache behavior depends on its utilization, and not just on cache size or application size alone.

4.6.1 Preemption delay analysis

We have implemented the analysis approach described in the preceding sections for direct mapped and associative instruction caches. We assume the distance d_2 metric of equation 4.14 and the strategy Sel_2, that compares all possible cache states.

The following diagrams show he analysis precision and analysis time compared to previously published approaches by other researchers. First, we compare the bound of the preemption delay considering only the preempted, only the preempting and both (preempting as well as preempted) task. Then, we evaluate the influence of the scaling parameter (the number of cache states which are at most allowed at each node during data flow analysis) on analysis precision. We describe the effects for changing the cache size, associativity and different benchmarks. Third, we show the impact of the scaling parameter on analysis time and memory consumption. Finally, we report the measured number of memory blocks of each set, as defined in equation 4.23 which influences the timing complexity of the intersection and merge operation (see section 4.3.4).

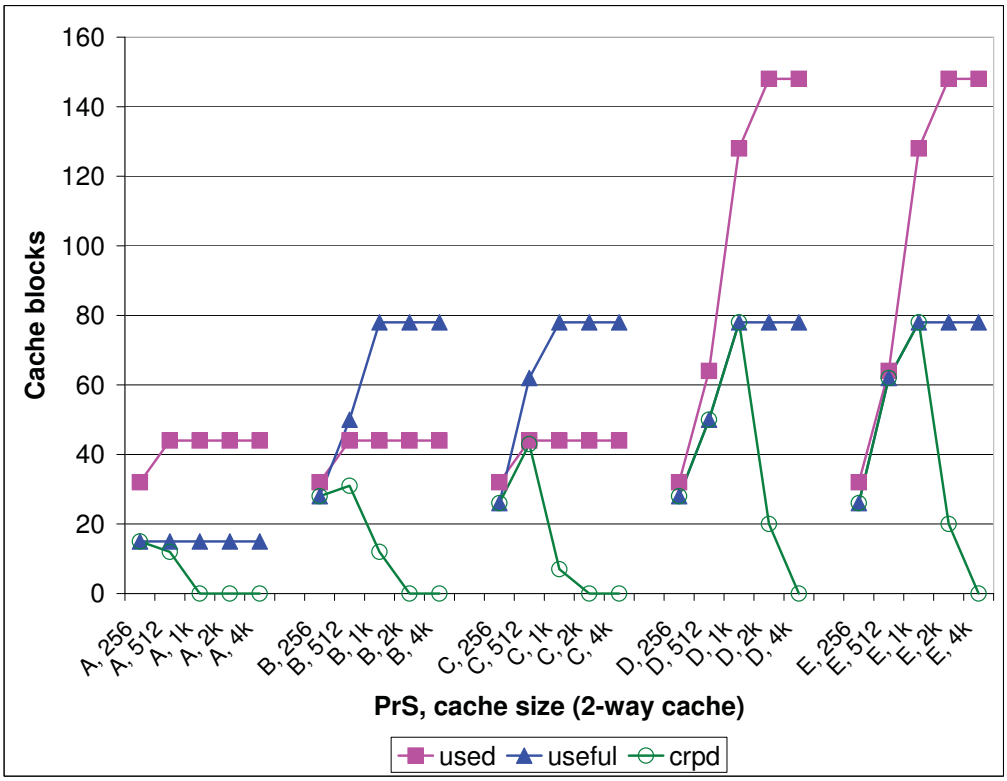

Figure 4.8. Used, useful and CRPD estimation for 2-way cache.

The following results show the estimated preemption delay if a task preempts another task once. Figure 4.8 shows the number of *used cache blocks of the preempting task*, the number of *useful cache blocks of the preempted task* and the cache related preemption delay (*CRPD*), as computed in equation 4.19 for the preemption scenarios in table 4.8. See also figure 2.2 and section 4.3.2.0 for a graphical overview and the formulas.

The curves for used and useful cache blocks are very similar and its shape can be structured in several phases. In the first phase, the cache size is the limiting factor. The number of cache blocks (used, useful) increases with the size of the cache (number of cache blocks is not sufficient). In the second phase, all useful cache blocks fit in the cache and the curves remaines on a constant level.

The $crpd$-curve can be structured in three phases. First, the curve increases with the cache size, as for the $used$- and $useful$-curve, since the cache size is the limiting factor until the curve reaches a maximum. In the second phase, the $crpd$ values are decreasing because the $useful$ and $used$ cache blocks do not fully overlap (smaller cache foot print index). In the third phase, the $crpd$ value is zero. This is because, the $used$ and $useful$ cache blocks of the preempting and preempted task respectively, are mapped to different cache sets and do not overlap. Note also, to reach phase three for a given preemption scenario it is not necessary that both tasks *entirely* fit in the cache.

These results clearly show, that just considering the $used$ or just $useful$ alone will lead to a pessimistic bound of the preemption delay.

As the next observation we note that the $crpd$ value is bounded by the minimum value of the $used$- and $useful$-curve. Its important to note, that the $useful$-curve is not below the $used$-curve because they are computed for *different* tasks (for example PrS B and C). Of course, $useful(\tau_i) \leq used(\tau_i)$ for the same task τ_i.

Figure 4.9 shows the influence of the cache associativity. The $used$ curve might increase with higher associativity while the $useful$-curve remains at the same level.

Figure 4.10 shows the bound of the preemption delay for a single preemption in number of cache blocks for increasing cache sizes. Observe the three phases of the $crpd$ curve. In phase one the value increases with the cache size; e.g. for 256B most preemption scenarios have the same value (total number of cache blocks of the cache. In the second phase the values decrease. In phase three, the preemption delay $crpd$ is zero.

The next figures 4.11, 4.12, and 4.13 show the impact of the scaling parameter. Figure 4.11 shows the impact when $z = 1, 10, 20, \infty$ for different cache sizes for a 2-way associative cache.

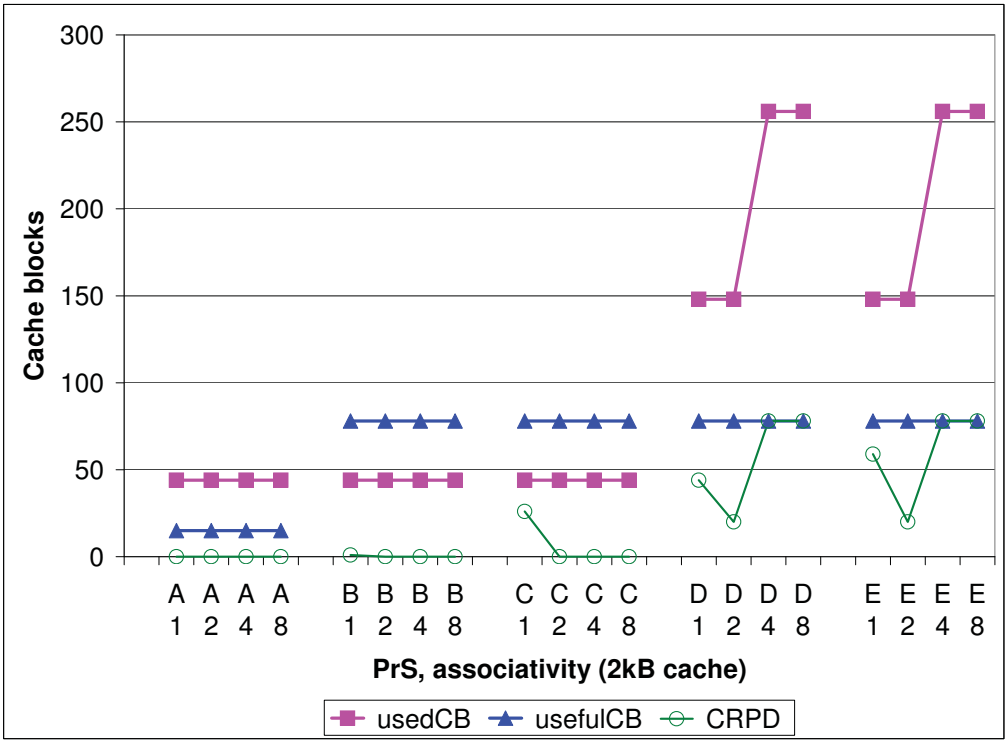

Figure 4.9. Used, useful and CRPD estimation for 2kB cache.

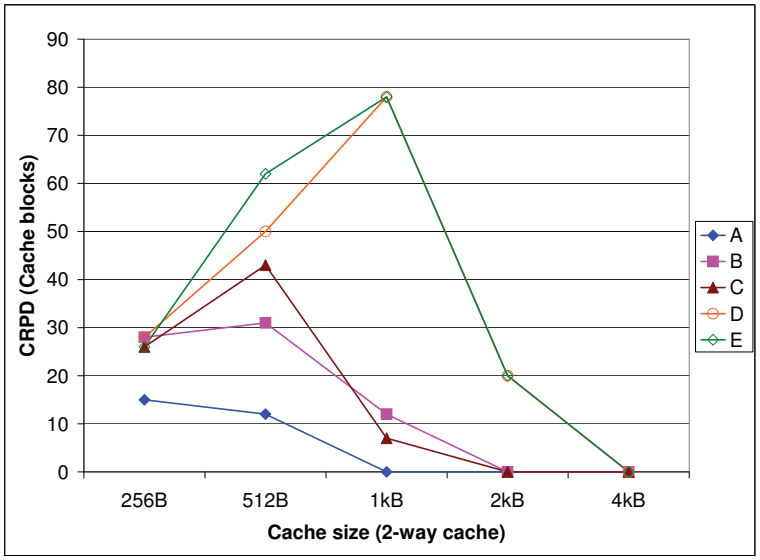

Figure 4.10. Three phases of *crpd* curve.

For $z = 1$ the results represent one cache state at each basic block as in approach by Lee [67]. The merge operation will operate exclusively on a single set.

For $z = \infty$, the results represent all possible cache states. The merge operation will never be applied. Thus this modeling is equivalent to the approach by Mitra et al

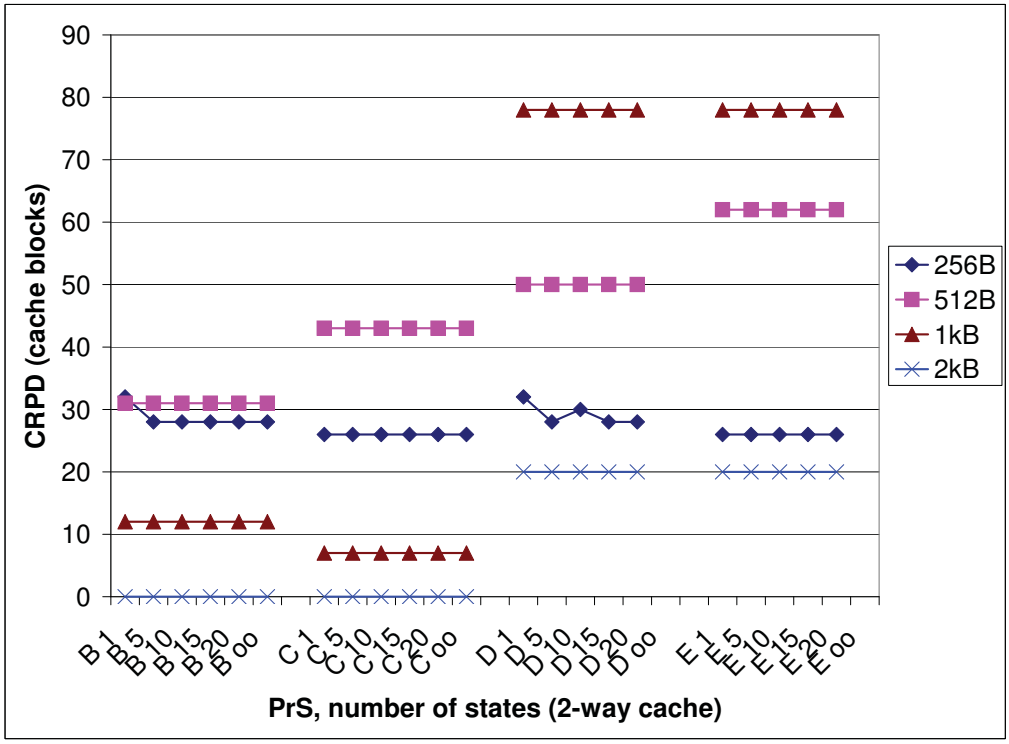

Figure 4.11. Impact of number of cache states.

[89]. In all the other cases, the parameter z corresponds to the bounded number cache states.

Figure 4.11 shows in most cases no improvement of the *crpd* estimation. The increase of CRPD in the scenario D for 256B cache at $z = 10$ is due to the ambiguity of the merging operation: If for different cache states, the metric will yield the same value and the CRPD of the merged states are different, then it can occur that a higher number of cache states can result in a higher CRPD bound. This is exmpained by an example for a direct mapped cache with two cache blocks. Assume a node in the control flow graph, where four cache states are reached and we are using metric d_2, as defined in equation 4.14, which counts the number of different cache blocks. The cache states shall be defined as follows:

$$c_1 = [\emptyset, \{m_1\}] \quad c_2 = [\emptyset, \{m_2\}] \quad c_3 = [\{m_3\}, \emptyset] \quad c_4 = [\{m_4\}, \emptyset]$$

Then, $d_2(c_i, c_j) = 2$ for all pairs of cache states $c_i\, c_j$ $(i \neq j)$. If $z = 2$, we could choose to merge c_1 and c_2 and in a second step c_3 and c_4 resulting to

$$\text{choosing } z = 2: \quad c_{1,2} = [\emptyset, \{m_1, m_2\}] \quad c_{3,4} = [\{m_3, m_4\}, \emptyset]$$

The preemption delay (CRPD) would in both cases be one, provided that both blocks are removed by the preeempting task. On the other hand, if we now choose $z = 3$,

we could merge c_1 and c_4, because the metric results the same value for all states, resulting to the three cache states:

$$\text{choosing } z = 3: \quad c_{1,4} = [\{m_4\},\{m_1\}] \quad c_2 = [\emptyset,\{m_2\}] \quad c_3 = [\{m_3\},\emptyset]$$

Thus the CRPD would be two cache blocks, provided that both blocks are removed by the preempting task. This change depends on the selected metric. If a pure decreasing CRPD value is desired, the definition of the metric must be improved to reflect the CRPD (for example a better metric might consider the number of non-empty cache blocks to avoid this non-monotonic behavior).

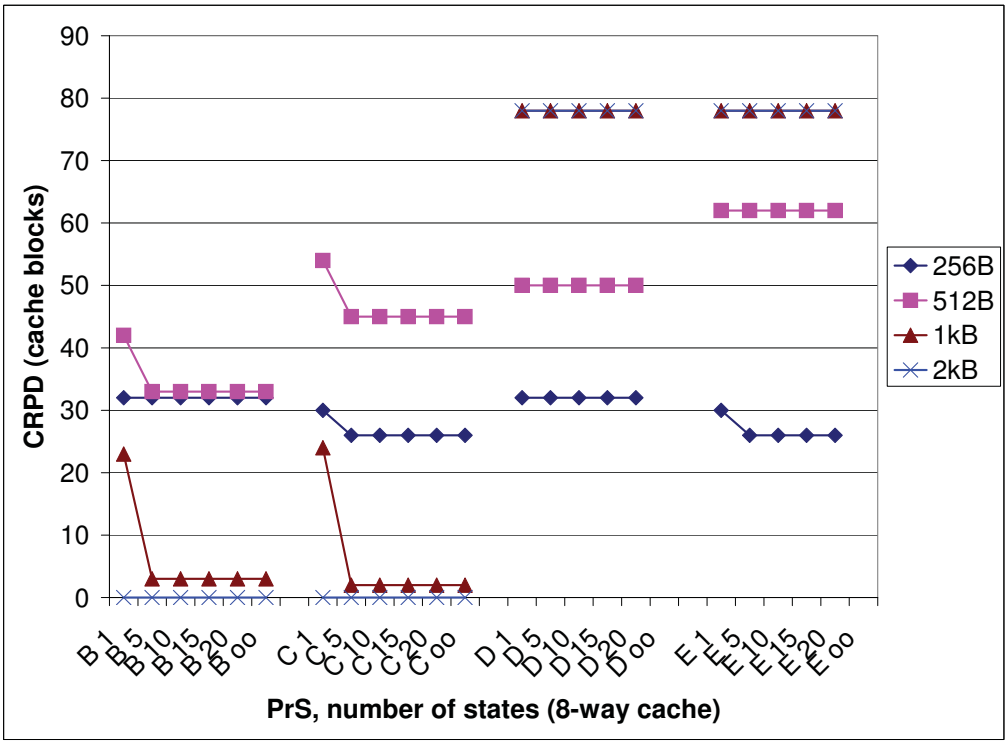

Figure 4.12. Impact of cache size.

For a higher associativity, figure 4.12 shows an improvement at already a small number of states. For PrS D and E no value for $z = \infty$ could be calculated because the memory consumption was too large (see figure 4.15). Note, the *crpd*-value for $z = 20$ is the same as for $z = \infty$ for all cases.

Figure 4.13 shows that for increasing associativity the analysis results are very similar.

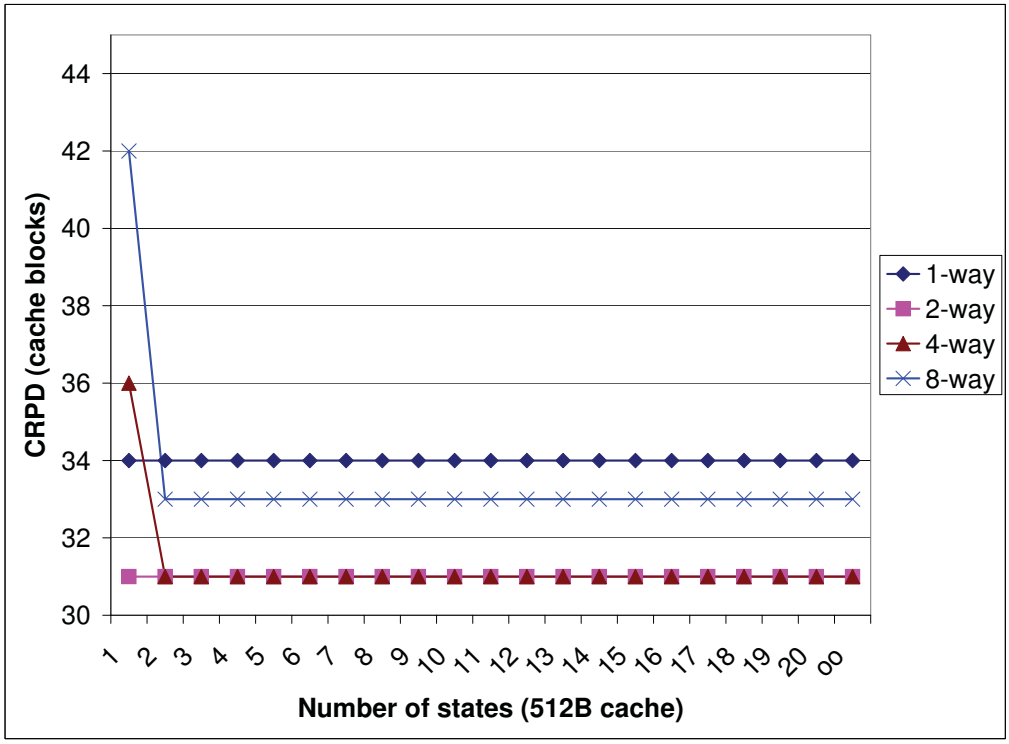

Figure 4.13. Impact of associativity.

