Timing analysis for time-predictable architectures

Amine Naji

SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS DEPUIS 1257

ENSTA ParisTech

Sorbonne Université

Ecole doctorale Informatique, Télécommunication et Électronique de Paris

Timing Analysis for Time-Predictable Architectures

Par Amine Naji

Thèse de doctorat d'Informatique

Dirigée par Albert Cohen

Présentée et soutenue publiquement le 12 juin 2019

Devant un jury composé de :

Rapporteurs	Mme. Isabelle Puaut	Professor, University of Rennes I
	M. Jens Knoop	Professor, Vienna University of Technology
Examinateurs	Mme. Karine Heydemann	Associate Professor, UPMC (Paris 6)
	M. Albert Cohen	Senior Research Scientist, Google
	M. Florian Brandner	Associate Professor, Télécom ParisTech
	M. Mathieu Jan	Research Engineer, CEA List

Contents

List of Figures

List of Tables

Abstract

With the rising complexity of the underlying computer hardware, the analysis of the timing behavior of real-time software is becoming more and more complex and imprecise. Time-predictable computer architectures thus have been proposed to provide hardware support for timing analysis. The goal is to deliver tighter worst-case execution time (WCET) estimates while keeping the analysis overhead minimal. These estimates are typically provided by standalone WCET analysis tools.

The emergence of time-predictable architectures is, however, quite recent. While several designs have been introduced, efforts are still needed to assess their effectiveness in actually enhancing the worst-case performance. For many time-predictable hardware, timing analysis is either non-existing or lacking proper support. Consequently, time-predictable architectures are barely supported in existing WCET analysis tools.

The general contribution of this thesis is to help filling this gap and turning some opportunities into concrete advantages. For this, we take interest in the Patmos processor. The already existing support around Patmos allows for an effective exploration of techniques to enhance the worst-case performance. This is delivered through the interplay between the hardware, the compiler, and the timing analysis. We thus not only provide some missing timing analysis support, but we also target hardware/software optimizations to enhance performance.

Main contributions include: (1) Handling of predicated execution in timing analysis, (2) Comparison of the precision of stack cache occupancy analyses, (3) Analysis of preemption costs for the stack cache, (4) Preemption mechanisms for the stack cache, and (5) Prefetching-like technique for the stack cache. In addition, we present our WCET analysis tool Odyssey, which implements timing analyses for the Patmos processor.

Résumé

En raison de la complexité croissante des architectures matérielles, l'analyse temporelle du logiciel temps-réel devient de plus en plus complexe et imprécise. Les architectures prédictibles des ordinateurs ont donc été proposées afin d'assurer un support matériel dédié à analyse temporelle. The but est de fournir des estimations plus précises de pire-temps d'exécution de programmes (WCET), tout en gardant le coût et la compexité de l'analyse minimal. Ces estimations proviennent typiquement d'outils dédiés à l'analyse WCET.

L'émergence de ces architectures spécialisées est, toutefois, assez récent. Bien que plusieurs designs d'architectures ont été proposés, des efforts sont encore nécessaires pour évaluer leurs capacités à améliorer les performances pire cas. Pour plusieurs composants matériels prédictibles, l'analyse temporelle est manquante ou partiellement supportée. En conséquence, les architectures prédictibles sont à peine supportées dans les outils d'analyse WCET existants.

Cette thèse s'inscrit dans les efforts pour combler ce manque et transformer le potentiel de ces architectures en avantages concrets. Pour cela, nous nous intéressons au processeur prédictible Patmos. Le support existant autour de la plateforme permet une exploration effective des techniques d'optimisation pour les performances pire cas. Ceci se base sur trois composantes étroitement liées, à savoir : le matériel, le compilateur, ainsi que l'analyse temporelle. Nous nous intéressons, donc, non seulement au support d'analyse temporelle, mais aussi aux optimizations du compilateur et du matériel en vue d'améliorer les performances pire cas.

Les principales contributions de cette thèse comprennent : (1) Une gestion des prédicats dans le flux d'anayse WCET, (2) Une comparaison de la précision des analyses d'occupancy pour le stack cache, (3) Une analyse des coûts de préemption pour le stack cache, (4) Des mécanismes de préemption pour le stack cache, et (5) Des techniques de prefetching pour le stack cache. En outre, nous présentons notre outil d'analyse WCET Odyssey. Notre outil implémente plusieurs de ces analyses et supporte le processeur Patmos.

Acknowledgements

I would like to express my deep gratitude to Florian Brandner for believing in me. I was fortunate to work closely alongside him throughout my thesis, and learned a great deal both technically and personally. Thank you for your positive energy. I could not have completed my work without you.

I would also like to thank Mathieu Jan for introducing me to this thesis. Your helpful nature and relevant advice contributed to the completion of this manuscript.

My gratitude goes to Albert Cohen for his guidance and encouragement throughout my research.

I owe so much to my family, particularly my mom. Thank you for encouraging me by any means.

Finally, my fiancée Jessica. I love you so much. Thank you for your patience, love, and support.

CHAPTER 1

Introduction

This first chapter provides a gentle introduction and lists our contributions. The rich information provided here makes it accessible for a wide range of readers. Specialists may find the first and the last sections sufficient for this matter. The chapter is organized as follows: We start off by presenting the general context and the motivation for our work. We then provide in Sections 2 and 3 basic background concepts related to real-time systems and timing analysis. Section 4 covers the meaning of performance in conventional vs. real-time computing systems. Section 5 reports some issues related to timing analysis for conventional computer architectures. Some time-predictable architectures are presented and compared in Section 6. Finally, we present our contributions in the last section.

1 Toward High-Performance Safety-Critical Systems

Computer technology has profoundly changed our society and shaped the way we live over the last few decades. Ever since the invention of the first transistor and the emergence of early digital computers, advances in electronics and computer science have made it possible to build computers that run software capable of supporting increasingly demanding features. Attainable levels of circuit miniaturization made it possible to realize faster and sophisticated computer architectures that implement multi-core technologies and other performance enhancing features. In parallel, compiler technology provided tools to build optimized software for the underlying computer architecture, enabling software engineers to develop large scale and complex projects. This multi-disciplinary synergy allowed use cases of computers to grow in vast and unpredictable ways. Remarkable achievements like social-networking, electronic voting, or some major scientific discoveries were not possible to attain without the deployment of high-performance computers in the heart of machines of all possible sizes.

There are other applications where computers showed most beneficial uses. In avionics, the automotive industry, or nuclear plants, there is a need in developing, so called, *safety-critical systems*. These systems are subject to special kinds of requirements, and are characterized by their highest degree of reliability and safety.

In contrast to other classes of systems, a safety-critical system failure may endanger human lives, lead to economical loss, or cause disastrous harm to the environment. They are often used as control systems that monitor the dynamics of a physical environment in real-time and thus are subject to strict timing requirements. For example, an automatic flight control system, like those used in most modern aircrafts, needs to continuously check inputs, such as the air dynamics, and activate the right controls *in-time* so that the desired flight conditions are maintained. These real-time control tasks are merely software programs carried-out by embedded computers that were specifically designed for that purpose. In that regard, one may think that these computers *must* also inherit decades of progress and wisdom accumulated in terms of high-performance computing.

Not quite, unfortunately. The problem is, because of their critical nature, these systems need to pass the most rigorous and extensive tests and verification processes, before being put in use. Particularly, timing analysis is crucial in order to check the satisfaction of the system's timing requirements and avoid failure. This typically consists of statically[1] analyzing the timing behavior of control tasks performed by the embedded computer and making sure they meet their respective timing constraints under *all* circumstances. The analysis identifies the worst-case timing behavior of each task, and provides an upper-bound estimation of its worst-case executing time (WCET). On the one hand, the WCET estimation has to be conservative in the sense that it covers all contributing factors to execution time variation, as well as their induced uncertainties. On the other hand, it is important that the estimated WCET bound be tight and the closest to the *actual* WCET, so that hardware resources are not wasted.

Modern computer architectures, however, are purposely designed to enhance the average-case performance of programs. This is done by optimizing execution times of frequent execution cases, which inevitably penalize those of infrequent cases. Unless the program is of extreme simplicity or consisting of a single execution scenario, the worst-case timing behavior is by definition extremely rare to occur and the performance gap with respect to other cases may therefore be large. Moreover, many of the built-in hardware components rely on complex and speculative behavior that overly increases the complexity of the timing analysis and induces huge amounts of uncertainties. The presence of uncertainties may seriously limit time-predictability, that is, the ability to precisely predict the timing behavior of the system. As a result, identifying the worst-case timing behavior becomes harder, and the computed WCET estimation is overly pessimistic. Components like caches, dynamic branch predictors, and deep out-of-order pipelines – often regarded as performance boosters – are unfortunately problematic for tight and precise timing analysis[2]. From a global standpoint, these architectures are often not timing-composable, meaning that a safe

[1] Before running the system.

[2] Depending on the memory hierarchy and performance, the timing behavior of a cache miss compared to that of a cache hit can be measured in order of magnitudes. In case the timing analysis is not able to predict, for instance due to complexity, the fact that some memory access is a cache hit, then the analysis has to consider that access as a cache miss. The resulting WCET estimation is indeed conservative. However, the system has to allocate more CPU time than what is actually needed, leading to a hardware resource waste. This is not to mention timing anomalies that describe non-intuitive situations where a local positive effect (say a cache hit) can lead to a global negative effect (increased pipeline stalls). Timing anomalies are hard to spot as they require tracking a long execution history which makes the timing analysis overly complex.

WCET estimation cannot be derived by separately studying the timing behavior of each component. The lack of this particular property prohibits the timing analysis problem to be addressed in a composable manner, adding further complexity to the issue. Due to these reasons, designers of safety-critical systems might sometimes be confronted with a paradoxical situation, where introducing high-performance features eventually leads to devastating loss in performance itself.

A typical workaround used in many system designs would consist of merely disabling these hardware components, sacrificing overall performance over tight and more predictable worst-case timing behavior [20]. Any more need in computing resources would be filled by introducing more embedded computers with, unfortunately, impeded performance. The result is often excessively complex electronic structures consisting of several distributed computers that need to interact and share resources with each others. One has just to take a look at the electronic architecture on modern vehicles to realize how complex this may turn out to be. It is estimated that modern cars contain around 50 embedded computers that communicate through networks consisting of kilometers of wirings and kilograms of copper [1]. To mitigate this issue, more attention was directed toward multi-core platforms, especially after the success they showed in the general-purpose computer market during the beginning of the millennium. Due to their hardware parallelism capabilities, these new computer architectures promise more computing power with little to no overhead as to cost, space, and power usage. This seems very convenient for many industrial applications. However, some issues may emerge when designing software for multi-core architectures. In fact, software that used to run in uniprocessor systems may not always scale well and perform better when ported to multi-core systems. Cache coherence, synchronization, and distribution of software functionality across computing resources are examples of typical issues that a software designer has to deal with in order to build a safe but yet efficient system. Due to hardware resource sharing, dependencies between applications may show up making it almost impossible to get timing guarantees without seriously impacting the performance advertised by these architectures.

On the other side, societal expectations for increased technology performance is constantly growing. The safety-critical systems are requested to carry more demanding features in need of computing power. In the automotive industry, for instance, autonomous vehicles must implement automated driving systems, responsible of processing huge amounts of information provided by networks of sensors and cameras in real-time, in order to make split-second driving decisions. Meanwhile, the driver is invited to take the seat of a mere passenger, enjoying the ride, and maybe all kinds of entertainment services provided by the vehicle's computer system. The direct implication of this growth in functional requirements is of course a dramatic increase in size and complexity of software to implement them. This phenomenon is observed across all sectors and has been well captured by a NASA study that covered various projects in the space and avionic domain [36]. In fact, the apparent increase in performance of modern computer architectures seems to encourage industries to rely on software to perform demanding safety-critical tasks.

Many computer vendors start to propose new computers with unprecedented levels of performance to keep pace with these high expectations. The brand new Nvidia computer Drive PX [6] is an example of such hardware platforms providing original computing capabilities aimed to support autonomous driving functionalities. It

embarks four high-performance artificial intelligence processors capable of performing 320 TOPS[3] of deep learning operations. But what is the meaning of performance anyway? Are such raw computational metrics a good way to realistically appreciate the performance of computers in the context of safety-critical systems? The experience accumulated in the matter seems to suggest otherwise, and the foreseen increase of software complexity is only going to get the process even more complicated. In fact, seeking more computing power without effectively addressing the problem of timing analysis deems most of the performance to be stuck in the *potential* bucket and never get a chance to be exploited in real life.
The work in this thesis is part of the efforts to reconcile performance with safety-critical systems and unlock the potential modern computers have to offer. Prior work already investigated computer architectures tailored for timing analysis [38, 120, 53, 91]. However, the T-CREST project [108] takes a comprehensive approach to performance in the context of real-time systems. In fact, it is based on the observation that performance depends not only on the architecture, but also on the timing analysis tool as well as the compiler support. Patmos is a time-predictable processor resulting from the T-CREST project. It implements special features and hardware components intended to simplify the analysis process and derive tight WCET estimates. Moreover, a port of the LLVM compiler infrastructure to Patmos is provided, which allows to introduce and perform WCET-related analyses and optimizations. We participate in this effort by providing several missing timing analyses for its time-predictability features. In this process, we also introduce hardware extensions that aim not only to derive more precise bounds, but also to reduce the actual WCET. Most of these techniques are integrated in our WCET analysis tool Odyssey. The outcome of this work may, ultimately, help defining computer architectures that are suitable for high-performance safety-critical applications. A detailed list of contributions is provided in Section 7.

2 Real-Time Systems

Safety-critical systems are often subject to timing requirements that should be met in order to avoid failure. Such systems require their inputs to be regularly controlled so that the adequate response is issued at the right moment, i.e., in real time. Computer systems operating according to that scheme are known as real-time systems. They retain special properties and functionalities necessary to handle temporal and functional aspects of the containing safety-critical system. What do we mean precisely by the term "real-time" and how do these systems manage these aspects?

2.1 Definition and Key Properties

In the context of computing, the term "real-time" might be the subject of misinterpretations among people outside specialized communities. While many attribute to it qualities such as speed or how fast some computer system may react to user stimuli, this picture does not capture the true properties and characteristics this

[3]Trillion operations per second. Although this metric is straightforward, it only represents peak execution rates of instructions performed by the CPU without considering other important elements of the computer architecture and the program.

concept actually grants. More exactly, computer systems qualified as real-time refer to those in which the logical correctness of the results they provide, and the time by which these results are provided, are of equal importance. It is a simple concept for which many analogies could easily be found in everyday life. For instance, in many combat sports like boxing or fencing, marking points is usually decided by how the player is successful in executing the right action within the right time frame. Surely the reaction time frame is usually perceived as *fast* with respect to human standards, however, depending on the situation this may not always be the case. What if we consider now the example of a ratepayer requested to submit tax returns within the next two months? The reactiveness in the earlier example is not more "real-time" than that in the later one by any aspect. A failure to correctly file and deliver the requested documents within the imposed time frame will probably lead to undesirable consequences. Applying this concept on computer systems is extremely useful in applications requiring the physical environment to be permanently controlled in order to produce and maintain the desired behavior. In that regard, the reaction speed to stimuli is merely dictated by timing requirements relative to the environment's dynamics. Such requirements are often characterized by deadlines by which the correct response has to be produced. The timeliness of a real-time system expresses how successful that system is in meeting its timing constraints.
Given the criticality of some applications, it is necessary to derive proofs as to the timeliness of the system before actually running it. However, not everything can be known during design time. Many factors can influence the behavior of the system, especially the evolution of input data. These uncertainties introduce non-determinism that has to be considered in the verification process. Achieving this goal requires the system to be predictable. This means that it must be possible to predict, at least to some extent, facts or properties about the system's evolution. Timing predictability expresses the degree of certainty regarding the timing behavior of the system. Therefore, it is desirable that the system allows to derive precise predictions with respect to the actual timing behavior. A system that lacks an adequate level of predictability will not be certified for use in safety-critical systems.
That being said, not all real-time systems are intended for safety-critical applications. Many do not require highest levels of timeliness to operate in acceptable conditions. Depending on the strictness of timing requirements, real-time systems can be categorized into three kinds:

- **Hard real-time systems:** as the name suggests they are subject to hard timing constraints, meaning that the system experiences a failure if any of the deadlines is missed. Hard real-time systems are typically used in safety-critical applications and have the highest levels of reliability and timeliness. Application examples include nuclear plants, power-train systems, flight control systems, and spacecrafts.

- **Firm real-time systems:** a result that comes after a deadline is useless. Missing the deadlines does not cause serious damage. However, the performance of the system might be severely degraded. Examples of firm real-time systems include robotic assembly lines or sensory data transmission.

- **Soft real-time systems:** deadline misses may be tolerated. A late result is still useful, although a degradation of performance may be noticeable. Soft real-

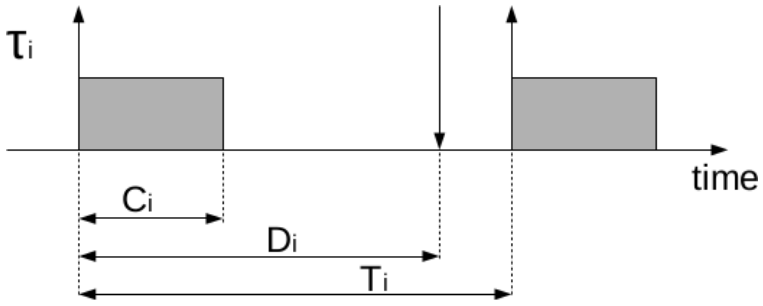


Figure 1.1 – Timing characteristics of task τ_i. T_i is the period of activation, C_i is the computation time, and D_i is the deadline relative to the activation date.

time systems are typically used in consumer applications such as audio-video streaming or video games.

In this work, we focus on hard real-time systems.

2.2 Embedded Systems

Practically speaking, most real-time systems are realized using embedded computer systems. In contrast to general-purpose computers, embedded computers are dedicated for specific applications or products. They are designed to carry-out distinct functionalities delivered by a larger system like those mentioned earlier. Like any computer system, they are essentially composed of many hardware and software parts. Performance and characteristics of embedded systems range widely, depending on application domains. For consumer applications, they usually consist of System on Chips (SoC) with multi-core processors, along with advanced peripherals such as graphics processing unit (GPU), and connectivity modules. Examples include Texas Instruments' *OMAP* and Qualcomm's *Snapdragon* Series. In real-time applications, embedded computers traditionally consist of simple micro-controllers with basic elements of a computer architecture. These computers typically operate at frequencies under 100MHz, and carry a memory system of several KBytes. However, as computing requirements increased lately, many industries get interested in deploying computers delivering high-performance capabilities. Among them, we can cite Renesas' multi-core platform *RCAR-H3*, or Kalray's many-core platform *Bostan*. In any case, embedded computers are often subject to constraints regarding hardware resources, price, space, or perhaps power consumption. For this reason, the embedded software is usually optimized for the targeted embedded computer architecture. In fact, software designers have to be mindful as to the strict needs of the application being developed. In some cases, an operating system may even not be required to properly implement and operate the desired functionalities.

2.3 Real-Time Tasks

Real-time systems typically control many aspects of system inputs at the same time. To achieve reactiveness, a real-time system may consist of several tasks that execute periodically, in a parallel fashion. A task is the basic unit of execution, and merely consists of a software program that processes input data to provide results. Depending on the adopted task model, each task is associated with different

parameters capturing their timing characteristics. Figure 1.1 illustrates some of these parameters for some task: (1) the period of activation T_i, (2) the CPU time needed to fully execute the task C_i, and (3) the deadline relative to each of its activations D_i. The real-time operating system (RTOS) manages the computer's hardware resources (such as CPU time) and allocates them to tasks according to their respective timing requirements. This is done using task scheduling that consists of finding a correct execution order of active tasks that guarantees the satisfaction of all their timing constraints. Scheduling has been (and still is) the subject of an extensive amount of research since the mid-20th century. Many scheduling algorithms exist in the literature along with several studies detailing advantages and drawbacks of each class of algorithms. However, static scheduling is often preferred in safety-critical systems due to its predictability and implementation simplicity [73]. In static scheduling all tasks' parameters involved in scheduling decisions are *known* in advance and may not change during the system's execution. An example of a static scheduling algorithm, is called Rate Monotonic (RM), where task priorities are assigned according to the task's period of activation, commonly denoted as T_i. Tasks with shorter periods have higher priorities to execute.

2.4 System Failure

Like any computer system, real-time systems are not totally spared from design flaws and bugs that may compromise one or many of their aspects. There are many examples of incidents that occurred in the past due to complications resulting from what may appear as mere software bugs [8]. While some of these flaws can today be detected using advanced software checking and testing techniques, failures due violation of timing requirements may have different causes that cannot be spotted via such methods. Missing a deadline can be the result of circumstances that the software designer did not anticipate. For instance, a task that took more time than expected to complete its execution. Or, a high peak load that the system cannot handle. Temporal aspects are particularly hard to verify in situations where the environment to be controlled is not reproducible in the lab. This is unfortunately often the case, and given the criticality of some applications, the blue screen of death is usually not an option. In the next section we discuss how timing guarantees could be derived in order to ensure the timeliness of real-time systems.

3 Deriving Timing Guarantees

One of the main concerns of building real-time systems, is to ensure the satisfaction of all their timing requirements. Due to many uncertainties, it may not be possible to predict with absolute precision the evolution of every aspect of the system. Therefore, some precision sacrifices are usually necessary, leading the designer to rather consider the worst-case scenario. Generally, the approach to derive timing guarantees consists of showing that in the worst-case all tasks fully terminate their execution by their respective deadlines. A good system design is thereby characterized by its ability to make reliable predictions based on the results obtained for the worst-case. In any case, the problem has to be addressed at all levels of the computer system.

3.1 Schedulability Analysis

Task scheduling offers means to find an execution order of tasks depending on their timing characteristics. However, it does not necessarily guarantee, on its own, the satisfaction of timing requirements[4]. Very often, a schedulability analysis is required to verify that no deadline is missed. Schedulability analysis is usually expressed by the mean of a schedulability test that answers the following question: Given a system of tasks and a scheduling algorithm, will all tasks meet their individual deadlines? In that regard, the test function takes as input the timing characteristics of tasks, and attempts to provide a positive or a negative answer as to the schedulability of the system. Equation 1.1 provides an example of a schedulability test for the Rate Monotonic algorithm [79]. This simple test is based on the total CPU utilization of the system, and a positive response is sufficient to declare all deadlines to be met. There are many other approaches to schedulability analysis. For more details, a comparative study of RM schedulability tests can be found in [35].

$$U = \sum_{i=1}^{n} C_i/T_i \leq n(2^{1/n} - 1), \tag{1.1}$$

where C_i is the computation time, T_i the activation period of the i-th task, and n the total number of tasks.

From this standpoint, schedulability tests may determine if a system of tasks successfully meets all its timing constraints. However, it is important to note that the response provided by the test directly depends on the tasks' parameters and how they are chosen. To achieve this, the system designer has to combine timing characteristics of both the environment to be controlled, and the computer performing the needed computations. This is typically done by first formulating timing requirements given the dynamics of the environment. Then, choosing the adequate computer system allowing to carry-out tasks defined by the software architecture within the timing requirements. In particular, the parameter C_i is tied to the computer architecture and its determination is crucial to verify the timeliness of the system. Failing to correctly fill this value may cause the schedulability analysis to provide a wrong answer, potentially leading to a system failure. Assuming a scheduling algorithm free of anomalies[5], this falls into determining the worst-case execution time (WCET) for each task program.

3.2 Timing Analysis

Intuitively, the determination of the WCET may look as a matter of execution time measurement. Some approaches in fact consist of performing dynamic analysis based on measurements. The principle consists of running the task, or parts of it, under some circumstances and perform measurements of the execution times

[4] A very simple example would be to consider a system of periodic non-preemptive tasks where some task never terminates its execution. Once started, the blocking task prevents all other tasks from execution and subsequently to meet their individual deadlines.

[5] A scheduling algorithm suffers from anomalies when any positive effect on the system, such as a relaxation of timing constraints or additional hardware resources, leads to an increase of task response times.

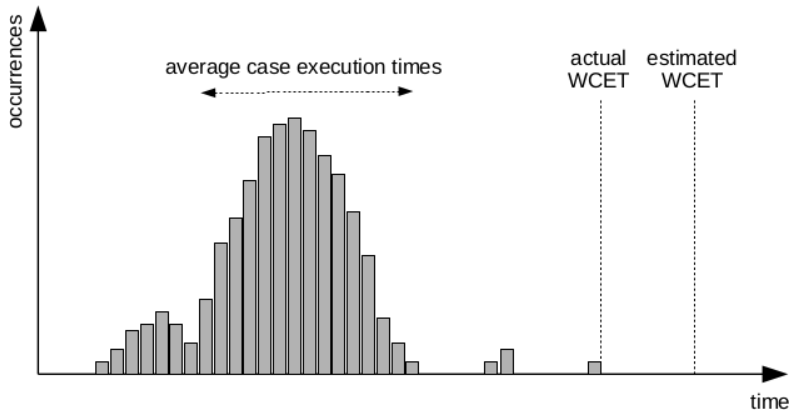


Figure 1.2 – Distribution of execution times. The actual WCET has to be upper-bounded by the WCET estimate provided by the timing analysis. The precision of the analysis determines the tightness of to the actual WCET.

in order to derive maximum bounds. While this method is simple and provides accurate execution times based on the actual hardware platform, it has some major issues. What exact circumstances should be applied in order to produce the WCET? This question is hard to answer as many factors influence the execution time. The evolution of system inputs as well as the initial hardware state may have a direct or indirect implication. Consequently, not choosing the right circumstances may lead to optimistic estimations for the WCET, and invalidate the results of the schedulability test. A typical distribution of execution times related to a real-time task is shown in Figure 1.2. Measurement-based approaches relying on optimistic circumstances will probably observe an execution time below the WCET. Conversely, the worst-case behavior is usually extremely rare to encounter. This rarity though does not constitute an argument to dismiss it, as given enough time, the occurrence of rare events become more probable. Therefore, there is a need in deriving the WCET using more reliable approaches to timing analysis.
To achieve this goal, it is first important to capture all factors involved in the execution time variation. Let us recall that a real-time system consists of a set of tasks running in a computer system. A task is basically a program performing computations on input data and producing results. The program is stored in the computer system that executes its instructions in a sequential order. Hence, the time spent on executing the program depends on these factors:

Input Data: Many of the program decisions are data dependent. The evolution of input data may thus cause the program to process data differently, leading to different execution times.

Program: The program may consist of many ”paths“ each representing a possible execution scenario. Depending on the chosen path, different operations may be performed leading to a variety of possible execution times.

Computer Architecture: The time spent on executing each instruction of a program path is a function of the computer architecture and the performance of its hardware implementation.

Other Tasks: Scheduling organizes tasks execution on the same hardware platform. When some task is executed, it changes hardware states, which may have an

influence on the execution time of subsequent tasks.

Therefore, determining the WCET requires the analysis to consider the combination of circumstances, on all these factors, that lead to the worst-case timing behavior. Safety-critical systems require maximum confidence about the derived timing guarantees. Also, they have to be obtained statically, i.e., before running the system. This process has therefore to be formally established using analytical methods. Static program analysis offers means to satisfy these two requirements. Static program analysis is a generic method [30, 48] that consists of determining facts or properties about the dynamic behavior of a program without actually running it. The idea consists of gathering information about possible execution scenarios of the program and then to use that information to track and predict the processor's timing behavior. The WCET will then correspond to the worst-case timing behavior produced by running the program on that specific processor.
That being said, uncertainties and precise predictions are two opposite things. Not everything can be known in advance regarding the aforementioned factors. For instance, some CPU vendor may decide not to provide every detail about the computer architecture. Furthermore, the evolution of input data is by definition non-deterministic in most systems. The implication on the timing analysis is that the determination of the exact WCET becomes often intractable. Rather, the timing analysis aims to provide an upper-bound estimate of the actual WCET. The good news is that static program analysis can take these uncertainties into consideration in order to conservatively derive those worst-case bounds. However, this is done at the expense of precision. The gap between the actual WCET and the WCET estimate is known as *pessimism*. Pessimism is the natural response of timing analysis that needs to be conservative in the presence of uncertainties. Therefore, it is important to keep them as low as possible for tight WCET estimates.
We cover more in detail the WCET analysis flow and its different steps in Chapter 3.

4 Meaning of Performance

Computers are built to process information and perform computing tasks. Evaluating how good computers are in performing these tasks is regarded as computer performance. Studying computer performance allows to spot performance bottlenecks within the computer system in order to build computers that are a better fit for the application. For this, the meaning of performance has to capture all important criteria relevant for the application in question. In real-time systems, time is one of the most primordial aspects that need to be considered for performance. However, the fact that real-time systems are computer-based systems impels us to consider performance in a more general manner. This section provides a clarification as to the meaning of performance in our context and whether standard architectures are a good fit for real-time applications.

4.1 Performance in Standard Computer Systems

During the past six decades, *making the common case fast* has been one of the design philosophies that were prominent in computer architecture [31]. This idea makes sense, because computing resources are limited, so they better be exploited

for what the user experiences most frequently. Making the common case fast is usually addressed by improving either the throughput, i.e., the amount of work done in a unit of time, or the observed execution time as previously depicted in Figure 1.2. Comparing computer systems with respect to these two aspects requires looking at the computer subsystem in a basic level and spot the involved factors. In the single CPU model, there are basically three low-level factors contributing to performance: (1) the operating frequency, i.e., the clock rate to run the CPU, (2) the instruction count, that is, the total number of instructions executed to fully complete the program, and (3) the average clock cycles per instruction (CPI) as each instruction may take a different amount of time, depending on operations they induce. Many of the software and hardware performance enhancing features that were gradually introduced during the single CPU era aim at enhancing these factors. In the micro-architectural level, making the common case fast has been achieved through the introduction of extra hardware components to the basic computer architecture. The general idea behind all these techniques is to make the CPU execute multiple instructions in a single CPU cycle, thus reducing the average CPI number. Here below we list some of these components, and how they improve computer performance:

Pipeline: A technique that implements a form of parallelism known as *instruction-level parallelism* (ILP). It works by dividing the execution of incoming instructions into different stages, allowing multiple instructions to overlap part of their execution (see [31] Section 4.10).

Branch Predictor: Attempts to predict the outcome of a branch. Instead of waiting for the branch to be resolved in a late stage of the pipeline, a branch predictor guesses its outcome so that the CPU can proceed with its execution (see [31] Section 4.8).

Superscalar pipeline: This is a more advanced form of ILP. Multiple-issue pipelines enable the CPU to execute more than one instruction per cycle by dynamically scheduling instructions, while respecting data/control dependencies (see [31] Section 4.10).

Cache: A fast and small memory structure holding frequently referenced data. Caches are part of the memory hierarchy, and provide a fast access to cached data preventing the CPU from stalling due to time consuming requests to main memory (see [31] Section 5.3).

Prefetcher: Attempts to anticipate future memory accesses by fetching the needed data from the main memory to the cache, before it is actually requested (see [31] Section 5.16).