4.6.2 Analysis time and memory consumption

In this subsection, the performance of the analysis framework itself is evaluated in terms of analysis time, memory consumption and other statistical measures. The analysis framework was executed on a 1.7 GHz machine with 4GB RAM.

Figure 4.14 shows the analysis time in seconds for a typical setup (e.g. 1-way associative cache) for different cache sizes and preemption scenarios. Note the logarithmic scale. This highlights the exponential growth in running time (PrS C took several hours). Only for the smaller benchmark, we could execute the analysis for $z = \infty$, no results could be obtained for PrS D and E. For caches with a higher associativity the analysis times are slightly higher (not shown in diagram).

Figure 4.15 shows the memory consumption of the analysis framework for a typical setup. While the memory consumption is about 3 to 100 MB for $z = 1$ to $z = 20$, the memory consumption for the larger task τ_5 in PrS C is in the range of Giga bytes for $z = \infty$; for PrS D and PrS E no results have been obtained because too many cache states were necessary.

Figure 4.16 and figure 4.17 show the maximum and average number of memory blocks in each set, as defined in section 4.3.4. While there exists a higher number of memory lines (about 20 for PrS E), the average number of memory blocks is between

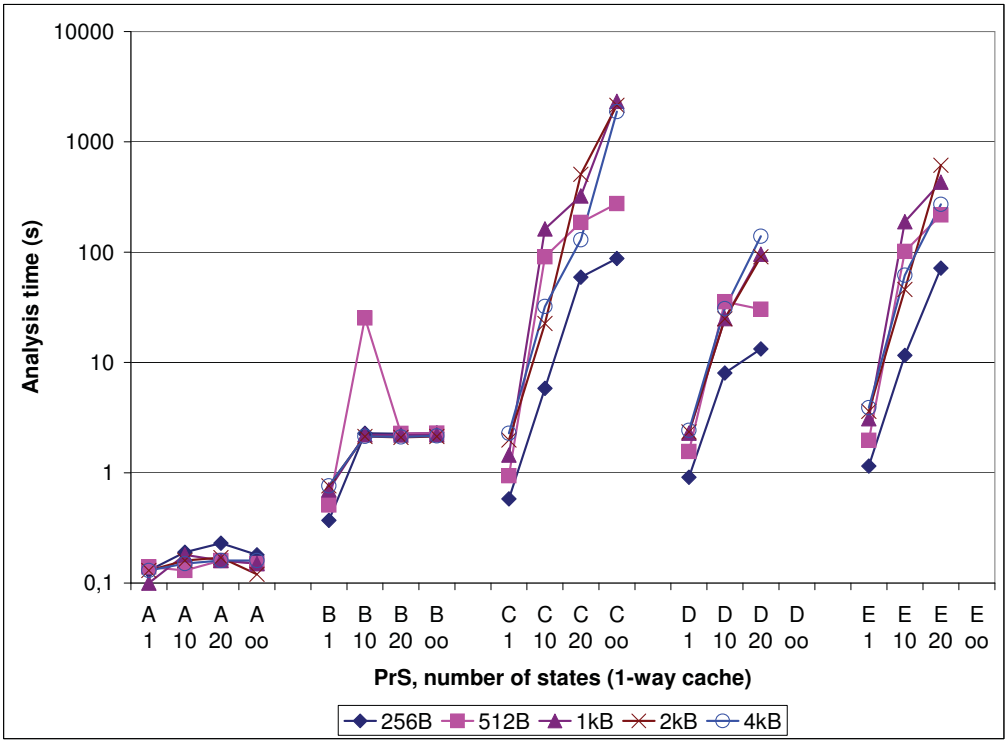

Figure 4.14. Analysis time.

one and two for all benchmarks and cache sizes. This has been checked for all setups, but cannot be shown here. These results give strong evidence that the number of X can be considered as a small constant for the timing complexity analysis as suggested in section 4.3.4.

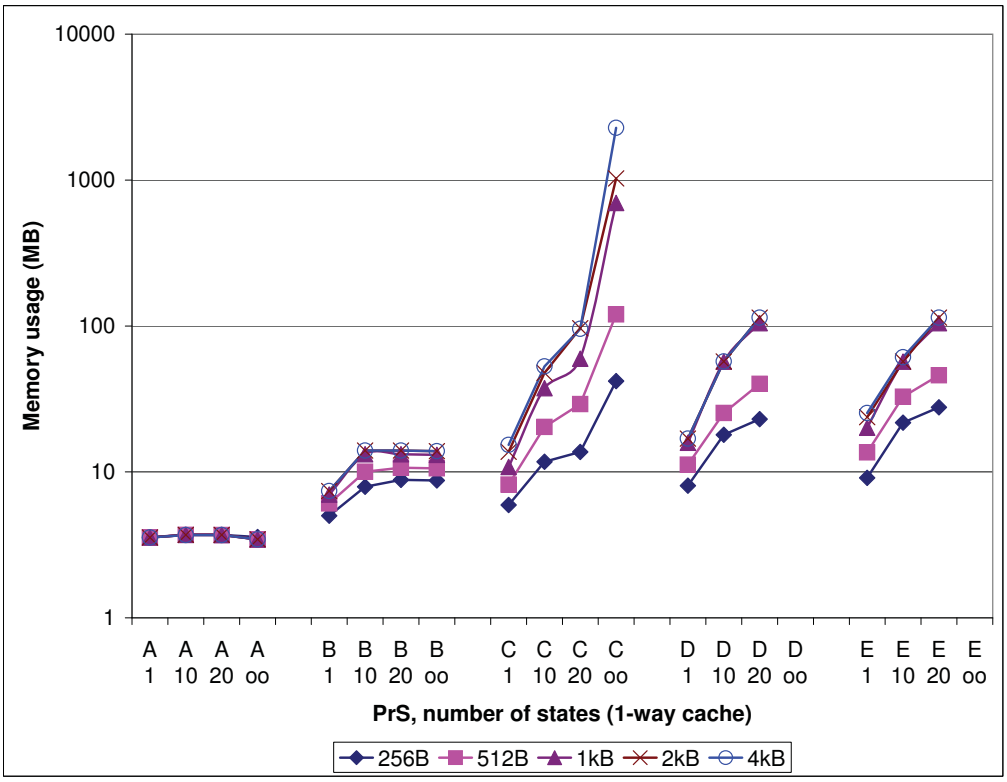

Figure 4.15. Memory consumption.

4.6.3 Response time analysis

Finally, we integrate the scalable CRPD estimation into a response time analysis as proposed in [113]. It reveals the impact of the cache related preemption delay on the total response time of a task. Our approach is compared to the approaches by Busquets-Mataix et al. [19] and Petters [92]. Busquets assumes that all used cache blocks of the preempting task are removed. However in their experiments they assume that the entire cache is flushed. In the following experiments we use the actual number of used cache blocks of the preempting task, which is computed by the global data flow analysis. Petters' presentation is based on the number of useful cache blocks. However the author only assumes a fixed percentage of cache content to be useful without performing any analysis. Again, because we have the total number of useful cache blocks available from global data flow analysis we use this number in Petters' computation. The approach by [67] needs an exponential number of equations for the ILP formulation and a re-implementation would have been too time-consuming for comparison purposes.

We compute the worst case response time for the task set $\tau_1, \tau_2, \tau_3, \tau_4$ with τ_1 as highest priority task and τ_4 as lowest priority task. The execution time of the whetstone benchmark was much greater than the other four, which is why we left it out.

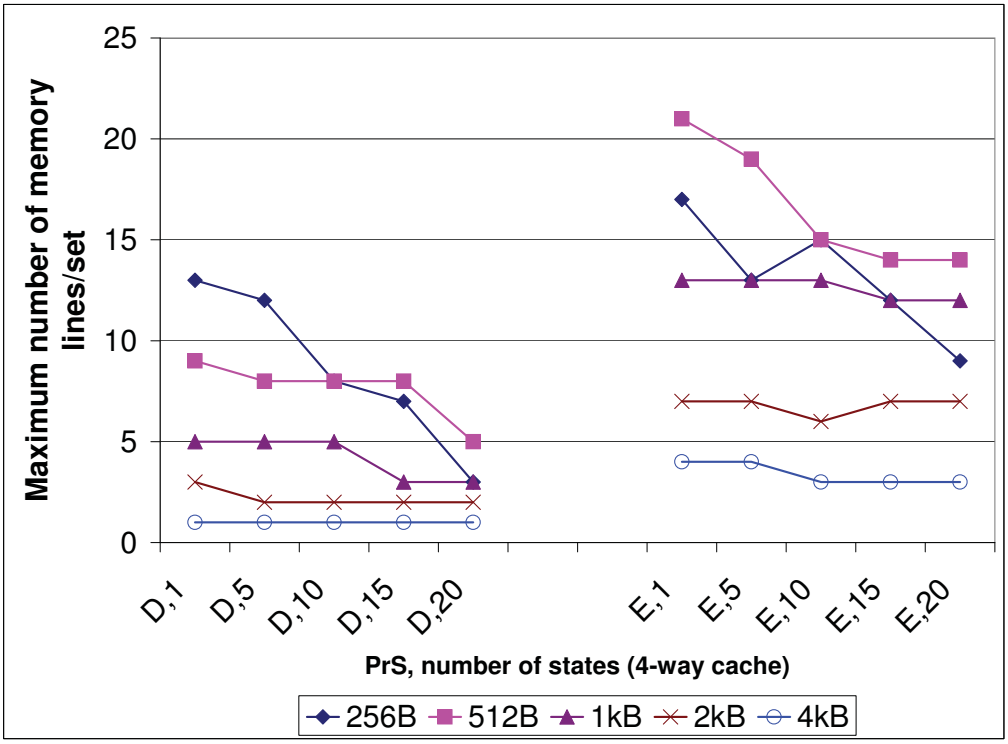

Figure 4.16. Maximum number of memory blocks per set.

Figure 4.18 shows the total preemption delay in clock cycles during the entire schedule for several cache sizes for a 4-way set associative cache. Compared to Busquets the data flow analysis with scaling factor 15 shows an improvement of 57%, 35%, 22% and 31% for 256B, 512B, 1KB and 2KB cache respectively. Compared to Petters our analysis shows an improvement of 39%, 43%, 59% and 70% for the given cache sizes.

The response time of task τ_4 is shown in figure 4.19. This value includes the preemption delay (CRPD) of figure 4.18 as well as the core execution times of each task and the time of higher priority tasks according to equation 2.3. The total preemption delay is calculated by the previously developed cache-aware response time analysis [113].

The vertical scale shows the response time in percentage of the response time that was calculated by Busquets approach. The reason is that the response times for different cache sizes are significantly different, such that a linear scale would be inappropriate. The value for Busquets is zero in all cases, because all other values were normalized to it. Nevertheless, we kept it in the figure to compare it to our results.

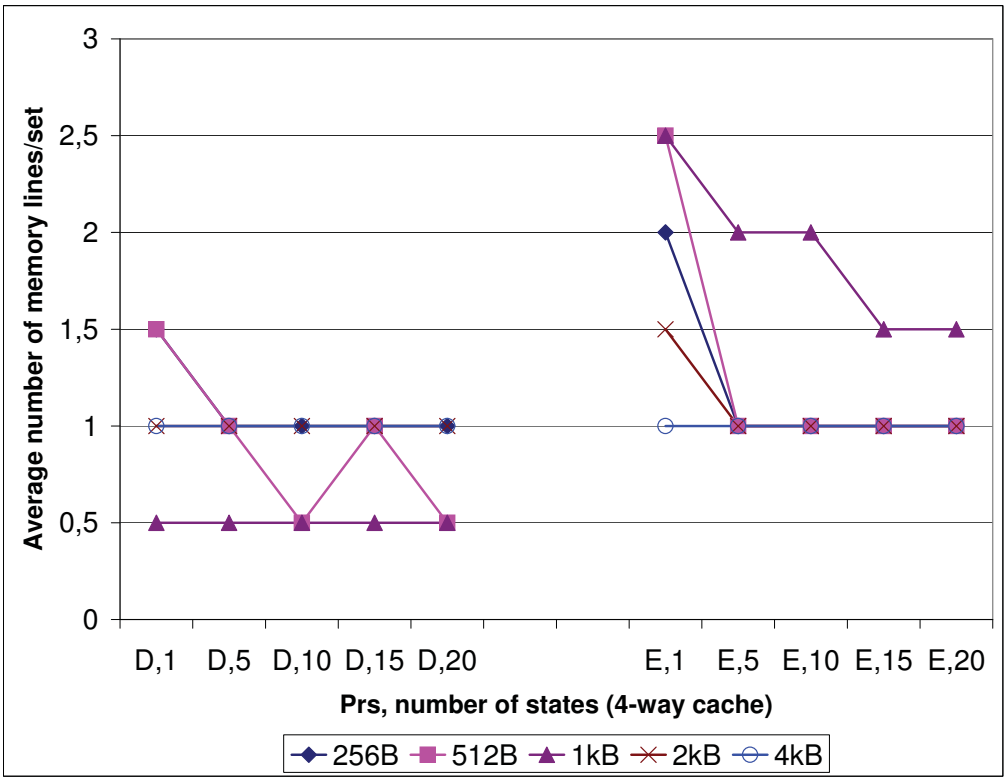

Figure 4.17. Average number of memory blocks per set.

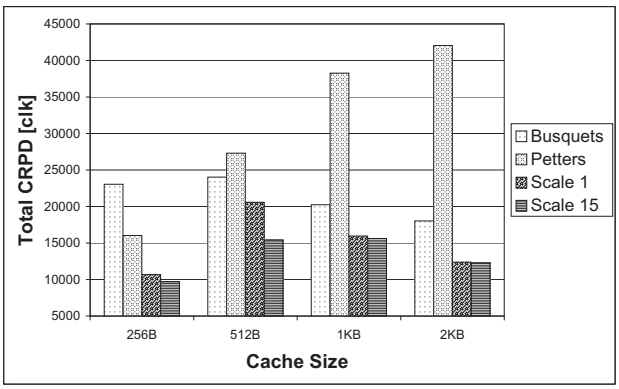

Figure 4.18. CRPD in clock cycles (clk) during the entire response time of τ_4.

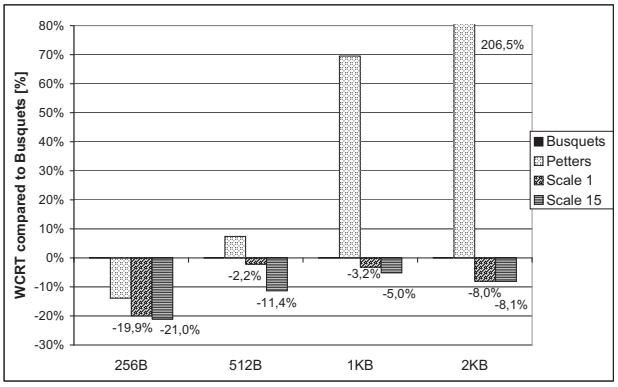

Figure 4.19. CRPD and core execution time (response time) for τ_4.

Besides Busquets, the results in Petters' approach, the set-based approach with scaling factor 1 and scaling factor 15 are depicted. The analysis precision for scaling factor 15 improves between 5% for 1KB cache and 21% for 256 Bytes cache.

4.7 Conclusion

Cache memory introduces unpredictable interference to task execution in real-time computing systems with preemptive scheduling. In this section we have proposed a scalable precision cache analysis for direct mapped and associative instruction caches. Then, we applied these results to a previously developed cache-aware response time analysis. The experiments have shown that a high analysis precision is already reaches for a small number of cache states, hence with a low time complexity. The approach is well suited for early design space exploration as well as for highly accurate real-time performance verification.

Assumptions and Limitations

Data caches are not considered. A timing analysis for data caches has been proposed e.g. in [133] [99] [112] and a preemption delay analysis for data caches in [98].

The approach is based on the control flow graph of a task which is constructed from the source code. Often software is secured by IP rights and is not available in source code. In this case, an a control flow graph of the application with associated instruction addresses could be an interface to our cache analysis technique.

As the control flow graph is used to represent the task execution, we only consider in-order issuing single-processors and constant cache miss penalty. Timing analysis of out-of-order pipelines and parallel resource allocation can lead to timing anomalies [105] and are more difficult to analyze.

Chapter 5

DATA CACHE ANALYSIS

5.1 Introduction

While processor speed is steadily increasing, main memory access time remains slow. Caches can significantly reduce the average memory access time, leading to a total shorter execution time. Timing behavior is becoming a serious problem in many embedded systems. In soft and hard real-time systems, timing guarantees are necessary to verify the functional behavior as well as to efficiently use hardware resources.

Current practice is to use cache simulation to determine the typical timing behavior, which is unsafe because not all program paths can be covered. A full coverage would require an exponential number of test data and would be too time consuming. Therefore, only a subset of all program paths is tested. An alternative approach is static timing analysis that delivers safe bounds of the worst case execution time. In the last decade, many techniques have been proposed for tasks running on a single processor architecture with a complex design, including pipelines [106], caches [71], and branch predictors [15].

There are several approaches to make caches more predictable and efficient. One approach is to partition the cache sets and to reserve these partitions for individual tasks [72]. The advantage is that cache lines do not have to be reloaded after interrupts and between consecutive executions of the same task. Also, cache behavior becomes (partly) orthogonal for tasks and, therefore, more predictable. Task layout techniques are suggested in [28], which aim at minimizing the inter-task interference in the instruction cache. Another approach is to lock frequently used cache blocks. Such techniques have been investigated in [62] [25] [97]. These approaches increase area and power cost as they require larger caches and main memory to become ef-

fective. Therefore, heterogeneous memory architectures with caches and scratch-pad SRAM have been introduced, like TriCore, where the scratch-pad can hold frequently used cache blocks. Compiler techniques for such architectures have been proposed by [90]. While cache partition and lock strategies are certainly a very useful add-on to improve cache predictability and efficiency, they do not solve the general cache analysis problem in which all tasks share the entire cache.

For a single task execution, the timing behavior for instruction caches has been extensively studied [71] [141]. However, data cache behavior has often been restricted in static worst case timing analysis. Previous approaches have focused on predictable memory access pattern, for example [38] [133] [99], or have simplified the timing behavior for input dependent memory accesses [78] [36].

The contribution is a worst case timing analysis for data caches that classifies memory accesses as predictable and unpredictable. For unpredictable memory accesses we propose a novel timing analysis framework that tightly bounds the impact on the existing cache contents as well as the number of cache misses for unpredictable memory accesses itself. For predictable memory accesses, we propose a local cache simulation and data-flow techniques. As a second contribution, we describe an implementation of the analysis framework. Finally, we compare the new approach with a previous approach and cache simulation.

The key benefit of the proposed approach is a re-target-able analysis framework to verify the timing behavior of data caches that considers input dependent memory accesses. The framework can be used for design space exploration as well as system level optimization.

5.1.1 Related work

In early work by [60] a heuristic argument of the *pigeon-principle* is proposed for direct mapped data caches. We explain the principle by an example where a memory access to an array occurs in a loop with a maximum of 100 iterations. Supposing that the array has ten elements, it can be concluded that 10 cache misses and 90 cache hits will occur given the following assumptions: 1.) the entire array fits in the cache, 2.) the cache is direct mapped and 3.) none of the array elements are replaced during program execution. Because none of the array elements are replaced there can be at most 10 (compulsory) cache misses all other have to be cache hits.

An integer linear problem (ILP) formulation was proposed by [70] for direct mapped data caches. A cache conflict graph is constructed in which each memory address corresponds to a variable in the equation system. Because the complexity scales with the

data size of arrays, is is unclear how efficient this approach will be for larger arrays. The dependency of index variables and array ranges are assumed to be given. This restricts the application domain to fully predictable array accesses. Unfortunately, no results from experiments are presented.

Abstract interpretation is used to predict data cache behavior in [36]. A persistence analysis determines the maximum number of cache blocks that remain in cache. This is used to calculate a lower bound on cache hits. Value analysis is used to calculate the possible range of array accesses. For an unpredictable array access, it is assumed that all cache blocks of an array are accessed, which replace many other cache blocks. This means, that a single access to an unpredictable array is modeled as if all elements of the array would have been accessed. However, only one cache block can be replaced. The method is only described in theory without providing experimental results.

Data flow analysis techniques have been applied to data cache analysis by [136]. The address range of data references is performed on low-level representation after code generation and all optimizations. Unknown data references are not considered and array ranges would have to be annotated by the user.

A symbolic simulation technique is proposed by [78] to detect predictable and unpredictable memory accesses. Unpredictable data structures are tagged as non-cache-able and consequently always require a cache miss. The main drawback is that input dependent memory accesses are classified as non-cache-able and are treated as always cache miss.

A different approach is to use cache miss equations(CME). Introduced by [38] and developed by [133], these CME compute re-use vectors to calculate cache accesses within loops. It is assumed that all array accesses are affine functions of loop induction variables and that loops are perfectly nested. In [99] the CME-framework is extended to consider scalar variables and to allow more sophisticated loop structures. However, the CME frameworks always assume that array index variables do not depend on input data.

5.1.2 Principle of analysis

The main open problem in data cache analysis is the timing behavior of unpredictable memory accesses. Trace-based simulations would be too time-consuming to cover the worst case. Existing data-cache frameworks exclude input dependency, such as the CME frameworks [38][133][99], or classify input dependent memory accesses as non-cache-able [78]. In [36] the impact of unknown data accesses on the cache contents is simplified by assuming that all elements of an array are loaded to

the cache for each array access. However, only one array element will be loaded to the cache.

To reduce this overestimation, we classify array accesses as predictable and unpredictable memory accesses, if the index expression can be statically computed or not. Each scalar variable is predictable, because the memory location is fixed. An unpredictable memory access has two influences on cache behavior. First, it has an impact on the current cache contents by possibly replacing some other cache block. Second, the access itself requires an additional cache miss, if the element is not in the cache.

For the first impact, a cache miss counter C_m is defined for each unpredictable memory access. The counter represents possible replacements of some cache blocks. We refine this idea, by incrementing C_m only if a useful cache block can be replaced by some element of the array. A useful cache block is a cache block that contains a data variable that is used a second time after the unpredictable access occurred. Only these cache blocks will require an additional cache miss in the future execution.

For the second impact, we bound the number of cache misses by comparing the array size and the execution counts as suggested in [60]. The key difference in this approach is the computation of the set of persistent cache blocks. In [60] no memory blocks are allowed to be replaced. We apply the pigeon-hole principle to those arrays whose elements are all persistent (e.g. are never replaced by other memory blocks).

For all predictable memory accesses we use local cache simulation and global data flow analysis. Finally, we set up an integer linear problem (ILP) to consider predictable and unpredictable memory accesses. Additional constraints for the pigeonhole principle are added to the ILP formulation.

5.1.3 Framework overview

The analysis framework is shown schematically in figure 5.1. The analysis is based on the source code of an application. An intermediate representation, the abstract syntax tree, is generated from the source code. Based on this representation, data dependency analysis identifies single data sequences and a control flow graph is constructed (section 5.2). In a second step, the actual memory addresses are extracted using the linked object code and are mapped to the control flow graph (section 5.3). Then, follows a cache behavior analysis for predictable and unpredictable data accesses (section 5.4) and, finally, the timing analysis computes the worst case program execution path (section 5.5).

Experimental results are provided in section 5.6. and section 5.7 concludes this section.

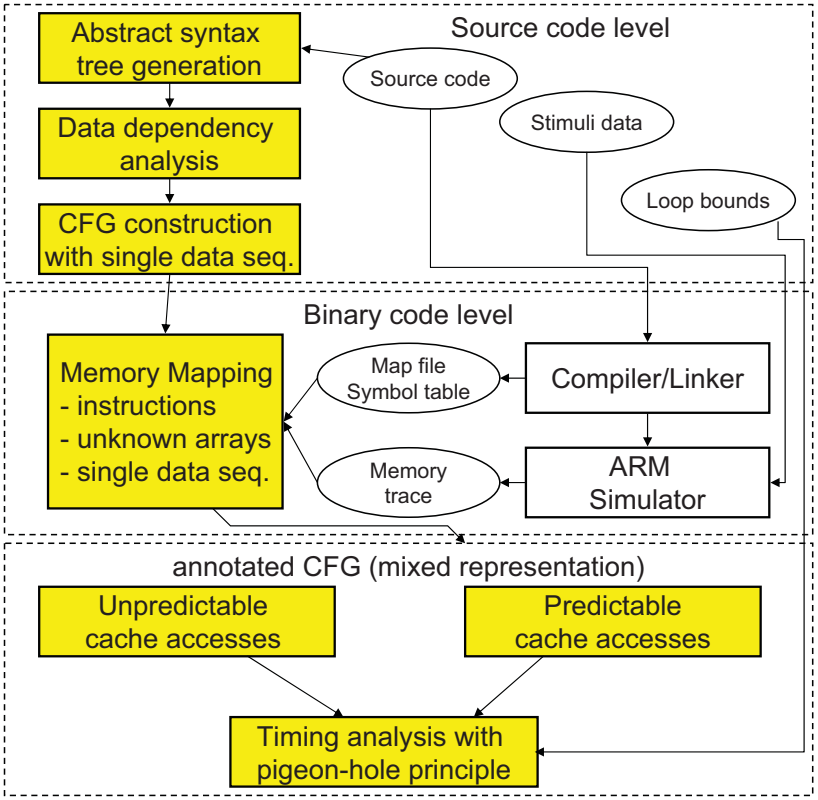

Figure 5.1. Overview of analysis framework.

5.2 Data dependency analysis

Data cache behavior has been simplified in worst case timing analysis because of unpredictable data accesses. A single instruction might access different memory locations during the execution of a program. In such cases, it is already difficult to predict which memory locations are accessed, and it becomes very complicated to decide whether such accesses are cache hits or misses. However, sometimes memory locations, for example array elements, are accessed in a predictable order. Two conditions are necessary: first, an input data independent control flow and, secondly, input data independent memory accesses.

We say that a program segment contains a *single data sequence* (SDS) if it contains only input independent control structures and input independent memory accesses. A program segment is a sequence of basic blocks, in which basic blocks are the smallest entity of a program [3]. An example source code is shown in figure 5.2. The `for`-loop in lines 11-12 contains such a SDS, because the condition $i < 100$ does not depend on input (or global data) and the index variable of $b[i]$ does not depend on input data. On the other hand, the program segment in lines 6-10 do not contain a SDS, because the

condition $a < 2$ depends on global variable a. Also, the array access in line 9 depends on the global variable a.

```
 1 int a;      // global variable
 2 int b[100]; // global array
 3 int main(){
 4     int i;
 5     int k;
 6     if (a<2){
 7      k = 10;
 8     } else {
 9      b[a] = 0;  // input dependent access
10     }
11     for (i=0;i<100;i++){
12      b[i] = i;  // input independet access
13     }
14     return 0;
15 }
```

Figure 5.2. Source code example.

In this section we present a methodology to identify SDSes. First, we discuss the proper analysis level in section 5.2.1. Then, we describe how to identify input dependent control structures in section 5.2.2 and input dependent memory accesses in section 5.2.3. Finally, we construct a control flow graph with SDS in section 5.2.4.

5.2.1 Analysis abstraction level

Single data sequences require input independent control structures and memory accesses. These properties can be computed by data dependency analysis which could be performed either on source code or on object code level. A source code level analysis would be based on a control flow graph, where nodes contain assignment statements in high level language. The drawback is that actual memory addresses cannot be derived because instruction set of the assembly language, linking information, compiler options and memory layout are not available. The exact memory mapping has to be derived in a second step from the actual binary by disassembling the object code. For a correct source code correspondence, optimizations beyond basic block level are not allowed. An approach to analyze input-dependency for high level programming language has been proposed by [39].

As an alternative, data dependency analysis could be executed on object code level where the actual memory layout information is available. From the object code a control flow graph can be generated, as suggested in [36][136][70]. However, data dependence analysis is difficult, because of indirect register addressing and the de-

pendence on the instruction set. For example the ARM instruction set contains conditional instructions, which makes value analysis complicated. Second, such an analysis is limited to the analyzed instruction set only, for every new processor a new data dependency module would be necessary.

We chose the first alternative, in which the data dependency analysis is done on source code level because it is hardware independent and value analysis is simpler than on object code level. The lack of actual memory addresses is resolved by annotating the control flow graph with the memory addresses using cache simulation in a second step (section 5.3.

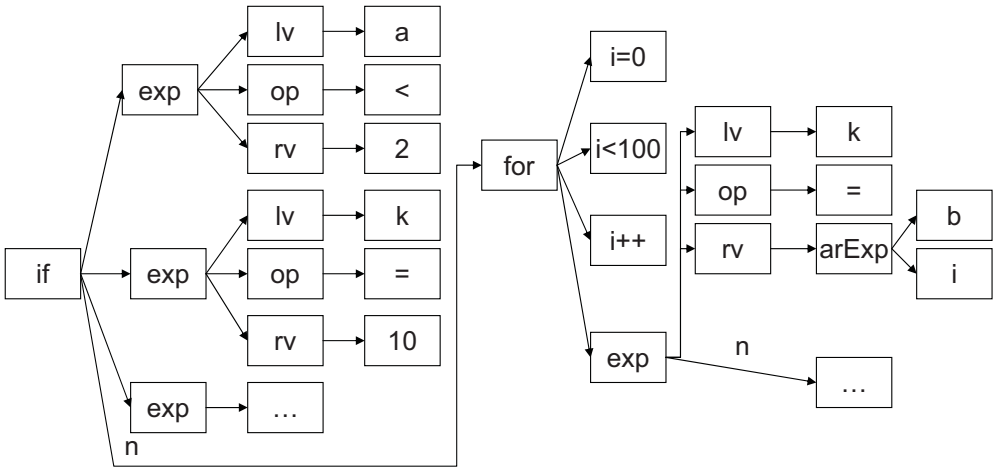

Figure 5.3. Abstract syntax tree.

For data dependency analysis, an abstract syntax tree (AST) [3] is generated from the source code program. The abstract syntax tree is used because it keeps direct correspondence between program segments and the control flow hierarchy [141]. An AST contains hierarchical nodes representing statements and edges that represent the hierarchy of statements. For a part of the source code of figure 5.2 a simplified abstract syntax tree is shown in figure 5.3. Edges connect a node with a lower-hierarchy node, for example, to a `then`, an `else` expression, or connect a node to the next node (n) on the same hierarchy level. The `for` statement has edges to the init-, condition-, body-, update and next-expression. An assignment expression contains a left value (`lv`), operator (`op`) and a right value (`rv`). An array expression (`arExp`) has two sub-expressions: the name of the array (`b`) and the index expression (`i`).

5.2.2 Program path classification

The execution path of a program depends on input data which makes it difficult to predict precise worst case timing bounds. But often a program contains only a single feasible program path. An example is a fast Fourier transformation (FFT) or a FIR filter algorithm. In such structures the control flow only depends on the internal state, but not on input data. For example, a loop with several `if-then-else` statements that only depend on the loop iteration variable. In this case we can use trace-based simulation. In previous work we have presented a methodology to identify single feasible paths (SFP)[139] and proposed a timing analysis for instruction caches [141].

An SFP is determined by analyzing the conditions of each control structure in the abstract syntax tree. A control structure that does not contain other hierarchical control structures is called a SFP, if the condition does not depend on input data, otherwise it is called a multiple feasible path (MFP). A control structure with an input independent condition that contains substructures with MFPs is also classified as a MFP.

The input dependency of conditions is determined by a symbolic simulation algorithm on the abstract syntax tree. Each variable is either classified as input dependent or input independent. As the algorithm traverses the tree, the left value of an assignment is classified as input dependent if at least one variable on the right value is input dependent. A symbol table stores the classification of each variable. If a node has several predecessors like a `then` or an `else` branch, a variable is also classified as unpredictable if it contains different values in the symbol tables of the predecessor nodes. Loops are analyzed twice to propagate assignment-statements to higher level hierarchic nodes.

A control structure is input dependent if it contains at least one input dependent variable in the condition, otherwise it is classified as input independent (SFP). Figure 5.4 shows a control flow graph (CFG) on the left side, where the loop nodes are a SFP, because the loop condition does not depend on input data.

5.2.3 Memory access classification

The prediction of memory accesses for a data cache is more difficult than for instruction cache behavior because a single instruction may access a range of memory locations. To detect single data sequences, data cache memory accesses are classified as predictable and unpredictable. Memory accesses by scalar variables and predefined array accesses are classified as predictable. Otherwise, accesses are classified as unpredictable.

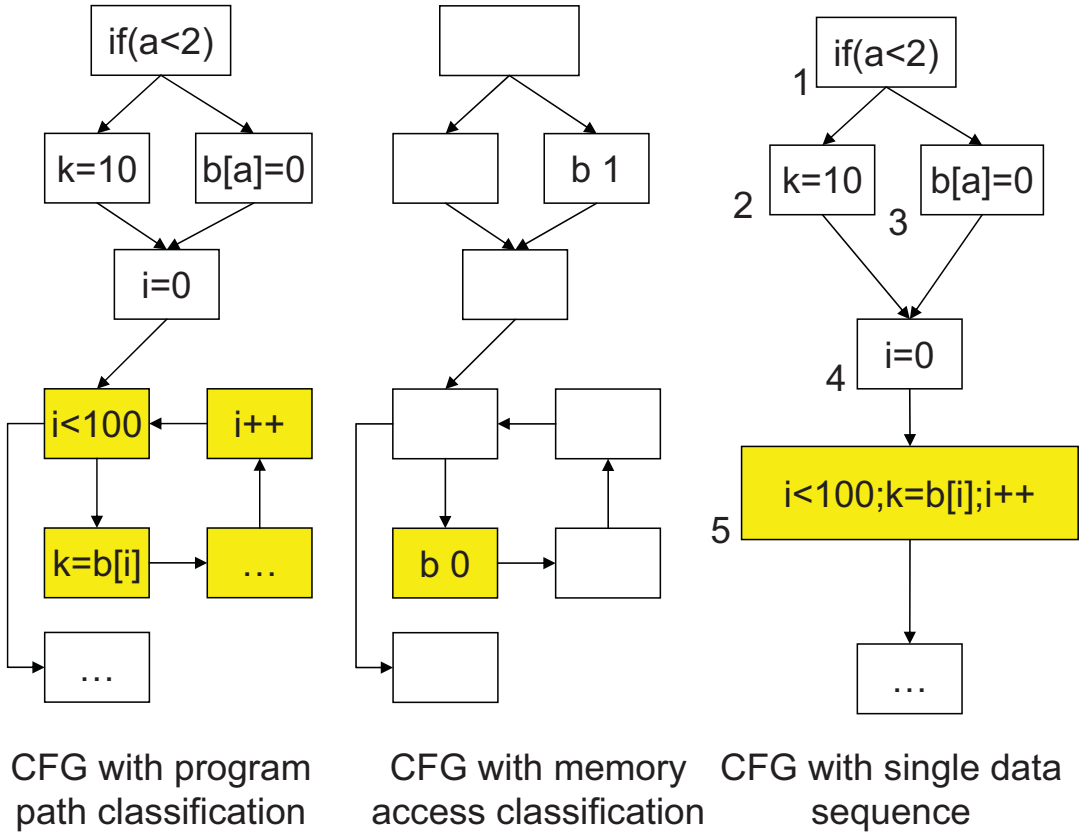

Figure 5.4. CFG in different analysis steps.

In section 5.2.2 we described a symbolic simulation algorithm on the abstract syntax tree to determine the input dependency of control structures. A similar algorithm can be used to determine the input dependency of memory accesses.

Initially, all scalar variables are classified as predictable, because the memory address is constant. Then, the analysis proceeds as described above, propagating the input classification of each variable in the abstract syntax tree.

Instead of using the condition expression, the index expression of the array access expression determines if the memory location is input dependent or not. We define a memory access by an array as predictable if all variables of the index expression are input independent, otherwise it is classified as not predictable.

For example, the array expression in line 9 of figure 5.2 is unpredictable, but the one in line 12 is predictable. The output of the data dependency analysis is a CFG constructed from AST which is shown in figure 5.4 (middle). For each array variable a boolean variable signals the data dependency. For example, the assignment `b[a]=0` is data dependent (`b 1`) but the assignment `k=b[i]` is not.

For each node a tree traversal algorithm on the AST identifies whether an array access is predictable or not. Furthermore, the c-lines of each expression are annotated.

5.2.4 Single data sequence construction

A single data sequence (SDS) is defined by the program path classification (section 5.2.2) and memory access classification (section 5.2.3): A single feasible path that contains only predictable data structures is a SDS. An example of a SDS is shown in figure 5.4 (right).

At the end of this analysis step, a control flow graph is available where nodes contain single data sequences or unpredictable array accesses. In each node it is specified whether an array access is predictable or not. Our implemented algorithm stores the c-lines as well, but for a better understanding of the methodology we omitted the c-lines in this presentation.

5.3 Memory address mapping

In section 5.2, single data sequences have been constructed based on source code. The actual memory addresses are necessary for a data cache analysis. In this section, we describe how memory addresses are computed.

The methodology consists of three steps. First, instruction addresses are mapped to the CFG (section 5.3.1). Second, the memory access trace of instruction and data addresses is generated by a processor simulator (section 5.3.2). And finally, the data accesses of the memory trace are mapped to the nodes of the CFG (section 5.3.3).

5.3.1 Instruction address mapping

In the first step, instruction addresses are mapped to the control flow graph. A `map`-file is constructed that contains a table of memory addresses of assembly instructions and the corresponding c-lines. In the proposed data-cache analysis methodology we assume that compiler optimizations beyond basic block boundaries are not used. With a debugging tool, e.g. `addr2line`[1], the correspondence of memory addresses to c-lines can be computed. Then, instruction addresses are mapped those nodes in the control flow graph which have the same c-line. Figure 5.5 shows a part of the map file for the example source code.

[1] Binutils tool suite http://www.gnu.org/software/binutils/

```
address  c-line     address  c-line
80a8     9          80e8     11
80ac     9          80ec     12
80b0     9          80f0     12
80b4     9          80f4     13
80b8     10          ...
...
```

Figure 5.5. Memory map file.