These hardware components are popular techniques to enhance performance and thereby are found in almost every modern computer architecture. However, it is important to note that some of them rely on assumptions, speculations or predictions that may turn out to be true, but also may turn out to be wrong, depending on the behavior of the program. Logically, their individual effectiveness has a direct implication on the overall performance of the system. Therefore, specific metrics for each component can be defined in order to study their behavior. For caches usually the miss or hit rates are used, whereas the mis-prediction rate is often used

to compare branch predictors. Huge research efforts were conducted in order to come up with strategies based on heuristics that attempt to maximize the profit in different situations. For instance, the Least-Recently Used (LRU) cache replacement policy is most effective when accesses to the same data are likely to happen again soon. The same policy may however exhibit side effects if data are accessed in a different pattern.

Despite the introduction of these innovative techniques in computer architectures, increasing the frequency has been a more tempting approach to achieve performance. However, due to power consumption issues that emerged at high frequencies, other approaches to performance had to be found. This is when multi-core architectures emerged. Multi-core architectures provide *task-level parallelism* (TLB), a technique through which tasks or sequences of instructions can run simultaneously, taking advantage of the potential parallelism offered by additional cores. There exist different ways to exploit the parallelism in order to enhance different aspects of system performance. One way consists of scheduling different programs in dedicated cores, which favors the utilization of subsystem resources and improves global responsiveness. Another way is by speeding-up the execution of a particular program by dispatching parts that can be parallelized over different cores. Multi-core architectures provide potential to considerably increase performance. However, this does not come for free, nor without concerns. For instance, any speedup improvement depends on the degree to which the program can be partitioned into parallel tasks that may run cooperatively. This implies the use of communication protocols and synchronization mechanisms that usually come with extra latencies and overheads. Contentions may also show up due to resource sharing among cores. To many of these problems, a good answer cannot be guessed by the computer architecture on its own. The programmer is requested, more than ever, to consider computer architecture to achieve performance, exposing many difficulties that were once abstracted. The software is thereby another contributor to performance.

Software can intervene at various levels of the computer system. In a high level, the organization of software architecture can have a considerable impact on performance. Decomposing the problem into smaller tasks can not only simplify the software development process, but also helps in allocating appropriate hardware resources for specific tasks and achieve better utilization of the system. Parallel programming is necessary to take advantage of underlying parallel hardware resources and achieve higher throughput. The algorithm used to solve a particular problem directly affects the system performance. Programmers use high-level programming languages such as Java, Python, or C++ in order to implement algorithms, which require (optimizing) compilers.

Compiler technology is used to transform the high-level language to a low-level language runnable by the machine. The compiler therefore has a fine knowledge of the program behavior, the underlying computer architecture, and its instruction-set architecture (ISA). It is a crucial component of performance as it affects both the instruction count and the average CPI count in various ways. This is typically done through optimizations that attempt to maximize or minimize certain aspects of the program such as the code size or its execution time. For instance, the compiler may inject special instructions in particular locations of the program telling the prefetcher to fetch data that will be used in the near future. In this case, the code size is increased. However, it potentially saves valuable CPU cycles spent on cache misses.

More about high-performance computing, challenges, and trends in the future can be found in this report [101].

4.2 Worst-Case Performance

Real-time systems are mostly concerned with respecting timing. Specifically, whether the system meets all its deadlines in the worst-case as discussed previously. Schedulability analysis combined with timing analysis can provide a positive or a negative answer. However, is a binary response enough to evaluate performance in the worst-case? Probably not. In fact, behind this response lays also the precision of the WCET estimates provided by the timing analysis. Pessimism represents imprecisions, that is the gap between the actual WCET and the estimated WCET bound. High amounts of pessimism can lead to devastating loss of performance, increase the cost, or deem the system infeasible when it perfectly is. For instance, a system for which its WCETs are overly pessimistic may cause the schedulability analysis to emit a negative response. Possible solutions may include reducing the workload through the removal of some features initially provided by the system, or deploying faster computers, probably more expensive, more energy consuming, and for which the resources are actually under-utilized. Improving the worst-case performance thus requires dealing with pessimism.
In addition to the precision of WCET bounds, reducing the actual WCET is also desirable. The main function of a computer is still to process information and perform computing tasks. In that regard, the actual WCET is still an interesting aspect to look at. Indeed, systems requiring a maximum control on timing behavior, not necessarily interested in performance, can rely on simple embedded processors. However, this might not be an option in complex projects where portability, productivity, and short time-to-market are required. We want to exploit performance enhancing features offered by the platform just as in standard computer architectures.

Computer Architecture: In a basic level, achieving more precision along with a reduced WCET relies on the computer architecture. On the one hand, the timing behavior of the processor is a function of the micro-architecture and its subsystems. On the other hand, the behavior of individual hardware components or the architecture as a whole may introduce uncertainties. These uncertainties are due, for instance, to a complex behavior that is hard to track by the timing analysis. In general, the predictability can be seen as the degree of certainty that we have regarding the future hardware states of the architecture. In order to increase predictability, while keeping the timing analysis simple, the architecture has to justify: (1) *timing-composability*, i.e., the timing behavior of a particular component is independent from that of other components. (2) *timing-predictability*, i.e., the ability to derive useful worst-case timing behavior of it hardware components, with reasonable overhead. The *random* cache replacement policy found in Cortex-R4-core-based processors [2] is not predictable as its behavior prevents a static prediction of its future states.

Predictability is thus an essential component of performance. But is predictability quantifiable? This question has been specifically investigated in [107]. It is argued that predictability on its own does not allow a meaningful comparison between computer architectures. We believe this is true because the process of deriving timing

guarantees involves more than the computer architecture itself. In fact, there are other factors that are worth the attention on an equal footing:

Timing Analysis Tool: This is obvious, the quality of the analysis tool certainly affects the results it provides. Timing analysis involves many successive steps during which lots of information is gathered, but also lost due to abstraction. Abstractions are often necessary to make the analysis practically feasible. Depending on the discarded information, some pessimism may sneak in due to the conservative nature of the analysis. Possible sources of uncertainties may include control-flow joins or simply the absence of an appropriate analysis that takes fully advantage of capabilities offered by a particular hardware component. In any case, there is always a trade-off to be found between the precision of the analysis and the induced overhead, i.e., time and memory footprint. Therefore, two timing analysis tools considering different sets of information, and adopting different analysis approaches, will probably yield more or less precise WCET estimates. Moreover, a timing analysis tool that takes overly long to provide extremely tight results might not be useful. We review some existing WCET analysis tools in Chapter 3.

Compiler Support: As in standard computer systems, compiler technology plays an important role in influencing worst-case performance. Through program analysis, the compiler gathers information regarding the program and its behavior. Therefore, it is a natural candidate to automatically perform WCET-related optimizations. This is done, for instance, through the reduction of execution time variations using techniques such as predication. A radical way to this approach is the *single-path programming* [97, 95]. This technique aims to produce a code with constant execution time in all circumstances, avoiding the need of timing analysis. Alternatively, the compiler can perform optimizations that help the timing analysis yield more precise results. A first attempt to WCET-aware compilation was proposed by Falk et al. using the WCC compiler [42]. WCC integrates the WCET analysis tool aiT [43] and provides a mapping between the internal program representations of the compiler and the analysis tool. In another work [24], Brandner et al. investigated means to identify program paths that are critical with respect to the global WCET. The *criticality* metric is obtained through a set of static analyses on the program. This opens an opportunity for the compiler to target parts of the program that are more relevant for optimizations.

The computer architecture, the timing analysis tool, and the compiler technology, all combined, constitutes building blocks of performance in our context. Their individual effectiveness and interplay will be reflected in the reduction of both the actual and upper-bound WCET.

5 Issues with Standard Architectures

Standard computer architectures are designed to make the common case fast. Real-time computing, on the other hand, requires worst-case performance. Historically, real-time applications relied on standard architectures that were once simple and very time-predictable. Today, however, these architectures evolved to include many sophisticated hardware techniques whose behavior depends on a long execution history, and dynamically tweak hardware states. As computing requirements increased in real-

time applications, it sometimes became obvious that these two visions of performance cannot cohabit, or even be compatible. By making the common case fast, techniques deployed in standard computer architectures tend to penalize infrequent cases by excluding them from performance enhancements. The worst-case is usually among the most infrequent ones. The induced time variation might therefore be spectacular and not help the timing analysis deriving precise WCET bounds. Here below we identify some important issues related to timing-analysis on conventional architectures.

Lack of timing-composability: One obvious problem observed in modern computer architectures is that they lack timing-composability. The behavior of some hardware components may have a major influence on others. The actions taken by the prefetcher might change the data cache state and potentially its future timing behavior. A guess from the branch predictor may cause the out-of-order pipeline to continue speculating on the predicted branch, leading to a variety of complex behaviors each with different outcomes on the execution time. Enumerating and analyzing these potential interactions introduces a considerable amount of complexity. To keep the analysis practically feasible abstraction is often needed. However, this usually comes at the expense of precision. Nevertheless, some efforts were conducted in order to model such behaviors. A model of an out-of-order processor is provided in [76]. However, there is some skepticism regarding the applicability of the approach [106].

Timing-anomalies: An unfortunate consequence of aggressive hardware optimizations is the presence of *timing anomalies*. A timing anomaly is a counter-intuitive situation where a local positive (negative) effect on timing behavior, induces a global negative (positive) effect on the execution time of the whole program. For instance, a cache miss that is a local negative effect may, in some circumstances, lead to a shorter execution time. Timing anomalies prevent the analysis from relying only on the worst-case timing behavior of individual instructions to compute a safe WCET bound. This adds even more complexity due to state-explosion. The presence of timing anomalies therefore impacts the applicability of timing analysis methods. All modern computer architectures exhibit timing anomalies. Processors based on out-of-order execution are found to cause timing anomalies [76]. The same holds for modern two-level branch predictors, where it is observed that a decrease in the number of loop iterations can actually increase the execution time [40]. Pseudo Round-Robin Cache policy is also found to cause timing anomalies even in an in-order execution scheme [116].

Unadapted compiler support: The compilers used for standard computer architectures traditionally perform optimizations to speed-up the common case. This introduces further variation on execution time as the worst-case is usually not optimized. Furthermore, it is not always possible to extract *raw* information from the already optimized machine code. This information has been discarded during the compilation process, maybe due to its irrelevance with respect to the common case performance. Also, it is sometimes difficult to tell what optimizations were exactly applied and transformations they induced. This makes it hard to map the source code to the machine code on which the timing analysis is actually performed. Therefore, it might not be possible to highlight, for the programmer, the parts responsible for performance problems.

Lack of documentation: Another recurrent concern when using standard com-

puter architectures is the absence of a detailed documentation that precisely describes the micro-architecture and the timing behavior of instructions. This documentation is necessary to build an accurate model of the processor, based on which precise and safe WCET estimates could be derived. Unfortunately, processor vendors sometimes do not provide such level of details, which may compromise the validity of the whole timing analysis approach [104].

The NGMP case: An example of conventional architectures sensitive to some of these issues is the *Next Generation Microprocessor* (NGMP). The NGMP is a multi-core architecture that was designed by Cobham Gaisler and the European Space Agency to enable future space missions. The platform is intended to carry-out applications with hard real-time requirements. In its current implementation, the GR740 [4] combines four LEON4 cores each implementing a single-issue in-order pipeline with an always-taken branch predictor. The memory hierarchy provides two levels of caches: two private L1 caches per core dedicated to instructions and data, as well as an L2 cache shared between cores. Moreover, a bus arbiter running a round-robin policy connects the L2 cache to the memory controller. In a quantitative study that evaluated inter-core interferences in the NGMP platform [46], it was observed that benchmarks exhibit slowdowns up to 20x when compared to an execution in a single-core configuration. The main issue was related to heavy contentions in the bus resulting from memory transfers issued by tasks being executed in different cores. Due to the round-robin policy, the timing behavior of some task may depend on the number of stores performed by other tasks and whether all their data fit into the L1 cache. This lack of timing-composability as well as the existence of an L2 cache makes the WCET analysis extremely complex as it needs to account for potential interferences in order to provide safe bounds.

Analyzing the processor's timing behavior in standard computer architectures is a difficult matter. The induced complexity makes the timing analysis pessimistic, if not infeasible. New alternatives were to be found.

6 Predictable Architectures as an Alternative

From the realization that conventional architectures, as of today, might not be the best fit for hard real-time applications, efforts recently multiplied to investigate new architecture philosophies. In particular, the so called *time-predictable* architectures are designed with worst-case performance in mind. The promise is that by providing hardware support for predictability, one can build a WCET analysis that is simpler and more precise. We review in this section most notable efforts and compare the architectures they resulted in.

6.1 Notable Efforts and Architectures

Many architectures have been proposed so far with a focus on different aspects and applications. Among these we can most notably cite:

SPEAR: The *Scalable Processor for Embedded Applications in Real-time Environment* (SPEAR) [34] is one of the first architecture designs for time-predictability. The architecture is rather simple: a RISC ISA with partial predication, a three-stage

single-issue pipeline, and a memory hierarchy consisting of on-chip ROM/RAM memories and no caches. The architecture, on the other hand, supports the single-path programming paradigm. Using predication, one can produce a code with a unique execution path whose execution time is independent of input data. As a result, the determination of the program's execution time can merely be done through measurement. This eliminates the need of building a complex timing model for the processor as the WCET analysis is no longer necessary. A downside of this technique is the waste of CPU cycles executing long blocks of instructions whose predicates evaluate to false.

PRET[6]**:** *PREcision Timed* is a project that aimed at providing a class of computer architectures for *repeatable-timing*. Repeatable-timing is the ability to repeat the exact timing behavior of the system if given the same inputs [37]. Edwards et al. argue that timing should be a repeatable property of a program, not of the program-processor couple. This concept is therefore different from time-predictability, which is interested in providing safe and tight WCET bounds given a program and an architecture. To support repeatable-timing, the PRET architecture equips the ISA with timing semantics and control over timing instructions. An example is the *deadline* instruction that stalls some thread until a lower bound deadline is reached. Moreover, branch and data hazards are eliminated by relying on a *thread-interleaved pipeline* (TIP). The TIP is a hardware multi-threading technique according to which each thread occupies at least one stage of the pipeline. Every cycle, the pipeline fetches an instruction from a different thread according to the round-robin policy. The memory hierarchy avoids using caches to eliminate related timing variations. Instead, instruction and data scratchpad memories (SP), which are statically managed by the program/compiler. Also, a predictable DRAM controller [99, 80] is introduced to ensure repeatable-timing behavior for the DRAM memory. The PRET architecture may consist of a multi-core platform. In such case, accesses of cores to the main memory is arbitrated according to the TDMA (Time-Division Multiple Access) policy. According to this scheme, every core is granted the access to memory in a dedicated time slot. The PRET principles were implemented in different machines targeting different applications [78, 126]. In particular, the PTARM [80] is better suited for hard real-time systems. The PTARM architecture is based on the ARMv4 ISA that can be extended with control over timing instructions. PTARM implements a five-stage in-order pipeline allowing to interleave up to four threads.

PREDATOR[7]**:** is a consortium that investigated computer architectures and methodologies to develop safety-critical systems. The project did not result in a new computer architecture, however, the contributors provided design principles to build processors tailored for timing analysis [120]. Moreover, they covered topics related to timing analyses for caches [51, 100, 27] and proposed compiler support for WCET-driven optimizations [42].

CoMPSoC[8]**:** is a template platform allowing to generate multi-core SoC architectures satisfying timing-composability and predictability [53]. The goal here is to be able to develop and validate applications independently from the rest. An application

[6] `https://ptolemy.berkeley.edu/projects/chess/pret/`
[7] `https://www.predator-project.eu/consortium.htm`
[8] `http://compsoc.eu`

consists of a set of tasks that may be mapped to different processor cores for parallel processing. The template dedicates to each application a virtual platform consisting of partitioned physical resources. Those resources could be either CPU cores, interconnect, or memory. Depending on the resource type, different partitioning techniques are proposed to guarantee time-composability between virtual platforms. In particular, a preemptive TDMA schedule is statically generated allowing each application to use a processor core on dedicated time frames. Moreover, resources are restored or reset to a neutral state between scheduling intervals. The platform is based on MicroBlaze cores implementing a conventional RISC ISA. Branch delay slots are supported and can be exploited using the proprietary MicroBlaze tool-chain. The characteristics of the pipeline are not explicitly mentioned, but according to MicroBlaze manual reference [5], it consists of an in-order single issue pipeline with either 3 or 5 stages. No caches or SP memories are used, the branch predictor is turned off as well. The NoC connecting the cores operates asynchronously, meaning that the NoC and each resource run on their own frequencies. To ensure time-predictable behavior of the DRAM, the CoMPSoC platform uses the Predator DRAM controller [15].

MERASA[9]**:** The *Multi-Core Execution of Hard Real-Time Applications Supporting Analysability* is a project that focuses on time-predictable SoC designs and tools for timing analysis. The MERASA architecture [91] targets mixed-criticality systems with hard real-time (HRT) tasks and non-hard real-time (NHRT) tasks. In particular, the SoC architecture combines CarCore cores [86], which are basically Infinion TriCore cores customized to support simultaneous multi-threading (SMT). The SMT relies on a super-scalar in-order pipeline capable of issuing up to two instructions at a cycle. The processor is able to issue two instructions in parallel from the same thread if an *address* instruction is followed by an *integer* instruction. Otherwise, the pipeline fetches instructions from two different threads. To each core one HRT task and up to three NHRT tasks are mapped. The isolation of the HRT task is ensured by prioritizing its execution in all pipeline stages. Moreover, the HRT task does not profit from caches. Instead, they rely on SP memories to perform fast and predictable accesses to instructions and the stack data. In particular, the dynamically managed instruction scratchpad (D-ISP) loads whole functions on calls and returns [84]. The D-ISP behaves pretty much like the method cache [33] with a FIFO replacement policy. In contrast to the method cache, the D-ISP has to make sure that the size of each individual function fits inside its memory. In the multi-core level, a real-time bus arbiter is deployed to handle memory requests from different cores. The bus arbiter implements different policies to schedule core requests, i.e., Round-Robin (RR), Fixed Priority (FR), and FIFO. Furthermore, the arbiter can be configured to force a bounded memory latency for requests issued by HRT tasks [92]. MERASA also introduces a DRAM controller providing a predictable memory access time [90]. A follow-up project called *Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability* (parMERASA[10]) has been initiated. The main goal is to provide tools and software support for parallelization in NoC platforms with up to 64 cores. The architecture combines commercial off-the-shelf PowerPC cores.

[9] `http://ginkgo.informatik.uni-augsburg.de/merasa-web/`
[10] `https://www.parmerasa.eu`

T-CREST[11]**:** The *Time-Predictable Multi-Core Architecture for Embedded Systems* project aims at building a time-predictable and composable platform with a focus on single-thread real-time performance [108]. The work is based on the observation that the architecture alone may not be sufficient to achieve best worst-case performance [107]. On the one hand, a strong compiler support that is aware of the hardware possibilities is needed to enable WCET-related optimizations. On the other hand, the WCET tool has to implement timing analyses for the time-predictable hardware to provide tight WCET estimates. The T-CREST project took interest in each of these aspects attempting to treat them as a white box. Patmos [110] is the time-predictable multi-core processor resulting from the T-CREST project. It implements a *Very Long Instruction Word* architecture with a dual-issue in-order pipeline. The ISA is designed to support different features for time-predictability. This includes a full-predication of instructions, which provides a complete support for efficient single-path programming. Another interesting feature is the support of typed `load/store` instructions. This is particularly important as the memory hierarchy relies on a separation of cache structures depending on the data access pattern. The idea is to rely less on standard caches in favor of specialized caches that are more predictable and easier to analyze. Patmos, thereby, implements two time-predictable caches: (1) the Method Cache [33], which operates on functions and makes it simpler to model the cache's behavior in timing analyses, and (2) the Stack Cache [11], which works on stack data and takes advantage of the stack structure simplifying timing analysis. A memory NoC can also be built based on Patmos processor cores [109]. The TDMA arbitration is used to guarantee a predictable access to the shared main memory. An SDRAM controller has been introduced [68] for Patmos. Many of the ideas used in the T-CREST project were first explored within the JEOPARD [12] (Java Environment for Parallel Real-Time Development) project. The JOP [13] (Java Optimized Processor) is somehow the precursor of Patmos and has very similar architectural characteristics, e.g., typed loads/stores, time-predictable caches, TDMA arbitration policy. One of the motivations for T-CREST is to overcome the intrinsic limitations of the Java environment.

6.2 Comparing Architectures

We summarize in Table 1.1 some key features these architectures present. We, furthermore, take a look at the corresponding compiler support as well as the available WCET tools for these architectures. A comparison based on these aspects is provided here below:

Memory hierarchy: A first observation is that most time-predictable architectures avoid using caches. This is because caches introduce execution time variations that need to be accounted for by a dedicated cache analysis in the WCET tool (see Chapter 3.Section 4). Caches are usually replaced by SP memories which are managed by the program. In contrast to the SP, it is hard to precisely determine the cache content before runtime. The SP may therefore present a more attractive option for predictability. However, the concern here is the increase in code size and

[11] `http://www.t-crest.org`
[12] `https://www.hipeac.net/network/projects/4686/jeopard/`
[13] `https://www.jopdesign.com`

		Time-Predictable Architectures				
HW Features		**SPEAR**	**PTARM**	**MERASA**	**CoMPSoC**	**Patmos**
ISA (all RISC-style)		SPEAR	ARMv4	TriCore	MicroBlaze	Patmos: - VLIW - typed ld/st - delay slots
Pipeline (all in-order)		3 stages, single issue	5 stages, single-issue, TIP	5 stages, dual issue, SMT	3 or 5 stages, single issue	5 stages, dual issue
Predication		Partial	Partial	Partial	–	Full
Branch Prediction		–	–	–	–	–
Prefetching		–	–	–	–	–
In-Core Memory	I$	–	–	Not for HRT tasks	–	Yes
	D$	–	–	Not for HRT tasks	–	Yes
	M$	–	–	–	–	Yes
	S$	–	–	–	–	Yes
	SP	–	Yes (instr., data)	Yes (D-ISP, stack data)	–	Yes (instr., data)
Bus Arbiter		–	TDMA	FP, RR, FIFO	TDMA	TDMA
DRAM controller		–	PRET	RTCMC	Predator	Patmos
Compiler Support		GCC: - predication - single-path	LLVM (timing semantics)	TriCore toolchain	MicroBlaze toolchain	LLVM: - predication - single-path - S$, M$ - delay slots
WCET Tools		–	Otawa, aiT, Heptane	Otawa, aiT	–	aiT, Otawa, Platin, LLVM

Table 1.1 – Comparison of time-predictable architectures. (I$: Instruction Cache. D$: Data Cache. S$: Stack Cache. M$: Method Cache. SP: Scratchpad.)

the difficulty to implement complex SP behaviors within the compiler. To avoid this issue, the D-ISP implemented in MERASA requires additional hardware compared to a regular SP. The logic is used to automatically manage loaded functions within the SP space, namely, to identify them and perform evictions when necessary. From a functional point of view, the D-ISP and the method cache behave exactly the same. The main difference is that the method cache relies on a tag memory for hit detection whereas the D-ISP implements a lookup table. However, the existing compiler and ISA supports for the method cache allows to split a function in case its size is too big

for the cache [57]. The D-ISP does not have such support. We believe that the use of caches is necessary for high-performance real-time systems. The time-predictable caches implemented in the Patmos processor present an interesting opportunity: The automatic hardware behavior can be *guided* by the program/compiler to achieve more predictability and analyzability. In fact, these caches can be seen as a midway between the conventional caches and the SP memories (see Chapter 2.Section 2.4).

Pipeline: The PTARM and MERASA architectures make use of hardware approaches to support multi-threading. The SMT implemented in MERASA is designed so that the HRT task is never delayed by NHRT tasks. To achieve this, however, hardware resources need to be duplicated for each thread. This includes the register file, the PC, and the instruction buffer. Moreover, the architecture restricts each core to carry-out at most one HRT task and up to three NHRT tasks. We believe this limits possible uses cases of the architecture. Similarly, the TIP pipeline of the PTARM processor requires the duplication of hardware resources. PRET machines are primarily designed for repeatable timing, this comes at a price. To achieve TIP, the processor switches the threads every cycle according to round-robin fashion. While this guarantees fairness among threads, it does not allow much flexibility in scheduling the threads. Furthermore, the PTARM processor behaves as a non-pipelined processor operating on a fourth of the clock frequency. Consequently, this makes PRET architectures rigid, more expensive to implement, and less suitable for high-performance real-time applications. On the other hand, the VLIW architecture in Patmos implements a dual-issue pipeline which improves the ILP within the same thread. The VLIW design reduces the pipeline complexity and puts the compiler to play a key role in optimizing the processor performance. We believe this opens the door to explore WCET-related optimizations and more flexibility to tweak the performance/predictability trade-off.

Predication, Prefetching, and Branch Prediction: In contrast to other predictable architectures, the Patmos ISA supports full predication. This not only allows to efficiently support the single-path programming, but also to perform branch optimizations to reduce branch hazards in a predictable fashion (see Chapter 2.Section 3.2). To the best of our knowledge, no time-predictable architecture supports prefetching and branch prediction features.

Extensibility: The ability to change and adapt elements of the computer architecture is vital to investigate and enhance the worst-case performance. For most architectures, it is hard to find source codes for simulators or models because either they are not publicly available or the links are dead. This is the case for the PTARM, CoMPSoC, MERASA and SPEAR architectures. Besides, some of these architectures rely on proprietary ISA which might not be always possible to extend (e.g., ARM, TriCore, MicroBlaze). In contrast, Patmos provides a publicly accessible repository [14] containing a cycle-accurate simulator written in C++, as well as an implementation in Chisel to generate hardware description used for FPGA synthesis. This allows for large possibilities in hardware configurations. The PRET machine FlexPRET that is intended for safety-critical systems implements the open and extensible RISC V ISA. A simulator and a Chisel implementation of the processor can still be found [15]

[14] `https://github.com/t-crest`
[15] `https://github.com/pretis/flexpret`

as of today.

Compiler Support: Another important aspect in this comparison is the availability and extensibility of the tool set to compile and optimize for worst-case performance. The MERASA and CoMPSoC platforms rely on proprietary compiler toolchains that might not be possible to extend. In [26] Broman et. al describe a vision of the software infrastructure for PRET machines. In particular, an intermediate language called PRETIL is to be used to hide low-level implementation details from the high-level language containing timing constructs. Moreover, a compiler must guarantee the compliance of the PRETIL timing semantics with the machine timing behavior. It is not clear where these efforts have gone so far, in fact, we could not find publications or source codes to evaluate the compiler. For Patmos, the compiler [96] consists of an adaptation of the LLVM compiler framework [69]. All Patmos features/components have a certain support in the compiler. Moreover, the source code is public and can be found in the T-CREST repository.

WCET Tools: In order for a WCET tool to support a particular processor, it has to have an abstract model for the processor timing behavior, supports its ISA, and implement timing analyses for its hardware components. To the best of our knowledge, there is no WCET tool that supports the MicroBlaze and SPEAR ISA. Consequently, there exists no tool that can provide WCET estimates for programs running on the CoMPSoC and SPEAR architectures. The PTARM processor relies on the popular ARM ISA and implements a simple architecture. Therefore, many tools like Otawa, aiT and Heptane could be used with potentially minimal adaptations. A support of the MERASA architecture is provided in Otawa including a timing analysis for the D-ISP [82]. Moreover, a dedicated WCET tool called ISPTAP [83] has been introduced and is specialized in the analysis of instruction memories. For Patmos, the support is barely existing and, at best, shredded into different tools such as aiT, platin, Otawa, or even the compiler's back-end. For instance, the platin tool acts like an interface between the compiler and aiT providing it with WCET-related information. It also implements some timing analyses that are not supported by aiT due to its lack of extensibility.

Compared to other time-predictable architectures, we believe Patmos provides interesting features and more flexibility to enhance the worst-case performance. The compiler infrastructure is already existing along with an open cycle-accurate simulator to extend the processor features. Moreover, Florian Brandner – who co-supervises the work of this thesis – has contributed closely to the T-CREST project thus providing his expertise in the matter.

In this work, we focus on the Patmos processor.

7 Contributions

Due to its comprehensive approach to worst-case performance, we propose to investigate WCET analysis on time-predictable architectures through the Patmos processor. The already existing support around Patmos allows for an effective exploration of techniques to enhance the worst-case performance. Moreover, treating the architecture/compiler/analyzer as a white box allows to address the problem while bypassing

some side issues that may interfere with our goal (e.g., the lack of documentation, the difficulty to extend the hardware/ISA, the opacity of tools, etc.).
First studies on the Patmos processor and its hardware components showed promising results [63, 12, 59], however they are rather incomplete. For instance, the standard stack cache analysis does not consider preemptive execution of tasks, an important feature, without which many systems may turn infeasible. Also, no prior work addressed timing analysis for predicated execution and how to handle it in WCET analysis work-flow. The support of time-predictable processors in WCET tools is barely existing. The Otawa tool [18] provides only a partial support for the Patmos processor. It does not support VLIW architectures nor predicated execution. Also, it lacks analyses for time-predictable caches. The same holds for the well established commercial tool aiT [43]. Only a stack cache analysis based on data-flow analysis is provided [118], and which we find incorrect (see Chapter 5).

The work in this thesis is part of the efforts to fill this gap in order to fully understand predictable architectures and investigate their potential in actually enhancing worst-case performance of hard real-time systems. More than that, we aim to explore some hardware extensions in the intent to increase the precision of WCET estimates, but also reduce the actual WCET itself. For this, we provide compiler support, as well as timing behavior analyses that take advantage of these hardware extensions. All in all, we contribute to the elaboration of a WCET timing analysis for the Patmos processor that supports some of its important time-predictability features.

We, thus, present here below our most important contributions:

Stack Cache: This specialized cache for stack data presents interesting properties for time-predictability, e.g., simple structure and behavior, not relying on addresses, cache accesses are guaranteed hits, and memory transfers only issued by few stack control instructions under simple circumstances. It is, therefore, a natural candidate for interest in the context of this work. We thus propose the following:

- Firstly, we compare the precision of occupancy analysis obtained from an abstract interpretation-based technique with that of the standard stack cache analysis [88]. The cache occupancy determines whether memory transfers need to be performed at stack cache control instructions. Therefore, the precision of occupancy analysis influences the tightness of timing behavior estimates.
- Secondly, we propose means to optimize preemption costs for the stack cache and extend the standard analysis to account for preemption-related delays [13, 87]. The simple structure of the cache does not allow it to be shared by multiple tasks. As a result, stack data of the preempted task needs to be saved/restored as part of the context switch operation. Instead of saving/restoring the whole cache content, our approach tracks *useful* data at program points where the preemption might occur.
- The analysis of preemption-related delays provides rich information at each program point. However, this may be of little use if the scheduler is not able

to exploit them for efficient context switching. We, thus, provide *preemption mechanisms* that realize the context switch operations taking advantage of the analysis results [87]. These mechanisms are implemented through simple hardware extensions and induce no overhead on memory footprint or execution time.