5.3.2 Memory trace generation

Trace generation is used to collect the addresses of the predictable data accesses. Because the addresses are constant, profiling with branch coverage is sufficient. The memory trace of a software program is computed by an off-the-shelf processor simulator. For this study, we use the RealView development suite [6], which is a cycle accurate processor simulator. In general, it suffices to use a functional simulator that outputs the memory trace. The simulator comes with the option to trace instruction and data addresses.

```
IT 000080A8 e59f0050 LDR   r0,0x8100
MNR4____ 00008100 00008414
IT 000080AC e5900000 LDR   r0,[r0,#0]
MNR4____ 00008414 00000000
IT 000080B0 e3500002 CMP   r0,#2
IS 000080B4 aa000001 bge   0x80c0
IT 000080B8 e3a0200a MOV      r2,#0xa
...
IT 000080EC e59f0010 LDR   r0,0x8104
MNR4____ 00008104 00008418
IT 000080F0 e7801101 STR   r1,[r0,r1,LSL #2]
MNW4____ 00008418 00000000
...
IT 000080EC e59f0010 LDR   r0,0x8104
MNR4____ 00008104 00008418
IT 000080F0 e7801101 STR   r1,[r0,r1,LSL #2]
MNW4____ 0000841C 00000001
...
```

Figure 5.6. Memory access trace.

An example of a memory access trace is shown in figure 5.6. The first letters denote the memory access type: `IT/IS` denotes instruction taken/skipped, `MNR4` denotes a memory access read (4 byte) and `MNW4` denotes a memory access write (4byte). The

second column states the address fetched, the third the hex code and the rest of the line is the mnemonic assembly instruction.

Sufficient test patterns for a complete branch coverage are necessary. Branch coverage of a program means that each branch is executed taken and not-taken. The number of test patterns of a full branch coverage scales linearly with the number of branches while a full path coverage requires exponential number of test data.

5.3.3 Data address mapping

The objective is to map data accesses to control flow nodes. The instruction addresses were mapped to the nodes and can be used as identifiers. If one or more data addresses follow an instruction address d_i in the memory trace, then these data addresses are mapped to the nodes that contains d_i. We assume an in-order execution model of the processor. Otherwise, it would be more difficult to identify the instruction requesting a data address. For predictable data structures, especially in single data sequences, all data accesses are mapped to the corresponding nodes.

For unpredictable memory addresses this is not possible because a different test pattern would result in a different access pattern. Therefore, we have to assume that all memory locations are possible. The symbol table of the disassembled binary contains the start address and the size of each variable, for example the array `b[]` has the start address `8418` and a size of `0x190h`. This information is used to compute the range of possible memory accesses for unpredictable data accesses and is mapped to the corresponding nodes.

Table 5.1 summarizes the address mapping of instruction as well as data accesses for two nodes. The last column states whether the node is a single data sequence(SDS) or not. Note that the unpredictable array access in B_3 is annotated by its start address and its size. For the SDS in B_5, the exact simulation trace is used. Because of space restrictions, only a small part is shown and the array access is annotated with the actual sequence of data addresses.

Table 5.1. Memory address mapping summary

B_{id}	c−line	I-address	D-address	SDS
3	9	80a8, 80ac 80b0 80b4	8100 8414 **8418/0x190**	no
5	11,12	80EC 80F0 $\cdots$ 80EC 80F0	8104 8418 $\cdots$ 8104 841C	yes

5.4 Cache behavior analysis

The key idea of our approach is the observation that an access to an unknown data structure can remove at most one cache block. While the previous approach by [36] assumes that all cache blocks of an unknown data structure are assumed to be loaded to cache, we present a novel method to overcome this pessimism.

It consists of three steps. First, the analysis of the impact of unpredictable data accesses on existing cache content (section 5.4.1), second, the analysis of data cache behavior of unpredictable memory accesses themselves (section 5.4.2), and finally, the analysis of predictable cache accesses (section 5.4.3).

5.4.1 Cache miss counter

In this section, we describe the influence of an unpredictable data access on existing cache content. Assuming that an array has the size of ten cache blocks, any of these ten cache blocks might be requested by an access to this array. Which cache blocks of the current cache contents could be possibly replaced? The approach by [36] assumes conservatively that all ten cache blocks are loaded to the cache and (possibly) replace ten existing cache blocks. However, at most *one* cache block is actually removed, but we cannot statically predict which one.

To overcome such overestimation, a miss counter $C_m(B_i)$ is defined at node B_i. Initially, the miss counter is zero. Whenever a cache access to an unknown data structure occurs, the miss counter is incremented by one. This miss counter represents the number of *additional* cache misses for (possibly) replaced cache blocks. Note that this miss counter does not represent the number of cache misses for unpredictable data accesses themselves. This will be described in section 5.4.2.

We give two algorithms to compute the cache miss counter. First, we could simply assume that `each` access to an unknown data address replaces a cache block. This is formalized in equation 5.1.

$$C_m(B_i) = |\{d| \quad \forall d \in Data(B_i)\}| \tag{5.1}$$

In other words, it is the cardinality of the set of unpredictable data accesses of node B_i. The set of data accesses is given by $Data(B_i)$ for basic block B_i. However, this would be too conservative. An additional cache miss is only required if a *useful* cache block is replaced. Useful cache blocks has been used by [66] [89] [111] to calculate the cache related preemption delay for fixed priority preemptive scheduling. A cache block is called useful if it may be available in basic block B_i (e.g. reaches this node) and may be accessed a second time via some outgoing path of B (e.g. is a live cache

block). Two data flow analysis algorithms are set up, one to compute the reaching cache blocks $RMB(B_i)$ and one to compute the live cache blocks $LMB(B_i)$ at node B_i. The intersection of these sets is the set of useful cache blocks $UCB(B_i)$. The concept of data flow algorithms are further explained in section 5.4.2.

We refine the calculation of additional cache misses by incrementing the miss counter $C_m(B_i)$ only if there exists at least one useful cache block in the range of an unpredictable data access. This is shown in Equation 5.2:

$$C_m(B_i) = |\{d|UCB(B_i) \cap range(d) \neq \emptyset \forall d \in Data(B_i)\}| \tag{5.2}$$

where $range(d)$ is defined as the set of addresses of array d:

$$\{address(d), \cdots, address(d) + size(d) - 1\}$$

5.4.2 Persistence analysis of unpredictable accesses

In case memory access are input dependent, heuristic arguments can be applied in a safe way. With the *pigeon-hole* principle, as described by [60], we can reduce the number of cache misses even though very little is known about data cache accesses. However, the three assumptions are very restrictive. In our novel analysis framework we also assume that the cache is direct mapped and that the entire array fits in the cache. These restrictions seem feasible. But the third assumption is too restrictive.

We will statically analyze by a persistence analysis which array addresses are never replaced . This persistence analysis checks that all elements of an array are still in the cache for all possible execution traces. The number of persistent cache blocks has been calculated in the approach by [36] with the conservative assumption that all cache blocks of unpredictable data structure are loaded to the cache. We use a similar technique, but with the goal to apply the pigeon-hole principle in section 5.5.2.

Persistent cache blocks can be computed by data flow algorithms [3]. These data flow algorithms have been used in preemption delay analysis by [66] [89] [111]. Given a control flow graph, and some property P, a data flow algorithm propagates the property P to all nodes.

For persistence analysis, as described in [36], we define P as the set of reaching cache blocks. An iterative algorithm is used to solve the following equations:

$$P_{in}[B_i] = \bigcap_{P \in pred(B_i)} P_{out}[P]) \tag{5.3}$$

$$P_{out}[B_i] = \{r \odot gen[B_i] | r \in P_{in}[B_i]\} \tag{5.4}$$

$$c \odot c' = \begin{cases} c' & if \quad c' \neq \emptyset \\ c & otherwise \end{cases} \tag{5.5}$$

The quantities $P_{in}[B_i]$ and $P_{out}[B_i]$ are computed for each basic block B_i. Initially, $P_{in}[B_i] = \emptyset$ and $P_{out}[B_i] = gen[B_i]$. The set $gen[B_i]$ is defined by the set of cache blocks that are loaded during the execution of basic block B_i by predictable and unpredictable memory accesses. For unpredictable memory accesses we assume that all cache blocks are accessed. We define P for the entire cache by considering each cache set separately. The $\odot$ operator replaces a cache block if $gen[B_i]$ is not empty, otherwise the cache block from some incoming path $P_{in}[B_i]$ is taken. We set $P[B_i] = P_{out}[B_i]$ when the fixed point is reached which contains all persistent cache blocks at B_i.

The persistence analysis has to be performed for each unpredictable data access d. The *gen* set for the node containing d is constructed by all possible addresses. During the data flow analysis, d is ignored in all other nodes. Otherwise, some memory addresses would be loaded to the cache and the persistence analysis would not deliver the worst case.

5.4.3 Predictable data accesses

Global data flow analysis is used to calculate the data cache behavior for predictable memory accesses. The iterative data flow analysis, as described in section 5.4.2, can be applied with the following modifications: The $gen[B_i]$ set is defined by the predictable data accesses only. The replaced cache blocks by unpredictable array accesses can be ignored because their potential interference is accounted by the miss counter $C_m(B_i)$. At the end of this analysis step, the number of cache hits $B_i(hits)$ and misses $B_i(misses)$ for each node B_i are computed.

5.5 Data cache timing analysis

After the cache access behavior for each node has been computed, we present now the last step of the analysis framework: the global timing analysis. Integer linear programming is used to calculate the worst case data cache behavior (section 5.5.1) for the whole control flow graph. Unpredictable data accesses are then accounted by applying the pigeon-hole principle (section 5.5.2. In section 5.5.3 we summarize the assumptions and limitations of the whole analysis framework.

5.5.1 Integer linear programming

Integer linear programming (ILP) is an established method to find the worst case execution path in timing analysis [71] [141]. Based on the control flow graph of a program, a linear optimization problem is constructed that maximizes the flow through the program. Each node B_i contributes a constant execution time c_i and a variable x_i denotes the execution count of node B_i. In the following we focus on the contri-

bution of data cache behavior and neglect instruction cache behavior. Therefore, our approach can be combined with any existing static timing analysis approach, e.g. that already considers instruction caches, pipelining and branch behavior. The objective function of the ILP is defined as:

$$max : \sum_{i \in B} c_i \cdot x_i \tag{5.6}$$

Where B denotes the set of all nodes of the control flow graph. A set of structural constraints is set up to model that the incoming flow of each node is equal to the outgoing flow:

$$\sum_{p \in pred(B_i)} e_{p,i} = x_i = \sum_{s \in succ(B_i)} e_{s,i} \tag{5.7}$$

where $e_{i,j}$ denotes an edge from node B_i to node B_j. The set of all predecessors is abbreviated as $pred(B_i)$, and the set of successors as $succ(B_i)$ for a node B_i. Additional constraints, e.g. for maximum number of loop iterations, can be defined by the engineer. The cache access time c_i for each basic block B_i is set to

$$c_i = c^{hit} \cdot B_i(hits) + c^{miss} \cdot B_i(misses) \tag{5.8}$$

where $B_i(hits)$ and $B_i(misses)$ denote the worst case number of cache hits and misses for basic block B_i of predictable data accesses, as computed in section 5.4.3. Cache hit and miss penalty time is denoted by c^{hit} and c^{miss} respectively. For unpredictable data accesses we apply the pigeon-hole principle.

5.5.2 Pigeon-hole principle

We extend the ILP to account for unpredictable memory accesses with the pigeon-hole principle [60], as described in s:dc:relatedwork. For a node B_i with a data dependent array d_k the execution count is separated into hits and misses. $Data(B_i)$ denotes the set of unpredictable data accesses.

$$x_i = x_{ik}^{hit} + x_{ik}^{miss} \quad \forall d_k \in Data(B_i) \tag{5.9}$$

For every *persistent* data access, which were computed in section 5.4.2 an additional in-equation regarding the pigeon-hole principle is added to the ILP:

$$x_{ik}^{miss} \leq range(d_k) \tag{5.10}$$

This in-equation reflects the fact that the number of misses is bounded by the maximum number of elements of the array since the array is persistent in cache. Then, the

objective function is modified to

$$max : \sum_{i \in B} \left(\sum_{d_k \in Data(i)} \left(c^{hit} \cdot x_{ik}^{hit} + c^{miss} \cdot x_{ik}^{miss} \right) \right) + c^{miss} \cdot C_m(B_i) \cdot x_i + c_i \cdot x_i \quad (5.11)$$

The second sum represent the cache behavior for unpredictable memory accesses (pigeon-hole principle) together with equation 5.9and 5.10. The additional cache misses regarding existing cache contents is modeled by term $c^{miss} \cdot C_m(B_i) \cdot x_i$ which multiplies the cache miss penalty c^{miss} with the cache miss counter $C_m(B_i)$ and the execution count x_i of the basic block B_i. The last summand $c_i \cdot x_i$ is the cache behavior for predictable memory accesses (equation 5.8).

Figure 5.7 shows an example ILP for the CFG of figure 5.4. We assume that the array b has 10 elements. Note, that the loop statement B_5 is a SFP, therefore, no loop bound is necessary.

max: $c_1x_1 + c_2x_2 + c^{hit}x_{31}^{hit} + c^{miss}x_{31}^{miss} + c^{miss}C_m(B_3)$
$+c_4x_4 + c_5x_5$
subject to

$x_1 = 1; x_1 = e_{1,2} + e_{1,3};$	// structural
$x_2 = e_{1,2} = e_{2,4}; x_3 = e_{1,3} = e_{3,4};$	// structural
$x_4 = e_{2,4} + e_{3,4} = e_{4,5}; x_5 = e_{4,5}$	// structural
$x_3 = x_{31}^{hit} + x_{31}^{miss}$	// pigeon-hole
$x_{31}^{miss} <= 10$	// pigeon-hole

Figure 5.7. Example ILP formulation.

5.5.3 Assumptions for data cache analysis

Our approach assumes a direct mapped data cache (write through, no-write allocate), with a constant cache hit and cache miss penalty and an in-order execution model of the processor. Further, we disable optimizations beyond basic block boundaries. We do not consider pointer arithmetics and dynamic data structures on the heap. For memory trace simulation, test patterns for full branch coverage have to be available. For the experiment, we supplied the test data manually. Branch coverage is often required for functional verification and should therefore be available for timing verification phase. The analysis framework does not support multiple function calls at the moment.

5.6 Experiments

This section presents results of the analysis framework for direct mapped caches.

Table 5.2. Benchmark description.

Task	Code	Data	C-Ln	Nodes	Name
τ_1	180	404	15	8	example1
τ_2	180	44	15	8	example2
τ_3	140	80	31	18	exchangesort
τ_4	1852	256	181	89	fft
τ_5	240	80	76	21	FIRFilter

Table 5.2 describes the benchmarks with total instruction memory (code) and data memory (data) in Bytes, number of c-lines (C-Ln) and number nodes (Nodes) of the control flow graph. The benchmark τ_1 is the example of figure 5.2. Benchmark τ_2 is based on τ_1 but the data access in the loop is input dependent and the size of the array is 10 elements. The maximum number of loop iterations is 1000. The benchmark τ_3 is a public domain sorting algorithm and τ_4, τ_5 are taken from [66]. We use g++ v3.0.4 with the option `-ast-original` to generate the abstract syntax tree, which is the input of data dependency analysis. We assume a 10 cycle cache miss penalty, 1 cycle cache hit time, and a fixed instruction length of 32 bits. In all experiments we use a direct mapped cache. The memory access trace was computed with the ARM9 processor simulator from ARM RealView Developer Suite, which contains the instruction as well as data addresses. Therefore, we simulated a unified cache. It could have also been applied to a separate data cache by filtering the data memory addresses. In the following we denote as the worst case execution time (WCET) the time to access this unified cache. The core execution time of the processor is not considered.

Table 5.3. WCET for data cache in [10^3clk].

Task	t_{cons}	t_{ana}	t_{sim}	$\frac{t_{cons}-t_{sim}}{t_{sim}}$	$\frac{t_{ana}-t_{sim}}{t_{sim}}$
τ_1	18.2	5.40	4.3	320%	25%
τ_2	78.3	29.3	20.1	290%	46%
τ_3	27.5	6.89	5.38	410%	28%
τ_4	807	285	205	290%	39%
τ_5	20.6	6.08	5.60	270%	8.6%

Results for the data cache timing behavior of these benchmarks for a 512 Byte direct mapped cache with block size of 16 bytes are shown in table 5.3 for a conservative analysis approach t_{cons}, for the proposed analysis framework t_{ana} and for cache simulation t_{sim}. The total worst case execution time (WCET) is given in 10^3 clock cycles (clk). The conservative estimate t_{cons} assumes two cache misses for each unpredictable memory access. The last two column expresses the overestimation of the conservative approach and our approach compared to cache simulation. The analysis precision of our approach is an improvement of an order of magnitude compared to the conservative approach.

Table 5.4. WCET in [10^3clk] for different cache sizes.

Cache, Task	t_{cons}	t_{ana}	t_{sim}	$\frac{t_{ana}-t_{sim}}{t_{sim}}$
$128-16\ \tau_2$	78.3	38.3	38.2	0.2%
$512-16\ \tau_2$	78.3	29.2	20.1	45%
$2048-16\ \tau_2$	78.3	29.3	20.1	46%
$128-16\ \tau_3$	32.3	11.7	11.1	5.4%
$512-16\ \tau_3$	27.5	6.89	5.38	28%
$2048-16\ \tau_3$	27.5	6.89	5.38	28%
$128-16\ \tau_5$	20.6	10.3	9.25	11%
$512-16\ \tau_5$	20.6	6.08	5.60	8.6%
$2048-16\ \tau_5$	20.6	3.77	3.54	6.5%

Table 5.4 shows the WCET for different cache sizes with 16 Byte cache block size only for some tasks due to space restrictions. While the conservative approach is very pessimistic, the proposed approach can be very close to the simulated results.

Table 5.5. WCET in [10^3clk] for different cache block sizes.

Cache, Task	t_{cons}	t_{ana}	t_{sim}	$\frac{t_{ana}-t_{sim}}{t_{sim}}$
$512-8\ \tau_2$	78.3	29.3	20.1	46%
$512-16\ \tau_2$	78.2	29.2	20.1	45%
$512-32\ \tau_2$	78.2	29.2	20.2	45%
$512-8\ \tau_3$	27.7	7.13	5.49	30%
$512-16\ \tau_3$	27.5	6.89	5.38	28%
$512-32\ \tau_3$	27.2	6.67	5.38	24%
$512-8\ \tau_5$	22.7	6.47	5.89	9.8%
$512-16\ \tau_5$	20.6	6.08	5.60	8.6%
$512-32\ \tau_5$	20.1	6.39	5.82	9.8%

Table 5.5 shows the results for different cache block sizes for a 512 Byte cache which shows that the deviation for different cache block sizes is small.

In summary, the results are improved by an order of magnitude compared to previous conservative analyses. Compared to cache simulation our approach yields an overestimation of 8% to 46%. Note, that simulation results might not include the real worst case behavior. Therefore, analysis results cannot directly be compared to simulation results. A second reason is that *nearly* input independent data access behavior cannot be detected by our analysis. E.g. memory accesses that are not fully input independent but contain some regular access patterns.

The time-complexity of the analysis is very low. For each task, the entire analysis including data dependency classification, memory mapping, data cache analysis, construction and solution of the ILP took less than one minute for calculating all nine cache configurations.

5.7 Conclusion

In this chapter we have proposed a novel worst case timing analysis framework for data caches. Input data dependency has been addressed as the key issue to deliver tight timing bounds. We have presented an analysis for unpredictable data accesses as well as predictable data accesses. The analysis framework is a combination of data-flow analysis, pigeon-hole principle and integer linear programming. Compared to previous conservative approaches, the results are an improvement of an order of magnitude while compared to simulation results our approach shows up to 46% overestimation. However, a direct comparison with simulation is not possible because some execution paths might not have been measured, and hence the worst-case path could have been omitted.

Chapter 6

INSTRUMENTATION POINT PLACEMENT

6.1 Introduction

Timing requirements are becoming increasingly important in embedded system design. Control systems continuously interact with their environment. To accurately implement a control function, these systems not only have to preform correctly but also have to provide the computed results within specified time bounds. Driven by the increased flexibility resulting from the use of microprocessors in control systems, more and more functionality is integrated into electronic control units (ECU). The automotive industry is one of the leading fields pushing these developments. Today, a new car contains about 30 ECUs and in luxury cars up to 70 ECUs are embedded [30]. As a consequence, applications have to be mapped to ECUs. Several applications are mapped to the same processor and thus have to be scheduled. Schedulability analyses [93] can be applied to guarantee that an application finishes before a specified time bound. As a key requisite, the worst-case execution time (WCET) has to be calculated for each application.

A simple technique to estimate the timing behavior is software tracing and sampling. Examples are RTA-Trace by ETAS [32], CodeTEST by Freescale [48], and a hybrid tracing and sampling approach used by Nokia [83]. Measurement points are inserted at the beginning and end of each function or around critical sections to monitor special events. The program execution is stimulated with a set of test patterns. However, safe timing bounds are difficult to guarantee, because only a subset of all test patterns is used.

As an alternative to software tracing, many static timing analysis approaches have been proposed, e.g. timing analysis language [85], timing schema [73], abstract inter-

pretation [35], implicit path enumeration [71], and many more. Some timing analysis tools are commercially available: aiT from Absint GmbH [4], Bound-T by Tidorum Ltd [128] and Rapitime by Rapita Systems Ltd [100]. An overview of academic and commercial WCET-tools can be found in [137].

Static analysis methods assume either cycle-level simulators or other precise hardware models to compute the execution time of individual basic blocks. However, the main difficulty is that precise hardware models are not available or would be too expensive to construct. In fact, only very few static analyses are being adopted in industry, despite the wealth of academic approaches.

Requirements for industrial strength WCET tools have been summarized in [64]: The tool must work with minimal user interaction. It must support model-based software development, e.g. Matlab/Simulink [82] with automatic code generation, e.g. TargetLink [126]. The method must integrate into the development chain of customers without modification of tools from the tool chain. And finally, the WCET analysis method must be easy to re-target to different hardware settings, i.e economic usability.

Measurement-based WCET analysis approaches, such as [14] [91] [139] [135], are attractive to meet these requirements, as they are easy to re-target and cost-efficient. The principle is to measure the execution time of an application on real hardware by inserting intrusive instrumentation probes. The main drawback in using instrumentation probes is that they disturb the temporal behavior of the application, i.e. the execution time of the program differs when the probes are removed. Furthermore, the initial hardware state is difficult to assert during the measurements.

To minimize the probe interference, measurement probes can be inserted only at the beginning and at the end of a function, but this requires test data for a path coverage. Path coverage means that every execution path has to be covered. The number of input data increases exponentially with the software structure. For example, a path coverage for 20 consecutive if-then-else statements would require 2^{20} (about 1 million) test patterns. In the following we use the term test patterns as a synonym of input data.

If the smallest structural entity of a program, also called basic blocks, were measured, test data for full branch coverage would be sufficient, but the increased measurement frequency would lead to a significantly overestimated WCET. Branch coverage means that each branch is executed both taken and non-taken. For the above example, $2 \cdot 20 = 40$ test data would be sufficient for full branch coverage. Thus the challenge is to select measurement points wisely, such that the effort to specify input data is minimized and the measurement probe overhead is kept as low as possible.

Three major contributions are presented. First, we propose a novel instrumentation methodology that reduces the number of instrumentation points but yet requires only test data for full branch coverage. The main advantage is the ability to measure longer pieces of code than basic blocks, while still requiring only test data for full branch coverage. As a second contribution, we present a WCET analysis framework, which integrates the instrumentation methodology and which supports important industrial requirements for WCET tools, including easy re-target-ability to new processors and a seamless integration to existing embedded design tools. As a third contribution, we present a case study of an automotive control application to demonstrate the applicability of our approach.

The rest of this chapter is structured as follows. In section 6.2 we review related work. In section 6.3 we present the novel instrumentation methodology. The experiments for the instrumentation methodology and the case study are presented in section 6.4 before we conclude in section 6.5.

6.2 Related Work

The related work for measurement-based approaches is presented in section 6.2.1 and our previous approach to WCET analysis is described in section 6.2.2.

6.2.1 Measurement-based WCET analysis

In measurement-based approaches, a program is partitioned into program segments, then the execution time for each program segment is measured on real hardware, e.g. on an evaluation or emulation board. Sophisticated debugging and hardware monitoring facilities, such as JTAG interface, provide low-intrusive measurement environments. Finally, the worst case execution path is calculated based on the execution time of each program segment. A program segment can be defined in several ways: as a single basic block [80] [75], a measurement block [91] or as a sequence of a varying number of basic blocks [135] [14] [139].

A source code instrumentation methodology MetaC, which is an extension of C, is presented in [80]. Meta-programming techniques are used to automatically monitor the program. The placement of instrumentation points or debugger statements is configurable. It has been applied to WCET-analysis where instrumentation points are inserted at each basic block. This framework could be combined with our proposed measurement methodology by extending MetaC. Another approach which is based on instrumenting basic blocks is presented in [75]. First, the program is instrumented at each basic block to monitor the execution count. Then, all measurement points are removed, and the execution time of the entire program path is measured. In a sec-

ond step, an equation system of linearly independent equations is set up representing the visited basic blocks as unknowns and the measured execution times of the program. The approach assumes that the execution times of basic blocks is input data independent. For complex pipelines and caches this approach is not applicable.

The approach by [91] is based on the granularity of measurement blocks. A measurement block is a piece of code, e.g. several basic blocks. However, the partition has to be done manually. Another algorithm to select instrumentation points is presented in [14]. Using the hardware debug interface Nexus restricts the selection of instrumentation points. The paper discusses how these restrictions can be resolved.

The WCET estimation method in [135] automatically partitions the control flow graph into program segments and automatically generates test data. The size of a program segment is determined by a path bound b. This means that each program segment contains at most b paths. The partition algorithm thus tradeoffs between the number of instrumentation points and the number of necessary measurements. Test data is automatically generated by a combination of genetic algorithms and model checking.

Compared to our proposed approach, the execution time is measured for the entire program segment, containing several paths. The worst case execution time for all test patterns is taken as worst case for the segment. If the set of test patterns is not complete, an underestimation is possible. Another difference is that our approach is based on the control flow structure while this approach is based on program paths. An example to clarify this difference is given in section 6.3.

6.2.2 SymTA/P approach

SymTA/P is hybrid analysis using static timing analysis and measurements [139] to calculate an upper and lower execution time bound of a C program running on a single processor. SymTA/P is an acronym of SYMbolic Timing Analysis for Processes. The control flow graph is partitioned into program segments containing single feasible paths (SFP). A SFP is a sequence of basic blocks, which is executed independently of input data. For example, a loop-statement consisting of several if-then-else statements whose conditions only depend on loop-iteration variables has a SFP. A FIRFilter and fast Fourier transformation are further examples.

Then, the execution time is measured for each SFP either on real-hardware, e.g. evaluation board using JTAG interface, or on a cycle true processor simulator. A conservative overhead is added to each program segment to account for the worst case hardware state, for example pipeline effects, at the beginning of each program

segment. The precision is higher when the SFP program segments are longer. Finally, integer linear programming (ILP) is used to calculate the best-case and worst-case execution time of the program. A research prototype has been developed, however, several limitations exist.

1 The partitioning into long SFP program segments is very limited for control intensive software applications, e.g. in automotive ECUs. Automatically generated C source code, e.g. by TargetLink [126] or ASCET-SD [9], contains long `switch-case` statements with frequent function calls in `if-conditions`. In this case, a SFP program segment has the size of a basic block. This leads to more frequent measurements and thus to a higher total overestimation. Other techniques are necessary to construct longer program segments even in the presence of input data dependency.

2 Program path analysis to identify SFPs is based on symbolic simulation and is restricted to a subset of C programming language. Most important, sub-function calls, `switch-case` statements, pre-compiler statements and `break`-statements are not supported. A program path analysis supporting the full C language would be necessary, if the WCET method were applied to automatically generated code.

3 Global WCET calculation has been limited to single functions. A WCET framework supporting sub-function calls is essential to be applicable to larger software applications.

The first limitation is addressed in section 6.3 in which a novel instrumentation point algorithm is proposed. The second and third limitation are important when the approach is going to be applied to an industrial project. These issues are resolved in appendix A.1.

6.3 Instrumentation point methodology

The objective of our approach is to minimize the number of instrumentation points while requiring only full branch coverage. Measuring all program paths, e.g. instrumenting at the beginning and at the end of each function, is too time-consuming. An exponential number of test patterns is necessary. On the other hand, instrumenting each basic block would lead to too many instrumentation points.

The key idea of our approach is to partition the program into program segments, which are longer than a single basic block, but short enough that test patterns for full branch coverage are sufficient. As a motivating example, Figure 6.1 shows two program segments S_1 and S_2. The if-condition statement is emphasized by c. S_1

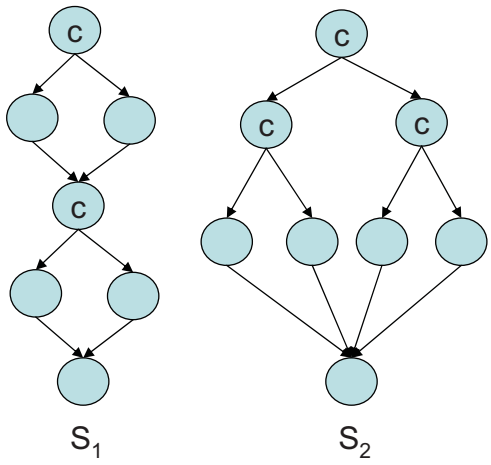

Figure 6.1. Control flow graph and program segment graph.

contains two consecutive `if-then-else` statements and program segment S_2 contains three nested `if-then-else`-statements (an `if-then-else`-statement with an `if-then-else` within the `then` as well as within the `else` branch). Both program segments S_1 and S_2 contain four paths, thus would be considered as a single program segment in [135]. The difference is the number of test patterns in terms of branch and path coverage. Test patterns for full branch coverage are sufficient for S_2, while a path coverage is necessary for S_1. Compared to the approach by [135], our approach detects those segments where full branch coverage is sufficient, e.g. S_2.

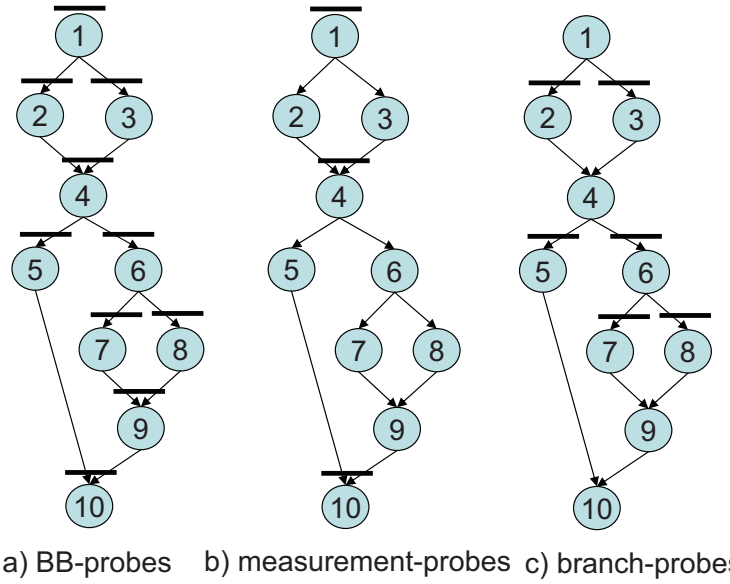

Figure 6.2. Control flow graph with instrumentation points.

The instrumentation methodology partitions the instrumentation probes into *measurement probes* and *branch probes*. An example of an instrumented control flow graph is shown in Figure 6.2 in three variations. Figure 6.2 a) shows a traditional instrumentation at each basic block. Figure 6.2 b) shows the placement of measurement probes and Figure 6.2 c) shows the placement of branch probes. The source code snippet for this control flow graph is shown in Figure 6.3. Instead of separately measuring 10 basic blocks we insert only 3 measurement probes.

This method has two gains. The first gain is a reduced number of time-critical measurement probes for which a time-overhead has to be accounted for in the WCET analysis. The purpose of the branch probes is to monitor which branch is taken during program execution. The execution of of branch probes is not time-critical, because only the c-line number is monitored during program execution. The second gain is that test patterns for full branch coverage are sufficient.

The method is structured in the following phases. First, program segments are identified and the control flow graph is partitioned in section 6.3.1. The placement of measurement and branch probes is described in section 6.3.2. Then, the measurement on real hardware is presented in section 6.3.3. Finally, the branch monitoring results and measurement results are back-annotated to the graph in section 6.3.4.

6.3.1 Program segment partitioning

The methodology is based on the control flow graph (CFG), in which basic blocks are represented by nodes control flow between basic blocks is represented by edges. The CFG is constructed from the source code of the program. First, we define some notations.

Definition. A *program segment* PS is defined by a sequence of basic blocks which contains at most one branching basic block on the same structural hierarchy level.

Definition. A *branching basic block* is a basic block with two or more outgoing edges. Then, $Br(PS_k)$ denotes all branching basic blocks of program segment PS_k.

Definition. The *structural hierarchy level* of a basic block, abbreviated as level, is given by its hierarchical depth level in the abstract syntax tree. The level of a basic block can be calculated as follows. The root node has depth level 1. The depth level increases by one for each branch statement and decreases by one when branches are joined.

Definition. A control flow graph containing program segments as nodes is called *program segment graph* (PSG).

We use the notation $PS_k = (B_1, \cdots, B_n, B_h)$, in which the basic blocks $B_1, \cdots, B_n$ are the basic blocks of the program segment PS_k and B_h the segment header of the next program segment. The first basic block of a program segment is called *segment header*. This definition simplifies the instrumentation methodology because measurement points are inserted before segment headers.

```
1  bool A,B;    // global variables
2  void f(void){
3    int c,d;     // local variables
4    if (A == 1){
5       c = 0;
6    } else {
7       c = 1;
8    }
9    if (B == 1) {
10     d = 0;
11   } else {
12     if (A == 0){
13          d = 1;
14     } else {
15          d = 2;
16     }
17   }
18   return c+d;
19 }
```

Figure 6.3. Source code example.

An example program snippet is shown in Figure 6.3. It consists of two consecutive `if-then-else` statements where the second one contains another `if-then-else` statement. The CFG and the partitioned program segment graph are shown in Figure 6.4. The level of basic block B_1, B_4, B_{10} is 1, the level of B_2, B_3, B_5, B_6, B_9 is 2 and the level of B_7, B_8 is 3. Branching basic blocks are B_1, B_4, B_6 where branching basic blocks B_1 and B_4 are on the same level. Therefore, the CFG is partitioned into 6 program segments, as shown in Figure 6.4 b). Note that the more than one branch statement can belong to a program segment, e.g. PS $(B_4, B_6, B_7, B_9, B_{10})$, because branching basic blocks B_4 and B_6 are on different levels.

A graph traversing algorithm is used to identify segment headers. Starting from the root node, the next branching basic block on the same hierarchy level is searched. This search continues at the found segment header and is finished when the last node of the control flow graph is reached. As a second step, all paths between two segments

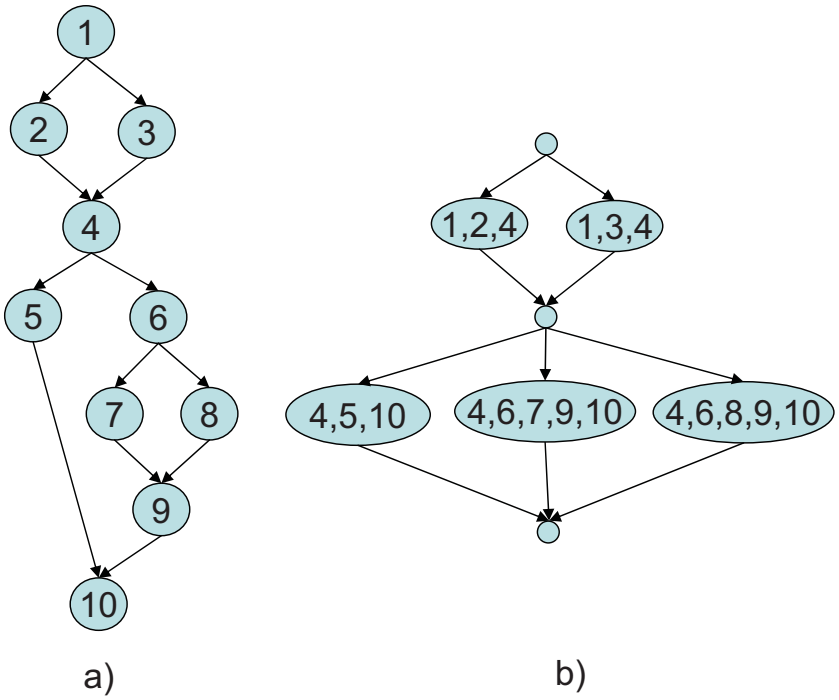

Figure 6.4. Control flow graph and program segment graph.

headers form a program segment. Finally, a program segment graph is created where the nodes are these program segments, and edges connect them.

6.3.2 Instrumentation probes

A *measurement probe*, $P_m(B_i)$ at a basic block B_i is an instrumentation point denoted by a tuple $P_m(B_i) = (l_i, t)$ in which l_i denotes the c-line and t a time stamp. In terms of programming language, it is an instruction that reads the system timer and stores it with the corresponding c-line l_i. A *branch probe* $P_b(B_i)$ at a basic block B_i is a tuple of the c-line l_i: $P_b(B_i) = (l_i)$. It is implemented by an instruction that saves the c-line l_i.

For an efficient system implementation, the basic blocks (bb), the branches (br), and the (yet to be measured) execution time (time) of each program segment are stored in a XML file. An example is shown in Figure 6.5.

Placement of instrumentation probes

Instrumentation probes are inserted before the segment headers of each program segment. The placement of the measurement probes are shown in Figure 6.6 in the source code and in Figure 6.2 b) in the control flow graph. In this example, three

```
<PS id="1"> <bb> 1,2,4 </bb> <br> 2 </br>  <time></time>\>
<PS id="2"> <bb> 1,3,4 </bb> <br> 3 </br>  <time></time>\>
<PS id="3"> <bb> 4,5,10 </bb> <br> 5 </br>  <time></time>\>
<PS id="4"> <bb> 4,6,7,9,10 </bb> <br> 6,7 </br> <time></time>\>
<PS id="5"> <bb> 4,6,8,9,10 </bb> <br> 6,8 </br> <time></time>\>
```

Figure 6.5. XML structure for program segments

```
bool A,B;   // global variables
void f(void){
  int c,d;    // local variables
  measurementProbe ( 4 );
  if (A == 1){
     c = 0;
  } else {
     c = 1;
  }
  measurementProbe ( 9 );
  if (B == 1) {
    d = 0;
  } else {
    if (A == 0){
        d = 1;
    } else {
        d = 2;
    }
  }
  measurementProbe ( 18 );
  return c + d;
}
```

Figure 6.6. Source with measurement probes.

```
bool A,B;   // global variables
void f(void){
  int c,d;    // local variables
  if (A == 1){
     branchProbe( 5 );
     c = 0;
  } else {
     branchProbe( 7 );
     c = 1;
  }
  if (B == 1) {
    branchProbe( 10 );
    d = 0;
  } else {
    branchProbe( 12 );
    if (A == 0){
       branchProbe( 13 );
       d = 1;
    } else {
       branchProbe( 15 );
       d = 2;
    }
  }
  return c + d;
}
```

Figure 6.7. Source with branch probes.

measurement probes are sufficient. Note, that no additional probes are necessary for the nested `if-then-else` statement in lines 11-16 in Figure 6.3.