- Finally, we investigate a prefetching-like technique that allows to anticipate and hide stack data transfers from/to the main memory [89]. The *eagerly* performed memory transfers have no side effects with the stack cache itself, however, interferences may occur with other hardware components. We explore a solution to avoid these interferences in the Patmos multi-core platform by making use of *free* TDM slots. In the absence of a TDM analysis that detects those free TDM slots, our solution guarantees that the worst-case performance is nevertheless preserved. On the other hand, the conducted experiments show an improvement of the average-case performance.

Predication: Predicated execution is problematic for tight WCET analysis. The mere fact that an instruction is executed (or not) may have a chain of effects on almost all analysis steps. As these analyses have to be conservative, one may simply ignore the predicates and consider the wost-case effect resulting from both possibilities. Unfortunately, this may yield very conservative results due to information loss after the branch elimination. Another approach consists of adapting each analysis so that it is aware of the predicates. This may, however, require considerable and redundant efforts. We provide a simple approach that consists of recovering control-flow edges *early* in the WCET analysis process [25]. In addition to effectively treating the predicates, this causes little to no effect on subsequent steps of the WCET analysis.

Odyssey: An open and fully-integrated WCET analysis tool for the Patmos processor. Odyssey is integrated within the LLVM compiler framework and is called right before code emission. The tool, therefore, has a complete and accurate view of the program to be executed. This integration brings many advantages and opportunities for the exploration of new techniques to enhance the worst-case performance. For instance, Odyssey does not operate on the program binary[16] in contrast to other WCET analysis tools. In fact, lots of information that may help timing analysis are normally discarded by the compiler once the binary is generated. Another opportunity is that a close relationship between the WCET tool and the compiler promotes WCET-driven compilation. As for the supported features, Odyssey implements: address analysis, classical analyses for conventional caches, and specialized stack cache and method cache analyses. Odyssey, furthermore, supports the Patmos VLIW architecture with predicated execution.

While the work in this thesis is conducted on the Patmos processor, the obtained results can be extended to similar architectures. In fact, many of Patmos' features and components are already being (or can be) used in other architectures. By ensuring time-composability, one can make use of most of the results of this work. This includes the timing analyses, the hardware extensions, as well as the corresponding compiler support. Patmos is merely an exploratory architecture that helps investigating new

[16] We are aware that WCET analyzers are often considered as stand alone tools and thus usually operate on the binary code. However, with respect to our goal of exploration, we consider the issues related to this process (e.g., loss of WCET-related information during the compilation, the front-end techniques to build the program model) out of our scope.

software/hardware techniques for worst-case performance.
The remainder of the manuscript is structured as follows: Chapter 2 describes the Patmos processor architecture and the compiler support in the LLVM framework. Chapter 3 provides the WCET analysis flow as well as the current state-of-the-art related to WCET analysis. In Chapter 4 we present our WCET analysis tool Odyssey and how the predication is handled in the analysis flow. We, then, take interest in the stack cache and compare in Chapter 5 the precision of existing analyses. The analysis of preemption delays for the stack cache is presented in Chapter 6. We propose, in Chapter 7, preemption mechanisms making use of preemption analysis results. In Chapter 8, we explore hardware extensions to realize a prefetching-like optimization on the stack cache. Finally, conclusions and future work are provided in the last Chapter.

CHAPTER 2

Patmos, a Time-Predictable Processor

The time-predictable design of computer architectures for the use in (hard) real-time systems is becoming more and more important, due to the increasing complexity of modern computer architectures. Many time-predictable architectures have been introduced so far, however, Patmos stands as an attractive candidate to study their effectiveness for enhancing worst-case performance. We provide here a detailed presentation of the time-predictable processor Patmos and walk-though its key features. The Chapter is structured as follows. We first provide a quick overview of the Patmos processor through its design principles. Then, we describe in Section 2 its computer architecture including the memory hierarchy. Finally, Section 3 presents the compiler support that is provided so far.

1 Overview and Design Approach

The awareness as to the necessity of deploying computer architectures that are suitable for timing analysis has risen since the beginning of the millennium. A number of studies has been conducted and resulted in various architectures and design rules for time-predictability [34, 80, 120, 53, 91]. With the emergence of different architecture designs, a natural question arises: which of them offers better time-predictability? Schoeberl investigated this [107], and argued that time-predictability alone does not allow a meaningful comparison between architectures. Instead, one may consider comparing the WCET of tasks obtained on different architectures. The resulting WCET bounds are, in fact, the result of the interplay between three closely related components: (1) the computer architecture, (2) the compiler support, and (3) the WCET analysis tool. The T-CREST project [108] incarnates this vision of wost-case performance and proposes a time-predictable multi-core architecture called Patmos [110].

Compared to conventional architectures where the hardware takes a major part in dynamically controlling/optimizing the average-case performance, the Patmos approach relies on a combination of software/hardware techniques to enhance the worst-case performance.

Hardware: The architecture design stands on three pillars: (1) reduce the actual WCET, (2) simplify static WCET analysis, and (3) make the resulting WCET estimates more precise. Here below, we respectively refer to these principles as Performance, Simplicity, and Predictability and we present an overview of what Patmos provides to support each of them.

- Performance: Reducing the actual WCET implies the deployment of hardware components and techniques to speed-up the execution time and enhance throughput. As such, the Patmos processor implements: a VLIW architecture that issues up to two instructions of the same thread, a 5 stages pipeline with full-forwarding to handle data hazards, and a memory hierarchy consisting of caches and SP memories. Moreover, the architecture provides interconnect to build a multi-core platform combining Patmos cores. The cores may also communicate in the form of message passing through a core-to-core NoC. A synthesis of the processor core was realized on the low cost Cyclone IV FPGA. The maximum operating frequency reached 80MHz according to [110].

- Simplicity: The architecture is timing-composable, i.e., the timing behavior of any hardware component is independent from the rest. This allows to decompose the complex WCET analysis problem into smaller and simpler ones. Sound WCET estimates can therefore be obtained by separately studying the timing behavior of each hardware component and combining their individual contributions. In Patmos, timing-composability is enforced through: a pipeline design with in-order execution and a single stalling stage, the use of the TDMA policy to arbitrate memory accesses between Patmos cores, and the non-deployment of any component/feature known to cause timing-anomalies (see Chapter 1.Section 5).

- Predictability: The architecture attempts to reduce sources of uncertainties that might contribute to timing variations. Examples of this include the dependency on input data, or heavily relying on components with a timing behavior that is hard to model and predict. To mitigate these issues, Patmos provides features like full-predication which allows to eliminate branches that may be dependent on input data. Moreover, the memory hierarchy implements specialized caches for different data access types, i.e., stack cache, method cache. These caches are qualified as time-predictable for their relatively simpler behavior and analyzability. The argument here is that by relying on time-predictable caches, the pressure is reduced on conventional caches for which a simple and precise timing-analysis can be challenging (see Chapter 3.Section 4). This is supposed, a priori, to yield more precise WCET estimates.

Software: To back many of the aforementioned hardware features, the architecture is more exposed to the compiler. Compared to conventional architectures, this means shifting some of the control from the hardware side to the software side. A direct implication is that a considerable amount of the optimization/control effort is now carried-out by the compiler. On the other hand, this opens opportunities to perform more adapted optimizations as the compiler naturally gathers lots of information about the program. To enable this level of control, the Patmos ISA is equipped with instructions to manipulate some of the processor's internal structures. Notable examples of this include: a set of typed load/stores instructions targeting different cache structures of the memory hierarchy, control instructions to manage

the time-predictable caches, as well as delayed and non-delayed branches/calls. On top of the ISA, the compiler applies techniques to manage the hardware and optimize the performance/predictability of the program. Examples include function splitting for the method cache, predication, and branch optimizations by exploiting delay slots.

In the following sections, we shall present these hardware features and the corresponding compiler support in more depth.

2 Patmos Computer Architecture

The Patmos processor is designed to enhance the worst-case performance of single threads. In this section, we present the computer architecture and describe its internal subsystem. In particular, we will be covering: the pipeline design, the register file, the ISA, the memory hierarchy, and the multi-core platform.

2.1 Pipeline and Register File

Patmos is a *Very Long Instruction Word* (VLIW) architecture that can issue and execute a bundle of two instructions in parallel in a five-stage, in-order pipeline: fetch (`FE`), decode (`DEC`), execute (`EX`), memory access (`MEM`), and register writeback (`WB`). Within the same bundle, the first slot may hold any instruction of the ISA, including those that might initiate memory transfers (e.g., load/store, branches, call, return). The second slot may only hold instructions that do not access to memory. Figure 2.1 illustrates a simplified representation of the pipeline with some of its local memories, i.e, the register file (`RF`), method cache (`M$`), the stack cache (`S$`), the data cache (`D$`), and the scratchpad (`SP`). Instructions are fetched from a special instruction cache, the method cache [33], which guarantees that an instruction fetch always hits in the cache. Method cache misses may only occur in the `MEM` stage during the execution of dedicated branch instructions (e.g., `brcf`) or function calls (e.g., `call`, `ret`). Similarly, misses in the data cache and the stack cache may only occur in the `MEM` stage. The fact that all misses are detected in the `MEM` stage has the benefit of simplifying the pipeline design and particularly the stalling logic. More importantly, since only the first instruction of the bundle is legally allowed to initiate memory transfers, a miss in two different caches may never occur at the same cycle. This, as a result, enforces timing-composability and simplifies the timing analysis. The `FE`, `EX`, and `DEC` pipeline stages are thus free from undesirable side-effects (apart from updating the `PC`), while the `MEM` and `WB` stages may cause side-effects on the processor's registers or caches. Predicate registers are read and written in the execute stage (`EX`). The processor thus is able to detect whether a predicated instruction needs to be nullified before any undesirable side-effects may become visible.
The register file (`RF`) consists of 32 general-purpose registers (`r0` through `r31`), and 16 special-purpose registers (`s0` through `s15`). All registers are 32 bits wide, except for the predicates that consist of eight single-bit registers (`p0` through `p7`) shadowed but the lowest byte of the special register `s0`. General purpose registers are read in the `DEC` stage and written in the `WB` stage. The pipeline supports full forwarding, which makes those registers available at the `EX` stage. Special-purpose registers, on the other hand, can be read and written in different stages depending on the type

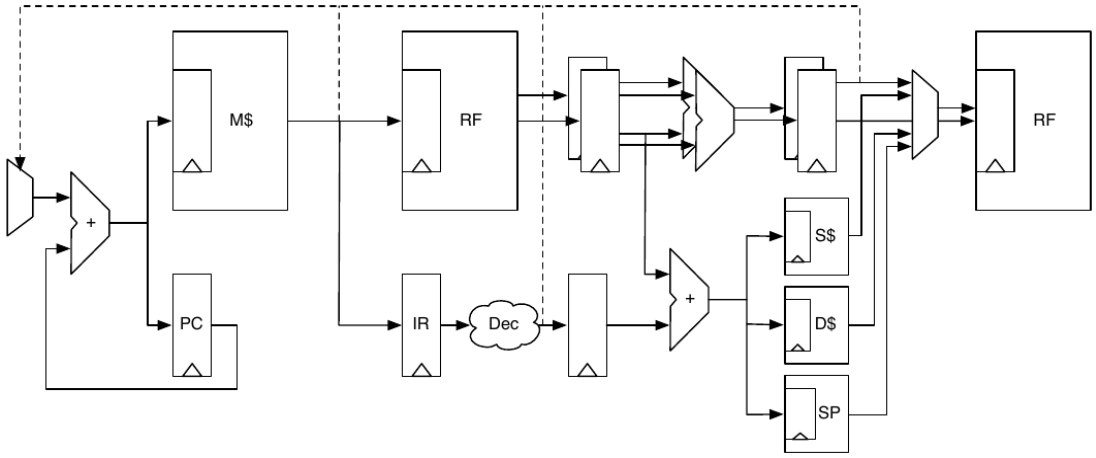


Figure 2.1 – Simplified representation of Patmos pipeline. Taken from Patmos Handbook [7].

of the register. Those special-purpose registers merely consist of various processor registers dedicated to manage specific functions of the processor. Examples include, the `ST` and `MT` pointers that manage the stack cache and are mapped to registers `s5` and `s6` respectively.

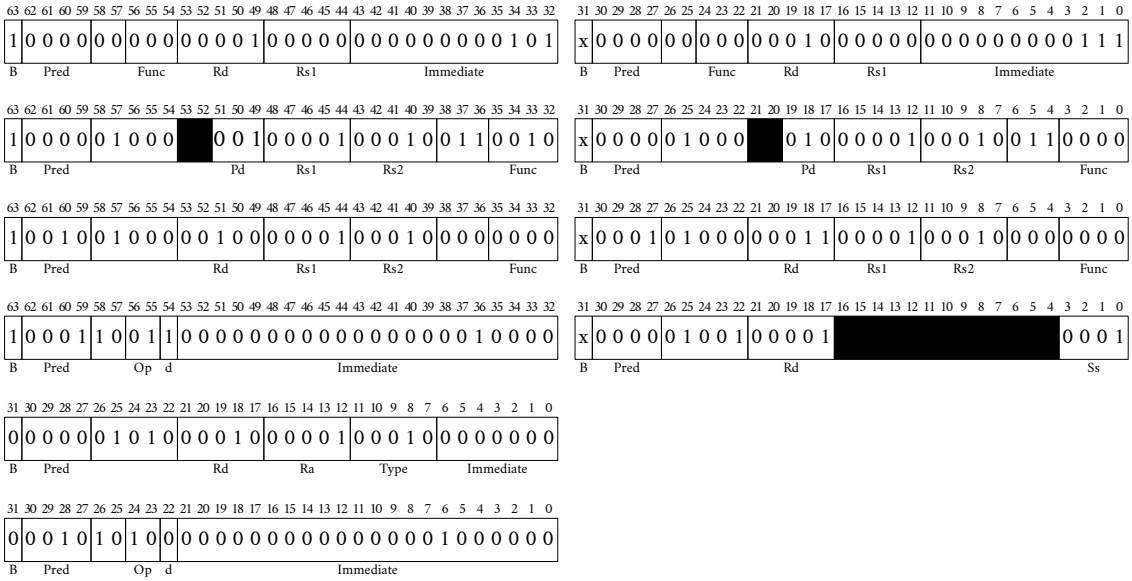
2.2 Instruction Set Architecture (ISA)

The instruction set of the Patmos processor follows the VLIW paradigm and may thus execute up to two instructions that are grouped into bundles at the same time. The adopted RISC-style restricts data memory transfers to specific classes of instructions, e.g., `load`, `store` and branches. This has the advantage of reducing the complexity of the pipeline and simplifying the timing analysis process. All Patmos instructions can execute in parallel, the only restriction is that only the first instruction of the same bundle is allowed to initiate memory transfers. Patmos instructions are 32 bits wide, the bundle size can therefore either be 32 or 64 bits wide. The compiler statically schedules the instructions and determines the size of the bundles. Moreover, it has to be ensured that bundles are 32 bits aligned.
Access to different cache structures is distinguished through typed `load` and `store` instructions, e.g, `lwc` to load one word[1] from the data cache, `lws` to load one word from the stack cache. Moreover, these `load`/`store` accesses come in different sizes, i.e., word (`lwc`/`swc`), half-word (`lhc`/`shc`), and byte (`lbc`/`sbc`). Also, unsigned `load`/`store` operations can be performed for half-word (`lhuc`/`shuc`) and byte-sized (`lbus`/`sbus`) accesses. In Patmos, `load` and `store` operations are subject to alignment restrictions. This means that the address of the access has always to be a multiple of its size in bytes, e.g., `lws`:4, `lhc`:2.
The Patmos ISA supports full predication of instructions. This means that every instruction can be predicated, including control-flow instructions. Predicated execution is used to decrease the number of branches. The ISA is, moreover, equipped with special instructions that control cache structures if necessary, e.g., `sres` and `sens` instructions for the stack cache and `brcf` to branch with method cache fill. Two variants of branch and call instructions are supported in Patmos. The *non-delayed*

[1] In Patmos, a word size is 4 bytes

```
(B1)        addi   r1 = r0, 5    ||        addi   r2 = r0, 7
(B2)        cmplt  p1 = r1, r2   ||        cmpeq  p2 = r1, r2
(B3) (p2)   add    r4 = r1, r2   || (p1)  add    r3 = r1, r2
(B4) (p1)   br     0x10          ||        mfs    r1 = s1
(B5)        lwc    r2 = [r1 + 0]
(B6) (p2)   brcfnd 0x40
```

(a) Patmos assembly code.



(b) Patmos machine code with the instruction format.

Figure 2.2 – Portion of a program in Patmos assembly language and its corresponding machine code. Each line in the machine code corresponds to a line in the assembly code.

variant simply inserts *stalls* into the pipeline preventing upcoming instructions to be fetched before the control-flow instruction gets executed. The *delayed* variant allows the compiler to statically insert instructions in the available *delay slots*. The number of delay slots is influenced by the pipeline design and at which stage the control-flow instruction is executed. In Patmos, there may be between 2 and 3 delay slots depending on the instruction (see Section 3.2).
The format of each instruction depends of the instruction class it belongs to. However, the most significant bit of the instruction bundle always determines the bundle size, i.e., 32 or 64 bits. Then, for each instruction of the bundle, the next four significant bits are reserved for the predicates. The opcode bits are located after the predicates and are used to identify the instruction in the decode stage. Furthermore, each instruction can take up to three register operands. The register addresses are located at fixed position so that the register file can be read in parallel to instruction decoding.

Example 2.1 *Consider the code portion written in Patmos assembly language shown in Figure 2.2a. The corresponding machine code according to the Patmos ISA is shown below. Each line of the machine code corresponds to a bundle in the assembly code. A bundle may hold up to two instructions. The first instruction of the bundle*

is the one on the left side.

B_1: The first bundle consists of two instructions `addi` *that perform binary addition. The first instruction of the bundle adds the content of* `r0` *to the immediate value* 5*, and stores the result in register* `r1`*. The second instruction of the bundle performs a similar operation. In the corresponding machine code, the bundle flag of the first instruction is set to* 1*, which indicates a bundle size of* 64 *bits. The bundle flag in the second instruction is not relevant. The operand Rd indicates the destination register, i.e.,* `r1` *and* `r2` *for the first and the second instruction respectively. Similarly, the operand Rs*1 *refers to the source register which is* `r0` *for both instructions. The value located in the immediate operand is then added to the content of source register identified by the Rs*1 *operand and stored in the destination register identified by the Rs*1 *operand. All instructions in Patmos are predicated. The Pred operand allows to conditionally execute the instruction based on the value of the predicate register it points to. For both instructions, Pred points to the read-only predicate register* `p0`*, which is always evaluated to true. This means that instruction is always executed.*

B_2: The first instruction of this bundle `cmplt` *sets the predicate register* `p1` *to* 1 *if the content of* `r1` *is strictly less than that of* `r2`*. On the other hand, the second instruction sets* `p2` *to* 1 *if they are equal. The encoding of the bundle is done similarly to the previous one. The operand Pd determines the predicate register to be set based on the result of the comparison.*

B_3: Both instructions in the bundle are conditional, meaning they execute only if the corresponding predicate is true. The `add` *instruction is similar to* `addi`*, only it takes a third register operand instead of an immediate value. As in previous instructions where the default predicate was* `p0`*, here the Pred operand in the machine code needs to indicate the corresponding predicate registers. Also notice that, when applied, the operands Rd, Rs*1*, and Rs*2 *are always located in the same place in the instruction format. This allows the operand registers to be read in parallel to instruction decoding.*

B_4: The bundle consists of a predicated `br` *instruction which performs a delayed* `PC` *relative branch. We assume that the branch target is located at the address* `PC`+ `0x10`*. Again, the Pred operand has to point to the corresponding predicate register, which is* `p1`*. To indicate that the control-flow instruction is delayed, the bit d that is part of the opcode is set to* 1*. The immediate operand is used to compute the target of the branch based on the current value of* `PC`*. Branch instructions can be executed in parallel with any instruction that does not access to main memory. The* `mts` *simply copies the content of the special register* `s1` *into the general-purpose register* `r1`*, therefore, no memory transfers are involved. As a result, the corresponding machine code sets the operand Ss to the source special register and Rd to the destination general-purpose register.*

B_5: The bundle consists of the instruction `lwc` *that loads the register* `r2` *with the content of the address stored into the register* `r1`*, displaced with an unsigned immediate value. The displacement value (here set to* 0*) is always left-shifted by* 2 *positions to ensure a* 32 *bits alignment of addresses. The* `lwc` *instruction is tied to the data cache and initiates a memory transfer if the accessed data is not cached. The bundle consists of a single instruction as the compiler could not find any instruction legally allowed to occupy the second slot. As a result, the bundle bit is set to* 0 *indicating a bundle size of* 32 *bits. The operand Ra refers to the register containing*

the address and Rd is the destination register into which the data is going to be loaded.

B_6: The last bundle consists of a predicated `br` instruction which performs a non-delayed `PC` relative branch with a potential method cache fill. Here the target of the branch is located at the address `PC` + `0x40`. This branch is legally allowed to occupy the second delay slot of the previous branch in the bundle B_4. The reason is that their respective predicates (`p1` and `p2`) are disjoint, meaning that they cannot both evaluate to true in any execution scenario. As a result, only one branch is known to be taken, or none (in which case both instructions are nullified). Similarly to the previous bundle, the compiler could not find any instruction to legally pair with the memory accessing branch instruction. The conversion to the machine code is done as in B_4.

2.3 Predication

Among many other features, Patmos supports predicated execution. The technique is used to decrease the number of branches and their associated penalties, especially in VLIW architectures that need to keep their parallel pipelines busy. Predication can also be beneficial for timing analysis as it allows to eliminate execution time variations through single-path programming [97, 95]. In Patmos, an additional predicate operand is attached to each instruction, which allows to refer to one out of the 8 predicate registers (`p0` through `p7`). The predicate operand, in addition, allows to invert the predicate value (e.g., `!p0`). Consequently, 4 bits of the instruction encoding are reserved for the predicate operand (3 bits for the predicate register, 1 bit for negation). Predicate register `p0` always evaluates to `true` and cannot be overwritten.
Predicate registers can be defined using dedicated comparison instructions (e.g., `cmpeq`), which allow to compare 32 bit integer values from general-purpose register operands and immediate values. These instructions allow to specify a destination predicate, which sets the predicate register accordingly. In addition, basic logical operations can be performed on two predicate register operands using dedicated logical-predicate instructions (e.g., `por`). The result of the operation is again written into a predicate register. Note that these instructions can be predicated themselves, which facilitates the handling of nested `if` statements.

2.4 Memory Hierarchy

One of the features Patmos offers is a specialization of cache structures depending on data access types, e.g., instructions, stack data, and other data. Compared to traditional architectures, the specialization of caches is intended to reduce the interference on conventional caches, allowing more precise and simpler static cache analysis. Patmos, thereby, supports a range of cache structures such as the *stack cache* that is dedicated for stack data, the *method cache* for instructions, conventional instruction and data caches, as well as a *scratchpad* memory, which is managed by the compiler. This allows to define a variety of configurations for the underlying hardware platform which is extremely useful for benchmarking purposes.

Standard Caches

Conventional caches may store either instructions, data, or both. Patmos dedicates a specific cache to data, and to instructions if the method cache is not to be used. The cache is located between the CPU and the main memory. When a memory access instruction is being executed, the CPU first checks the availability of the accessed element in the cache via its address in the main memory. Two outcomes are possible. The data is present in the cache and can be used immediately, a *cache hit* has then occurred. Otherwise, we say that a *cache miss* has occurred. Cache misses are particularly time consuming as a request has to be made to main memory in order to fetch the missing element. Once the main memory responds, the element in question is added to the cache, which may potentially evict some other cached elements. Both read and write memory accesses are concerned with the cache.
Cache write policies determine how memory stores are handled. There are basically two options on a *write hit*. In the *write-through* policy, data elements are written both into the cache and the main memory. On the other hand, the *write-back* policy writes data elements only into the cache. This may, however, cause data to be incoherent between the cache and the main memory. Consequently, once incoherent data is evicted, the corresponding data in the main memory is updated. The write-back policy results in fewer memory transfers and therefore often yields better performance. However, it is harder to analyze as incoherent data need be tracked before eviction. This is why the *write-through* policy is a better choice for predictability.
Concretely, the cache space holds a set of k entries. To each entry is associated a *Valid Flag*, a *Tag*, and the data. The valid flag indicates whether the data is valid or not, while the tag represents the address information of the data held by the entry. The data consists of a sequence of bytes commonly called a *cache block* or *cache line*. A cache block represents a contiguous space in the main memory that we refer to as a *memory block*. Note that a memory block may contain multiple bytes or words. Therefore, an access to a single data element results in fetching the whole memory block, and placing it in the corresponding cache entry.
There are, furthermore, three main ways to structure the cache based on how memory blocks are mapped to the cache. The *direct-mapping* scheme is fast and simple to implement and merely consists of placing each memory block in exactly one dedicated cache entry. The *fully-associative* structure reduces contention by allowing memory blocks to be mapped to all available entries. This structure is relatively complex and may be slow as it involves performing a search over all cached entries in order to find a particular memory block. In the *set-associative* cache, each set of memory blocks is mapped to a dedicated set of entries called a *cache set*. In that regard, it combines features of both direct-mapped and fully-associative caches. The maximum number of entries in a cache set is called the *associativity* of the cache and denoted as s.
Depending on the cache size and structure, it may not be possible to hold all data elements of a program in the cache. The cache may need to evict some cache blocks. *Cache replacement policies* define a way to manage cache content in the case of conflicts, i.e., accesses to memory blocks that are mapped to the same cache set. In case the corresponding cache set is full, the replacement policy decides which cache block is to be evicted. Patmos supports two popular replacement policies. The *Least-Recently Used* (LRU) is a scheme under which the *oldest* accessed cache line is the one to be evicted. Whereas under the *FIFO* policy the first accessed cache lines are evicted first.

Cache replacement policies aim to profit from the *locality of memory references*, i.e., cache reuse. Depending on the memory access pattern of a program, we distinguish two basic kinds of localities. *Temporal locality* refers to a situation where the reuse of a data element happens in short time duration. This is the case, for instance, for loops where the same data is likely to be accessed in next iterations. The *spacial locality*, on the other hand, designates a situation where data elements to be accessed are stored in address locations close by. This is typically the case of array accesses. Improving the locality leads generally to a more effective utilization of the cache, which translates into higher cache hit rates.

Stack Cache

The stack cache is implemented as a ring buffer with two hardware registers holding pointers [11]: *stack top* (ST) and *memory top* (MT). The top of the stack is represented by ST, which points to the address of all stack data either stored in the cache or in main memory. MT points to the top element that is stored only in main memory. The stack grows towards lower addresses. The difference $\texttt{MT} - \texttt{ST}$ represents the amount of occupied space in the stack cache. This notion of *occupancy* is crucial for the effective analysis of the stack cache behavior. Clearly, the occupancy cannot exceed the total size of the cache's memory $|SC|$. The stack cache thus has to respect the following invariants:

$$\texttt{ST} \leq \texttt{MT} \tag{2.1}$$

$$0 \leq \texttt{MT} - \texttt{ST} \leq |SC| \tag{2.2}$$

Data that is present in the cache is accessed using a dedicated class of load and store instructions (e.g, `lws`, `sws`). As mentioned earlier, these load/store accesses can be performed in different sizes, i.e., word, (unsigned) half-word, or (unsigned) byte. Moreover, accesses to the stack cache have to be aligned with the access size S. For simplicity, we may use fictive `lds` and `sts` instructions to refer to load and store accesses that occur in one cache block, regardless of the code block size.
The frame-relative address (FA) of such a memory access is added to ST and the sum is used to index into the ring buffer, i.e., the address within the ring buffer is given by $(\texttt{FA} + \texttt{ST}) \mod |SC|$. Recall that the stack load and store instructions are always cache hits. The compiler (or programmer) thus has to ensure that accessed data actually is available in the cache using dedicated stack cache control instructions. More formally, it has to be ensured that $(\texttt{FA} + \texttt{S}) \leq (\texttt{MT} - \texttt{ST}) \leq |SC|$ before every stack cache access. Cache block sizes are chosen such that no explicit check is required in the hardware implementation. This can be achieved by ensuing that the cache block size is a multiple of the alignment of the largest load/store instruction, i.e., 4 bytes for Patmos' `lws` instructions.

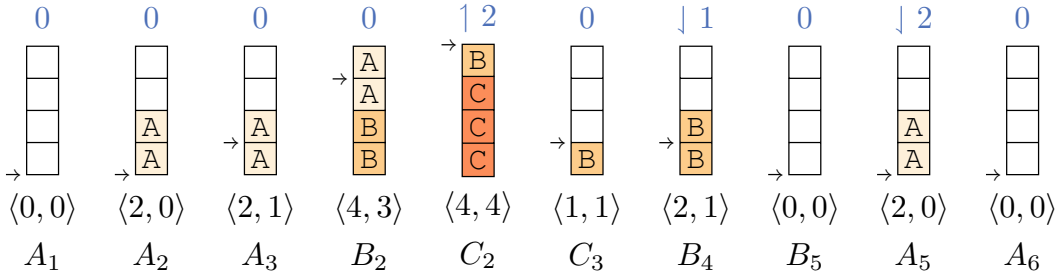
Stack Cache Operations: The stack cache control instructions manipulate the two stack pointers and initiate memory transfers to/from the cache from/to main memory, while preserving Equation 2.1 and 2.2. Memory transfers, and thus also the updates of the various pointers, are performed at the granularity of cache blocks, which can be parameterized in size. Depending on the configured block size, the memory transfers might be misaligned with the transfer size of the underlying memory system (e.g., the burst size of DRAMs). For brevity we do not cover this issue here and refer

```
(A1) func A()
(A2)   sres 2
(A3)   sws 0 = r9
(A4)   call B
(A5)   sens 2
(A6)   sfree 2
```

```
(B1) func B()
(B2)   sres 2
(B3)   call C
(B4)   sens 2
(B5)   sfree 2
```

```
(C1) func C()
(C2)   sres 3
(C3)   sfree 3
```

(a) Program consisting of 3 functions, reserving, freeing and ensuring space on the stack cache.



(b) The evolution of the stack cache state through the program. Assumed cache size: 4. The arrow → indicates the position of the LP pointer in the cache space. The number N above the cache state indicates the number of cache blocks filled ↓, or spilled ↑. The pair $\langle m, n \rangle$ below the cache state indicates the current occupancy m and the effective occupancy n.

Figure 2.3 – Example of a program and the stack cache state at particular points.

to Abbaspour and Brandner [10], where techniques to handle alignment issues are discussed.
A brief summary of the memory transfers associated with each control instruction is given below, further details are available in [11]:

sres k: Subtract k from ST. If this violates Equation 2.2, i.e., the cache size is exceeded, a memory *spill* is initiated to decrement MT until $\texttt{MT} - \texttt{ST} \leq |SC|$. Cache blocks are then transferred to main memory.

sfree k: Add k to ST. If this violates Equation 2.1, MT is set to ST. Main memory is not accessed.

sens k: Ensure that the occupancy is larger than k. If this is not the case, a memory *fill* is initiated to increment MT until $\texttt{MT} - \texttt{ST} \geq \texttt{k}$. Cache blocks are then transferred from main memory.

The stack load and store instructions only access the stack cache's ring buffer and thus exhibit constant execution times. This is particularly true for stack store instructions, which only modify the cached value. Modifications are not immediately propagated to the backing main memory. The stack cache's policy to handle stack store instructions thus resembles traditional *write back* caches.

Lazy Pointer (LP): An extension of the original stack cache allows to track coherent cache data [12]. Similar to MT and ST, LP is a pointer (realized as a hardware register) that satisfies the following equation: $\texttt{ST} \leq \texttt{LP} \leq \texttt{MT}$. The additional pointer divides the stack cache content into two parts: (1) cache data between ST and LP is potentially incoherent with the corresponding addresses in main memory, while

(2) data between LP and MT is known to have the same value in the cache and in main memory – the data is known to be coherent. Coherent data can simply be excluded from memory spill operations, i.e., it can be treated as if the data were not in the cache. We thus can refine the notion of occupancy: $\texttt{LP} - \texttt{ST}$ denotes the *effective occupancy* of a stack cache with a lazy pointer. Accounting for the effective occupancy allows to improve the sres instruction, with only slight modifications. The sfree instruction also requires minor modifications to correctly update the LP, while the sens instruction remains unchanged. In addition, the stack store instruction (sts) has to update LP whenever coherent data may be modified [12], i.e., LP is pushed upwards to ensure $\texttt{FA} + \texttt{ST} \leq \texttt{LP}$.

Example 2.2 *Consider functions A, B, and C shown in Figure 2.3a and a stack cache whose size is 4 blocks, i.e.,* $|SC| = 4$. *Figure 2.3b depicts the evolution of the stack cache state in particular program points. In this representation, the ST pointer is* fixed *at the bottom of the cache state. The MT pointer* moves *along the cache blocks indicating the end of the cache content and the beginning of the main memory address space. The actual operations on the pointers are performed according to the description we mentioned above. The occupancy is easily determined by counting the number of* colored *blocks. The arrow* $\rightarrow$ *represents the LP pointer. Cache blocks that are above LP are coherent with the main memory. Those below LP are potentially not coherent and their number determines the effective occupancy. We also assume that the cache is initially empty in the program entry point* A_1. *All pointers MT, ST, and LP, therefore, point to the same address, i.e.,* $\texttt{MT} = \texttt{LP} = \texttt{ST}$.
First, the sres A_2 *reserves two cache blocks in the cache space. For this, ST is decreased by* $k = 2$ *blocks which increases the occupancy by* 2 *blocks. The same holds for LP pointer as the newly reserved cache content is not yet initialized, therefore, it is treated as coherent with the main memory. In contrast to the occupancy, the effective occupancy remains* 0. *When the cache block of index* 0 *(with respect to ST) gets initialized by the sws* A_3, *the LP pointer is increased so that it points to the next sequence of data that is known to be coherent. LP now points to the cache block of index* 1, *the effective occupancy* $\texttt{LP} - \texttt{ST}$ *is therefore set to* 1. *The sres* B_2 *reserves two cache blocks. This time, LP points cannot decrease as before since the cache block just below is not coherent with the main memory. This adds two more blocks to the effective occupancy. Function C is called and the sres* C_2 *attempts to reserve* 3 *cache blocks. However, the cache is already full (occupancy is* 4*) and a further decrease of ST will violate the condition* $\texttt{MT} - \texttt{ST} \leq |SC|$. *To resolve this, MT will need to be decreased by the amount of* $k - [\,|SC| - (\texttt{MT} - \texttt{ST})\,] = 3$ *cache blocks. In the standard stack cache implementation, this will cause* 3 *cache blocks to be spilled. The lazy pointer extension is beneficial here as we can use the fact that some data is already coherent with the main memory and just remove it. So instead of spilling the* 3 *cache blocks located immediately below MT, only* $k - [\,|SC| - (\texttt{LP} - \texttt{ST})\,] = 2$ *cache blocks need to be transfered to the main memory. The sfree* C_3 *frees* 3 *cache blocks reserved by Function C, which increases ST by* 3 *cache blocks and reduces the occupancy to* 1. *Since the occupancy is less than* $k = 2$, *the sens* B_4 *fills the cache with* $k - (\texttt{MT} - \texttt{ST}) = 1$ *cache blocks. This causes MT to be increased accordingly, however LP is not affected as the newly filled blocks are naturally coherent with the main memory. The sfree* B_5 *causes a decrease of both LP and MT pointers such that* $\texttt{MT} = \texttt{LP} = \texttt{ST}$. *The sens* A_5 *fills back* 2 *cache blocks, however, the effective occupancy still evaluates to* 0 *as data is coherent.*

Method Cache

In contrast to the conventional instruction cache, the method cache operates on whole functions. The first implementation of the method cache was realized in the Java Optimized Processor (JOP) [105]. The design promises many advantages from a timing analysis standpoint. As for the stack cache, memory transfers can only be induced by few instructions whose locations are statically known by the compiler, i.e., `call`, `ret`, and `brcf`. This drastically limits the memory access cases to consider which makes the analysis simpler. Another advantage is that in contrast a conventional instruction cache where misses occurs in the `IF` stage, method cache misses may only occur in the `MEM` stage, similarly to other caches. As explained in Section 2.1, the single-stalling stage prevents the caches from conflicting each others on misses, which enforces timing-composability of the platform. Moreover, at misses the method cache can profit from memory bursts to efficiently transfer chunks of code from the memory in a shorter time. The relatively bigger size of code blocks has, also, the advantage to require less tag memory compared to a regular instruction cache, hence, reducing contentions.
The global structure of the method cache follows that of fully-associative organization. The entries represent a contiguous block of instructions known as the *code block*. Code blocks are statically formed by the compiler based on the *Control-Flow Graph* [2] (CFG) of functions and considering cache size and tag limitations. Possible implementations of the method cache depend on the desired replacement policy, and whether the code block size is fixed or variable [33]. A variable code block size implementation allows for a better utilization of the cache space. However, it only supports the FIFO replacement policy. Whereas an implementation with fixed-size code blocks can be associated either with the LRU or the FIFO replacement policies. The LRU replacement policy being a better choice for predictability. The cache state can only be altered by three instructions:

`call`: For absolute and indirect function calls.
`ret`: For function returns.
`brcf`: For PC-relative and indirect branches between different code blocks.

All other branch instructions are guaranteed hits and do not transfer control outside of the current code block. This may induce less memory requests, however, without proper compiler support large code blocks may be evicted before being fully-utilized.

2.5 Multi-Core and Bus Arbitration Policy

The Patmos multi-core platform can be built based on a set of Patmos cores connected through Network-on-Chips (NoC). As illustrated in Figure 2.4, Patmos dedicates a specific NoC for: (1) core-to-core communication that takes the form of message passing between processor scratchpad memories [65] and (2) transfers from/to the shared external main memory [109].
The memory NoC connects Patmos cores to the memory controller via a memory tree and applies a TDM schedule to arbitrate accesses to the main memory. For simplicity, we assume that each processor core may transfer a single memory burst

[2] A representation of all paths that might be traversed during the execution of the program / function. See Chapter 3 for a formal description.

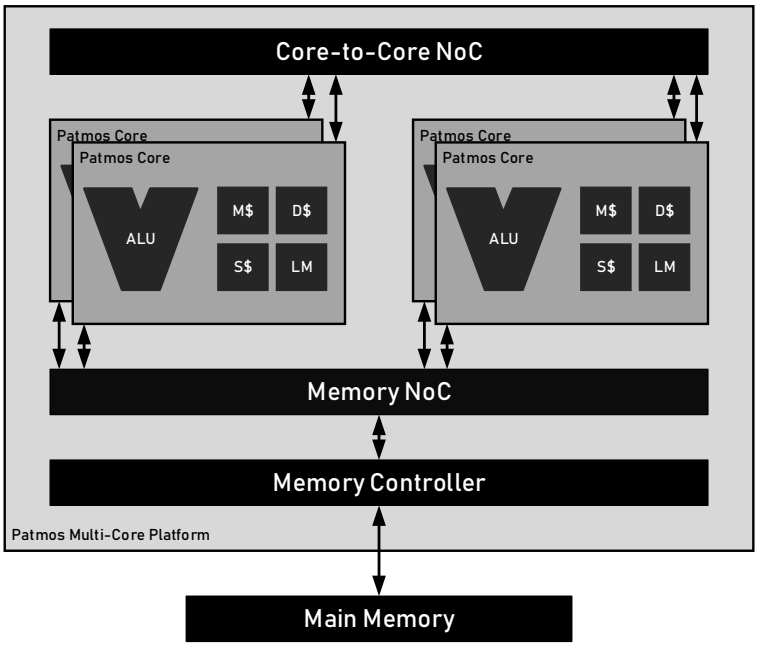


Figure 2.4 – Patmos multi-core platform.

from/to main memory in a dedicated *TDM slot*. Transfers may only be initiated at the beginning of a TDM slot. The slots are periodically scheduled in a *TDM period*. The duration of a period then depends on the number of cores n and the duration of a TDM slot k and is given by $n \cdot k$ cycles. We assume that the memory controller is able to process transfers with arbitrary start addresses and lengths. The actual memory transfer is, however, performed at the granularity of bursts, i.e., the start address and length are aligned accordingly to the burst size (excess data is either masked or discarded).
The core-to-core NoC connects Patmos cores to each others and basically consists of links, network interface, and routers. Each core is connected to a network interface which ensures the logic connection between the core and the network. Each network interface is connected to a router which itself is connected via links to its neighbors forming a 2D bi-torus structure. The routers rely on a static TDM schedule to route the stream of packets to the corresponding core.

3 Compiler Support

The compiler plays a central role in defining the performance and predictability of the real-time application. To achieve its role fully, the T-CREST project advocates for a complete support of Patmos' predictability features, as well as a tight integration into the WCET analysis process [96]. In this section, we briefly review the compiler toolchain, and present the support it provides for Patmos' features.

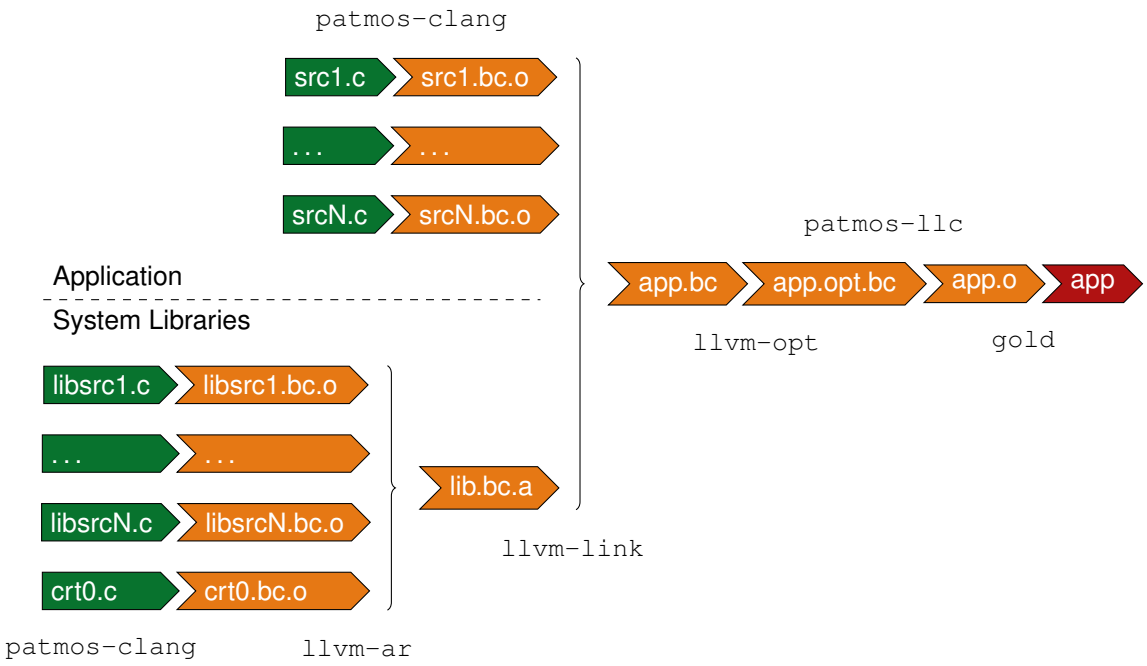


Figure 2.5 – The compilation toolchain.

3.1 Toolchain Overview

The Patmos toolchain is based on the LLVM compiler framework[3]. Figure 2.5 illustrates the compilation steps and the involved tools. The compilation flow starts by translating each source file into an intermediate representation called LLVM Intermediate Representation (LLVM-IR). This process is carried-out by the `patmos-clang` front-end that generates a *bitcode* file for each source file. This also applies to user and system libraries, which are also linked together at the bitcode-level using `llvm-link`. Moreover, a control-flow graph representation of the program can be obtained after the link stage. The main advantage of this is that a complete view of the program is available to subsequent optimization stages including the back-end [23]. A set of usually machine-independent analyses and optimizations are then performed by the `llvm-opt` tool.
The back-end mainly consists of `patmos-llc` that translates LLVM bitcode into Patmos machine code according to the ISA. A set of machine-dependent analyses and optimizations are performed in this stage right before code emission (see Section 3.2). The generated code is an ELF binary containing symbolic addresses that get translated to physical addresses by the `gold`[4] linker which defines the final data and memory layout.

3.2 Support for Patmos Features

In order to fully take advantage of the predictability features offered by the Patmos platform, a proper support needs to be provided by the compiler back-end infrastructure. We review here the compiler support to manage some of the hardware features within the `patmos-llc` tool.

[3] `http://www.llvm.org`
[4] `https://www.gnu.org/software/binutils/`

Stack Cache

Compiler support for the stack cache has been presented in the stack cache original paper [11]. The compiler manages the stack frames of functions quite similar to other architectures with the exception of the ensure instructions. Stack frames are allocated upon entering a function (`sres`) and freed immediately before returning (`sfree`). A function's stack frame might be (partially) evicted from the cache during calls. Ensure instructions (`sens`) are thus placed immediately after each call. The evicted data is consequently reloaded into the cache if needed after each call. We also restrict functions to only access their own stack frames. Data that is shared or too large can be allocated on a *shadow stack* outside the stack cache.
A more relaxed (but is not supported yet) placement of stack cache control instructions can be performed, which allows for varying frame sizes within functions in order to pass function arguments or for optimizations. The placement merely needs to be *well-formed* [63], which means that each `sres` has to be followed by matching `sfree` instructions on all execution paths (similar to well-formed braces).

Method Cache

The compiler ensures the formation of code blocks based on *code regions* and according to cache limitations, i.e., cache size, tag memory, and a user-specified maximum code block size. A code region is a subgraph of the CFG of a function with a unique-entry. This is necessary so that the access to the code block is performed based on its entry address in the branch/call instructions. The compiler performs the splitting of functions whose size is larger than the maximum code block size. The splitting is crucial to cache performance. In fact, splitting functions into too small code blocks may cause a code size overhead and many small memory transfers that may not fully profit from the memory bursts.
The algorithm presented in [57] splits the function by creating code regions that grow as the CFG is traversed. Maximizing the size of code blocks allows to reduce the needed tag memory necessary for caching functions. The algorithm starts by creating a region at the CFG root and traverses the CFG in a topological order. A basic block is included in the region of its predecessors if (1) the resulting total size is less than the maximum code block size and (2) all its predecessors are part of the same region. A violation of those conditions induces the creation of a new region. Reducible loops are handled during a preprocessing step by either adding the whole loop to the growing regions, or by making the loop header a region header. Irreducible loops are handled by creating an artificial loop header to which all edges reaching the original loop headers are directed to. The resulting structure is a reducible loop and is handled as such. There may also appear cases of computed branches induced, for instance, by the `switch` statement in the `C` language. Those are handled by creating artificial nodes so that an SCC [5] is formed with all the involved basic blocks. The splitting is then performed as before. Note that these transformations are not performed on the actual code and are only required for the computation of code regions/blocks for the method cache.

[5] In graph theory, a Strongly Connected Component (SCC) is a directed graph where every node is reachable from every other node.

Instruction	Delay Slots	Cache Fill
`call`	3	yes
`br`	2	no
`brcf`	3	yes
`ret`	3	yes

Table 2.1 – Example of control-flow instructions and their delay slots.

Predication

The LLVM code generator is able to produce predicated code at several stages. Firstly, the instruction selector is able to recognize simple *select statements* that are directly compiled to conditional moves. A generic if-conversion optimization is also available, which allows to eliminate conditional branches of complex control flow and produce predicated code. Finally, Patmos-specific code transformations are available to generate single-path programs [50].
The *Application Binary Interface* (ABI) defined by Patmos states that all predicate registers are callee saved. This means that, during a function call, the called function needs to save and restore any predicate register that it modifies. Predicate registers cannot be used to pass arguments to other functions. Predicates are, in terms of the ABI, strictly function local. Consequently, predicates can be freely used across function calls, while called functions are independent from predicates computed before entering the function. It is still possible that call instructions themselves are predicated, i.e., the function is called conditionally.

Branch Optimizations

Branch hazards are side-effects of pipelined architectures. Non-delayed branches cause the pipeline to stall until the branch is resolved. While this behavior is predictable, relying only on non-delayed branches may severely impact the performance. To mitigate their negative effects, the compiler can deal with branches in different ways. One consists of avoiding branch instructions in the first place by eliminating them. This class of techniques includes predicated execution, which we have just covered. Another one consists of performing code transformation techniques such as function inlining and loop unrolling. In function inlining, the compiler replaces some call instructions with the body of the function being called. Similarly, loop unrolling replicates the code of a loop body many times to reduce or eliminate loop control instructions. An obvious side effect of code replicating techniques is the increased code size.
Another approach consists of statically filling branch delay slots with instructions. Those instructions are ideally useful and must not depend on the outcome of the branch. Moreover, it must be ensured that their execution does not interfere with the branch being executed. The compiler performs this optimization by looking for such instructions and moving them to fill the available delay slots. In Patmos, the number of delay slots depends on the branch/call instruction (see Table 2.1). Each delay slot represents a bundle that can fit up to two instructions. Moreover, since Patmos supports predicated execution, the compiler is able to use predicates to place other branches in delay slots. For this, the compiler makes sure that the predicates are disjoint, i.e., only one branch is known to be taken at any moment at runtime.

Filling delay slots allows for better code utilization while minimizing branch hazards in a predictable fashion.

CHAPTER 3

Static WCET Analysis Framework

As the complexity of computer architectures and real-time applications grows, deriving tight and sound WCET bounds becomes a challenging task. The process, in fact, requires a deep knowledge not only about the program structure and constraints, but also the underlying computer architecture and its internal subsystems. A rigourous methodology becomes necessary to handle the complexity and provide safe WCET estimates. In this Chapter, we present some important results in static WCET analysis. Section 1 describes the overall analysis flow and its involved phases. Section 2 gives some basic background regarding static program analysis. In Section 3 we present the standard data-flow analysis framework. Then, from Section 4 through Section 7 we review relevant timing analysis results for some hardware components and features. A description of some WCET analysis tools is provided in Section 8. Finally, the conclusion is presented in the last section.

1 The Analysis Work Flow

We start this chapter by providing an overview of the static WCET analysis work flow. The goal of static WCET analysis is to derive a sound and tight WCET estimate of a program to be run on a particular computer. By sound we mean that the WCET estimate is always an over-estimation of the actual WCET. A tight estimate means that the WCET bound is precise and as close as possible to the real one. To achieve this goal, the WCET analysis applies various techniques that stem from static program analysis. In general, the analysis consists of several phases, illustrated in Figure 3.1.

CFG Reconstruction: The typical starting point of a classical WCET analysis is to reconstruct the program's control-flow from the machine code, i.e., the binary code. The control-flow graph (CFG) represents all possible paths the program can take during its execution. This is necessary for the WCET analysis as it allows to track the processor's timing behavior along each of these possible paths. To achieve this, the machine code has to be parsed and decoded in order to detect branch or call instructions causing a split of the control-flow. The target address of branches

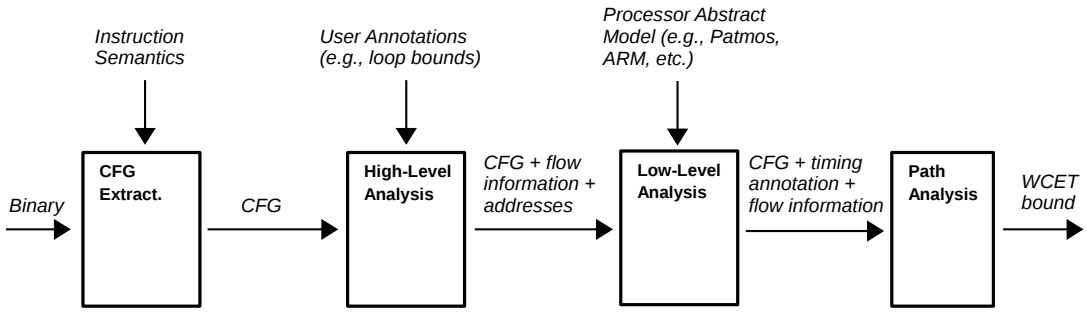


Figure 3.1 – General WCET analysis work flow.

and calls can often be determined by simply inspecting the memory layout and the symbol table of the program. Moreover, this step requires a certain knowledge of the processor's instruction set architecture (ISA) and its semantics. Section 2 provides formal description of the CFG and other concepts related to static analysis.

High-Level Analysis: The CFG serves as representation of the program, this phase consists of extracting necessary and useful information from it. The information is related to the behavior of the program and will subsequently be used to bound its execution time and tighten the WCET estimates. This phase is often carried-out using a set of data-flow analyses (DFA) for which the framework is described in Section 3. The collected information usually consists of addresses and flow-facts:

- Addresses: The machine code might not explicitly show the target address of all branches/calls or the address of data being referenced. In fact, some of them might be dynamically computed at runtime and thus stored in local registers. This situation is commonly encountered in calls by pointer where the value of the pointer is computed depending on certain conditions. Another example is the switch-case statement of the `C` language which is usually implemented using a jump table to determine the target address depending on the case value. Similar situations can also be found for addresses of referenced data. To determine addresses, a popular approach is to apply a value range analysis (VRA). The VRA analysis is a form of DFA which consists of tracking the interval of values stored in registers at every point of the program. As we will see, both high-level and low-level analysis are sensitive to the address information. Therefore, the tightness of the provided results can directly affect the precision of the WCET analysis as a whole.

- Flow-Facts: Without flow constraints, the CFG will represent an infinity of possible paths for which the WCET cannot be bounded. Flow-facts represent information about the control flow of the program that is necessary to compute the WCET or to make it more precise. Examples of this include loop iteration bounds, the depth of recursive calls, or the relative bounds which is often used for nested loops. Some flow-facts can be determined statically through a set of dedicated DFA analyses that might use the results of the VRA analysis. In the case they cannot be determined automatically, the user has to provide the information manually in the form of annotations. Flow-facts can also be used to eliminate infeasible paths, i.e., those that will never be taken under any circumstances. There can be different causes leading to infeasible paths, the main one is the correlation of conditional instructions. An example of this is the existence of paths

with consecutive contradictory conditions. Studying the relationship between conditions leading to different paths can help determining infeasible ones. The immediate benefit is the simplification of the CFG due to the elimination of such paths and the corresponding dead code which is never executed. The implication on the WCET analysis is a reduction of the complexity and the increased precision of the estimates.

Low-Level Analysis: The program itself does not disclose timing information on it. The worst-case timing behavior of the program is function of both the program *and* the processor. In particular, the processor spends a certain number of cycles to execute each instruction of the program. If all instructions would always execute in a constant time, then timing analysis would be a trivial problem to solve; one would just perform a longest path search on the CFG to maximize the number of instructions that might be part of a program execution. Computer architectures usually implement several hardware components, each targeting a specific aspect of performance in an interplay with other components. The execution time of some instruction is then determined by the timing behavior of the components being involved, which itself depends on their individual state right before the execution. This creates timing variations as different states may potentially lead to different timing behaviors.

These timing variations are essentially caused by *timing accidents*. Those are situations leading to an increase of the execution time of some instruction, also referred to as a *timing penalty*. An example of this is a cache miss on a load instruction. The execution time of the load instruction dramatically varies depending on whether the referenced data is present in the cache. In computer architectures supporting different levels of caches, the execution time of the load instruction may take a range of values. The concept also applies to branch miss-predictions, bus delays, pipeline hazards, and delays related to a DRAM memory refresh.

The goal thus is to derive potential hardware states that may occur during program execution. This is usually done through a set of data-flow analyses (or similar analyses) that *simulate* the program execution and capture its impact on the hardware state. For each of these hardware components, a dedicated analysis determines the set of its potential states at each program point. The conservative analysis needs then to investigate those potential states in order to determine whether there is a potential timing accident and, if so, associate the corresponding timing penalty. Given this information, the local worst-case timing of instructions can be derived and annotated to the CFG.

As different processors may come with different characteristics, it is necessary to provide an abstract model of the processor. This consists of a description of the computer architecture on which the analyses can run to conservatively derive the timing behavior for all instructions of the ISA. This typically includes information regarding the CPU registers, pipeline stages, memory hierarchy, and how instructions affect the processor state. The abstract model can be either provided as an input to the tool, or being part of the tool itself.

The difficulty here resides in the complexity of the computer architecture and the ability to build a valid and precise abstract model from a, usually, incomplete documentation. In general, the more complex the computer architecture, the more difficult it will be to derive a precise worst-case timing behavior for instructions.

Precisely tracking the complex hardware behavior may be expensive in terms of computational and space complexity. Abstraction may be used to handle the complexity, however, the precision loss results in a larger set of potential hardware states with possible timing accidents. Also, the absence of information regarding some hardware component may require considering all its possible states to ensure the soundness. To add more difficulty to the matter, conventional architectures with complex interactions between hardware components are subject to timing anomalies. Identifying such situations requires tracking a long execution history, often deeming the whole process infeasible. Time-predictable architectures address this issue by imposing some design rules to simplify the behavior of hardware components and thus eliminate timing anomalies. The absence of timing anomalies suppresses the need to track the execution history in order to provide sound WCET estimates. Moreover, the timing composability allows to separately study the timing behavior of hardware components.

Path Analysis: Once the local WCET is determined for each basic block in the CFG, this step consists of computing an estimation of the program's WCET. This is typically performed using the *Implicit Path Enumeration Technique* (IPET) [119]. The idea consists of combining flow information obtained from the high-level analysis with the local WCETs of basic blocks, in order to formulate an integer linear program (ILP). Each edge of the CFG is associated with (1) a time cost denoted as $\hat{c}_i$ representing the contribution of the basic block to the global WCET, as well as (2) a count variable, x_i, representing the number of times the basic block is executed. A safe estimation of the program's WCET is obtained by finding the *worst-case execution path* (WCEP) through the annotated CFG. The WCEP is obtained by performing a longest path search that maximizes the goal function $\sum_{i=1}^{N} \hat{c}_i x_i$, where N is the total number of basic blocks. Solving the ILP requires expressing a set of flow equations that take into consideration different constraints. In particular, structural constraints are those stemming directly from the program's control flow. For instance, the equations must verify that the sum of in-flow execution reaching some basic block is equal to the sum of out-flow execution. Also necessary to formulate and solve ILP are the loop bounds which are either automatically computed during the high-level analysis or manually provided by the user. Once the problem is formulated, the actual solving is typically performed using standalone tools such as `CPLEX` [1] or `GLPK` [2]. The result is often provided in the form of CPU cycles.

2 Basic Concepts for Program Static Analysis

In order to formally study and analyze the timing behavior of programs, we first need to set the theoretical foundations and terminology related to static program analysis. This section provides formal definitions of most important concepts.

2.1 Control-Flow Graph

In a first step, we view a program as a collection of instructions that execute in some order. The control-flow graph (CFG) is a fundamental representation of the program

[1] `https://www.ibm.com/fr-fr/analytics/cplex-optimizer`
[2] `https://www.gnu.org/software/glpk/`

that determines the execution order of instructions. Very often, program instructions are organized into straight-line code sequences called the basic blocks.

Definition 1 *(Basic-Block) A basic-block (BB) is a maximum sequence of consecutive instructions that has a unique entry point, and a single exit point.*

The first instruction of the sequence corresponds to the entry point of the basic block. This can be the first instruction of a function, the target of a branch instruction, or a fall-through after a conditional branch. The exit point, on the other hand, typically consists of a control-transfer instruction that directs the flow of execution to another basic block. Examples include branch (`br`, `brcf`, `brnd`, `brcfnd`), call (`call`, `callnd`), and return (`ret`, `retnd`) instructions. The call instruction may, sometimes, not be considered as an exit point of the basic block. For simplicity, we explicitly end the basic block at call instructions. Also, in some cases the exit point may not always be a control transfer instruction, and that a basic block may just fall through its successor.
Given the description above, one can formally model the execution order of basic blocks by looking at execution flow relations. Using this, we can define a CFG for a single function, also known as an *intra-procedural* CFG.

Definition 2 *(Control-Flow Graph) The control-flow graph (CFG) of a single function is a directed graph $G = (N, E, r, t)$. Nodes in N represent basic blocks and edges in $E \subseteq N \times N$ the execution flow. Nodes r and $t \in N$ denote unique entry and exit points respectively. To each node $u \in N$ corresponds a program point p_u. Additionally, we define $Succs(v) = \{u \mid (v, u) \in E\}$, the set of immediate successors of v, and $Preds(v) = \{u \mid (u, v) \in E\}$, the set of immediate predecessors of v.*

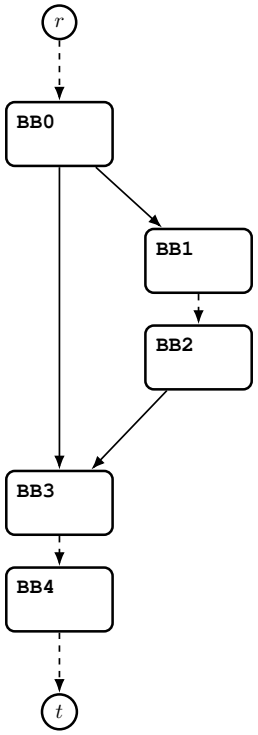
In a sense, the CFG represents all possible execution *paths*, i.e., all possible sequences of CFG nodes starting from the entry point to the exit point. It is, thus, a convenient way to represent not only functions but also any substructure such as loops or conditional statements. Moreover, the information contained in the CFG allows to perform flow-sensitive analyses intended for program optimization, we cover this aspect in the next section. Note that the CFG nodes also represent individual instructions. In this case, additional edges are needed to represent the execution flow of instructions within the same basic block.