Placement of branch probes

Branch probes are inserted in each branch. Figure 6.7 shows the source code with inserted branch probes and Figure 6.2 c) shows the positions in the control flow graph. The position of branch probes is given for each program segment PS_k in $Br(PS_k)$.

6.3.3 Execution time measurement

As described in the previous section, the source code is instrumented, first, with measurement probes and, second, with branch probes. The probe instruction is translated to the actual hardware timing instruction. An example for a measurement probe is given in the case study in section A.2. Then, the two source files are compiled and separately executed on the evaluation board or processor simulator. In the next step these timing results are back-annotated to the PSG.

6.3.4 Back-annotation of timing results

An example output of measurement probes is shown in Figure 6.8 a) and of branch probes in Figure 6.8 c).

```
timing 4 1001
timing 9 1020
timing 18 1025
timing 4 1054
timing 9 1066
timing 18 1090
```

(a)

```
timing 4 9 19
timing 9 18 5
timing 4 9 12
timing 9 18 24
```

(b)

```
branch 5
branch 10
branch 7
branch 12
branch 13
```

(c)

Figure 6.8. Output of timing and branch probes.

Individual timing items are merged to a reduced timing item with the syntax: `timing` l_{start} l_{end} t, where l_{start} is the first and l_{end} the last line of the program segment. This is shown in Figure 6.8 b). The measured execution time between l_{start} and l_{end} is given by the variable t. The variable t is the difference between the two system timer values of the timing statement at l_{end} and l_{start}, as shown in Figure 6.8 a).

In the next step, the available branches and these timing-statements are annotated back to the corresponding program segment. Then, the execution time t is annotated to the matching program segment. This match is unique, because two program segments with the same l_{start} and l_{end} share at most a subset of branches. But the set of branches cannot be the same, then it would be the same program segment. After this step, the execution time for each program segment is available. A WCET analysis could then be used to calculate the longest execution time of the program.

6.4 Experiments

In this section we evaluate the instrumentation methodology for several benchmarks. Table 6.1 lists the lines of code (LoC), the number of nodes in the control flow and program segment graph, and the number of instrumentation points for each benchmark.

task	LoC	graph nodes		i-points			red.[%]
		BB-CFG	PSG	BB	P_m	P_b	$\frac{BB-P_m}{BB}$
paperExample	19	10	5	9	3	6	66%
parttrain	275	105	53	42	11	31	73%
fft	182	87	1	24	2	0	92%
jfdctint	375	95	3	14	5	0	64%
statemate	1276	538	305	272	148	124	46%
case study	5791	1102	893	865	354	511	59%

Table 6.1. Comparison number of graph nodes (CFG and PSG) for different benchmarks and lines of code (LoC). Number of instrumentation points (i-points) at each basic block (BB), number of measurement probes P_m, number of branch probes P_b, and reduction (red.) of time-critical measurement points in %.

The benchmark `paperExample` is the example described earlier. Benchmark `parttrain` is a packet receiver algorithm in a network server and `fft` is a Fast Fourier Transformation algorithm, both are taken from [141]. The benchmark `statemate` is automatically generated code of an automotive control application and `jfdctint` is a discrete-cosine transformation on a 8x8 pixel block; both are taken from Mälardalen WCET research group[1]. The `case study` is automatically generated code of the case study which will be presented in section A.2.

The ratio program segments in the program segment graph and the number of basic blocks (BB) in a CFG for three benchmarks is about 0.5, while the ratio for the benchmarks `fft` and `jfdctint` is about 0.03. This significant reduction is caused by the detection of single feasible paths. Both algorithms consist of input independent loops and thus entire functions are single feasible paths. Second, the number of instrumentation points are compared. The number of instrumentation points at each basic block (BB), the number of intrusive measurement probes P_m, and the number of non-intrusive branch probes P_b. Intrusive means in this context that the insertion of

[1] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

the probe disturbs the temporal behavior during program execution. Branch probes are non-intrusive because only executed branches are monitored.

In the last column, the reduction of intrusive measurement points is shown. On average, the number of intrusive measurement points is reduced by 67% compared to an instrumentation at each basic block. Since in both cases, test patterns for full branch coverage are sufficient, this reduction directly reduces the overestimation in the WCET calculation method. The time complexity of the measurement methodology is very low. The construction of the control flow graph from source code, the partitioning into a program segment graph, and the instrumentation of measurement and branch probes took less than one minute for each benchmark.

6.5 Conclusion

Performance validation is a key issue for real-time embedded systems. Measurement-based WCET analysis approaches are an cost-efficient solution as the execution time is measured on real-hardware. We have proposed a new instrumentation methodology that minimizes the number of instrumentation points and requires only test patterns for full branch coverage. The results for the instrumentation methodology show an average reduction of 67% of intrusive measurement points compared to an instrumentation at each basic block.

Chapter 7

SUMMARY AND OUTLOOK

7.1 Summary

Embedded systems are prevalent in today's society and will be even more pervasive in the future. Then, requirements for an efficient system design will be more demanding. Caches are needed to bridge the gap between slow main memory access time and ever increasing processor clock frequency. Because of its complex design, a cache leads to a dynamic timing behavior and is thus often avoided for real-time applications. Besides the dynamic timing effects within a single task execution, context switch delays have to be considered when preemptive task scheduling is used. Since fixed priority preemptive scheduling is very attractive for its ability to guarantee short response times and a high utilization, timing verification approaches have to consider their impact on cache behavior.

In this thesis we have presented methods to analyze timing behavior of associative instruction caches for fixed priority preemptive scheduling. First, we have developed a novel cache-aware schedulability analysis that bounds the preemption delay much tighter than previous conservative approaches. The key idea is to distinguish between task preemptions and actual cache interference and thereby reducing the total number of preemptions. The additional time complexity to identify and to calculate the preemption delay is bounded by $O(n \cdot E_{j,i} \cdot \log(E_{j,i} \cdot n))$ in which n denotes the number of tasks to be scheduled and $E_{j,i}$ denotes the number of task activations of a higher priority task τ_j during the response time of task τ_i. As the time delay for a single preemption we used the state-of-the-art approach that considers the preempted as well as the preempting task.

As a refinement, we have focused on multiple task preemptions. While previous approaches assume that each preemption takes place at the worst case preemption point, we analyze all preemptions in context. We determine the worst case preemption scenario, e.g. the set of worst case preemption points that yield the total worst-case delay. The cache interference at a preemption point is analyzed in connection with all previous cache interferences of the preemption scenario.

Motivated by an automotive case study, we analyzed the effect of multiple task executions on cache timing behavior. We presented an analysis that considers previous task activations and thereby revised the empty cache assumption in previous cache analysis approaches.

As a bound for a single preemption delay is the key factor for potential overestimation, we studied the characteristics of two data flow analysis techniques which bound the preemption delay most accurately among all related work. These techniques differ substantially in time-complexity and analysis precision. We presented a combined analysis that allows to gradually scale the analysis precision and analysis time complexity. The results have shown that a significantly higher analysis precision can be achieved with only a small increase of time-complexity.

While the above techniques are geared to instruction caches, we have proposed a timing analysis for direct mapped data caches for a single task execution. Input data dependency of memory accesses are the main cause of unpredictability. A combination of symbolic execution, data flow techniques and integer linear programming is used to bound the cache access time even for accesses to memory addresses that are dependent on input data. Compared to previous conservative approaches, the results show an improvement of an order of magnitude while compared to simulation results our approach shows up to 46% overestimation. However, a direct comparison with simulation is misleading because not all execution paths were simulated and thus the worst-case path might not have been simulated.

Finally, we integrated the presented cache analyses into the SymTA/P prototype. As SymTA/P is a measurement-based WCET-analysis approach, the precision depends on the measurement overhead. We have presented a novel instrumentation point methodology that reduces the number of instrumentation points while assuming only full branch coverage. The results show a reduction of over 60% of instrumentation points compared to previous instrumentation techniques that instrument each basic block and which also requires a full branch coverage.

7.2 Outlook

In this thesis we have presented a sophisticated analysis framework to make caches more attractive for real-time systems. However, much research will still be necessary until caches will be used in real-time or safety critical applications. First, we point out some short-term goals which aim at improving the proposed analysis framework and then we point out some long-term research goals.

Short term goals

The cache-aware scheduling analysis has been applied to fixed priority preemptive scheduling. First, it would be valuable to apply the methodology to other scheduling policies, such as TDMA or Round Robin.

In TDMA and Round Robin scheduling policies the task execution is sliced. For preemption delay calculation, one has to search for the maximum delay only during this time window instead for the entire task. This has the potential gain of tighter bounds of the preemption delay. Also if the task sequence is known a priori, tighter bounds can be given for initial cache state (warm cache state). If preferred preemption points were used (e.g. in linear code and not during execution of loops) coupled with time-slicing techniques, the cache-related preemption delay costs could further be minimized.

Second, the analysis for multiple preemptions supports only single functions. An extension for sub-functions is essential if the analysis is to be applied to more complex applications.

Third, associative instruction caches with LRU replacement strategy are assumed. Many cache architectures include complex features, like critical word first, sub-block placement, streaming to CPU, write buffers, victim caches and multi-level caches. Our analysis consists of a modular framework that captures the cache behavior in a single module. Thus, this cache module could be extended feature by feature.

Third, we assumed direct mapped data caches in the timing analysis for input dependent memory accesses. This analysis could be extended to associative caches. Furthermore, the cache related preemption delay analysis framework could be applied to data caches. Currently, data cache analysis restricts the source code to single functions. For larger applications, these limitations have to be resolved.

If these extensions were implemented, a sophisticated framework including single task cache analysis and cache-related preemption delay analysis supporting both instruction as well as data caches would be available. Such a framework could be integrated into any WCET-analysis methodology, like SymTA/P, aiT by Absint or

Rapitime by Rapitasystems. This integration would also be of great value for system level analysis, like SymTA/S [56] [101].

Long term goals

Long-term research directions might include bounding main memory access times, designing a real-time cache and comparing different cache prediction techniques.

In traditional schedulability analyses the WCET is assumed, while in WCET approaches the cache miss penalty and the memory access time are considered as constant. The bound of memory access times has to be conservative, e.g. the worst case memory access time. As memory access times are dynamically influenced by bus arbitration and timing behavior of memory controller, assuming always the worst case memory access time will lead to high overestimations of the WCET and, thus, to an overestimated response time. In [119] [104] [103] it is discussed, how system level schedulability analysis could be combined with lower-level WCET analysis to more tightly bound memory access times.

Designing a cache that is fast but also predictable is a major challenge. It would be very interesting to design an alternative cache architecture that meets high timing requirements while being analyzable with an adequate static timing analysis.

Another future research direction would be to compare the cache performance of cache locking approaches, cache partitioning approaches and cache-aware schedulability analyses. Then, an optimal configuration using all three techniques could be suggested for a given application.

Appendix A
WCET Analysis

A.1 SymTA/P analysis framework

The SymTA/P methodology to calculate the WCET and its limitations have been briefly described in section 6.2.2. In this section we describe the refined analysis including the following main contributions:

1 The proposed instrumentation methodology is integrated in the SymTA/P.

2 The input dependency analysis for single feasible path identification is rigorously revised. The proposed method is based on GNU GCC compiler's abstract syntax tree. The advantage is that the full C syntax is supported making SymTA/P feasible for larger software projects.

3 The BCET and WCET calculation supports sub-functions. From practical perspective this issue is essential for larger software projects.

An overview of the framework is shown in Figure A.1. The hybrid approach consists of a path analysis module (Frontend) based on source code, an execution time measurement module (Backend) based on object code, a cache analysis module, and a WCET calculation module (ILP solver). In the following each module is described in more detail. A detailed description of the installation and usage of the tool can be found in [114] and at the website of the institute [121].

A.1.1 Program path analysis (Frontend)

The program path analysis module corresponds to the Frontend in Figure A.1. Based on the C source-code, the program path classification and the control flow graph of a software task are created by a Frontend parser, as shown in Figure A.2.

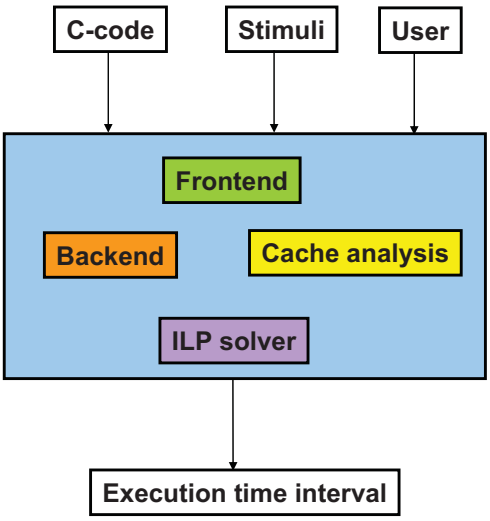

Figure A.1. Overview of SymTA/P analysis framework.

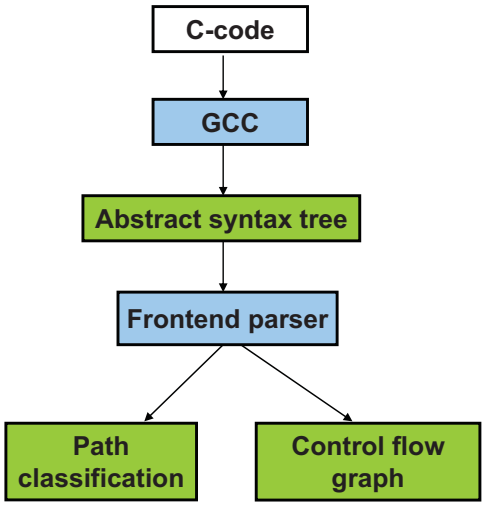

Figure A.2. Frontend of SymTA/P.

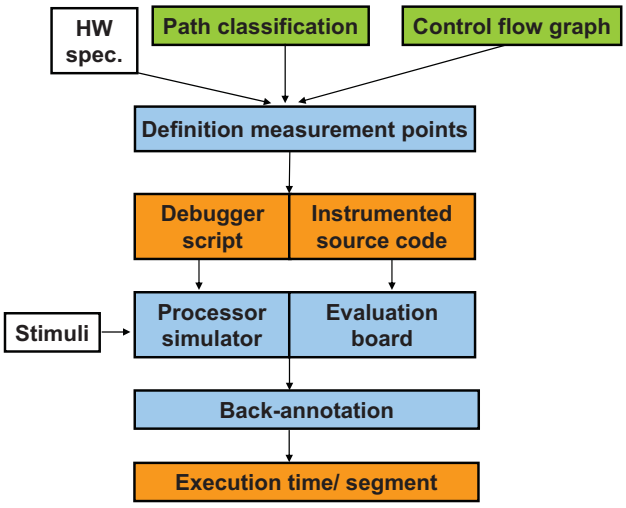

Figure A.3. Backend of SymTA/P.

This Frontend parser reads the abstract syntax tree, which is generated by GNU GCC, and performs a symbolic simulation on the abstract syntax tree to determine input data dependency.

A.1.2 Execution time measurement (Backend)

The execution time measurement, called *Backend*, is shown in Figure A.3. Off-the-shelf processor simulators or cost-efficient evaluation boards can be used. Based on the control flow graph and the path classification, the measurement instrumentation

framework of section 6.3.1 is implemented in the `Definition measurement points` phase.

If an evaluation board is used, the C source code is instrumented once with instrumentation probes and once with branch probes. Each probe is a function call, that is defined in terms of hardware dependent assembly instructions. Then both instrumented programs are executed and the results are annotated back to the program segment graph according to section 6.3.4. If a cycle accurate processor simulator is used, a debugger script is automatically generated that starts and stops the execution of the program. The output is back annotated to the program segment graph according to section 6.3.4. During this measurement, a safe initial state cannot be assured in all cases. Therefore an additional time delay is added to conservatively assume the worst case. At the end of this module the execution time of each program segment is available.

The Backend provides a framework that is easy to re-target to new processors. In general the Backend requires the following: First, test patterns have to be specified for full branch coverage. Usually these test patterns are already available from previous functional test phase in industrial projects. SymTA/P does not provide a test pattern generator. However, the WCET-calculation module highlights the source-code lines that were not measured. Second, the implementation of the instrumentation and branch probes have to be defined for the new target processor. And third, the measurement framework has to be adapted to the new hardware. This includes the communication to load the program to the board, to start and stop the program, to access system timers, and to write back timing results to the host PC.

A.1.3 Cache Analysis

The cache analysis in SymTA/P is shown in Figure A.4.

Based on the control flow graph and the memory map file that is created from the binary, a static cache analysis [141] computes the cache access behavior for each segment. The requirements for cache analysis include the memory map file of the program and the cache configuration, e.g. cache size, associativity, and block size.

A.1.4 WCET Computation (ILP Solver)

The longest and shortest paths in the control flow graph are found by an integer linear programming (ILP). This module is shown in Figure A.5. The measured execution time and the cache access behavior are combined to the total execution for each program segment. The ILP is constructed using structural constrains which are auto-

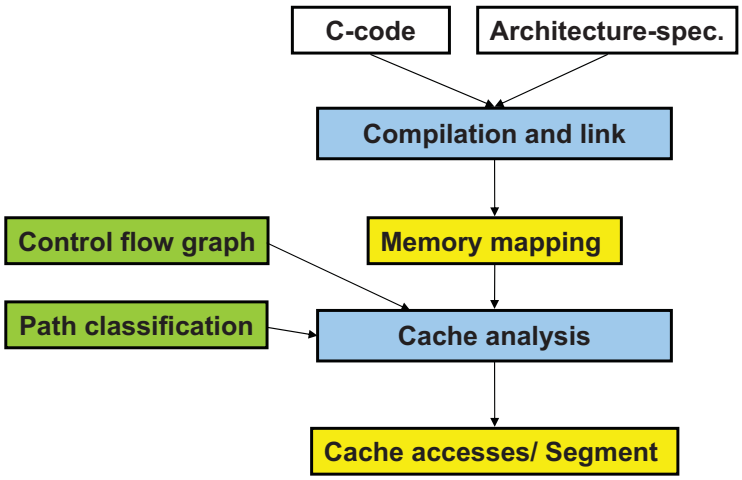

Figure A.4. Cache analysis of SymTA/P.

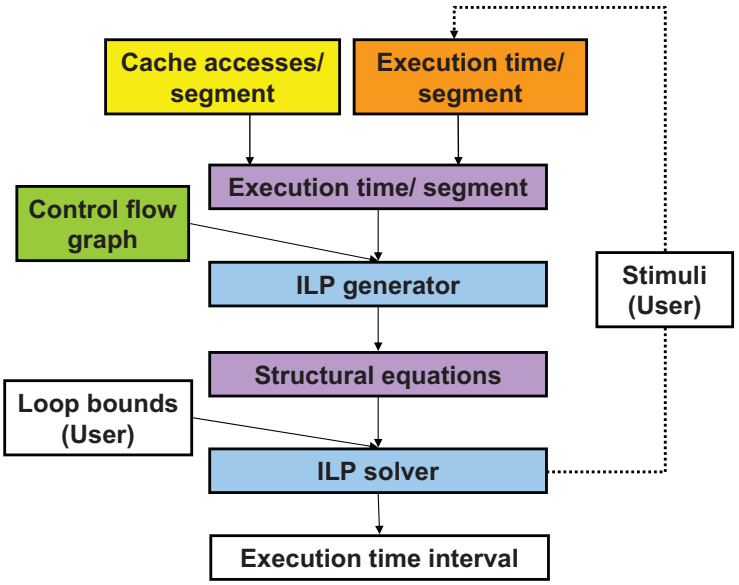

Figure A.5. WCET calculation (ILP Solver) of SymTA/P.

matically generated from the control flow graph. Loop bounds have to be specified by the user to bound the maximum number of loop iterations.