Definition 3 *(Path) A path is an ordered sequence of nodes $(v_1, \ldots, v_n)$ such that, for $i < n$, all edges (v_i, v_{i+1}) are in E. A path π is said to pass through a node u, denoted by $u \in \pi$, if $\exists\, i, 0 < i \leq n : v_i = u$. We assume that all nodes are reachable from the root node r, i.e., there exists a path from r to every node in the CFG, and that the sink node t is reachable from every node.*

Example 2.1 *Consider the function `F()` written in Patmos assembly language shown in Figure 3.2a. In this example, we build its corresponding CFG illustrated in Figure 3.2b. The function `F()` simply performs two function calls one of which depends on the value of the general purpose register `r9`. Additionally, `r1` and `r2` are used to count the number of branches and calls performed during the execution. The compiler schedules the instructions and sets the order of basic blocks in the memory layout. Moreover, the compiler uses branches to direct the flow of execution from*

```
func F()
BB0:
       sres   1
       mov    r1 = r0
       mov    r2 = r0
       cmpneq p1 = r9,  0x0
 (p1)  br     0x6            //cond. branch to BB3
 (p1)  addi   r1 = r0,  0x1
       nop
BB1:
       addi   r2 = r0,  0x1
       callnd G            //non-delayed call
BB2:
       sens   1
BB3:
       addi   r2 = r0,  0x1
       callnd H
BB4:
       sens   1
       ret
       sfree  1
       nop
       nop
```

(a) Definition of the `F()` function in the Patmos assembly code assuming a single-issue execution. The labels on the left represent basic blocks.



(b) The control-flow graph of the `F()` function.

Figure 3.2 – Portion of a program in Patmos assembly language and its corresponding machine code. Each line in the machine code corresponds to a line in the assembly code.

a basic block to another. The function consists of five basic blocks each of which is represented by a node in the CFG with identical names.
The basic block `BB0` starts with the `sres` instruction which, also, represents the function entry point. Consequently, the edge $(r, BB0)$ connects the root node r to the basic block `BB0`. The `cmpneq` instruction compares the content of `r9` to the immediate value 0 and sets the predicate register `p1` if the comparison evaluates to true. The `br` instruction is a delayed conditional branch with no cache fill that is executed depending on the value of the predicate register `p1`. The `br` has 2 delay slots, meaning that the CPU can execute up to two more instructions before the branch actually takes effect. Moreover, the content of `r1` is incremented using `addi` if the branch is taken. If the branch is not taken, then `BB0` falls into `BB1`. These two possible executions are represented by two edges (`BB0, BB3`) and (`BB0, BB1`) respectively.
In `BB1`, a call to the function `G()` is performed and the `r2` is incremented. This call is non-delayed meaning that the pipeline cannot fetch any more instructions until the call instruction passes the `MEM` stage. Therefore, the non-delayed call marks the end of the current basic block. When the function `G()` returns, the flow of execution is directed to the next instruction located right after the call. This instruction is located in `BB2`, which itself falls into `BB3`. We choose to represent the edge (`BB1, BB2`) as dashed as the execution flow is transfered indirectly from `BB1` to `BB2` (due to the

call).
The edges (`BB2, BB3`) and (`BB0, BB3`) reach the basic block `BB3` and form a join point. In this basic block, a function call to `H()` is performed with an increase in the `r2` register. A dashed edge (`BB3, BB4`) is then used as explained before. The basic block `BB4`, executes until reaching the instruction `ret` which is a delayed return with 3 *delay slots. The compiler schedules instructions in the available delay slots before the return takes effect. This marks the end of the basic block `BB4` and the function `F()`. The edge (`BB4, t`) is then used connects the last node to the tail node.*
In this intra-procedural CFG, there are two possible execution paths depending on whether the branch is taken or not. The path $\{r, \texttt{BB0}, \texttt{BB1}, \texttt{BB2}, \texttt{BB3}, \texttt{BB4}, t\}$ *corresponds to the scenario where the branch is taken. Assuming the called functions do not alter the contents of registers `r1` and `r2`, these registers hold the values* 1 *and* 2 *respectively at the end of the execution. The other possible path* $\{r, \texttt{BB0}, \texttt{BB3}, \texttt{BB4}, t\}$ *represents the scenario with the branch not taken. The contents of `r1` and `r2` registers then evaluates to* 0 *and* 1 *respectively at the end of the execution.*

2.2 Loops

Programs spend most of their execution time in loops. They are, therefore, natural candidates for compiler optimizations and WCET analysis. Considering the CFG definition, we can identify loops as a non-trivial strongly connected component (SCC) or trivial SCC with a self-edge.

Definition 4 *(Loop) A loop is a set of nodes* $L \subseteq N$ *forming an SCC subgraph in the CFG. An edge* (u, v) *is called an entry edge of a loop* L*, if* $u \notin L$ *and* $v \in L$*. The node* v *is then called an entry node. We call a loop with a single entry node reducible, all other loops are irreducible. Similarly, we call a CFG reducible when all its loops are reducible, the CFG is otherwise irreducible.*

Most loop optimization techniques target reducible loops with a single entry node. Several approaches can be used to identify loops within the CFG. A popular one relies on dominance-based relationships between the nodes of the CFG.

Definition 5 *(Dominance) Let* u *and* v *be nodes in the CFG. The node* u *dominates the node* v *if every path from the root node* r *to* v *passes through* u*. The node* u *strictly dominates* v *if* u *dominates* v *and* $u \neq v$*.*

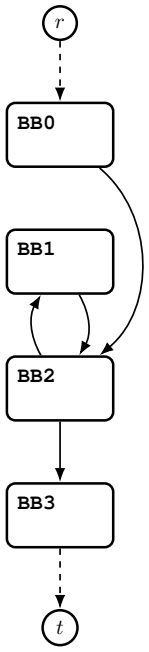
Definition 6 *(Post-Dominance) Let* u *and* v *be nodes in the CFG. The node* u *post-dominates the node* v *if every path from the node* v *to the sink node* t *passes through* u*.*

Once dominators are found in the CFG, loops can be identified by investigating their corresponding loop headers (i.e., entry nodes). Loop headers are useful to identify *natural* loops.

Definition 7 *(Loop Header) A loop header is a node* h *within a loop such that no other node in* L *strictly dominates* h*. Edges of the form* (u, h)*,* $u \in L$*, leading to a loop header are called back edges.*

```
func H()
BB0:
       sres   1
       br     0x4             //branch to BB2
       addi   r1 = r0,  0x9
       nop
BB1:
       subi   r1 = r1,  0x1
BB2:
       cmpneq p1 = r1,  0x0
 (p1)  brnd   -0x2           //cond. branch to BB1
BB3:
       ret
       sfree  1
       nop
       nop
```

(a) Definition of the `H()` function in the Patmos assembly code assuming a single-issue execution.



(b) The control-flow graph of the `H()` function.

Figure 3.3 – Example of a program in the Patmos assembly language and its corresponding CFG demonstrating the use of a single loop.

Natural loops are reducible loops with a single loop header and one or potentially several back edges. Each back edge has a corresponding natural loop.

Definition 8 *(Natural Loop) A natural loop of the back edge (u, h) is a reducible loop consisting of the smallest set of nodes $L \in N$ that includes the back edge and for which the only predecessors outside L are those of the header h.*

Loops may also be nested. In such a case, their hierarchy in the SCC may be represented using a loop nesting forest [98] that we do not cover in this section. A loop that contains no other loop is called an inner loop.

Example 2.2 *Consider the function `H()` written in Patmos assembly language shown in Figure 3.3a and its corresponding CFG illustrated in Figure 3.3b.*
The function simply sets and decrements the content of the register `r1` until it reaches 0*. The function consists of four basic blocks each of which is represented by a node in the CFG with an identical name. The CFG is constructed just similarly as in Example 2.1.*
In `BB0`, the function initializes the content of `r1` and performs a delayed branch to `BB2`. In `BB2`, a conditional branch to `BB1` is performed based on the result of the comparison performed by the `cmpneq` instruction. Then, `BB1` falls into `BB2` right after `r1` is decremented. This cycle is repeated until `r1` evaluates to 0*, in which case `BB2` falls through `BB3`. The function then returns to the caller.*
The nodes `BB1` and `BB2` form a loop body and the edge $(\texttt{BB0}, \texttt{BB2})$ is its entry edge. The node `BB2` is the loop header as no other node of the loop strictly dominates it (i.e., all paths from r to `BB1` pass through `BB2`). Also, `BB1` does not post-dominate `BB2` as there is a path from `BB2` to t that does not pass through `BB1` (i.e., $\{\texttt{BB2}, \texttt{BB3}, t\}$).

The loop is also a natural one of the back edge (`BB1`, `BB2`). Finally, the node `BB3` post-dominates all the nodes of the function as every path from any node to the sink node t passes through `BB3`.

2.3 Inter-procedural CFG and Call Graph

In general, a program consists of several functions that call each others to realize a higher-level computing task. In order to represent the whole program, an *inter-procedural* CFG (ICFG) needs to be constructed. The ICFG is a super-graph that combines intra-procedural CFGs of all program functions using call and return relations between them. This is done by further introducing two types of edges representing the call sites and returns. The node with call site is then connected to the node representing the first basic block of the callee (i.e., the function to be called). The return edge connects the last basic block of the callee to the successor of the calling node in the intra-procedural CFG.

Definition 9 *(Inter-Procedural CFG) The inter-procedural CFG (ICFG) of a program is a directed graph $G = (N, E, r, t)$. The set N is the union of all nodes of all intra-procedural CFGs of functions of the program. Nodes r and $t \in N$ denote the program unique entry and exit points respectively. The set E is the union of all edges in all intra-procedural CFGs, extended with the set of call and return edges. We assume that each call site is associated with a corresponding call instruction. A Call edge (u, v) connects the calling node u to the node v representing the first basic block of the callee function. A return edge (g, h) connects the returning node g to the node h such that $h = Succ(u)$ in the caller's CFG.*

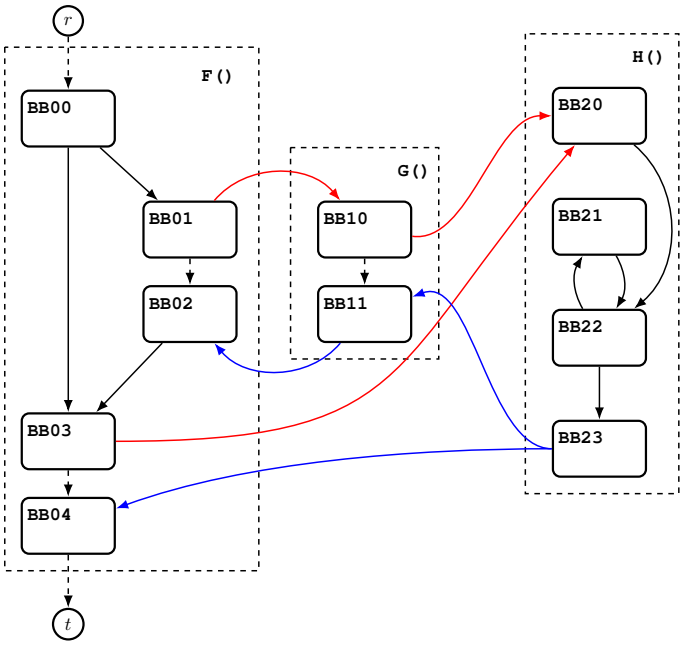
Depending on the analysis problem, it may be sufficient to represent only the call relations between functions. For this, we use the call graph.

Definition 10 *(Call Graph) The Call Graph (CG) of a program is a directed graph $C = (F, A, s)$. Nodes in F represent functions and edges in A call sites. Node $s \in F$ is the program's entry point. We assume that each call site is associated with a corresponding call instruction of the CFG of a function.*

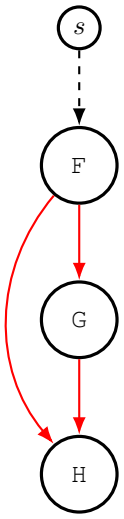
The CG can also be used in combination with intra-procedural CFGs to perform inter-procedural analyses. In this case, analyses operate on individual CFGs to model the effects of functions and assign summaries to the corresponding edges in the CG. Moreover, path analysis often operates on the program's ICFG. In order to compute the WCET bound, time costs resulting from the execution of basic blocks are typically assigned to ICFG edges. Assigning costs to a graph results in a weighted graph.

Definition 11 *(Weighted Graph) A weighted control-flow (or call graph) is associated with a function $\mathcal{W} : N \cup E \to \mathbb{N}$ ($\mathcal{W} : A \cup F \to \mathbb{N}$) to each node and/or edge in the graph.*

Example 2.3 *Consider a program consisting of three functions `F()`, `G()`, and `H()`. The functions `F()` and `H()` are defined as in the previous examples, whereas `G()` consists of a straight-line code with a single call site to `H()`. In this example, we*



(a) ICFG of a program consisting of functions `F()`, `G()` and `H()`. Call edges are represented in red, whereas return edges are in blue.



(b) The corresponding CG of the program.

Figure 3.4 – Example of an ICFG of a program and its corresponding CG.

build the ICFG shown in Figure 3.4a and the corresponding call graph shown in Figure 3.4b. For the ICFG, we first need to build the individual CFGs of its functions. We use the already built CFG of `F()` *and* `H()` *and make one for* `G()` *as shown in Figure 3.4a. These CFGs are connected using call and return edges to represent call relations between functions. In our example,* `F()` *calls* `G()` *in* `BB01`*, therefore, the call edge* (`BB01`, `BB10`) *connects* `BB01` *to* `BB10` *which is the callee's first basic block. On the other hand, the return edge* (`BB11`, `BB02`) *connects the returning node* `BB11` *to* `BB02`*, which is the immediate successor of the calling node* `BB01`*. The same is observed for call relations* `G()`/`H()` *and* `F()`/`H()`*. In the call graph, each function of the program (i.e.,* `F()`*,* `G()` *and* `H()`*) is represented by a single node. The node s represents the program entry point which is located in the function* `F()`*. Moreover, each call site is represented using a directed edge from the caller to the callee. For instance, the function* `F()` *calls both the* `G()` *and* `H()` *functions in two different call sites. Therefore, each call site is represented by a unique edge.*

3 Data-Flow Analysis Frameworks

Data-flow analysis (DFA) is a framework allowing to determine facts or information about the program without executing it. It was developed by Gary Kildall in 1973 [66]. Later, Cousot and Cousot provided, in their seminal work [30], a semantic foundation for building analyses in the framework, as well as means to reason about their correctness. The applicability of abstract interpretation extends data-flow analysis. Unless explicitly mentioned, we will follow the tradition of data-flow analysis as presented in [14].

Data-flow analysis is a powerful technique and has been extensively used in compilers

for program optimization. The usefulness can be illustrated through popular optimization techniques such as *register allocation* which exploits the results of *liveness analysis*. This analysis allows to determine, for each program point, the set of potentially *live* variables, i.e., those that might be read before they are written again. Variables that are not live can be safely eliminated as they do not influence program results. Other classic optimization techniques based on DFA include *constant propagation*, and *dead code elimination*. From a WCET standpoint, DFA frameworks can be defined to determine the timing behavior of hardware components and programs using them. We will cover some of these techniques in the subsequent sections.
DFA is based on the observation that the manipulation of data during program execution has a direct impact on the program state. This includes variables, stack data, and even low-level components of the computer architecture such as internal CPU registers, or cache memory. A program execution can thereby be seen as a sequence of transformations of an abstract program state. In DFA, we statically *simulate* and observe how the computer state is transformed along all possible execution paths, so that assertions could be made regarding particular aspects of the program at specific program points. Those assertions represent the safe and most precise approximations that could be obtained for a program point, regardless of the reaching path.
One of the virtues of DFA is being a framework whose components can be defined to solve a variety of static analysis problems. A DFA is typically defined by a tuple $A = (\mathcal{D}, T, \sqcup)$ where:

$\mathcal{D}$	is the abstract domain representing the set of values the information can take at any program point.
$T : \mathcal{D} \to \mathcal{D}$	is the transfer function modeling the effect of instructions on the information.
$\sqcup : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$	is a join operator applied on domain values to merge information in program joins.

Together with an (I)CFG an instance of an (inter) intra-procedural DFA can be formed, yielding a set of data-flow equations. The equations are typically solved by iteratively applying these functions until a fixed-point is reached [14]. In the following, we describe in detail the DFA framework and its properties.

3.1 Abstract Domain

The information that could be derived might be very complex to compute, and tracking it may not be possible due to efficiency and practical considerations. For instance, tracking the numerical values of variables during a liveness analysis is irrelevant for the problem and just adds unnecessary complexity to the process. It is thus important to find a suitable representation of the information that can be processed efficiently and captures the relevant characteristics of the information that is to be derived about the program states. This is achieved through abstraction. An abstraction is merely a representation of the *concrete* program states with respect to some particular aspect, potentially dropping, simplifying, or summarizing the information of the concrete state. By concrete we mean the actual state resulting from an actual execution on the computer.

Based on this abstraction, the DFA analysis associates with each program point a data-flow value representing a particular fact. The set of all possible values that could be taken is known as the *abstract domain* that we denote as $\mathcal{D}$. Assuming programs terminate, the domain in general consists of a finite set of values summarizing all possible concrete states generated by the program. The nature of the information depends on the analysis problem. However, since we are interested in formulating a general framework to handle (large) classes of analyses, the domain values need to be comparable. Again, the nature of the information and the meaning of the comparison is problem-dependent. In any case, the domain needs to form a partial order.

Definition 12 *(Partial Order) A partial order on a set $\mathcal{S}$ is a binary relation $\sqsubseteq$ over $\mathcal{S} \times \mathcal{S}$ such that:*

1. *$\sqsubseteq$ is reflexive: $\forall x \in \mathcal{S}, x \sqsubseteq x$.*
2. *$\sqsubseteq$ is transitive: $\forall x, y, z \in \mathcal{S}$, $x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$.*
3. *$\sqsubseteq$ is anti-symmetric: $\forall x, y \in \mathcal{S}$, $x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$.*

Furthermore, the partial order allows to define the following upper and lower bound relations. Assuming $x \in \mathcal{S}$, and $\mathcal{X} \subseteq \mathcal{S}$:

upper bound (ub): x is a ub of $\mathcal{X}$ iff $\forall x' \in \mathcal{X}$, $x' \sqsubseteq x$.

lower bound (lb): x is a lb of $\mathcal{X}$ iff $\forall x' \in \mathcal{X}$, $x \sqsubseteq x'$.

least upper bound (lub): x is the lub of $\mathcal{X}$, denoted $\sqcup\mathcal{X}$, iff $\forall x' \in \mathcal{X}, x' \sqsubseteq x$, and $\forall x'' \in \mathcal{S}$ such that $\forall x' \in \mathcal{X}$, $x' \sqsubseteq x''$, we have $x \sqsubseteq x''$.

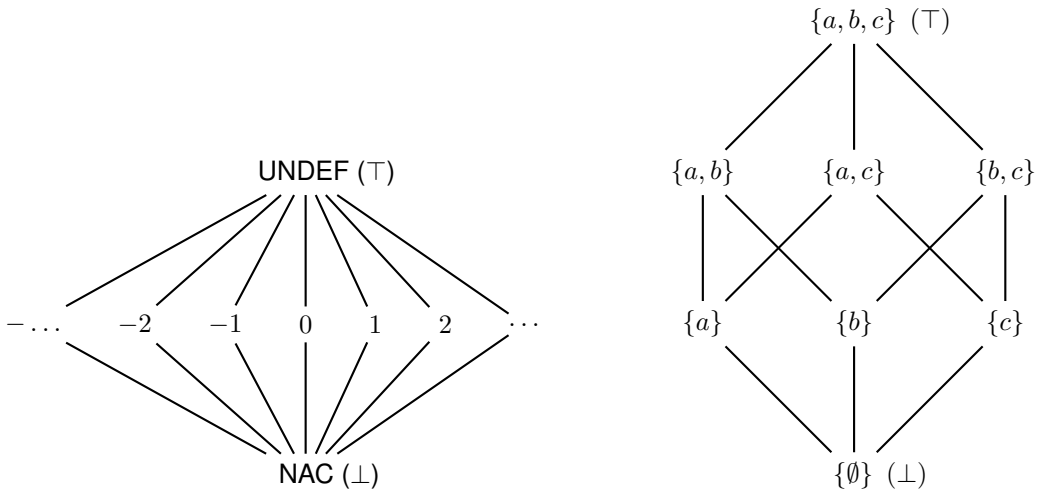
greatest lower bound (glb): x is the glb of $\mathcal{X}$, denoted $\sqcap\mathcal{X}$, iff $\forall x' \in \mathcal{X}, x \sqsubseteq x'$, and $\forall x'' \in \mathcal{S}$ such that $\forall x' \in \mathcal{X}$, $x'' \sqsubseteq x'$, we have $x'' \sqsubseteq x$.

The lub is also called the *join* of the partial order $\mathcal{S}$, whereas the glb is referred to as the *meet*. The meet and the join can also operate on individual elements x and y of the partial order $\mathcal{S}$. These operators are useful for the DFA as they provide means to merge any pair of information contained in $\mathcal{D}$ at join points. Depending on the analysis problem, either the join, the meet, or both might be required to perform the merge.
Given the discussion above, we can define the domain $\mathcal{D}$ as a lattice.

Definition 13 *(Lattice) A lattice $\mathcal{L}$ is a partial order where every two elements have a unique greatest lower bound and least upper bound.*

However, in order to guarantee that the join operator exists for every possible subset of values in $\mathcal{L}$, we need to consider a complete lattice.

Definition 14 *(Complete Lattice) A lattice $\mathcal{L}$ is complete iff for each subset $\mathcal{S}$ of $\mathcal{L}$, both $\sqcup\mathcal{S}$ and $\sqcap\mathcal{S}$ exist in $\mathcal{L}$.*



(a) Domain for constant propagation analysis (b) Domain for liveness analysis

Figure 3.5 – Examples of some domains

This means that even the empty and infinite subsets of $\mathcal{L}$ must have a lub and a glb in $\mathcal{L}$. By convention, a particular data-flow value denoted as $\top$ is used to represent the largest greatest element. In that regard, the value $\top$ is the lub of all possible subsets $\mathcal{S}$ of $\mathcal{L}$. Conversely, the data-flow value $\bot$ is the glb and could be used for instance to initialize the analysis.

Example 3.1 *In Figure 3.5a we illustrate the domain for constant propagation analysis. The analysis simply consists of determining whether variables are constants, and if so, track their unique values. The interpretation of the domain is as follows: The value $\top$ represents the absence of knowledge regarding whether the variable is a constant or not. The variable in this case is considered undefined. If the variable is deemed constant at a program point, then the corresponding numerical value is assigned to that point. The value $\bot$ means that the variable is certainly not a constant, which is a different information from the above. In liveness analysis, we determine the set of potentially live variables at program points. Therefore, the domain in Figure 3.5b has to represent all possible combinations of sets of variables. Again, the value $\top$ represents the case where no information is known, that is, all variables might be live. On the other hand, the value $\bot$ is interpreted as no variable is live at this point.*

3.2 Transfer Functions

So far, we defined the general mathematical structure to represent information held in the domain. Now, we need to capture transformations that are induced by instructions. Transformations to the information depend primarily on the analysis problem. For instance, liveness information may only be impacted by read and write operations on variables. Other instructions are completely transparent from the analysis point of view as they have no implication on the liveness.
Very often, the effect an instruction of the CFG has on the input information depends on the instruction class it belongs to (e.g., read, write). It is, therefore, convenient to define a transfer function for instruction classes and map the instructions in the

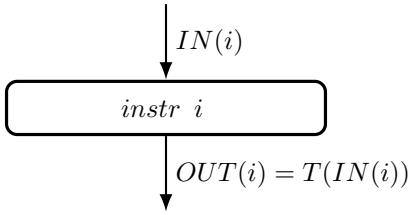


Figure 3.6 – The transfer function transforms the input information.

CFG to their corresponding instruction class. For simplicity, we will use i to refer directly to the instruction class of some instruction in the CFG. The impact of the instruction i on the data-flow value $d \in \mathcal{D}$ can simply be captured by a transfer function $T_i : \mathcal{D} \to \mathcal{D}$ in T that changes input value $\mathrm{IN}(i) = d$ reaching the instruction i into output $\mathrm{OUT}(i) = d'$, where $d' \in \mathcal{D}$. This process is illustrated in Figure 3.6. Another important property of a transfer function is monotonicity, i.e., order preservation. This is often necessary to guarantee the termination of the analysis.

Definition 15 *(Monotonic Function) A transfer function T is monotonic iff:*

$$\forall x, y \in \mathcal{L} : x \sqsubseteq y \Rightarrow T(x) \sqsubseteq T(y)$$

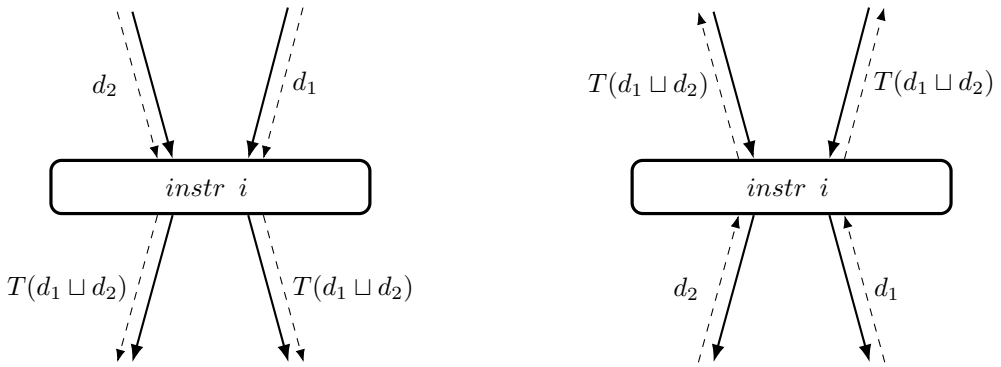
3.3 Forming DFA equations

Once a transfer function and a join operator is defined for the domain, one can now form DFA equations to be solved. Forming DFA equations needs also to take into consideration the *direction of the analysis*. In many cases, we want to collect and propagate the information along the paths according to the execution flow. However, we might also be interested in gathering information from the *future*. For instance, determining whether some variable is live at some program point p depends on whether the variable may be used in any path starting with p before a definition. We call a DFA *forward* if the information flows in the same direction as the control flow, otherwise, it is called *backward*. A forward DFA analysis starts at the CFG entry point. The forward flow direction is modeled through data-flow equations formed immediately before and after each instruction.

$$\mathrm{OUT}(i) = T(\mathrm{IN}(i)) \tag{3.1}$$

$$\mathrm{IN}(i) = \bigsqcup_{p \in Preds(i)} \mathrm{OUT}(p) \tag{3.2}$$

The output information $\mathrm{OUT}(i)$ is computed by applying the transfer function to the input information $\mathrm{IN}(i)$. The input information is itself conservatively approximated using the join operator $\sqcup$ over all the predecessors $p \in Preds(i)$ of instruction i, as illustrated in Figure 3.7a. Conversely, a backward DFA analysis starts at the program exit point and information is propagated in the opposite direction of the flow, i.e., from $\mathrm{OUT}(i)$ to $\mathrm{IN}(i)$. Moreover, the join operates on successors of instruction i rather than predecessors, as shown in Figure 3.7b.



(a) Information merge in forward DFA. (b) Information merge in backward DFA.

Figure 3.7 – Direction of the information flow in forward and backward analyses. Dashed arrows represent the flow of information, while the regular arrows represent the flow of execution.

$$\mathrm{IN}(i) = T(\mathrm{OUT}(i)) \tag{3.3}$$

$$\mathrm{OUT}(i) = \bigsqcup_{s \in Succs(i)} \mathrm{IN}(s) \tag{3.4}$$

Example 3.2 *Considering again the example of liveness analysis, we can determine that the flow of information is backward. A variable v is considered live at a program point p if there is a* use *before a* definition *of v along some path starting at p. A use implies a read operation (e.g., load) on the variable, whereas a definition is assimilated to an assignment (e.g., store). We define the $Use(i)$ and $Def(i)$ functions to capture the set of variables V that have been respectively read and assigned by the instruction i. Note also that some instructions may perform at the same time a read and an assignment.*

$$Use(i) = \begin{cases} V & , \textit{if } i = \textit{read} \\ \emptyset & , \textit{otherwise} \end{cases} \qquad Def(i) = \begin{cases} V & , \textit{if } i = \textit{assignment} \\ \emptyset & , \textit{otherwise} \end{cases}$$

Given the use and def information, we can compute the set of live variables at program points. Recall that the domain is defined as a lattice representing all possible combinations of sets of variables existing in the program (see Figure 3.5b). The transfer function captures the effect of an instruction i on the liveness information such that: (1) If a variable v is used in i then v is live at IN(i), and (2) if v is live at $OUT(i)$ and is not included in $Def(i)$ then v is also live at IN(i).

$$IN(i) = Use(i) \bigcup (OUT(i) \setminus Def(i)) \tag{3.5}$$

$$OUT(i) = \bigcup_{s \in Succs(i)} IN(s) \tag{3.6}$$

$$\tag{3.7}$$

Algorithm 1 Example of a work-list algorithm to solve a forward data-flow problem.

```
1:  procedure AnalyzeCFG(G)
2:      Let N be the set of nodes in G
3:      Let n_0 be the entry node in N
4:      Let d_0 be some data-flow value in D
5:      WL := ∅
6:
7:      for each node n in N do                    ▷ Initialize data-flow variables
8:          if n = n_0 then
9:              IN(n) = d_0
10:         OUT(n) = ⊤
11:         WL := WL ∪ n                    ▷ insert instruction into the work-list
12:
13:     while WL ≠ ∅ do                                  ▷ process the work-list
14:         Get n from WL                  ▷ get some instruction from the work-list
15:         WL := WL \ n                   ▷ remove the instruction from the work-list
16:         if n ≠ n_0 then                ▷ check if OUT information has changed
17:             IN(n) = ⊓_{p∈Preds(n)} OUT(p)          ▷ perform a join operation
18:         d = F_i(IN(n))                 ▷ apply transfer function on IN information
19:         if d ≠ OUT(n) then             ▷ check if OUT information has changed
20:             OUT(n) = d                          ▷ if so, update OUT information
21:             WL := WL ∪ Succs(n)        ▷ and insert i successors into the work-list
```

Moreover, the analysis provides the set of variables that may *be live at any path starting from i. Therefore, the liveness information at $OUT(i)$ is determined as the union of the information at $IN(s)$ for each successor $s \in Succs(i)$. The join operator $\sqcup$ is thus defined as a union $\cup$ involving domain values at join points.*

3.4 Solving DFA equations

Once DFA equations are formed at program points, the next step consists of actually solving them. By ensuring that the domain is a complete lattice, and that the transfer functions are monotonic, the existence of a fixed point solution is guaranteed for the analysis problem. The result expressed by the fixed point represents the fact that holds for the particular program point, regardless of the chosen execution path.

Definition 16 *(Fixed Point) A fixed point of a function $T : \mathcal{L} \to \mathcal{L}$ is value $v \in \mathcal{L}$ such that:*

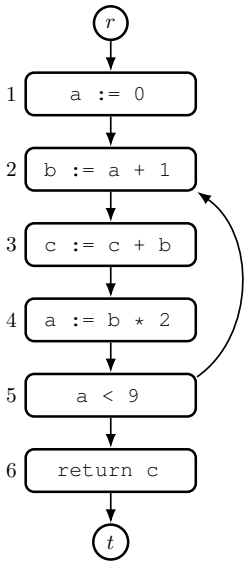
$$T(v) = v$$

Data-flow analyses can easily be implemented using a work-list algorithm that iteratively computes data-flow values until a fixed-point is reached. We present in Algorithm 1, an example of such an implementation strategy for a forward data-flow analysis.