Compared to previous work the ILP generation is extended for function calls. The call tree of all functions is constructed. The ILP of the functions at leaf nodes, which call no other functions, are constructed first. Then the ILP construction advances in a bottom-up fashion until all functions are considered.

If for some program segments no execution time was assigned, the ILP solver issues a warning. In this case, the user can specify additional test patterns (stimuli)

and can re-run the timing measurement (Backend). Finally the total best-case and worst-case execution time are available.

A.2 Case study

Setup

The WCET-analysis is applied to a safety critical automotive control application for a micro-hybrid vehicle. It consists of a C-source file with about 5800 lines of code. The largest function has about 4200 lines. Totally, there are 11 functions in the application. Parts of the presented results have been previously published in [110].

Test patterns are manually defined using Matlab/Simulink environment [82]. The test patterns are structured in test sets and test cases. A test case corresponds to a set of input signals for a single simulation step of the application. A test set corresponds to a sequence of input signals. Totally, 65 test sets are specified, with totally 272022 test cases (about 4185 test cases for each test set).

The measurement framework consists of a C167CR evaluation board, a C compiler (Tasking), bootloader (I+ME Actia) and a tailor made measurement methodology for the C167CR board. The C167CR evaluation board is equipped with 2kB internal (on-chip) RAM for registers and stack and 256kB static RAM for code. The board does not provide ROM. Communication is only possible over serial bus, JTAG or other debug interfaces are not available.

Process description

We briefly describe each step in the SymTA/P analysis framework. In the path analysis module, the control flow graph is constructed for each function and the code is instrumented with measurement and branch probes. The implementation of these probes for the C167CR processor is shown in Figure A.6. The measurement probe is a macro which outputs a system timer value and the branch probe outputs only the c-line.

```
#define measurementProbe(l) T6CON &= ~(0x0040);\
   printf("timing %u %lu00\n", (l),(( (unsigned long int) \
   (T5) << 16 ) | T6)* (TICK_NS/100) ); T6CON |= 0x0040;
#define branchProbe(l) printf("branch %u\n", (unsigned int) l);
```

Figure A.6. Implementation of probe functions for C167 evaluation board.

In this case study, 354 measurement probes and 511 branch probes were inserted in the source code. The control flow graph (for basic block instrumentation) consists of 1102 nodes and the program segment graph contains 893 program segments. These numbers are totals, e.g. for all 11 functions.

Once the execution time has been measured on the C167 board, they are annotated back to the program segment graph. Finally, the longest execution path is calculated by integer linear programming. For a conservative timing analysis, the worst case system state for each program segment has to be analyzed. The C167 processor has a four-stage pipeline and the memory access to the static RAM can be considered constant. The overhead is determined as follows: The time clock runs at half the speed as the processor. Therefore the measurement precision is two instructions. Assuming 100 ns= 1 clock cycle, the total error is at most 400 ns. The measurement instruction was measured to be 300-400 ns (=2 instructions, one instruction to start and one instruction to stop the timer). The worst case pipeline state is to assume a pipeline flush, which means a new instruction has to start at instruction fetch. This delay can be bounded by 4 clock cycles (400ns). In summary, the measurement instruction may take between 300-400 ns, which can be subtracted from the execution time for a program segment. The additional overhead for a conservative pipeline state is 400 ns. Therefore the overhead of 100 ns (1 clk) is added to the execution time for each program segment.

Results

Independent tracing was run with all test pattern to compare the static timing analysis by SymTA/P. The application was instrumented at the beginning and at the end of the application. The distribution of the execution times for all test cases, which were determined by simulation in Figure A.7. The longest execution time was 491 micro seconds (μs). The range of execution time is between 326 and 491 μs. It can be seen, that some execution times occur more frequent than others. This has motivated the notion of *process modes* and the study of *contest sensitive* timing analysis, in which for a given set of input data the execution time is determined [118].

The static timing analysis of SymTA/P determines the WCET of 797 μs which is about 62% longer than the tracing result. However this cannot be considered as an overestimation, because it cannot be guaranteed that the worst case path was actually measured during tracing.

The time spent in each phase is summarized in table A.1. The total time was about 54 hours for the entire case study.

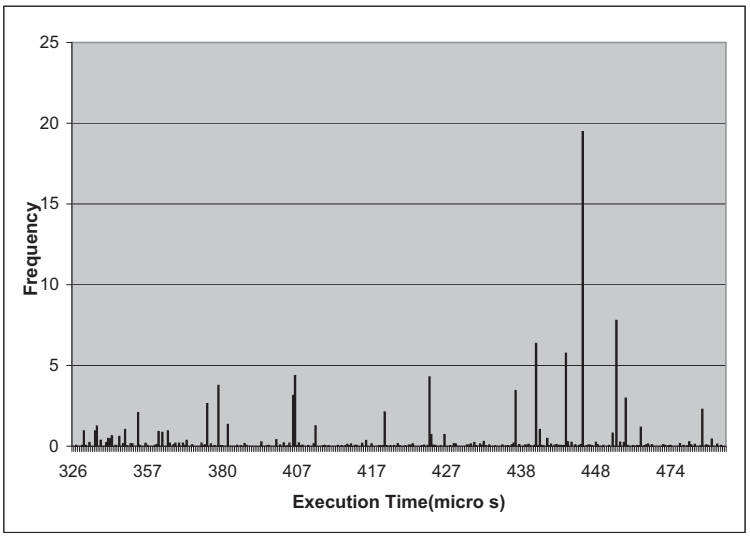

Figure A.7. WCET calculation (ILP) of SymTA/P.

Nr	Phase	Time
1.	Program Segment Clustering (CFG)	6min
1.	Instrumentation of measurement points (Board)	1min
2.	Configuration and compilation in Tasking EDE	5min
2.	Execution on C167 board (measurementProbes)	22h
2.	Execution on C167 board (branchProbes)	20h
3.	Back-annotation of timing results	11h
4.	WCET calculation (ILP Solver)	1min
Total		54 h

Table A.1. Time requirement for each analysis phase.

The execution on C167 board includes the communication via serial line. Most of the time is due to this communication which is very slow. This running time could be reduced, when the measurement results are stored in on-board static RAM. However, the available C167CR evaluation board has only a 256kB RAM. The running time could be significantly reduced by using a more sophisticated measurement methodology, e.g. JTAG interface or by using a board with sufficient on-board memory. The bottleneck was the communication between C167 board and the host PC. The back-annotation software is a prototype and has not yet been optimized, which is a technical issue.

List of Figures

List of Tables

List of publications

Refereed journal articles and book chapters

1 F. Wolf, J. Staschulat, and R. Ernst. Hybrid Cache Analysis In Running Time Verification Of Embedded Software. *Journal of Design Automation for Embedded Systems, Kluwer Academic Publishers*, 7(3):271–295, 2002.

2 M. Jersak, K. Richter, R. Racu, J. Staschulat, R. Ernst, J.-C. Braam, and F. Wolf. *Formal Methods for Integration of Automotive Software*, chapter 2. Embedded Software for SoC, Kluwer Academic Publishers, 2003.

3 Jan Staschulat and Rolf Ernst. Scalable Precision Cache Analysis for Real-Time Software. *Accepted for ACM Transactions on Embedded Computing Systems*, 2006.

4 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *Accepted for ACM Transactions on Programming Languages and Systems*, 2006.

Refereed conference and workshop articles

1 F. Wolf, J. Staschulat, and R. Ernst. Associative Caches in Formal Software Timing Analysis. In *IEEE/ACM Design Automation Conference*, pages 622–627, New Orleans, USA, June 2002.

2 Jan Staschulat and Rolf Ernst. Synergetic Effects in Cache Related Preemption Delays. In *International WCET Workshop (Satellite of Euromicro Conference on Real-Time Systems)*, Catania, Sicily, Italy, June 2004.

3 J. Staschulat and R. Ernst. Multiple Process Execution in Cache Related Preemption Delay Analysis. In *ACM International Workshop on Embedded Software (EMSOFT)*, pages 278–286, Pisa, Italy, Sept. 2004.

4 J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *EUROMICRO Conference on Real-Time Systems (ECRTS)*, pages 41–48, Palma de Mallorca, Spain, July 2005.

5 Jan Staschulat, Rolf Ernst, Andreas Schulze, and Fabian Wolf. Context Sensitive Performance Analysis of Automotive Applications. In *Designers Forum at Conference on Design, Automation and Test in Europe (DATE)*, pages 165–170, Munich, Germany, March 2005.

6 Jan Staschulat, Simon Schliecker, Matthias Ivers, and Rolf Ernst. Analysis of Memory Latencies in Multi-Processor Systems. In *International WCET Workshop (Satellite of EUROMICRO Conference on Real-Time Systems)*, Palma de Mallorca, Spain, July 2005.

7 Jan Staschulat and Rolf Ernst. Scalable Precision Cache Analysis for Preemptive Scheduling. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 157–165, Chicago, Ill, USA, June 2005.

8 Simon Schliecker, Matthias Ivers, Jan Staschulat, and Rolf Ernst. Combined System and Task Level Performance Analysis for MPSoC. Poster session at ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Chicago, Ill, USA, June 2005.

9 J. Staschulat and R. Ernst. Worst Case Timing Analysis of Input Dependent Data Cache Behavior. In *EUROMICRO Conference on Real-Time Systems*, Dresden, Germany, July 2006.

10 S. Schliecker, M. Ivers, J. Staschulat, and Rolf Ernst. A Framework for Response Time Calculation of Multiple Corelated Events. In *WCET Workshop (Satellite of EUROMICRO Conference on Real-Time Systems)*, Dresden, Germany, July 2006.

11 J. Staschulat, J. C. Braam, R. Ernst, T. Rambow, R. Schlör, and R. Busch. Cost-Efficient Worst-Case Execution Time Analysis in Industrial Practice. In *2nd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Paphos,Cypris, Nov. 2006.

Other publications

1 Jan Staschulat and Rolf Ernst. Cache Effects in Multi Process Real-Time Systems with Preemptive Scheduling. Technical report, IDA, TU Braunschweig, Germany, November 2003.

2 Jan Staschulat and Rolf Ernst. CRPD Independence for Multiple Process Execution. Technical report, IDA, TU Braunschweig, March 2004.

3 Jan Staschulat. SymTA/P Performance Verification for Complex Embedded Systems, Manual, Version 1.2. Technical report, IDA, Technical University Braunschweig, Germany, August 2005.

Bibliography

[1] A. Agarwal. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Kluwer Academic Publishers, 1989.

[2] A. Agarwal and S. D. Pudar. Column-associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches. In *ACM International Symposium on Computer Architecture*, pages 16–19, San Diego, CA, USA, May 1993.

[3] A. V. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, GB, 1988.

[4] aiT - Worst-Case Execution Time Analyzer, Absint GmbH. www.absint.de, 2006.

[5] F. Angiolini and L. Benini. Polynomial-Time Algorithm For On-Chip Scratchppad Memory Partitioning. In *ACM Intl. Conference on Compilers, Architecture and Synthesis of Embedded Systems (CASES)*, San Jose, CA, U.S.A, 2003.

[6] ARM: Realview Development Suite. http://www.arm.com.

[7] ARM. *ARM 9 Family Flyer, ARM DOI 0034-5/09.04*, September 2004.

[8] R. Arnold, F. Mueller, D.B. Whalley, and M. Harmon. Bounding Worst-Case Instruction Cache Performance. In *IEEE 15th Real-Time Systems Symposium (RTSS)*, pages 172–181, San Juan, Puerto Rico, Dec. 1994. IEEE Computer Society Press.

[9] ASCET-SD Embedded Control Development Systems for Automotive Solutions. ETAS, August 2002.

[10] S. Basumallick and K. Nilsen. Cache Issues In Real-Time Systems. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, USA, June 1994.

[11] Thomas Beierlein and Olaf Hagenbruch, editors. *Taschenbuch Mikroprozessortechnik*. Fachbuchverlag Leipzig im Carl Hanser Verlag, 1999.

[12] L. Benini and G. D. Micheli. Network on Chips: A new SoC Paradigm. *IEEE Computer*, 35(1):70–78, 2002.

[13] M. Berkelaar and J. Dirks. ILP Solver lp_solve v. 2.2: Source code, libraries and documentation. ftp.es.ele.tue.nl, 1997.

[14] A. Betts and G. Bernat. Issues using Nexus Interface for Measurement-Based WCET Analysis. In *WCET Workshop (Satellite of EUROMICRO Conference on Real-Time Systems)*, Palma de Mallorca, Spain, 2005.

[15] F. Bodin and I. Puaut. A WCET-oriented Static Branch Prediction Scheme For Real-Time Systems. In *EUROMICRO Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[16] A. Burns. *Preemptive Priority Based Scheduling*, chapter An appropriate engineering approach, pages 225–248. Prentice-Hall International, Inc., 1994.

[17] J. V. Busquets, J. J. Serrano, and A. Wellings. Hybrid Instruction Cache Partitioning for Preemptive Real-Time Systems. In *9th Euromicro Workshop on Real-Time Systems*, Toledo, Spain, June 1997. IEEE Computer Society Press.

[18] J. V. Busquets-Mataix, D. Gil, P. Gil, and A. Wellings. Techniques To Increase The Schedulable Utilization Of Cache-Based Preemptive Real-Time Systems. *EUROMICRO Journal of Systems Architecture, Elsevier*, 46:357–378, 2000.

[19] Jose Vicente Busquets-Mataix and Andy Wellings. Adding Instruction Cache Effect To Schedulability Analysis Of Preemptive Real-Time Systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 204–212, Bosten, MA, USA, June 1996.

[20] G. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2002.

[21] G. Buttazzo. *Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 2002.

[22] G. Buttazzo. Rate Monotonic vs. EDF: Judgment Day. In *ACM International Workshop on Embedded Software*, Philadelphia, PA, USA, Oct. 2003.

[23] C-Lab, A collection of WCET Benchmarks. http://www.c-lab.de/download, 2004. Paderborn, Germany.

[24] A. Campoy, Isabelle Puaut, Angel Ivars, and Jose Mataix-Busquets. Cache Contents Selection For Statically-Locked Instruction Caches: An Algorithm Comparison. In *EUROMICRO Conference on Real-Time Systems*, Palma de Mallorca, Spain, July 2005.

[25] Marti Campoy, A. Perles Ivars, and J. V. Busquets-Mataix. Static Use of Locking Caches in Multitask Preemptive Real-Time Systems. In *IEEE Real-Time Embedded System Workshop*, Dec. 2001.

[26] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1997.

[27] Matteo Corti, Roberto Brega, and Thomas Gross. Approximation Of Worst-Case Execution Time For Preemptive Multitasking Systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 178–198, Vancouver, Canada, June 2000. Springer.

[28] A. Datta, S. Choudhury, A. Basu, H. Tomiyama, and N. Dutt. Satisfying Timing Constraints Of Preemptive Real-Time Tasks Through Task Layout Technique. In *IEEE Proc. of 14th Intl. Conference on VLSI Design*, pages 97–102, Jan. 2001.

[29] Jean-Philippe Dauvin. Semiconductor Market And Industry Mega-Trends 2005-2010. In *MEDEA DAC*, May 2005.

[30] A. Deicke. The Electrical/Electronic Diagnostic Concept of the New 7 Series. In *Convergence International Congress & Exposition on Transportation Electronics*, Detroit, MI, USA, 2002.

[31] ETAS. *ERCOSEK - Automotive Real-Time Operating System*. http://www.etas.info.

[32] ETAS. *RTA-TRACE Datasheet*, June 2004.

[33] In-Car Enforcement Technologies Today, Brochure. European Transport Safety Council (ETSC), 2005.

[34] D. W. Whalley F. Mueller and M. Marmon. Predicting Instruction Cache Behavior. In *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems (LCTES)*, June 1994.

[35] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, Germany, 1997.

[36] Christian Ferdinand and Reinhard Wilhelm. On Predicting Data Cache Behaviour for Real-Time Systems. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, Montreal, Canada, June 1998.

[37] Freescale Semiconductor. *MPC7448 PowerPC Processor, Document Number MPC7448POWPCFS*, 2005.

[38] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: A Compiler Framework For Analyzing And Tuning Memory Behavior. *ACM Transactions on Programming Languages and Systems*, 21(4):703–746, July 1999.

[39] J. Gustafsson, B. Lisper, P. Puschner, and R. Kirner. Input-Dependency Analysis for Hard Real-Time Software. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Guadalajara, Mexico, Jan. 2003.

[40] J. C. Palencia Gutierrez, J. J. Gutierrez Garcia, and M. Gonzalez Harbour. Best-Case Analysis for Improving the Worst-Case Schedulability Test for Distributed Hard Real-Time Systems. In *Euromicro Workshop on Real-Time Systems*, pages 35–44, Berlin, Germany, June 1998. IEEE Computer Society Press.

[41] Article in Newspaper: Mikrosysteme machen Autos sicherer. Handelsblatt, 08.Mai 2006.

[42] M. S. Hecht and J. D. Ullman. A Simple Algorithm for Global Data Flow Analysis Problems. *SIAM J. Computing*, 4(4):519–532, Dec. 1975.

[43] S. Heithecker, A. Lucas, and R. Ernst. A Mixed QoS SDRAM Controller for FPGA-Based High-End Image Processing. In *IEEE Workshop on Signal Processing Systems (SIPS'03)*, Seoul, Korea, August 2003.

[44] William Henderson, David Kendall, and Adrian Robson. Improving the accuracy of scheduling analysis applied to distributed systems computing minimal response times and reducing jitter. *Journal of Real-Time Systems, Springer Netherlands*, 20(1):5–25, Jan. 2001.

[45] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers, 1996.

[46] M. D. Hill. A Case for Direct Mapped Caches. *IEEE Computers*, 21(12):25–40, Dec. 1988.

[47] Steven Hill. The ARM10 Family of Advanced Microprocessor Cores. In *Symposium on High-Performance Chips at Stanford University (HOT Chips 13)*, 2001.

[48] Nat Hillary. *Beyond Profiling Gaining Control of Software Performance*. Freescale Simiconductor, Nov. 2005.

[49] IBM. *PowerPC 750GX Product Brief*, 2004.

[50] Infineon Technologies AG. *TC1775 User's Manual V.2.0, 32 Bit Single-Chip Microcontroller: System Units*, February 2001.

[51] Infineon Technologies AG. *TC1796 User's Manual, V1.0, 32 Bit Single-Chip Microcontroller: System Units*, June 2005.

[52] Infineon Technologies AG. *TriCore 1, 32-bit Unified Processor Core, Volume 1: v1.3 Core Architecture, User's Manual V1.3.5*, Feb. 2005.

[53] Intel Corporation. *Intel 80200 Processor based on Intel XScale Microarchitecture, Datasheet - Commercial and Extended Temperature (80200T).*

[54] ITEA. ITEA2 Blue Book - European Leadership In Software-Intensive Systems And Services, 2005.