The algorithm operates on the function's CFG denoted as G, where N is the set of nodes in the CFG. Usually, nodes in the CFG represent basic blocks. For simplicity, we assume that each node represents a single instruction in the function. This has

			1st		2nd		3rd	
Point	**Use**	**Def**	**OUT**	**IN**	**OUT**	**IN**	**OUT**	**IN**
6	{c}	{}	{}	{c}	{}	{c}	{}	{c}
5	{a}	{}	{c}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c}
4	{b}	{a}	{a,c}	{b,c}	{a,c}	{b,c}	{a,c}	{b,c}
3	{b,c}	{c}	{b,c}	{b,c}	{b,c}	{b,c}	{b,c}	{b,c}
2	{a}	{b}	{b,c}	{a,c}	{b,c}	{a,c}	{b,c}	{a,c}
1	{}	{a}	{a,c}	{c}	{a,c}	{c}	{a,c}	{c}

(a) Liveness information at program points.



(b) Program CFG.

Figure 3.8 – Example of liveness analysis. Taken from [3].

no implication on the correctness of the results, but it may require more space and computation time due to the increased number of edges and nodes. In case the nodes represent basic blocks, transfer functions can also be defined by composing the sequence of their instructions.

The algorithm consists of two main phases. First, data-flow variables are initialized for every instruction of the function (l. 7). Depending on the analysis problem, a particular value $d_0 \in \mathcal{D}$ is assigned to initialize the input information $\text{IN}(n_0)$ at the entry node n_0. This could be, for instance, $\top$ if no information is known at the function's entry point. Output information is also assigned a data-flow value that we denote as $\top$. Every node n for which the $\text{OUT}(n)$ information has been initialized is then put into the work-list WL for processing.

The second phase consists of actually computing the fixed-point solution. The analysis processes nodes present in the WL, and terminates when the work-list WL becomes empty (l. 13). The processing order may be different depending on whether the analysis is forward or backward. In a forward analysis, it is more convenient to process the nodes of the work-list in a post order as a particular node is visited only when all its predecessors have been visited. In our case, the entry node n_0 will be picked-up first. Every node that is visited needs to be removed from the work-list. As the input information for n_0 has been initialized (l. 9), there is no need to compute a data-flow value for it. However, the $\text{IN}(n)$ information of other nodes needs to be computed before applying the transfer function. For this, a join operation needs to be applied in order to safely merge information potentially reaching a join point. A temporary data-flow value d is computed by the transfer function, and then compared to the current $\text{OUT}(n)$ information for node n. If the information has changed then the value d is assigned to $\text{OUT}(n)$, and its successors are added to the work-list. Otherwise, the next node is picked-up from the work-list.

Example 3.3 *Consider the CFG shown in Figure 3.8a. In this example, we compute liveness information at program points as shown in Table 3.8b. For simplicity, we will refer to statements with their corresponding node number (showen in the left of*

each node). Since the analysis is backward, it is convenient to iterate the CFG from bottom-up so that the fixed-point is reached faster. Moreover, the analysis is initialized by assigning $OUT(i)$ and $IN(i)$ of each program point to $\emptyset$. The statement in node 6 performs a read operation on the variable $\{c\}$. Therefore, the $Use(6)$ function returns the variable $\{c\}$ whereas $Def(6)$ returns $\emptyset$ as no assignment is performed. As a result, by applying the equations from Example 3.2 we get $OUT(6) = \emptyset$ and $IN(6) = \{c\}$. Similarly, the statement in node 5 performs only a read, this yields $Use(5) = \{a\}$ and $Def(5) = \emptyset$. The $OUT(i)$ information is computed using the union of the $IN(s)$ information of each successor s. The node 5 has two successors (i.e., nodes 6 and 2). Given $IN(5) = c$ and $IN(2) = \emptyset$ (as not visited yet) we obtain $OUT(5) = \{c\}$. Consequently, the information $IN(5)$ holds the value $\{a, c\}$. As node 4 has only one successor, $OUT(4) = IN(5) = \{a, c\}$. However, the corresponding statement performs both a read on b and an assignment on a. This gives $Def(4) = \{b\}$ and $Use(4) = \{a\}$. Using these values we obtain $IN(4) = \{b, c\}$. Note that the variable a is not live anymore in $IN(4)$ as it is assigned and not used in the node 4. By following the same reasoning, we determine that $OUT(3) = \{b, c\}$ and $IN(3) = \{b, c\}$. For node 2, we determine that $IN(2) = \{a, c\}$, which is different than the previous value (i.e., $\emptyset$) used to compute the $OUT(5)$ information. This causes a new iteration of the analysis which now has to recompute the value $IN(5)$ using the newly obtained $OUT(2)$ value. The value at $IN(5)$ still evaluates to $\{a, c\}$. A third iteration establishes that the results are stable and that a fixed-point has been reached.

3.5 Intra-procedural and Inter-procedural DFA

A DFA can be either intra-procedural or inter-procedural. In intra-procedural analysis, only the CFG of individual procedures or functions is considered. Call relations are ignored and, therefore, no information is propagated along call and return edges. Depending on the analysis problem, the data-flow values obtained at program points might not always be precise, sometimes even wrong. To capture the effects of function calls, the intra-procedural analysis has either to be combined with an inter-procedural analysis or use approximations that correspond to the side effects of calls.

Inter-procedural data-flow analysis (IDFA) additionally considers the call relations between functions. The analysis can run either by considering an ICFG as previously defined in 2 or by combining the call graph information with functions' CFGs. In this case, additional data-flow equations are constructed modeling function calls and returns [14]. Often these analyses are context-sensitive, i.e., the analyses distinguish between (bounded) chains of functions calls. Such a chain of nested function calls is then called a call string (or context string), which defines a calling context that can be distinguished from other parts of the program calling the same function. Call strings typically have a length limit. The longer the call strings, the higher the ability to distinguish different contexts. Consequently, the analysis results are more precise. Increasing the call string length may also increase the computational complexity and the required memory footprint, since additional data-flow equations are created for each context. A call string length of zero corresponds to a context-insensitive data-flow analysis.

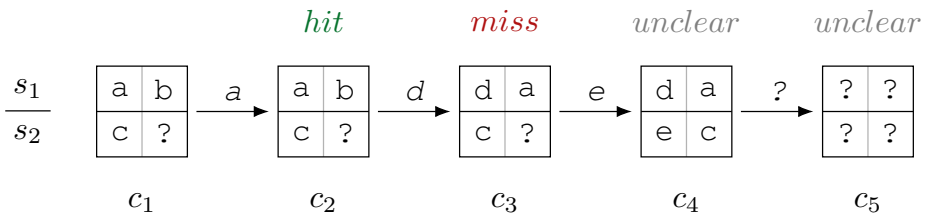


Figure 3.9 – The evolution of the (actual) cache state depending on memory accesses. Assuming 2-way set-associative cache with LRU replacement policy. The memory blocks 'a', 'b' and 'd' map to the cache set s_1, whereas 'c' and 'e' map to s_2.

4 Standard Cache Analyses

One of the most fundamental operations CPUs do is to transfer data from/to the main memory. However, the gap in terms of speed between the CPU and the memory circuits dramatically impacts the execution time of programs. Caches are relatively small and fast memories that attempt to hide long memory latencies. Unfortunately, the small size and the organization of the cache does not allow to completely eliminate data transfers to/from the slow main memory. As a result, variations in the execution time may be introduced due to cache behavior, which need to be accounted for during WCET analysis. In this section, we review some analysis techniques for traditional caches. A brief presentation of caches and their organization has been provided in Chapter 2.Section 2.4.

4.1 Goal and Challenges

In architectures that are free from timing anomalies, it is perfectly safe to ignore the cache and assume every memory access in the program as a cache miss. While this would greatly simplify the WCET analysis, it would also ignore any benefits of caching, causing overly pessimistic WCET estimates. We want to profit from the performance improvement provided by the cache, while having guarantees with respect to the worst-case timing behavior of the program. To achieve this, we need to build a cache analysis that predicts the cache behavior when executing a particular program. The analysis has to determine cases of cache misses/hits that may potentially occur during the execution of each basic block. The information of cache misses/hits can then be used to annotate a program's CFG so that the corresponding costs in CPU cycles can be accounted for during the path analysis phase presented in Section 1.
The outcome of each memory access depends on the cache state, i.e., information on the cache content prior to the access. Therefore, the cache analysis tracks the cache states through execution paths and captures the effects of potentially accessed data elements on the cache state itself. This information is used by the analysis to determine how often hits and misses occur during the execution of the program.

Example 4.1 *Figure 3.9 illustrates an example of a cache state and its evolution assuming a 2-way set-associative cache with LRU replacement policy. This representation of the cache state is* concrete *as it shows the* actual *content of the cache. The initial cache state c_1 holds the cache blocks 'a' and 'b' in the cache set s_1, in addition to 'c' and an undetermined cache block denoted as '?' both in the cache*

set s_2. The memory access to 'a' is a hit since the information on the cache state c_1 shows the presence of the memory block in the cache. On the other hand, the access to 'd' is determined as a miss since the memory block is not in the cache as shown by the cache state c_2. That being said, precisely predicting the behavior of the cache can be challenging in many cases. In order to determine whether a particular access is a hit or miss, the information regarding both the access address and the cache state need to be precise. The access to the memory block 'e' can either be a hit or miss because the cache state c_3 is not precise regarding the content of the cache set s_2. The cache block '?' may hold either 'e' or any other memory block mapped to the same cache set. However, the cache state c_4 can still be determined precisely. When the accessed memory block is not known, a set of possible cache states have to be considered as a result. For instance, if the accessed memory block is mapped to s_1 this causes that memory block to be put in the first position and 'a' to be evicted. However, if the accessed address is mapped to s_2 then 'c' is to be evicted. Therefore, not knowing the address can lead to pessimistic results.

Cache hit/miss classification: We cover in Section 4.2 a DFA-based analysis for LRU cache that attempts to classify memory accesses into two categories: *Always Miss* or *Always Hit*. Cache states are tracked using an *abstract cache state* (ACS) representation as a domain. The ACS allows to summarize and efficiently track the cache states resulting from possible execution paths. The transfer functions are sensitive to `load` and `store` instructions.

Possible issues: A precise cache analysis has to address further issues that may cause the determination of hits and misses unclear. These issues are often related to:

- Control-flow joins: Combining two cache states with different contents at join points can lead to less precision. A simple example is when a memory block is cached in one cache state but not in the other. The resulting cache state is not known whether it holds the memory block anymore. This means that if the memory block is accessed again later it won't be clear if it will be a hit or a miss. Unfeasible path elimination can reduce the number of paths that can never be taken which may enhance the precision of the cache states.

- Access addresses: When the access address is not known, it is not only hard to determine its outcome (hit/miss), it may also not be possible to precisely compute its implication of the cache state (see the transition from c_4 to c_5 in Example 4.1). Practically, instruction cache analyses are less concerned with this issue since addresses are statically known. Data cache analyses, on the other hand, are particularly sensitive to this [123] as load and store instructions may access to multiple locations of the memory. We do not cover the techniques to enhance the precision of address information in this section.

- Execution context: Instructions involving memory accesses may, in some situations, exhibit a different timing behavior depending on the calling context. This is typically the case for loops and functions called from different locations of the program. Ignoring the execution context can provide very pessimistic information to the analysis. While function calls can be traditionally dealt with using context sensitivity on the ICFG, loop structures are very common and need approaches with a good complexity/precision trade-off. We mention some techniques that help with such situations in Section 4.3.

Age	0	1	2	$A-1=3$
Content	$\{a\}$	$\{b,d\}$	$\{\}$	$\{f\}$

Figure 3.10 – Example of an ACS content assuming a cache of associativity 4.

- Task preemptions: In preemptive task models, a preempting task may alter the cache state left by the preempted task. The cache analysis has to determine how preemptions impact the cache behavior when the preempted task is resumed. This aspect is further discussed in Section 4.4.

Most of the work on cache analysis addresses those issues assuming the LRU replacement policy. We review in the following some seminal results.

4.2 Cache Analysis Based On Access Classification

Ferdinand et al. proposed a DFA-based cache analysis [44, 45] which attempts to classify each memory access into one of the two categories: *always hit* (AH) to indicate that an access is a guaranteed hit, or *always miss* (AM) for a guaranteed miss. Intuitively, classifying accesses as such may be done through the investigation of the *actual* cache content, also known as the concrete cache state, on every possible path leading to a program point. However, since the number of paths may grow exponentially depending on the program's CFG, this may cause the analysis to not scale well. Instead, the classifying analysis defines an abstract cache state (ACS) that represents a set of CCSs resulting from possible executions.

> **Abstract Domain:** The ACS associates each potentially accessed memory block with an age that corresponds to its position in the cache. Let MB_l be the set of memory blocks that are mapped to some cache set l. The age some memory block $b \in MB_l$ can take is a natural number within the range $[0, \ldots, A-1]$, where A is the associativity of the cache. The memory block b is assigned the youngest age 0 when b is accessed, whereas the age $A-1$ represents the last position in the cache set l before possible eviction. When b is evicted, it is simply removed from the ACS. The ACS of the cache set l can be expressed as follows:
>
> $$ACS = MB_l \times [0, \ldots, A-1] \tag{3.8}$$
>
> A representation of the ACS is illustrated in Figure 3.10. Memory blocks in MB_l that are visible in the ACS are cached, others are not cached. For a complete representation, the ACS combines the abstract state related to its individual cache sets. This constitutes the domain for the data-flow analyses. The value $\top$ represents the set of all possible CCS. The value $\bot$, on the other hand, represents the empty set of CSS indicating that the program point has not yet been visited by the analysis.

The idea then will consist of tracking, at each program point, the minimum and maximum age for all potentially accessed memory blocks. The minimum age results from an optimistic execution scenario, that is, no other path leading to the program point can result in a younger age. Given this information, it is possible to detect memory accesses where a cache miss will always occur. It suffices to check the

minimum age at the program point prior to a memory access. If the minimum age is larger than $A-1$, then all executions of the memory access are guaranteed misses and can be classified as AM. Conversely, the maximum age represents a pessimistic scenario where a greater age cannot be obtained for any concrete execution. Therefore, if the maximum age is smaller or equal to $A-1$ the access is a hit, and shall thus be classified as AH. The memory access is classified as NC, for *not classified*, if it can neither be classified as AH or AM (i.e., maximum age is $> A-1$ and the minimum age is $< A-1$).
The above classification gives rise to two data-flow analysis problems commonly known as the *may analysis* and the *must analysis*. They are both forward and typically operate on the program's ICFG with context sensitivity.

Must Analysis: The must analysis computes the maximum ages and detects AH accesses. The join operator performs an intersection between the ACSs reaching the join point and simply selects the maximum ages of memory blocks. Assuming c_1 and $c_2 \in ACS$ the join operator $\sqcup_{MUST}$ is given as follows:

$$c_1 \sqcup_{MUST} c_2 = \{(u,a) | \exists (u,a_1) \in c_1, (u,a_2) \in c_2 : a = \max(a_1, a_2)\}$$

The transfer function is only sensitive to memory accessing instructions. Moreover, accesses to non-conflicting memory blocks do not alter the state $c \in ACS$ related to the cache set l. When a memory block u is being accessed, the age of each conflicting block needs to be computed using the update function $update_{MUST}$. A simplified version of the transfer function T_{must} is given as follows:

$$T_{must}(c,i) = \begin{cases} c & , \text{if } MB_l(i) = \emptyset \\ \bigsqcup\limits_{u \in MB_l(i)} update_{MUST}(c,u) & , \text{otherwise} \end{cases}$$

The update is performed using the age_{MUST} function which computes the ages of u and the conflicting memory blocks $\forall v \in MB_l$ based on their relative ages prior to the access (contained in $c \in ACS$). There can only be three different cases: (1) the accessed memory block u is v itself in which case v gets the youngest age 0, (2) u is younger than v which has no implication on the age of v, and (3) u is older than v which requires the age of v to be incremented as a result of moving u to the first position. Moreover, the memory blocks with an age $> s$ are evicted from c.

$$update_{MUST}(c,u) = \{(v,a) | v \in MB_l : a = age_{MUST}(c,u,v)\}$$

$$age_{MUST}(c,u,v) = \begin{cases} 0 & , \text{if } u = v \\ age(c,v) & , \text{if } age(c,v) \geq age(c,u) \\ age(c,v)+1 & , \text{if } age(c,v) < age(c,u) \end{cases}$$

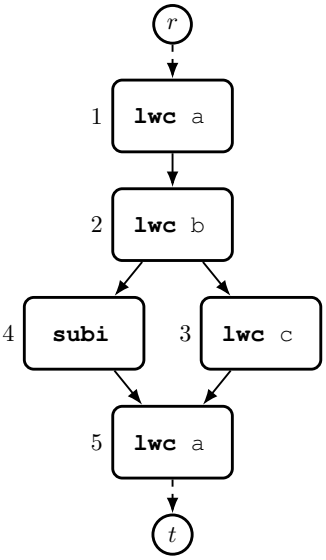
The analysis is initialized by assigning $\top$ to the program entry point (if the initial cache state is not known), and $\bot$ to all others.

	May	
Point	**IN**	**OUT**
1	$\{\}$	$\{(a,0)\}$
2	$\{(a,0)\}$	$\{(b,0),(a,1)\}$
3	$\{(b,0),(a,1)\}$	$\{(c,0),(b,1),(a,2)\}$
4	$\{(b,0),(a,1)\}$	$\{(b,0),(a,1)\}$
5	$\{(b,0),(c,0),(a,1)\}$	$\{(a,0),(b,1),(c,1)\}$

(a) May analysis results at program points.

	Must	
Point	**IN**	**OUT**
1	$\{\}$	$\{(a,0)\}$
2	$\{(a,0)\}$	$\{(b,0),(a,1)\}$
3	$\{(b,0),(a,1)\}$	$\{(c,0),(b,1),(a,2)\}$
4	$\{(b,0),(a,1)\}$	$\{(b,0),(a,1)\}$
5	$\{(b,1),(a,2)\}$	$\{(a,0),(b,2)\}$

(b) Must analysis results at program points.



(c) Program CFG with memory accesses.

Figure 3.11 – Example of May and Must analyses.

May Analysis: The may analysis defines a similar transfer function and join operator to compute the minimum ages of memory blocks. The join operator selects the minimum ages of memory blocks at join points.

$$c_1 \sqcup_{MAY} c_2 = \{(u,a) | \exists (u,a_1) \in c_1, (u,a_2) \in c_2 : a = \min(a_1,a_2)\}$$

Additionally, the may update function $update_{MAY}$ relies on the age_{MAY} function, defined here below, to compute the ages of conflicting memory blocks (i.e., those mapped to the same cache set) due to a memory access. Also, memory block with ages $> A-1$ are evicted from the cache state.

$$age_{MAY}(c,u,v) = \begin{cases} 0 & \text{, if } u = v \\ age(c,v) & \text{, if } age(c,v) > age(c,u) \\ age(c,v)+1 & \text{, if } age(c,v) \leq age(c,u) \end{cases}$$

Example 4.2 *Consider the CFG in Figure 3.11c. In this example, we perform May and Must cache analyses as shown in Table 3.11a and Table 3.11b respectively. Both analyses start with an empty abstract cache state. In the node* 1*, the memory block 'a' is referenced. By applying the* T_{may} *transfer function to the* $IN(1)$ *information, we obtain* $OUT(1) = \{(a,0)\}$ *which associates the age* 0 *to the memory block 'a'. The information* $OUT(1)$ *is passed to* $IN(2)$ *on which we apply the* T_{may} *transfer function. The memory block 'b' is referenced in node* 2*, this yields the value* $OUT(2) = \{(b,0),(a,1)\}$ *as 'b' was not present in the cache state. Accessing 'b' increases the age of 'a' by* 1*. The value* $OUT(2)$ *is then propagated to its two successors (i.e., nodes*

3 *and* 4*). By applying the same reasoning, we obtain the information* $OUT(3) = \{(c,0),(b,1),(a,2)\}$ *as '*c*' is referenced in node* 3*. The* $OUT(4)$ *information remains the same as no memory access is performed at node* 4*. At control-flow joins, the analysis performs the* $\sqcup_{MAY}$ *operator which selects the minimum ages of both cache states. This yields the information* $IN(5) = \{(a,0),(b,1),(c,1)\}$*. The memory block '*a*' is again referenced at node* 5*. However, this does not change the cache state as the age of '*a*' was* 0 *at* $IN(5)$*. The must analysis results in similar information except for the* $IN(5)$ *information where the join occurs and* $OUT(5)$*. This is because the* $\sqcup_{MUST}$ *selects the maximum ages of the referenced caches blocks in the cache state. This, therefore, yields* $IN(5) = \{(b,1),(a,2)\}$*. The value* $OUT(5) = \{(a,0),(b,1)\}$ *is obtained by updating the age of '*a*' to* 0 *as it is referenced in node* 5*.*

Note that the must analysis is crucial for WCET analysis as it allows to detect accesses with AH classification. In the absence of timing anomalies, the may analysis is not mandatory for the WCET analysis process. However, it is used to obtain precise BCET bounds, and perform related optimizations. Memory access classifications can be easily translated into timing costs. The AH classification usually requires no costs, whereas accesses classified as AM has to account for CPU cycles spent on the interaction with the main memory. Since the WCET analysis has to be conservative, an NC classification counts as an AM (always assuming the absence of timing anomalies). In that regard, the cost of uncertainties is considered as high as a miss.

4.3 Improving Precision in Loops

Many memory accesses can still be classified as NC. Often, the may and must analyses do not provide conclusive results in situations where executions of the same memory access may exhibit different timing behaviors. A typical case of this includes loop structures. Intuitively, the first iteration in a loop allows to load accessed memory blocks into the cache, while the next ones profit from the hits. The problem with the loop structure is the must join operator $\sqcup_{MUST}$ that conservatively merges ACS information of edges reaching the loop header, i.e., join point between the loop's back edge and the edge reaching the loop from outside. When the merge occurs, the intersection removes any cache block that were not accessed before the loop. This makes any potential access to memory blocks from within the loop forgotten. Another possible side effect is the eviction of the cache block prior to the loop. There exist essentially two approaches to handle this situation:

Virtual Inlining and Virtual Unrolling (VIVU): This technique [81] simply consists of peeling the first iterations off the loop and applying analyses just as before. Assuming the ACS is empty when entering the loop, the first access to the memory blocks during the first iterations will result in AM classification. However, the resulting ACS reaching the loop header contains updated age information that accounts for the performed accesses with, potentially, a reduced maximum age. When propagated to the loop with the remaining iterations, the updated cache state might help classifying those accesses as AH. The unrolling could be applied as many times to classify as much accesses as possible. In this case, each unrolled iteration represents a context.

Persistence Analysis: The idea of this approach is to bound the number of cache misses to occur within some scope, i.e., a region of the program such as a loop or a function. A block is said to be *persistent* within that scope if it remains in the cache once loaded. Therefore, as long as the execution stays in the scope, at most one cache miss needs to be considered for each referenced memory block. There exist two main variants of persistence analysis. The first one is based on DFA and was first introduced by Ferdinand et al. [45], and later corrected by Huynh et al. [61]. This variant is very similar to the classifying analysis except with a slight modification of the ACS; The range of considered ages is extended to $[0, \ldots, A]$ such that evicted memory blocks take the maximum age A. This is vital in order to distinguish misses due to eviction from those due to the first access. Using the transfer function and the join operator of the access classification analysis, the determination of persistent memory blocks is simple. At the end of a scope, the memory blocks with the age A are considered non-persistent. For loops, the persistent ACS will prevent the join operator from forgetting any potential access to memory blocks from within. Another approach, commonly called scope-based analysis, has been proposed by Huynh et al. [61]. Instead of tracking ages, the analysis collects, for each referenced memory block, the conflicting referenced blocks in the loop. The memory block is considered persistent if the number of conflicting memory blocks fits into the cache set.
Note that the two approaches are not comparable. This means that there exist some situations where the VIVU approach will be able to classify memory accesses that the persistence analysis variants will not be able to, and vise-versa. Intuitively, the advantage of persistence analysis is that it may yield a better precision/complexity trade-off compared to VIVU. In complex loop structures involving nested loops, applying VIVU is likely to create many execution contexts leading to an accelerating increase of complexity.

4.4 Preemption Costs

So far, the analyses we mentioned assume a straight execution of programs. In preemptive scheduling, a running task can be preempted by a higher priority task. While preemptions allow, in general, a better system schedulability, they may induce some time penalties that need to be accounted for. The majority of those penalties are caused by additional cache misses experienced by the preempted task, due to memory accesses in the preempting task. The additional execution time related to those cache misses is commonly known as the cache-related preemption delay (CRPD).
The computation of the CRPD in conventional caches is based on the notion of Useful Cache Blocks (UCBs). Some memory block b is called a UCB at program point p if, (1) b may be cached at p, and (2) b may be reused in another program point q that may be reached from p without being evicted. Liu et al. [70] use UCBs to tighten the WCET estimation. First, the number of UCBs at all execution points are calculated using data-flow analysis. Then for all tasks, a preemption cost table is constructed that defines the preemption cost at each point, which depends on the number of UCBs and on the worst-case visit count of each point. Based on this table and using integer linear programming, the worst-case preemption delay of a task is calculated. the notion of definitely-cached UCB (DC-UCB) was later introduced in [16] to detect cache misses that are included in the CRPD bound as well as in the

WCET bound. It was shown that this approach gives safe CRPDs, when combined with an upper bound of the WCET. The results show significant improvements over the original approach based on UCBs [70].
Also part of the preemption cost is the time spent on context switch operations, i.e., saving/restoring internal CPU registers. Prior work investigated the use of hardware support to optimize context switching. Tune et al. [122] and Mische et al. [85] introduce hardware support to optimize the context switching in real-time systems, but at the register file level. For instance, Tune et al. [122] use dedicated hardware for scheduling threads in an SMT-based processor. The hardware scheduler is also able to save/restore the registers of a thread to a special on-chip memory, the *Thread Control Block* (TCB). The TCB requires two separate ports, in order to eliminate any interference from parallel accesses to the TCB from the running program and the hardware scheduler. Other work, such as [114], optimize the average cost of context switching, but due to lacking predictability these methods are unsuited for real-time systems.

5 Stack Cache Analyses

Now we review existing approaches to analyze the timing behavior of the stack cache. The stack cache is dedicated to stack data, and is implemented as a simple ring buffer (a detailed description of its behavior and structure is provided in Chapter 2). From a timing analysis standpoint, the stack cache is intended to reduce execution time variations induced by the conventional data cache, while still being simpler to analyze. A considerable advantage for the stack cache is that it does not rely on addresses. Additionally, all its accesses are guaranteed hits and data transfers are performed only at specific program locations controlled by the compiler. Only two stack control instructions that may initiate memory transfers depending on the cache occupancy (i.e., $Occ = \texttt{MT} - \texttt{ST}$):

`sres k`: Issues a spilling of cache blocks if the resulting occupancy due to the reserve exceeds the stack cache size (i.e., $Occ + k > |SC|$).

`sens k`: Fills the stack cache with cache blocks if the occupancy at the ensure instruction is strictly less that the function frame size (i.e., $Occ < k$).

The worst-case (timing) behavior of these instructions only depends on the worst-case spilling and filling of `sres` and `sens` respectively, which can be bounded by computing the maximum and minimum cache occupancy [63].

Example 5.1 *Figure 3.12 shows the worst-case spilling or filling to occur given a sequence of stack cache control instructions. We assume that the minimum and maximum occupancy at cache state* c_1 *is equal to* 1 *and* 3 *respectively. The* `sres` 2 *reserves* 2 *cache blocks, which may cause a spilling if* $Occ + k > |SC|$*. The (concrete) occupancy level is comprised in the range defined by the occupancy bounds at* c_1*. Therefore, by considering the maximum occupancy we predict that the worst-case case spilling to occur is equal to* 1 *cache block. The* `sfree` 2 *reduces by* 2 *both the minimum and maximum occupancy but does not induce any memory transfers. The resulting minimum and maximum occupancy at* c_3 *is respectively* 1 *and* 2*. The* `sens` 1 *ensures that the occupancy level is at least equal to* 1*. No transfers are issued*

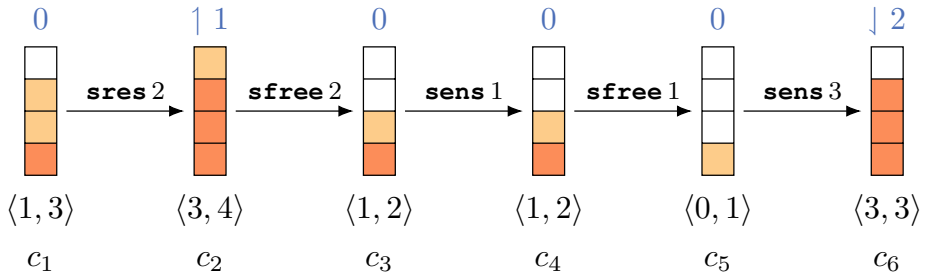


Figure 3.12 – Worst-case spilling/filling depending on max/min occupancy. The number N above the cache state indicates the number of cache blocks filled ↓, or spilled ↑. The pair $\langle m, n \rangle$ below the cache state indicates the current min. occupancy m and the max. occupancy n. (■) blocks represent max. occupancy and (■) blocks represent min. occupancy.

since the minimum occupancy is equal to 1. *The* `sfree` 1 *yields a minimum and maximum occupancy at* c_5 *of* 0 *and* 1 *respectively. The* `sens` 3 *issues a filling of* 3 *cache blocks as the minimum occupancy was equal to* 0.

The stack cache is also concerned with some of the issues described in Section 4.1 (i.e., control-flow joins, execution context, and task preemption). However, in comparison to analyses on conventional caches, the information to track for the stack cache analysis is much simpler. In conventional caches, the analysis needs to track the minimum and maximum ages for all memory blocks referenced in the program. The stack cache analysis merely tracks minimum and maximum occupancy bounds.
Two stack cache timing analyses have been proposed so far, both rely on different approaches to track the occupancy bounds.

Inter-procedural DFA-based Analysis (IDFA): Tracks occupancy bounds exclusively based on an inter-procedural DFA performed on the program's ICFG [118].

Standard Stack Cache Analysis (SCA): Breaks down the problem into smaller steps and combines information from both intra-procedural DFA-based analyses and analyses on the call graph [63].

This section briefly presents these two approaches, more details with examples are provided in Chapter 5.

5.1 Inter-procedural DFA-based Analysis

The domain of the IDFA-based approach are positive integer values in $\mathcal{D} = \{0, \ldots, |SC|\}$, where $|SC|$ represents the stack cache's size. Since both, the minimum and the maximum occupancy are needed, two analysis problems have to be defined. We mention here only the maximum occupancy analysis (for full description refer to Chapter 5). The analysis starts at the program entry, where the occupancy is assumed to be 0. It then propagates occupancy values along all execution paths of the program, while considering the effect of the instructions along the path. Only the stack control instructions (see Section 2.4 of Chapter 2) can have an impact: (1) `sres` instructions increase occupancy by their argument k, (2) `sens` instructions make sure that the occupancy is larger than k, and (3) `sfree` instructions reduce

the occupancy by k. The resulting data-flow equations for an instruction i are given below:

$$\mathrm{OUT}_{Occ}(i) = \begin{cases} \min(\mathrm{IN}_{Occ}(i) + k, |SC|) & \text{if } i = \texttt{sres } k \\ \max(\mathrm{IN}_{Occ}(i), k) & \text{if } i = \texttt{sens } k \\ \max(0, \mathrm{IN}_{Occ}(i) - k) & \text{if } i = \texttt{sfree } k \\ \mathrm{IN}_{Occ}(i) & \text{otherwise} \end{cases}$$

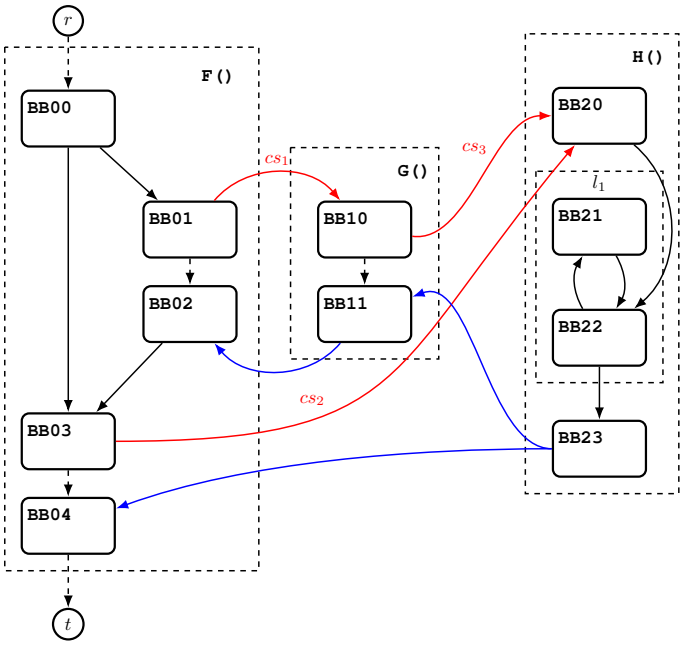
The occupancy right before an instruction (due to control-flow joins) is derived by taking the maximum occupancy from any of its predecessors ($Preds$), except for the program's entry. In the case of inter-procedurual analysis, predecessors can also be calls or returns from other functions:

$$\mathrm{IN}_{Occ}(i) = \begin{cases} 0 & \text{if } i = \texttt{entry} \\ \max_{s \in Preds(i)}(\mathrm{OUT}_{Occ}(s)) & \text{otherwise} \end{cases}$$

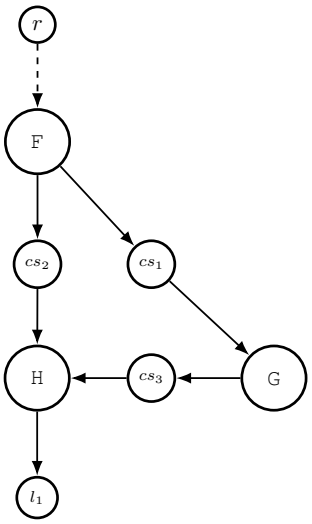
The analysis is initialized by assigning the value 0 to the program entry point (i.e., $\mathrm{IN}_{Occ}(entry)$) and also to all the $\mathrm{OUT}_{Occ}(i)$ values in the ICFG. Context sensitivity can easily be ensured by adding context information to the data-flow equations of the respective instructions.
This model is also implemented in Absint's aiT timing analyzer tool [118], however, we find the minimum occupancy analysis to be incorrect (see Chapter 5).

5.2 Standard Stack Cache Analysis

The stack cache analysis (SCA) [63] relies on similar DFA analyses. However, instead of a single, large inter-procedural DFA, several smaller function-local analyses are used. The impact of other functions at function calls in these DFAs are modeled through minimum and maximum *displacement* values, which represent the minimum/maximum amount of data potentially evicted from the the stack cache during a function call. Displacement values are computed by performing shortest/longest path search on a program's call graph whose weights represent the reserved stack space k. Complex context-sensitive analysis can thus be avoided.
The analysis is based on the observation that the occupancy at any instruction within a function can be computed from the occupancy at the function's entry and the displacement of all the potential function calls on any path leading to the particular instruction. The minimum occupancy thus can be computed by considering the initial minimum occupancy and the maximum displacement. Likewise, the minimum displacement allows to derive the maximum occupancy.
The program is thus analyzed in several steps. First, the minimum and maximum displacement of each function is computed using longest/shortest path searches on a weighted call graph. Next, local DFAs are performed to compute local lower and local upper bounds on the minimum and maximum occupancy within each function assuming a stack cache that is full at function entry. Finally, the concrete occupancy bounds are computed for each function considering the occupancy bounds at its respective callers and the previously computed local occupancy bounds. The final phase can deliver fully context-sensitive information, if so desired.
This analysis was implemented and validated against run-time measurements in [12].



(a) ICFG of a program consisting of functions `F()`, `G()` and `H()`.



(b) Scope graph of the program.

Figure 3.13 – Example of a program and its corresponding scope graph.

6 Method Cache Analysis

Huber et al. presented a scope-based method cache analysis [59], which is based on persistence analysis from standard caches [61]. The goal of the analysis is to determine, for each code block b, a set of conflict-free scopes (program portions) $\mathcal{S}(b)$ such that: (1) every access to b is performed in at least one scope $S \in \mathcal{S}(b)$ and (2) all accesses to b within a scope S exhibit at most one cache miss. The number of cache misses of a memory block b can be bounded by the sum of the execution frequencies of all the scopes in $\mathcal{S}(b)$. The process of finding the conflict-free scopes leaves out some essential details (e.g., formal definitions, how recursions are handled, examples). We try, nevertheless, to summarize our understanding of the approach and the steps involved.

The analysis starts by building a *scope graph*. The scope graph G is an acyclic and hierarchical flow-graph of the program, whose nodes can be either a function, a call site, or a loop. Intuitively, each node represents a set of instruction sequences, each of which, represents one possible execution of the program fragment covered by the corresponding scope. Moreover, the set of instruction sequences represented by a child node is included in one of instruction sequences represented by its parent. In that regard, the root node represents all the possible executions of the program. A leaf node represents the set of the instruction sequences resulting from possible executions of the corresponding program portion (i.e., function or loop).

Then, for each scope node N of G, an *access graph* is constructed and associated with memory accesses that occur within the scope. The access graph is used to model memory accesses that are performed within the scope node N. Additionally, this representation later is used to detect potential conflicts inside a scope.

Example 6.1 *Consider the ICFG of a program shown in Figure 3.13a. Figure 3.13b*

shows the resulting scope graph of the program. The nodes represent (1) the three functions of the program (i.e., `F()`, `G()` *and* `H()`*), (2) call sites (cs_1, cs_2, and cs_3) as well as (3) a loop located in the function* `H()` *(i.e., l_1). The function* `F()` *is the program entry function, therefore, it is a child of the root node r that represents all possible executions of the program. The function* `F()` *calls* `G()` *and* `H()` *at dedicated call sites. Consequently, the scope A has two children cs_1 and cs_2 each of which has a child that corresponds to the callee (G and H respectively). Similarly, the function* `G()` *calls* `H()` *in the call site cs_3. The node H is thus a child of both cs_2 and cs_3 nodes. Finally, the loop l_1 is located in function* `H()`*. The corresponding scope node of l_1 is then a child of node H.*

The first scopes directly provided by the scope nodes might be conflicting. The conflict detection is performed by traversing the scope graph G from the leaf-nodes upwards. For each scope node N check whether N is conflict-free with respect to all code blocks accessed within the scope. For the method cache, a conflict-free scope is defined as follows:

Definition 17 *(Conflict-Free Scope) A scope node N is conflict-free with respect to all the code blocks accessed within the scope if (1) the number of code blocks fit into the cache's associativity and (2) the total size of code blocks does not exceed the method cache size.*

In the case a scope is conflicting, it is decomposed into a set of conflict-free single-entry regions. Each of the regions becomes a scope satisfying the conflict-free conditions. For any code block b in a scope S, the scope is stored in $\mathcal{S}(b)$. To achieve this, the access graph is traversed in topological order while collecting visited code blocks in a region of the access graph of function or loop representing a *seed* scope. If including accessed code blocks in the seed scope is conflicting with respect to the aforementioned conditions, then the seed scope is split and a new seed scope is explored. The seed scopes continue to grow according to the conditions above until all nodes of the access graph are visited. Note that function calls imply that all code blocks of a function are added to a seed scope, i.e., the function's scope is merged into the seed scope.
Once the set of conflict-free scopes is determined, they can easily be integrated in IPET to bound the maximum number of cache misses. Accounting for cache miss costs is done similarly to the classical persistence analysis. Each conflict-free scope of a code block b is associated with a variable representing the number of cache misses, which is bounded by the scope execution count and the code block's execution count.
With respect to the description above, we notice the following remarks. First, it is not clear how the analysis handles recursive functions. In its current version, the analysis does not allow to merge scopes of a called function into multiple seed scopes. Therefore, each call of the same function may cause unnecessary cache misses to be accounted for. Moreover, the evaluation performed in the paper does not allow to draw a conclusive picture regarding the effectiveness of the approach. If the cache analysis itself was run on programs' CFGs, the path analysis through IPET was fed by the execution path observed during an execution of the programs. This, in fact, does not allow to account for the effects the analysis may have on the determination of the WCEP.

7 WCET Analysis with Predication

Predication is a challenge for tight WCET analysis. A predicated instruction is executed only if the predicate evaluates to true. This has an implication on the hardware state of almost all the CPU internal components. WCET analysis tools such as aiT or Otawa ignore the predicates. They are not considered in cache analysis, pipeline analysis, nor VRA for address analysis. This may lead to overly pessimistic results as each analysis needs to consider the worst-case of both possibilities. How this is then handled in other applications?
Predicated computer architectures received considerable attention in the 1990s with the development of VLIW and EPIC architectures that tried to exploit instruction-level parallelism through static compilation techniques rather than hardware [47]. Various compiler optimizations have been developed targeting the transformation of regular code into predicated code [93, 117] and the optimization of predicated code [39, 113]. A common problem for these optimizations is the need to understand the relations between predicates [62, 112], i.e., which predicates can be live at the same time. The underlying machine code may evolve through optimizations in the compiler, which might require these analyses to be performed multiple times. The analyses thus need to be fast and only reason about predicate relations that can be deduced from the *structural* relations between predicates. Information on the actual conditions, e.g., the tested values, are not captured. The work might help to reduce some of the overhead induced by useless code duplications. The techniques are, in addition, concerned with the analysis of the predicates themselves and do not allow to obtain other analysis results.
Hu [58] addressed this issue by refining the semantics of predicated code and redefining several typical concepts used in compilers/static analyzers (e.g., dominance and data dependencies). She also showed how predicate-aware data-flow analysis can be realized using the example of reaching definitions. Similar techniques could be applied to many other analysis techniques, including those used in typical WCET analyzers. However, this would require a considerable engineering effort in order to adapt all existing analyses accordingly.

8 Existing Static WCET Analysis Tools

Many commercial and research tools for WCET analysis have been proposed during the last couple of decades. In the context of this work we only focus on static program analysis tools, which provide safe WCET bounds. In this section, we present some of them.

AbsInt's aiT: is a well-established commercial tool [43], which, for instance, has been used during the certification of the latest Airbus airplanes [115]. AbsInt's tool provides all ingredients of a typical WCET analysis: CFG reconstruction, value-range analysis, low-level analysis (pipeline, caches), as well as a path analysis through IPET. For caches, aiT implements standard cache analysis for LRU but also for non-LRU replacement policies such as Pseudo-LRU and Pseudo-Round Robin. Analysis of an LRU cache is based on both a classifying and a persistence analysis. The tool also supports a CRPD analysis based on the notion of UCBs in order to bound the number of cache misses due to preemption. For the stack cache, an

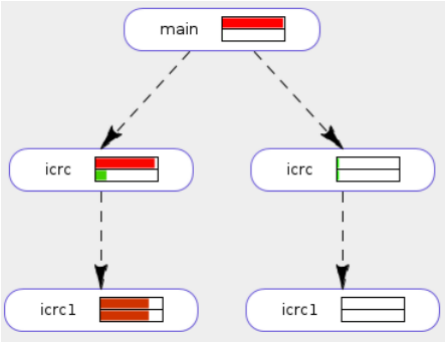


Figure 3.14 – Visual feedback generated by the OTAWA analysis tool.

occupancy analysis has been proposed in [118] (see Section 5). We find this analysis to be incorrect and propose a fix in Chapter 5. Cache analysis results are then used in combination with the pipeline analysis that assigns an execution time bound to each basic block of the CFG depending on the context. aiT also uses the *prediction file* for ILP analysis. The technique consists of building a global state graph for the possible hardware states and solve the ILP by finding the worst-case path through the graph. This allows to exclude paths that are architecturally-infeasible yielding more precise WCET bounds. However, the graph to analyze can be extremely large depending on the architecture. The Absint tool provides visual feedback to engineers showing the WCET bound and the WCEP. For each function of the program it is indicated whether the function is on the WCEP or not. Additionally, the local and cumulative contribution of the function (and its basic blocks) to the WCET bound can be visualized in the form of a graph similar to that of Figure 3.14. The WCEP can also be exported holding the disclosed information regarding the timing contribution.

Otawa: is a research tool [18] for WCET analysis that is developed and maintained by the TRACES team of the University of Toulouse, France. Otawa is essentially a generic framework consisting of a set of C++ classes allowing the implementation of static analyses related to WCET analysis. The tool is made in the intention of making the analyses independent from the hardware and its ISA. Besides this aspect, Otawa employs mainly the same analysis flow and comes with similar features than those of AbsInt's tool. Control-flow information can be obtained either through different flow analyses or by manually providing them in a *flow facts* file. Moreover, the oRange tool [22] can be used complementarily with Otawa for high-level analysis in order to determine flow facts and loop bounds. For the low-level analysis, Otawa provides the classical LRU cache analysis (both for data and instructions), a pipeline analysis, as well as a dynamic branch predictor analysis. Otawa can provide a visual feedback regarding the contribution of functions to the WCET (see Figure 3.14). Both tools additionally may provide cycle counts and execution frequencies of code fragments (basic blocks) along the WCEP. Otawa supports a range of computer architectures based, for instance, on ARM and Power PC cores. To the best of our knowledge, no support of a time-predictable processor is yet proposed, therefore no predictable cache analyses have been implemented yet in Otawa.

Chronos: is an open source tool [74] for WCET analysis from the National University of Singapore. Again, the general analysis flow is very similar to that

of Otawa and aiT. The analyzer operates on the program binary and first tries to extract its CFG. Then, a control-flow analysis is performed in order to collect flow information and compute loop bounds. In case the analysis is unable to find such information, the user can manually provide loop bounds through annotations, or designate infeasible paths. The tool then performs the processor behavior analysis using an abstract configurable model. Chronos implements the classical instruction cache analysis, an out-of-order pipeline analysis, and a dynamic branch prediction analysis. Due to the potential presence of timing anomalies, the tool provides means to handle the complexity through a technique that avoids enumerating all possible schedule cases [75]. Once execution time bounds are determined for each basic block, an ILP problem is formulated using the flow information provided earlier. A solver then is invoked in order to determine the WCEP corresponding to the WCET.

Heptane: is a WCET tool [102] developed and maintained by the ALF research team from the IRISA/INRIA institutes in Rennes. Heptane consists of two main parts. First, the *Extract* part constructs the program CFG from the binary generated from the compilation process. The provided CFG holds basic information such as loop bounds that are provided by the user as well as instruction addresses. The *Analysis* part is where the WCET analysis actually takes place to provide the WCET bound. This consists of both high-level and low-level analyses. The high-level analysis mainly consists of the IPET-based technique to formulate the WCET calculation problem. Heptane does not implement any automatic loop bound analysis which requires the user to provide all the bounds manually. The loop bounds are provided in the source code using predefined macros. The low-level analysis implements address analysis based on VRA, instruction and data cache analyses, as well as a pipeline analysis. In particular, the cache analysis is based on must/may analyses and persistence analysis. Moreover, multi-level cache analysis is also supported based on the approach described in [54, 72]. The analysis results are gradually annotated into a CFG and later used in the pipeline analysis to take into account possible timing accidents. The pipeline analysis assumes an in-order single-issue pipeline free of timing anomalies. A graphical representation of the CFG can be exported highlighting the WCEP along with some statistics. Heptane supports a limited number of targets (only ARM and MIPS-based).

SWEET: is a Swedish research tool [9] developed and maintained by the Malardalen University. The tool mainly focuses on deriving flow-fact information such as loop bounds and unfeasible paths. As such, Sweet implements various analysis techniques to compute flow information. In particular, the tool implements the *abstract execution* (AE) approach which is a form of symbolic execution based on abstract interpretation. The technique first computes a range of values for variables using abstract interpretation, then *executes* the program using the derived ranges. AE can be used to compute loop bounds and unfeasible paths generally providing more precise results than traditional approaches. Other analysis approaches include program slicing and conventional value analysis. SWEET performs analyses on programs represented in the ALF language which can be generated from `C` and other programming languages.

Platin: is a toolkit [96] for compiler and WCET analysis integration introduced as part of the T-CREST project. Platin plays different roles in the process of

deriving WCET estimates. The tool interacts with the LLVM compiler to extract both platform dependent and independent information that is relevant for the WCET analysis. The information is exported in the PML (Program Metainfo Language) format and includes, for instance, program structural information (e.g., function, basic blocks, instructions), flow-facts, memory addresses, and the results of some timing analyses implemented in the back-end (e.g., stack cache). Moreover, platin ensures that the obtained platform independent information corresponds to the machine code [60]. Platin can optionally invoke the SWEET tool to perform automatic high-level analysis and extract more precise flow-facts information. Flow-facts can, furthermore, be extracted from simulation traces produced by pasim (Patmos simulator). The collected information can then be exported to either aiT or Otawa tools to perform low-level analysis and formulate an IPET-based WCET problem. Platin implements also a built-in WCET analysis tool supporting an IPET-based WCET analysis as well as some timing analyses that are not supported by other WCET analysis tools (e.g., method cache).

9 Conclusion

Throughout this chapter, we first presented the static WCET analysis work flow as well as basic concepts for static timing analysis. Then, we covered seminal results with a focus on timing analyses for time-predictable components/features. We notice the following limitations and provide our answers:

- Existing WCET tools do not handle predicated execution. In order to be conservative, both high-level and low-level analyses take into account execution possibilities and consider the worst-case. This introduces pessimism that may impact the precision of the calculated WCET bound. We introduce in Chapter 4 an approach that allows to recover the hidden control-flow in the CFG and show how it integrates into our WCET tool for Patmos called Odyssey. This occurs early in the WCET analysis work flow, particularly, in the CFG reconstruction phase.
- So far two analyses have been proposed, however, no prior work has compared the precision of occupancy bounds. Moreover, the IDFA-based analysis is found to be incorrect. We provide in Chapter 5 a correction for the IDFA-based analysis and compare the precision of occupancy bounds provided by both analyses. We furthermore investigate situations where one analysis outperforms the other in terms of precision.
- Stack cache analyses assume a straight execution of the program and thus do not support preemptive task systems. We extend in Chapter 6 the standard stack cache analysis to account for and optimize timing penalties due to preemption. Our analysis provides a bound of preemption costs that would be induced at each program point. The information provided by our analysis of preemption costs is rich. A natural question raises as to the usability of the information by the scheduler to optimize the preemption costs. As a response, we introduce in Chapter 7 preemption mechanisms that support different preemption approaches while being cheap and simple to implement.
- Timing analysis in conventional caches might be challenging typically due to the hit/miss detection that relies on addresses. Hardware optimizations for

conventional caches (such as prefetching) introduce further timing variations and hardware states that might be overly complex to track. The stack cache on the other hand does not rely on addresses and all its accesses are guaranteed hits. We thus explore in Chapter 8 prefetching mechanisms for the stack cache and investigate their impact on the WCET in multicore configurations.

CHAPTER 4

Analysis of Predicated Programs in Odyssey – a Fully-Integrated WCET Analysis Tool

Predication is a combination of software and hardware techniques that aim to reduce branches and their potentially high penalties. However, a downside of predicated instructions is the precise worst-case execution time (WCET) analysis of programs making use of them. Predicated memory accesses, for instance, may or may not have an impact on the processor's cache and thus need to be considered by the cache analysis. Predication potentially has an impact on all analysis phases of a WCET analysis tool. We thus explore a preprocessing step that explicitly unfolds the control-flow graph, which allows us to apply standard analyses that are themselves not aware of predication. The approach is implemented and integrated into Odyssey, our WCET analysis tool for Patmos. Odyssey is open and fully integrated into the LLVM compiler framework and thus has a complete view of the program. In this chapter we cover predicated execution and how it can be handled in a WCET analysis flow. The first section explains how predication works and why it is important for predictability. Section 2 presents our WCET tool Odyssey and supported features/analyses. A motivating example of our approach is presented in Section 3. Section 4 describes our approach consisting of recovering the hidden control-flow of predicated code. Then we present the results from experiments in Section 5 and the conclusion in Section 6.

1 Outline

Predicated instructions can be problematic during WCET analysis. Side-effects of predicated instructions need to be analyzed, which depend on the runtime value of the instruction's predicate. This may have an impact on many analysis steps, including value range analysis, loop bounds analysis, infeasible path analysis, but also the cache and pipeline analyses. Predicated memory accesses, for instance, may or may not have an impact on the processor's cache, depending on the predicate. The simplest solution for the analysis would be to ignore predicates and conservatively consider the effect of both cases. This may result in very conservative results, since the implicit information available in the program's original control-flow before the

elimination of branches is entirely lost. The analysis could also be extended to be aware of predicates. Note, however, that this may require changes to virtually all analysis steps in a WCET analyzer and may thus require a considerable engineering effort.
We, thus, explore a much simpler solution that consists in recovering the (hidden) control-flow. Instructions that *define* (set) a predicate are handled similar to branch instructions and lead to a control-flow split. The succeeding instructions are then *duplicated*, once assuming that the predicate evaluates to `true` and once assuming that the predicate value is `false`. Subsequent instructions that are predicated with that predicate are now trivial to handle: an instruction either always corresponds to a `nop` or always corresponds to the regular unpredicated instruction. We implement this approach in our WCET analysis tool Odyssey. Based on our evaluations, we show that the unfolding does not result in excessive code duplication and yields a moderate code size increase.

2 Odyssey: a Fully-Integrated WCET Analysis Tool for Patmos

A fundamental aspect of this thesis consists of treating the contributing components to performance/predictability as a *white box*. The WCET analysis tool, in particular, has to be open and flexible to integrate timing analyses for time-predictability features. Moreover, it has to be *close* to the compiler so that critical information regarding the program and its control-flow is directly accessible. Prior work [96] in the T-CREST project proposed a method for the integration of aiT – the commercial analysis tool of AbsInt – and the LLVM compiler for Patmos. The process relies on a

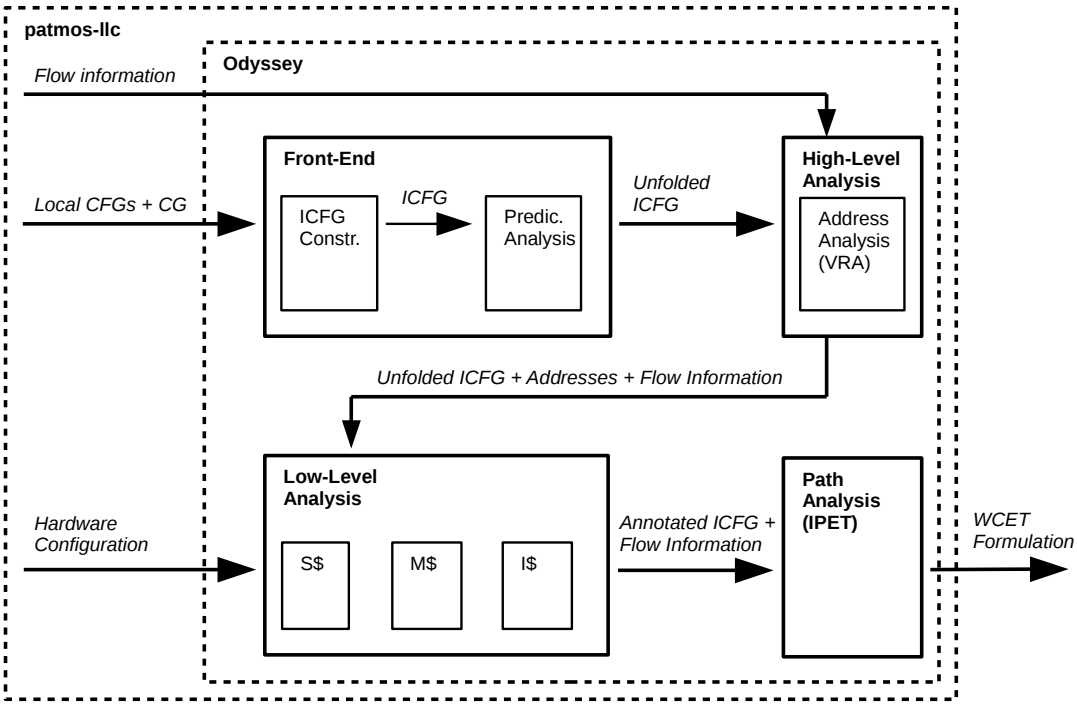


Figure 4.1 – Overview of the Odyssey WCET analysis tool.

set of intermediate tools and description languages that play the role of an interface between the compiler and the commercial WCET tool. We advocate for an even closer and simpler integration of the WCET tool and the compiler.
Our WCET analysis tool, Odyssey, is open and fully integrated into the LLVM compiler framework (a description of the compilation flow is provided in Chapter 2.Section 3). More precisely, the tool is located in the `patmos-llc` back-end and consists of an LLVM *ModulePass* performed right before code emission. The pass uses the entire program as a unit, providing a *Control-Flow Graph* (CFG) for each function, as well as machine code inside basic blocks. Odyssey has, therefore, a complete and accurate representation of the program to be executed in the Patmos platform. Figure 4.1 illustrates different analyses that we have implemented so far.

Front-End: Although LLVM provides lots of structural information about the program, it is convenient to represent that information in the form of an ICFG on which subsequent analyses will operate. The process of building the ICFG supports context sensitivity and takes into consideration predicated execution:

ICFG Construction
: Odyssey starts off by first building a complete inter-procedural CFG of the program. This is done by identifying the program's entry point using the call graph and LLVM's function local CFGs. Instruction bundles of the input basic block are successively processed until a control-flow instruction is reached. This leads to the creation of an ICFG node for the already visited instructions and triggers the processing of those reachable by the control-flow. ICFG nodes are connected using ICFG edges representing the control-flow. Since calls are not necessarily found at the end of LLVM's basic blocks, they are handled slightly differently. ICFG nodes are terminated by the call and connected to target function by call edges. In addition, we link an ICFG node to its immediate local successor, through a special edge called `BBLink` as illustrated in Figure 4.2. Call contexts are supported by creating, for each context, the corresponding nodes and edges until the maximum call string length is reached. During the ICFG creation, a considerable amount of information is collected and serves as annotations for the CFG and subsequent analyses. This, for instance, includes loop bounds, the identification of method cache block headers, as well as their corresponding sizes. Moreover, a loop nesting forest is constructed representing hierarchical relations for nested loops.

Predication
: Predicated execution is handled during the ICFG construction. We cover this in this chapter and show how *hidden* control-flow can be recovered from predicated instructions in LLVM's CFG. Handling predication in the ICFG step makes all subsequent timing analyses unaffected by this feature.

High-Level Analysis: In this step we basically collect information that is necessary to compute the WCET bound and also useful to make it more precise. This particularly consists of flow information and addresses of memory accesses:

Flow Information
: One of the advantages of Odyssey is that LLVM already provides control-flow-related information. Loop bounds that cannot be computed by LLVM can be provided by the user at the source-code level,

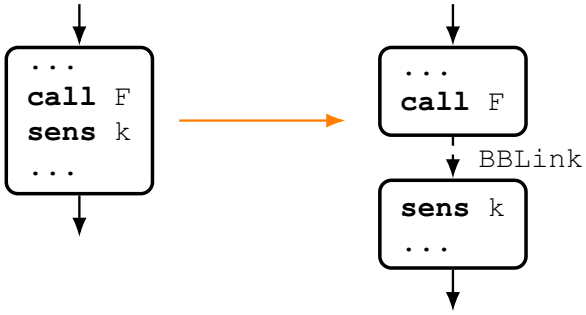


Figure 4.2 – Transforming an LLVM basic block with a call (left) into two ICFG basic blocks (right).

and can later be retrieved at the machine-code level in the form of *pseudo* instructions. This can be done using transformation techniques that map flow information from source code to machine code [67]. Furthermore, since we do not operate on the program's binary code, we are not required to recover branch targets, in contrast to other WCET tools. This information is already provided by LLVM.

Address Analysis — This step consists of two independent parts. First, we compute actual addresses of global variables and function symbols. The analysis is able to produce the complete symbol table of programs. Then, we use a modified value range analysis to determine address ranges.

Low-Level Analysis: Odyssey implements dedicated analyses for the stack cache, method cache, and the instruction cache. Those analyses typically operate on the ICFG and determine potential timing penalties to be accounted for:

Stack Cache — We currently support the standard stack cache analysis [63] (SCA) as well as the IDFA-based analysis for occupancy bounds [118, 88] (both presented and compared in Chapter 5). We also support a fix that reduces side-effects on the occupancy (at `sens` instructions) due to function recursion. This is simply done by propagating the occupancy at the call through `BBLink` edges. The result of the analysis consists of timing penalties due to spilling and filling issued by `sres` and `sens` instructions respectively. They are are associated to the ICFG in the form of annotations to be considered by the IPET phase.

Method Cache — A method cache analysis based on access classification is currently implemented. The analysis also takes into consideration a hardware optimization that implicitly evicts code blocks marked as *disposable*. Work in progress explores heuristics for disposable marking in order to reduce cache misses in large loops (see Chapter 9).

Instruction Cache — Odyssey supports conventional classifying cache analysis. The analysis relies on the address analysis to determine potentially accessed memory blocks.

Path Analysis: An ILP problem is formed based on control-flow constraints and

the results of previous analyses (see Chapter 3.Section 1). The problem can then be solved using usual standalone ILP solver tools.

To perform many of the aforementioned program static analyses, Odyssey implements an inter-procedural DFA engine. This allows to define and perform analyses on the ICFG representation where analysis information is attached to ICFG edges.
The implementation work we conducted constitutes a basis for further support of other predictability features of Patmos. This also favors tighter interactions between the compiler and the WCET analysis tool and enables more possibilities to explore WCET-related optimizations. This will also allow us to compare the outcome of predictable architectures against more standard hardware settings. In the remainder of this chapter, we present how predicated execution is handled in Odyssey.