[55] M. Jersak, K. Richter, R. Racu, J. Staschulat, R. Ernst, J.-C. Braam, and F. Wolf. *Formal Methods for Integration of Automotive Software*, chapter 2. Embedded Software for SoC, Kluwer Academic Publishers, 2003.

[56] Marek Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, Technical University Braunschweig, 2005.

[57] M. Joseph and P.K. Pandya. Finding response times in a real-time system. *The Computer Journal (British Computer Society)*, 29:390–395, Oct. 1986.

[58] N. P. Jouppi. Improving Direct-Mapped Cache Performance By The Addition Of a Small Fully-associative Cache And Prefetch Buffers. In *ACM International Symposium on Computer Architecture*, pages 28–31, Seattle, USA, May 1990.

[59] Daniel Kästner and Stephan Thesing. Cache Sensitive Pre-Runtime Scheduling. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems(LCTES)*, Montreal, Canada, June 1998.

[60] S.-K. Kim, S. L. Min, and R. Ha. Efficient Worst Case Timing Analysis Of Data Caching. In *IEEE Real-Time Technology and Applications Symposium*, pages 230–240, Bosten, MA, USA, June 1996.

[61] David B. Kirk. Process Dependent Static Cache Partitioning For Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 181–190, Huntsville, Al, USA, Dec. 1988.

[62] David B. Kirk. SMART (Strategic Memory Allocation for Real-Time) Cache Design. In *10th IEEE Real-Time Systems Symposium (RTSS)*, pages 229–239, Santa Monica, CA, USA, Dec. 1989.

[63] David B. Kirk and Jay K. Stronsnider. SMART (Strategic Memory Allocation for Real-Time) Cache Design Using the MIPS R3000. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 322–330, Lake Buena Vista, FL, USA, Dec. 1990.

[64] Raimund Kirner, Peter Puschner, and Ingomar Wenzel. Measurement-Based Worst-Case Execution Time Analysis using Automatic Test-Data Generation. In *Proc. 4th Euromicro International Workshop on WCET Analysis*, Catania, Italy, June 2004.

[65] Judita Kruse. *Distributed Design Methodology for Embedded Systems, in preparation*. PhD thesis, Technical University Braunschweig, 2007.

[66] C.-G. Lee, J. Hahn, Y.-M. Seo, S.-L. Min, R. Ha, S. Hong, C.-Y. Park, M. Lee, and C. S. Kim. Analysis Of Cache-Related Preemption Delay In Fixed-Priority Preemptive Scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[67] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Bounding Cache-Related Preemption Delay For Real-Time Systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, November 2001.

[68] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Enhanced Analysis Of Cache-Related Preemption Delay In Fixed-Priority Preemptive Scheduling. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 187–198, San Francisco, CA, USA, Dec. 1997.

[69] J. Lehoczky, L. Sha, and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. 10th IEEE Real-Time Systems Symposium (RTSS)*, pages 166–171, Santa Monica, CA, USA, December 1989.

[70] Y.-T. S. Li, Sharad Malik, and Andrew Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. In *Proc 17th IEEE Real-Time Systems Symposium (RTSS)*, pages 254–263, Washington, DC, USA, Dec. 1996.

[71] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis Of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[72] J. Liedtke, H. Härtig, and M. Hohmuth. OS-Controlled Cache Predictability For Real-Time Systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symposium*, pages 213–227, Montreal, Canada, June 1997.

[73] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C. S. Kim. An Accurate Worst Case Timing Analysis For RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–603, July 1995.

[74] T. H. Lin and W. S. Liou. Using Cache to Improve Task Scheduling in Hard Real-Time Systems. In *IEEE Workshop on Architecture Support for Real-Time Systems*, pages 81–85, December 1991.

[75] M. Lindgren, H. Hansson, and H. Thane. Using Measurements To Derive The Worst-Case Execution Time. In *Proc. of 7th IEEE Int. Conference on Real-Time Computing Systems and Applications (RTCSA*, Cheju Island, South Korea, Dec. 2000.

[76] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.

[77] Gabriele Luculli and Marco Di Natale. A Cache-Aware Scheduling Algorithm For Embedded Systems. In *Proc. 18th IEEE Real-Time Systems Symposium (RTSS)*, pages 199–209, San Francisco, CA, USA, Dec. 1997.

[78] Thomas Lundqvist and Per Stenström. A Method to Improve the Estimated Worst-Case Performance of Data Caching. In *Proc. of 6th IEEE Intl. Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 255–262, Hongkong, China, Dec. 1999.

[79] J. Madsen, S. Mahadevan, K. Virk, and M. Gonzalez. Network-on-Chip Modeling for System-Level Multiprocessor Simulation. In *IEEE Real-Time Systems Symposium (RTSS)*, Cancum, Mexico, Dec 2003.

[80] T. Maier-Komor, A. von Bülow, and G. Färber. MetaC and its use for automated source code instrumentation of C programs for real-time analysis. In *EUROMICRO Conference on Real-Time Systems, Work in Progress*, Palma de Mallorca, Spain, July 2005.

[81] A. Marti, X. Molero, A. Perles, F. Rodriguez, and J.V. Busquets. Combined Intrinsic-Extrinsic Cache Analysis For Preemptive Real-Time Systems. In *25th IFAC Workshop on Real Time Programming*, pages 49–55, Palma, Spain, May 2000.

[82] The Mathworks Inc.: Matlab/Simulink. http://www.mathworks.com/.

[83] Edu Metz, Raimondas Lencevicius, and Teofilo F. Gonzalez. Performance Data Collection Using a Hybrid Approach. In *CM SIGSOFT Software Engineering Notes, Proceedings of the 10th European Software Engineering Conference*, volume 30(5), 2005.

[84] Jeffrey C. Mogul and Anita Borg. The Effect Of Context Switches On Cache Performance. In *ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pages 75–84, Santa Clara, CA, USA, April 1991.

[85] A. Mok, P. Amerasinghe, M. Chen, and K. Tantisirivat. Evaluating Tight Execution Time Bounds Of Programs By Annoations. In *Proc. 6th IEEE Workshop on Real Time Operating Systems and Software*, Pittsburgh, USA, May 1989.

[86] F. Mueller. Compiler Support for Software-based Cache Partitioning. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems (LCTES)*, La Jolla, CA, USA, June 1995.

[87] F. Mueller. Timing analysis for instruction caches. *Journal of Real-Time Systems, Springer Netherlands*, 18(2/3):209–239, May 2000.

[88] H. Muller, D. May, J. Irwin, and D. Page. Novel Caches For Predictable Computing. Technical Report CSTR-98-011, Department of Computer Science, University of Bristol, 1998.

[89] Hemendra Sigh Negi, Tulika Mitra, and Abhaik Roychoudhury. Accurate Estimation Of Cache-Related Preemption Delay. In *ACM/IEEE Int. Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Newport Beach, CA, USA, Oct. 2003.

[90] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. *Memory Issues in Embedded Systems-On-Chip: Optimizations and Exploration*. Kluwer Academic Publishers,Norwell, MA, 1999.

[91] Stefan M. Petters and Georg Färber. Making Worst Case Execution Time Analysis For Hard Real-Time Tasks On State Of The Art Processors Feasible. In *Proc. of 6th IEEE Intl. Conference on Real-Time Computing Systems and Applications (RTCSA)*, Hongkong, China, Dec. 1999.

[92] Stefan M. Petters and Georg Färber. Scheduling Analysis with Respect to Hardware Related Preemption Delay. In *Workshop on Real-Time Embedded Systems (Satellite of the IEEE Real-Time Systems Symposium)*, London, UK, Dec. 2001.

[93] Paul Pop, Petru Eles, and Zebo Peng. Analysis and optimization of heterogeneous real-time embedded systems. *IEE Proceedings - Computers and Digital Techniques*, 152(2):130–147, March 2005.

[94] The PowerPC 440 Core. IBM Microelecttronics Division, 1999.

[95] Betty Prince. *Semiconductor Memories*. A Wiley-Teubner Publication, 1990.

[96] Isabelle Puaut. Cache Analysis Vs Static Cache Locking For Schedulability Analysis In Multitasking Real-Time Systems. In *EUROMICRO Conference on Real-Time Systems*, Vienna, Austria, June 2002.

[97] Isabelle Puaut and David Decotigny. Low-Complexity Algorihtms For Static Cache Locking In Multitasking Hard Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, Austin, Texas, USA, Dec. 2002.

[98] H. Ramaprasad and F. Mueller. Bounding preemption delay within data cache reference patterns for real-time tasks. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr 2006.

[99] Harini Ramaprasad and Frank Mueller. Bounding Worst-Case Data Cache Behavior by Analytically Deriving Cache Reference Patterns. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 148–157, 2005.

[100] Rapitime - Worst-Case Execution Time Analyzer, Rapita-Systems Ltd. www.rapitasystems.com, 2006.

[101] Kai Richter. *Compositional Scheduling Analysis Using Standart Event Models*. PhD thesis, Technical University of Braunschweig, 2005.

[102] Jim Robertson and Kalpesh Gala. *Application Note: Instruction and Data Cache Locking on G2 Processor Core*. Freescale Semiconductor, Inc, April 1999.

[103] S. Schliecker, M. Ivers, J. Staschulat, and Rolf Ernst. A Framework for Response Time Calculation of Multiple Corelated Events. In *WCET Workshop (Satellite of EUROMICRO Conference on Real-Time Systems)*, Dresden, Germany, July 2006.

[104] Simon Schliecker, Matthias Ivers, Jan Staschulat, and Rolf Ernst. Combined System and Task Level Performance Analysis for MPSoC. Poster session at ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES), Chicago, Ill, USA, June 2005.

[105] J. Schneider. Cache And Pipeline Sensitive Fixed Priority Scheduling For Preemptive Real-Time Systems. In *21st IEEE Real-Time Systems Symposium (RTSS)*, pages 195–204, Orlando, Florida, USA, Nov. 2000.

[106] J. Schneider and Christian Ferdinand. Pipeline Behavior Prediction For Superscalar Processors By Abstract Interpretation. *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 34(7):35–44, May 1999.

[107] F. Sebek. Cache Memories And Real-Time Systems. Technical Report MRTC 01/37, Mälardalen University, Sweden, Oct. 2001.

[108] Filip Sebek. Measuring Cache Related Pre-emption Delay On A Multiprocessor Real-Time System. In *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, London, UK, Dec. 2001.

[109] Jonathan Simonson and Janak H. Patel. Use Of Preferred Preemption Points In Cache-Based Real-Time Systems. In *IEEE International Computer Performance and Dependability Symposium*, pages 316–325, April 1995.

[110] J. Staschulat, J. C. Braam, R. Ernst, T. Rambow, R. Schlör, and R. Busch. Cost-Efficient Worst-Case Execution Time Analysis in Industrial Practice. In *2nd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Paphos,Cypris, Nov. 2006.

[111] J. Staschulat and R. Ernst. Multiple Process Execution in Cache Related Preemption Delay Analysis. In *ACM International Workshop on Embedded Software (EMSOFT)*, pages 278–286, Pisa, Italy, Sept. 2004.

[112] J. Staschulat and R. Ernst. Worst Case Timing Analysis of Input Dependent Data Cache Behavior. In *EUROMICRO Conference on Real-Time Systems*, Dresden, Germany, July 2006.

[113] J. Staschulat, S. Schliecker, and R. Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *EUROMICRO Conference on Real-Time Systems (ECRTS)*, pages 41–48, Palma de Mallorca, Spain, July 2005.

[114] Jan Staschulat. SymTA/P Performance Verification for Complex Embedded Systems, Manual, Version 1.2. Technical report, IDA, Technical University Braunschweig, Germany, August 2005.

[115] Jan Staschulat and Rolf Ernst. Synergetic Effects in Cache Related Preemption Delays. In *International WCET Workshop (Satellite of Euromicro Conference on Real-Time Systems)*, Catania, Sicily, Italy, June 2004.

[116] Jan Staschulat and Rolf Ernst. Scalable Precision Cache Analysis for Preemptive Scheduling. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 157–165, Chicago, Ill, USA, June 2005.

[117] Jan Staschulat and Rolf Ernst. Scalable Precision Cache Analysis for Real-Time Software. *Accepted for ACM Transactions on Embedded Computing Systems*, 2006.

[118] Jan Staschulat, Rolf Ernst, Andreas Schulze, and Fabian Wolf. Context Sensitive Performance Analysis of Automotive Applications. In *Designers Forum at Conference on Design, Automation and Test in Europe (DATE)*, pages 165–170, Munich, Germany, March 2005.

[119] Jan Staschulat, Simon Schliecker, Matthias Ivers, and Rolf Ernst. Analysis of Memory Latencies in Multi-Processor Systems. In *International WCET Workshop (Satellite of EUROMICRO Conference on Real-Time Systems)*, Palma de Mallorca, Spain, July 2005.

[120] G. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 116–127, Sept. 2001.

[121] SymTA/P - Worst-Case Execution Time Analysis Tool, Project Homepage. www.ida.ing.tu-bs.de/research/projects/symta.

[122] B. Tabbara, A. Tabbara, and A. Sangiovanni-Vicentelli. *Function/Architecture Optimization and Co-design of Embedded Systems*. Kluwer Academic Publishers, 2000.

[123] Y. Tan and V. Mooney. Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multitasking Real-Time Systems. In *International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 181–199, Sept. 2004.

[124] Y. Tan and V. Mooney. Timing Analysis for Multi-tasking Preemptive Real-Time Systems. In *Conference on Design, Automation and Test in Europe (DATE)*, Paris, France, Feb. 2004.

[125] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall Inc., 1992.

[126] dSpace: TargetLink. http://www.dspace.com/, 2006.

[127] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction By Seperate Cache And Path Analyses. *Journal of Real-Time Systems, Springer Netherlands*, 18(2/3), May 2000.

[128] Bound-T - Worst-Case Execution Time Analyser, Tidorum Ltd. www.bound-t.com, 2006.

[129] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time systems. *Journal of Real-Time Systems, Springer Netherlands*, 6(2):133–152, March 1994.

[130] Hiroyuki Tomiyama and Nikil D. Dutt. Program Path Analysis To Bound Cache-Related Preemption Delay In Preemptive Real-Time Systems. In *ACM Int. Symposium on Hardware Software Codesign (CODES)*, pages 67–71, San Diego, CA, USA, May 2000.

[131] S. Udayakumaran and R. Baruah. Compiler-Decided Dynamic Memory Allocation For Scratchpad Based Embedded Systems. In *ACM Intl. Conference on Compilers, Architecture and Synthesis of Embedded Systems(CASES)*, San Jose, CA, U.S.A., 2003.

[132] J.T.J. van Eijndhoven, K. A. Vissers, E. J. D. Pol, M. J. A. Tromp, P. Struik, R. H. J. Bloks, P. van der Wolf, A. D. Pimentel, and H. P. E. Vranken. Trimedia CPU64 architecture. In *In Proc. of IEEE International Conference on Computer Design*, pages 586–592, Austin, Texas, Oct. 1999.

[133] X. Vera, B. Lisper, and J. Xue. Data Caches in Multitasking Hard Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, 2003.

[134] Xavier Vera. *Cache And Compiler Interaction - How To Analyze, Optimize And Time Cache Behavior*. PhD thesis, Mälardalen University, Sweden, 2003.

[135] Ingomar Wenzel, Bernhard Rieder, Raimung Kirner, and Peter Puschner. Automatic timing model generation by cfg partitioning and model checking. In *Conference on Design, Automation and Test in Europe (DATE)*, Munich, Germany, March 2005.

[136] R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing Analysis for Data and Wrap-Around Fill Caches. *Journal of Real-Time Systems, Springer Netherlands*, 17(2-3):209–233, 1999.

[137] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *Accepted for ACM Transactions on Programming Languages and Systems*, 2006.

[138] WindRiver. *VxWorks*, 2006. http://www.windriver.com/announces/vxworks/.

[139] F. Wolf, R. Ernst, and W. Ye. Path Clustering in Software Timing Analysis. *IEEE Transactions on VLSI Systems*, 9(6):773–782, 2001.

[140] F. Wolf, J. Staschulat, and R. Ernst. Associative Caches in Formal Software Timing Analysis. In *IEEE/ACM Design Automation Conference*, pages 622–627, New Orleans, USA, June 2002.

[141] F. Wolf, J. Staschulat, and R. Ernst. Hybrid Cache Analysis In Running Time Verification Of Embedded Software. *Journal of Design Automation for Embedded Systems, Kluwer Academic Publishers*, 7(3):271–295, 2002.

[142] Fabian Wolf. *Behavioral Intervals In Embedded Software*. Kluwer Academic Publishers, 2002.

[143] A. Wolfe. Software-Based Cache Partitioning for Real-Time Applications. *Journal of Computer and Software Engineering, Ablex Publishing*, 2(3), 1994.

[144] Andrew Wolfe. Software-based cache partitioning for real-time applications. In *International Workshop on Responsive Computer Systems*, September 1993.

[145] Lance Zheng. *Guidelines for Cache Management, TriCore1 Modular (TC1M) 32-bit Unified Processor, Application Note AP3267a*. Infineon Technologies AG, October 2001.

Glossary

This glossary contains often used abbreviations and mathematical terms.

Abbreviations

BB,B	Basic block
CFG	Control flow graph
BCET	Best case execution time
BCRT	Best case response time
CRPD	Cache related preemption delay
ILP	Integer Linear Programming
LCS	Set of live cache states in Mitra's approach
LMB	Set of live memory blocks in Lee's approach
RCS	Set of reaching cache states in Mitra's approach
RMB	Set of reaching memory blocks in Lee's approach
SymTA/P	Symbolic Timing Analysis for Processes
SymTA/S	Symbolic Timing Analysis for Systems
USE	Set of useful cache blocks
WCET	Worst case execution time
WCRT	Worst case response time

Mathematical terms

B_i	Blocking time of a task τ_i, 27
C_i	Worst case execution time of a task τ_i, 27
$crpd(j,i,R_i^n)$	General term for preemption delay27
$\delta_{j,i}$	Single preemption time delay if task τ_j preempts τ_i, 52
$\delta_{j,i}(n)$	n-th marginal preemption delay if task τ_j preempts τ_i, 65
$\Delta^s_{j,i}(R_i^n)$	Simplified calculation of preemption delays, 52
$\Delta_{j,i}(R_i^n)$	Advanced calculation of preemption delays, 55
$E^n_{j,i}$	Number of activations of τ_j during the interval R_i^n, 52
$gen^c[B]$	Set of generated memory blocks at cache set c at B. The definition depends on the data flow algorithm
$hp(i)$	Set of higher priority tasks then task τ_i, 27
$LCS^c[B]$	Set of live cache states at B, 42
$LMB^c[B]$	Set of live memory blocks of cache set c at B, 35
P_i	Period of task τ_i, 27
R_i	Response time of a task τ_i, 27
$RCS^c[B]$	Set of reaching cache states at B, 42
$RMB^c[B]$	Set of reaching memory blocks of cache set c at B, 35
$S^n_{j,i}$	Preemption scenario, if task τ_j preempts τ_i n times, 65
t_{miss}	Time to reload a single cache block, 28
$USE^{\tau}_{lee}[B]$	Set of useful cache blocks at B in Lees approach, 40
$USE^{\tau}_{mitra}[B]$	Set of useful cache blocks at B in Mitras approach, 43
$USE^{\tau}_{scale}[B]$	Set of useful cache blocks at B in scalable approach, 97