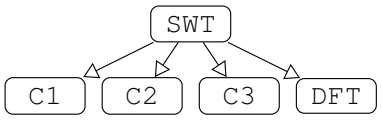
3 Handling Predication: Motivating Example

Before giving a formal description of our proposed approach, we will give a simple motivating example. Figure 4.3 illustrates the implementation of a `switch` statement using a jump table and predication. The original C code is shown in Subfigure 4.3a and the resulting *Control-Flow Graph* (CFG) from LLVM in Subfigure 4.3b. Basic blocks C1 through `C3` represent the three `case` statements, while basic block `DFT` corresponds to the `default` statement. The code of the `switch` statement itself can be found in the `SWT` block, whose machine code is shown in Subfigure 4.3c.
The machine code uses a jump table (`jt`) that is implemented as an array holding the addresses of basic blocks `C1` through `C3`. After verifying that the value of variable `x` is within the array bounds (`cmplt`), the address of the destination block is loaded (`lwc`) and control is transferred (after a 1 cycle load delay slot) via an indirect branch (`brcfnd`). If `x`'s value exceeds the array bounds, a conditional branch (`brcf`) immediately transfers control to basic block `DFT`. Note that this branch instruction has 3 branch delay slots and that one of these slots even contains another branch instruction.
The predicated code may pose several challenges in a WCET analyzer. One particular challenge is the reconstruction of the program's CFG from the binary machine code, which usually represents the input to most WCET analysis tools (see Chapter 3). The compiler placed a branch instruction in one of the branch delay slots of another branch. In this example it is trivial to detect that the predicates of the respective branches are disjoint. However, different predicates might be used, which makes it difficult to reconstruct the actual control-flow from such code. In our case this is not necessary, since the analysis is part of the compiler and thus has direct access to its intermediate representation.[1]
Another challenge, as noted before, are potential side-effects on caches or registers caused by predicated instructions. This issue is resolved by unfolding the hidden control-flow from the predicated code – as depicted by Subfigure 4.3d. Each time when a predicate register is defined (`cmplt`) a control-flow split is performed at the level of the control-flow graph. The instruction defining the predicate is then treated in a similar way as conditional branches and subsequent code is duplicated considering both of the potential predicate values (`true` or `false`). The basic block

[1]Note that this also solves many unrelated issues during the control-flow reconstruction from binary code such as computed branch targets, function pointers, et cetera.

```
switch(x) {
  case  1: ... break;
  case  2: ... break;
  case  3: ... break;
  default: ... break;
}
```

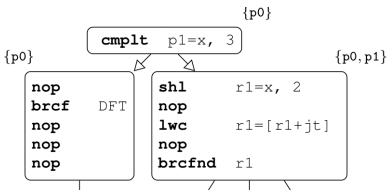
(a) Initial C code



(b) Original control-flow graph

```
       cmplt  p1=x, 3
( p1)  shl    r1=x, 2
(!p1)  brcf   DFT
( p1)  lwc    r1=[r1+jt]
       nop
( p1)  brcfnd r1
```

(c) Code of basic block SWT



(d) Unfolded control-flow graph

Figure 4.3 – Implementation of a simple `switch` statement using a jump table, the corresponding control-flow graph, the predicated machine code of basic block `SWT`, and the unfolded control flow.

on the left side of the subfigure here corresponds to an execution where predicate register `p1` evaluates to `false`, while the basic block on the right is executed only when `p1` evaluates to `true`. In fact, each basic block is associated with a set of predicates that are known to be `true` when entering the basic block (indicated in the top corner of each block). Inversely, predicates that do not appear in this set are known to be `false`.[2] Note that the predicates in the code are no longer needed. Predicated instructions are either duplicated *unconditionally* or are otherwise replaced by an explicit `nop`.
In the unfolded control-flow graph it is now much easier to analyze the instructions' side-effects. The load (`lwc`) from the jump table, for instance, is only executed when the variable `x` is known to be less than 3 (`cmplt`). This means that any side-effects of this instruction on the data cache are only visible in basic blocks `C1` through `C3`, but not in basic block `DFT`. Another implicit side-effect concerns the value of variable `x` after the comparison. Due to the control-flow split at the `cmplt` instruction, it is very easy for a value range analysis to show that the value of `x` has to be larger then 3 when reaching basic block `DFT`. Delayed branches and potential redefinitions of register operands make this much more challenging in the original CFG. The algorithm to construct such an unfolded CFG, while considering predication and delayed branches, is discussed in the next section.

4 Control-Flow Unfolding

Algorithm 2 shows a simplified version of our approach. The presented algorithm assumes a single issue architecture, which avoids the need to handle several parallel uses and (re-)definitions of predicates, parallel branches and predicate operations, et cetera. The algorithm also assumes that the predicates of branches that appear

[2]This is safe, since LLVM inserts pseudo definitions on all program paths where a register is not defined.

in branch delay slots are disjoint, i.e., only a single branch is known to be taken at any moment at runtime. Lastly, the presented approach only operates on the CFG of a single function. Extensions to the algorithm, included in the actual implementation, which allow us to handle these cases are briefly highlighted later. Finally, the algorithm invokes several helper functions whose code is not shown. We will briefly define these functions in an informal way before discussing the algorithm in detail.

Helper Functions

Several helper functions are needed in order to operate on individual instructions in LLVM's intermediate representation. The function NEXT allows to obtain the instruction immediately following an instruction i in its parent basic block, PKILL returns the set of predicate registers whose live ranges end after instruction i, while the functions PDEF and ISPREDDEF allow to obtain/test whether an instruction defines a predicate register. The function ISNOP is used to test whether an instruction i is nullified given the current set of predicates P.
Several helper functions are related to branches, allowing to test for branch instructions (ISBRANCH), obtain the number of the branch's delay slots (BRANCHDELAY), and obtain the successor basic blocks to which control may be transferred by a branch (BRANCHTARGETS). FALLTHROUGHTARGET is used to obtain the fall-through target basic block of the last instruction of a basic block, i.e., control is transferred to another basic block without an explicit jump or branch instruction.
Finally, three functions are related to the construction of the enriched intermediate representation of our analysis tool. GETCFNODE allows to obtain the control-flow node associated with a start instruction f, the remaining number of delay slots d, and a set of predicates P – if such a control-flow node was created before. Nodes are created using the function MAKECFNODE, which *duplicates* all instructions between the instruction f and e provided as arguments. The new node is also associated with d, the number of branch delay slots remaining, and P, the set of predicates. Finally, MAKECFEDGE creates control-flow edges between two control-flow nodes provided as arguments.

Discussion of the Algorithm

Algorithm 2 consists of two functions: the algorithm's main function UNFOLDCFG and the recursive function UNFOLD, which actually constructs the unfolded CFG. The latter function's parameters f (first) and l (last) represent machine instructions that need to be unfolded next. The integer argument d (delay) is needed to track branch delay slots across control-flow node boundaries. The argument T (targets), likewise, is used to track the set of potential branch targets during the handling of branch delay slots. The function's last argument P (predicates) represents the set of active predicates known to be `true`.
The algorithm starts off by processing the CFG of a function provided by LLVM (l. 31), considering the instructions at the function's entry point, which are not in a branch delay slot ($d = \infty$) and not executed under any specific predicate condition ($P = \{\texttt{p0}\}$). This triggers the recursive processing of all instructions in the CFG provided by LLVM and the construction of the unfolded CFG. The actual unfolding then proceeds in two steps.

Algorithm 2 Simplified algorithm to recover the hidden control-flow from predicated code by code duplication on a single-issue architecture.

```
1: function UnFold(MachInstr f, MachInstr l, Pred d, BasicBlockSet T, PredSet P)
2:     if n =GetCFNode(f, d, P) then return n    ▷ Check if control-flow node exists
3:     PredSet L = P; Pred pd = p0; MachInstr e = f          ▷ Initialize variables
4:     for each instruction i between l and f do
5:         L = L \ PKill(i)                                  ▷ Remove dead predicates
6:         e = i                                             ▷ Track end of control-flow node
7:         if ¬IsNop(i, P) then                              ▷ Skip nop instructions
8:             if IsPredDef(i) then                          ▷ Predicate definition
9:                 pd = PDef(i)                              ▷ Track defined predicate
10:                break                                     ▷ Immediately split control-flow
11:            else if IsBranch(i) then                      ▷ Branch instruction
12:                d = BranchDelay(i)                        ▷ Track branch delay slots
13:                T = BranchTargets(i)                      ▷ Track branch target(s)
14:        if d = 0 then break                   ▷ Split control-flow after branch delay
15:        d = d − 1                             ▷ Update remaining branch delay slots
16:    CFNode n = MakeCFNode(f, e, d, P)         ▷ Create a new control-flow node
17:    if e = l ∧ T = ∅ then                                 ▷ Handle fall-through
18:        T = FallThroughTarget(l)
19:    else if d ≠ 0 ∧ pd ≠ p0 then          ▷ Handle split due to predicate definition
20:        for each P′ ∈ {L ∪ pd, (L \ pd) ∪ {p0}} do         ▷ Compute successor predicates
21:            CFNode n′ = UnFold(Next(e), l, d, T, P′)
22:            MakeCFEdge(n, n′)
23:        return n
24:    for each s ∈ T do                     ▷ Create successor control-flow nodes
25:        for each P′ ∈ {L ∪ pd, (L \ pd) ∪ {p0}} do         ▷ Compute successor predicates
26:            Let f′, l′ be the first/last instruction of s in
27:                CFNode n′ = UnFold(f′, l′, ∞, ∅, P′)
28:                MakeCFEdge(n, n′)
29:    return n
30: procedure UnFoldCFG(G)
31:    Let f, l be the first/last instruction of the entry block of G in UnFold(l, f, ∞, ∅, {p0})
```

First, all the instructions between the arguments l and f of function UnFold are analyzed (l. 4 – 15) in order to find locations where the control-flow needs to be split. A split may be necessary due to one of the following reasons: (a) the end of the original basic block of LLVM is reached (fall-through), (b) a branch effectively transfers control to another basic block after its branch delay slots, or (c) a predicate definition is encountered.
Each instruction is analyzed in turn. Instructions that are nullified under the

current predicate set P (IsNop, l. 7) are ignored. Two instruction classes need special attention, since they may cause a control-flow split: instructions that define a predicate (IsPredDef) and branches (IsBranch). Predicate definitions are handled similar to branches in traditional CFGs and immediately lead to a control-flow split (**break**), remembering the current end location (e), and the newly defined predicate (pd). Branches, on the other hand, may cause a delayed control-flow split (BranchDelay). The variable d tracks the number of remaining branch delay slots (the variable is initialized to ∞ if the analyzed code is not in a branch delay slot). Once this counter reaches 0 the actual control-flow split occurs (l .14). Note that predicate definitions may also appear in branch delay slots. In this case the variable d is passed as an argument to subsequent recursive calls to the function UnFold. Since a branch was encountered, the branch targets also need to be remembered and potentially passed on the recursive calls using variable T. If no control-flow split is encountered by the analysis, i.e., no instruction defines a predicate or branches, the **for** loop terminates normally. This only happens for basic blocks with a fall-through. The set of live predicates (L) is tracked additionally, while instructions are processed. This set is initialized with the incoming argument P (l. 3) and updated whenever the live range of a predicate ends (l. 5). Note that a location, where the live range of a predicate ends, could be exploited to rejoin the control flow. The algorithm does not take advantage of this and essentially *extends* the live ranges of predicates up to a control-flow split.

The second step of the algorithm (l. 17 – 29) is concerned with the actual construction of the unfolded CFG. After leaving the **for** loop, a new control-flow node is created representing the instructions between f and e that are executed under the predicate set P (l. 16). It remains then to discover and unfold the successor control-flow nodes, depending on the nature of the control-flow split determined during the first step.

Fall-throughs (case a from above) and completed branches (case b) always transfer control to another basic block in LLVM's CFG. The main difference is that no branch target is known for fall-throughs (case a), which can simply be obtained using the function FallThroughTarget (l. 18). The remainder of the processing is identical (l. 24 – 29). Each branch target is analyzed through a recursive call to UnFold, considering the new set of active predicates P' as well as the branch target's first/last instruction. Note that a predicate definition (case c) may coincide with cases (a) and (b). The active predicates are thus computed from the live predicates L and the newly defined predicate pd (l. 25). The targets are then visited by the algorithm with the predicate `true` ($L \cup pd$) and `false` ($L \setminus pd$). Note that `p0` always remains `true` and consequently needs to be readded.

The remaining control-flow splits, that are not covered by the previous paragraph, are due to predicate definitions (case c), which may either occur in the middle of basic blocks or in a branch delay slot. Both situations require a slightly different handling (l. 19 – 23), since control is *not* (yet) transferred to another basic block of LLVM's CFG. The set of active predicates P' is computed as in the regular case. However, the remaining instructions (Next, l. 21) of LLVM's current basic block need to be analyzed after the control-flow split, while remembering potentially ongoing branches. This is accomplished by passing the values of d and T to the recursive invocation of UnFold. This allows the recursive invocation to correctly track the branch targets and the number of branch delay slots, i.e., if the split actually occurred in a branch delay slot.

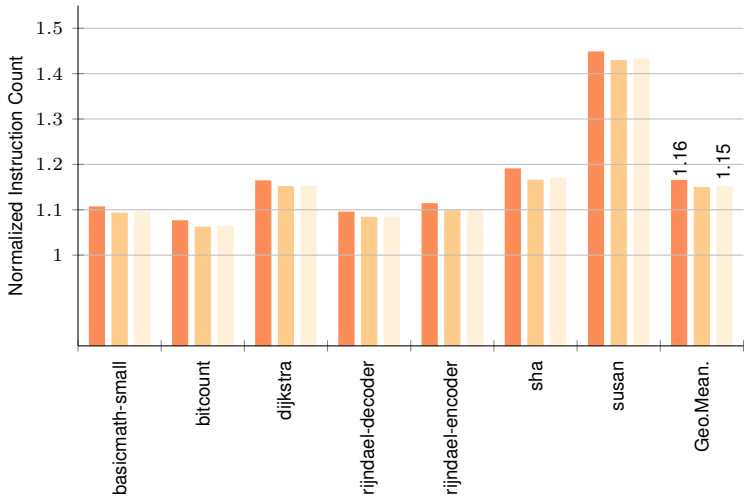


Figure 4.4 – Increase in the number of instructions due to unfolding for the delayed (■), mixed (■), and non-delayed (■) configurations with VLIW instruction bundles, normalized to the size of LLVM's original CFG (lower is better).

Extensions The presented algorithm is a somewhat simplified version of the actual implementation. Most notably, the Patmos processor can fetch and issue multiple operations in parallel using instruction bundles. This means that corner cases may arise that need to be considered. For instance, predicate definitions and branches can be combined into the same bundle. The implemented algorithm considers function calls and returns, whose branch delay slots also need to be accounted for. The handling of function calls was omitted for simplicity. These extensions slightly complicate the algorithm, but do not impact its overall structure.
Another, more involved extension, is the handling of branches nested within other branch delay slots. The presented algorithm is only correct as long as the predicates of the nested branches are disjoint. This can be handled by replacing the arguments d and T of the function UnFold by a stack data structure. This allows to track all executing branches at the same time, detect the completion of a branch, and split the control-flow accordingly.

Complexity The UnFold function essentially performs a depth first search on the CFG provided by LLVM. Each instruction is processed once for every set of potentially active predicates, whose number can be bounded by 128 (2^7). Note that `p0` is always `true` and thus cannot impact this bound. The algorithm thus is linear in the number of instructions and control-flow edges.

5 Experiments

The following section presents the results of experiments measuring the overhead induced by unfolding. The implementation is part of the Odyssey analysis framework, which is based on LLVM 3.5. The unfolded CFG is merely used for analysis purposes and essentially represents an additional annotation layer on top of the data structures of LLVM. The binary code of the analyzed programs is thus not modified. The extended version of the previously described algorithm was applied to a subset

of the TACLe benchmarks [41], i.e., those adopted from the MiBench suite. The programs were compiled with optimizations enabled (`-O2`), while varying the issue-width (single-issue vs. VLIW) and the compiler's handling of branch delay slots (non-delayed only, delayed only, mixed). This results in 6 configurations overall. The size of the unfolded CFGs for each of these configurations is compared against the original instruction count in LLVM's CFG.
Figure 4.4 shows the normalized increase in the number of instructions for the three configurations with VLIW instruction bundles. As can be seen, the overhead induced by unfolding is usually low, ranging between 10% and 20%. The `susan` benchmark is the only exception, showing an increase between 43% and 45%. The if-conversion optimization is particularly effective for this benchmark, covering larger regions and producing more complex predicates. Note that the observed overhead does not come as a surprise. It is well known that the share of conditional branches in typical programs roughly falls into a similar range as the observed overhead. This indicates that, overall, only a few instructions are duplicated by the unfolding algorithm for each computed *condition* – despite the fact that the algorithm artificially *extends* the live ranges of predicates.
The runtime overhead of the proposed algorithm is negligible and amounts to $0.1s$ on average, which represents 0.9% of the code generation time (excluding other WCET analysis steps). Also note that the unfolded CFG allowed us to improve other analyses. The value range analysis, for instance, is able to take advantage of control-flow splits at predicate definitions as explained in Section 3.

6 Conclusion

In this work a lightweight approach to the handling of predicated code in WCET analyzers was presented. Predicate definitions are treated similar to conventional branches and immediately lead to a control-flow split. Subsequent instructions are then analyzed twice, once assuming that the predicate evaluates to `true` and once assuming it evaluates to `false`. The hidden control-flow in predicated code is recovered and explicitly represented in an unfolded CFG. The presented algorithm is able to perform the desired control-flow unfolding and keep track of branch delay slots for a simplified single-issue architecture. The actual implementation in Odyssey is able to handle parallel instruction bundles, function calls, and nested delayed branches. Our preliminary evaluation shows that the unfolding does not result in excessive code duplication and yields a moderate code size increase of about 16% on average.

CHAPTER 5

Comparing the Precision of Stack Cache Occupancy Analyses

The previous chapter introduced our tool Odyssey, and showed how predication can be handled in a way that does not affect subsequent analyses. In this chapter, we take interest in the stack cache, Patmos' simplest cache structure that is dedicated for stack data. In order to propose effective optimizations, one first needs to inspect existing timing analysis approaches, and compare them. So far, two analyses have been proposed, each relying on a different approach to analyze the cache occupancy. Occupancy analysis is used to bound the spilling and filling cost induced by `sres` *and* `sens` *control instructions respectively. Therefore, its precision directly impacts the stack cache timing analysis. We, thus, compare the precision of stack cache occupancy analyses, and discuss situations where one outperforms the other. This chapter is structured as follows: Section 1 provides a short introduction and reminds of some of the stack cache properties (full description is provided in Chapter2.Section 2.4, for static program analysis see Chapter 3.Section 3). We then present the two approaches to analyze the occupancy bounds for the stack cache. The analyses are evaluated in Section 3 before concluding.*

1 Outline

The stack cache is a predictable cache design that was shown to be analyzable, while efficiently handling memory accesses to stack data at low (hardware) cost. Data accesses are, by definition, guaranteed cache hits, the content of the cache thus has to be managed explicitly using three stack cache control instructions: (1) `sres` k allows to reserve k words on the stack, (2) `sfree` k can be used to free previously reserved stack space, and (3) `sens` k, finally, can be used to make sure that at least k words are available in the cache. Only the reserve (`sres`) and ensure (`sens`) operations may initiate time-consuming memory transfers and thus need to be considered during timing analysis. The worst-case timing behavior of these instructions only depends on the worst-case spilling and filling of `sres` and `sens` respectively, which can be bounded by computing the cache occupancy bounds (i.e., maximum and minimum occupancy).
Stack cache occupancy bounds, and the associated spill/fill costs can be computed

using the proposed standard Stack Cache Analysis (SCA) [63]. The approach splits the analysis problem into several smaller steps, using context-insensitive data-flow analyses to capture function-local properties and longest/shortest path searches on the call graph to model calling contexts. An alternative solution would be to simply model the problem as a traditional inter-procedural DFA (IDFA) [14]. This appears simpler to implement, as the various steps of SCA are modeled in a single concise analysis. However, the impact on analysis precision has not been investigated so far. Indeed, overestimating the occupancy can increase the spill costs associated with `sres` instructions, while underestimating the occupancy can increase the fill costs of `sens` instructions. This chapter thus compares the precision of the two analysis approaches with respect to the attained maximum/minimum occupancy bounds. The chapter consists of two main sections. The first section presents each approach in detail with examples. The second one evaluates their precision using a series of benchmarks and discusses the obtained results.

2 Cache Occupancy Analyses

We present how to compute the cache's occupancy, which can be used to bound timing, using a tailored stack cache analysis (SCA) and an inter-procedural DFA-based approach (IDFA).

2.1 Standard Stack Cache Analysis

As all memory accesses (`lds`/`sts`) through the stack cache are guaranteed hits, the timing behavior of the stack cache only depends on the amount of data spilled or filled by `sres` and `sens` instructions, respectively (see Section 2.4 of Chapter 2). In the case of the standard stack cache this amount can be bounded by analyzing the cache's maximum/minimum occupancy, i.e., $\texttt{MT} - \texttt{ST}$. The *standard stack cache analysis* (SCA) proceeds in three phases:
First, the maximum/minimum displacement is computed for each function. These values indicate the largest/smallest number of cache blocks reserved during the execution of a function (including nested calls). The displacement can be used to efficiently compute the occupancy across function calls, since it allows to bound the number of blocks evicted from the stack cache. Due to the placement of stack cache control instructions,[1] the additional amount of stack space reserved at a given program point in a function, with respect to the function entry, is constant. In our case, it simply corresponds to the value of the parameter `k` of the `sres` instruction of the enclosing function. The problem thus can be modeled as a longest/shortest path search on a weighted call graph, where the edge weight of each call site is given by the amount of stack space allocated by the *calling* function. The minimum displacement is then given by the shortest path from a node to an artificial sink node. Likewise, the maximum displacement is given by the longest path.
Next, the maximum filling at `sens` instructions is bounded using a function-local data-flow analysis that propagates the *maximum* displacement from call sites to the succeeding ensure instructions. In our case every `call` is immediately followed by an `sens`, rendering this analysis trivial. The maximum filling at `sens` instruction can

[1] This applies to the restricted placement from above as well as *well-formed* programs.

Concept	Description	Analysis
Occupancy	Number of cache blocks occupied in the stack cache	–
min. Displacement	Min. number of blocks evicted during function call	shortest CG path
max. Displacement	Max. number of blocks evicted during function call	longest CG path
worst-case Occ.	Local bound of max. Occ. assuming full stack cache	DFA+min. Disp.
max. Filling	Min. occupancy before `sens` instructions	DFA+max. Disp.
max. Spilling	Max. occupancy before `sres` instructions	CG+w.-c. Occ.

Table 5.1 – Summary of concepts used by the traditional Stack Cache Analysis (SCA).

be bounded by computing the minimum number of cache blocks in the cache after the corresponding `call` instruction, i.e., the minimum occupancy. The minimum occupancy *after* the call has to be smaller than the occupancy before that call, since the called functions may only evict blocks from the cache. It cannot exceed $\max(0, |SC| - D(f))$, where $D(f)$ is the maximum displacement of the called function f and $|SC|$ the stack cache size. If this bound is smaller than `k`, the argument of the `sens` instruction, filling may occur. The maximum amount of filling can then be computed by subtracting the computed bound from `k`.
Finally, the worst-case occupancy is computed for each call site within a function using a function-local data-flow analysis. This is done by assuming a full stack cache at function entry. Subsequently, an upper bound of the occupancy is propagated to all call sites in the function, while considering the effect of other function calls and `sens` instructions. Function calls may evict stack data and thus lower the occupancy bound, depending on the *minimum* displacement of the called function (since the maximum occupancy after the call needs to be computed). The worst-case occupancy after a call cannot exceed $\max(0, |SC| - d(f))$, where $d(f)$ indicates the minimum displacement of the called function f and $|SC|$ the stack cache size. Ensure instructions on the other hand may increase the bound through filling, i.e., the worst-case occupancy after an `sres` has to be larger or equal to `k`, the ensure's argument. The maximum spilling at `sres` instructions is finally computed by propagating occupancy values through the CG, such that the maximum occupancy at the entry of a function is derived from the minimum of either (1) the maximum occupancy of its callers plus the size of the caller's stack frame (`k` of the corresponding `sres`) or (2) the worst-case occupancy bound computed by the local data-flow analysis. Note, that the latter case allows to consider spilling of other `sres` instructions that may reduce

```
(A1) func A()
(A2) sres 2 ⟨0⟩
(A3) call B
(A4) sens 2 ⟨2⟩
(A5) sfree 2
```

```
(B1) func B()
(B2) sres 1 ⟨0⟩
(B3) call C
(B4) sens 1 ⟨0⟩
(B5) call D
(B6) sens 1 ⟨1⟩
(B7) sfree 1
```

```
(C1) func C()
(C2) sres 1 ⟨0⟩
(C3) sfree 1
```

```
(D1) func D()
(D2) sres 4 ⟨3⟩
(D3) sfree 4
```

Figure 5.1 – A program consisting of 4 functions, reserving, freeing and ensuring space on the stack cache (cache size: 4). The annotations in angle brackets, e.g., ⟨2⟩, indicate the maximum filling/spilling behavior of stack cache control instructions.

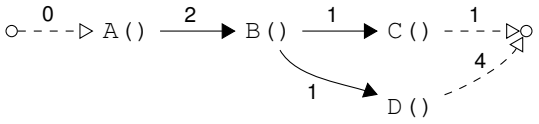


Figure 5.2 – A weighted call graph representing the program from Figure 5.1. The edge weights indicate the amount of stack space reserved in the respective functions, and can be used to compute the minimum/maximum displacement.

the occupancy before reaching the call site. Since the maximum spilling is computed on the call graph, fully context-sensitive bounds can be computed efficiently for all functions in a program.
Table 5.1 summarizes the various concepts used by the traditional SCA in order to efficiently bound the maximum filling/spilling at `sens` and `sres`, respectively.

Example 2.1 *Consider functions* `A`*,* `B`*,* `C`*, and* `D` *shown in Figure 5.1 and a stack cache whose size is 4 blocks. First, the displacement computation is performed on the weighted call graph shown in Figure 5.2. Function* `B`*, for instance, may call* `C` *or* `D`*. The maximum displacement thus has to account for the stack space reserved by* `B` *and by these two functions, which evaluates to either* $2 = 1 + 1$ *(*`C`*) or* $5 = 1 + 4$ *(*`D`*). For functions* `A`*,* `B`*,* `C`*, and* `D`*, respectively, the minimum/maximum displacement values evaluate to:* 4/7*,* 2/5*,* 1/1*, and* 4/4*. Then, the maximum filling of* `sens` *instructions is computed. Consider, for instance, the call from* `A` *to* `B` *(*A_3*). Since the maximum displacement of* `B` *is* 5*, the minimum occupancy after the call evaluates to* $0 = \max(0, 4 - 5)$*. The corresponding* `sens` *instruction (*A_4*) consequently has to fill both of* `A`*'s cache blocks (*$2 - 0$*), which is indicated by the bound in angle brackets* $\langle 2 \rangle$*. The displacement of* `C` *is only* 1*, which yields a minimum occupancy of* $3 = \max(0, 4 - 1)$ *after instruction* B_3*. The stack cache is thus large enough to hold both stack frames of* `B` *and* `C` *and no filling is needed as indicated by the bound* $\langle 0 \rangle$ *at instruction* B_4*. Next, the worst-case occupancy before call instructions is computed using a function-local data-flow analysis. The DFA determines that the worst-case occupancy before the call from* `B` *to* `D` *(*B_5*) is* $3 = 4 - 1$*, due to the call from* `B` *to* `C`*. Before all other call instructions the worst-case occupancy is* 4*, since no other call may lower the maximum occupancy before reaching them. Finally, the maximum occupancy is propagated through the call graph, starting at the program's entry function* `A`*. The maximum occupancy at the entry of* `A` *consequently is* 0*. For the call from* `A` *to* `B` *(*A_3*) a maximum occupancy of* 2 *is computed as the minimum of the call's worst-case occupancy (*4*) and the maximum occupancy at the entry of* `A` *plus the size of* `A`*'s stack frame (*$0 + 2$*). The maximum occupancy at the entry of* `D` *is similarly computed from the call's (*B_5*) worst-case occupancy (*3*) and the maximum occupancy at the entry of* `B` *plus the size of* `B`*'s stack frame (*$2 + 1$*). Since the size of* `D`*'s stack frame is equal to the stack cache size, all content of the stack cache has to be evicted by its reserve instruction. This results in a worst-case spilling of* 3 *blocks, as indicated by the bound* $\langle 3 \rangle$ *at instruction* D_2*. The bounds derived for the other* `sres` *and* `sens` *instructions are also indicated in angle brackets in Figure 5.1.*

2.2 Inter-procedural Data-flow Analysis

Another approach that is much simpler consists of computing and tracking the occupancy bounds exclusively using the DFA framework. For this, a set of data-flow

equations has to be formed and solved using for instance an ICFG representation of the program. (see Section 3 of Chapter 3). The analysis needs to define: (1) an abstract domain $\mathcal{D}$, (2) transfer functions $T_i \in T$ to model the effect of instructions, and (3) the join operator $\sqcup$ to conservatively merge information at join points.

Abstract Domain: The domain of the IDFA approach are positive integer values in $\mathcal{D} = \{0, \ldots, |SC|\}$, where $|SC|$ represents the stack cache's size. Since both, the minimum and the maximum occupancy are needed, two analysis problems have to be defined. We will start with the maximum occupancy.

Maximum Occupancy: The analysis starts at the program entry, where the occupancy is assumed to be 0. It then propagates occupancy values along all execution paths, while considering the effect of the instructions along the path. Only the stack control instructions (see Section 2.4 of Chapter 2) can have an impact: (1) `sres` instructions increase occupancy by their argument k, (2) `sens` instructions make sure that the occupancy is larger than k, and (3) `sfree` instructions reduce the occupancy by k. The resulting transfer function T_i for an instruction class i are given by the data-flow equation below:

$$\mathrm{OUT}_{maxOcc}(i) = \begin{cases} \min(\mathrm{IN}_{maxOcc}(i) + k, |SC|) & \text{if } i = \texttt{sres } k \\ \max(\mathrm{IN}_{maxOcc}(i), k) & \text{if } i = \texttt{sens } k \\ \max(0, \mathrm{IN}_{maxOcc}(i) - k) & \text{if } i = \texttt{sfree } k \\ \mathrm{IN}_{maxOcc}(i) & \text{otherwise} \end{cases}$$

The join operator $\sqcup$ is defined by simply taking the maximum occupancy for all operands. Therefore, the occupancy right before an instruction (due to control-flow joins) is derived by taking the maximum occupancy from any of its predecessors ($Preds$), except for the program's entry. In the case of inter-procedural analysis, predecessors can also be calls or returns from other functions:

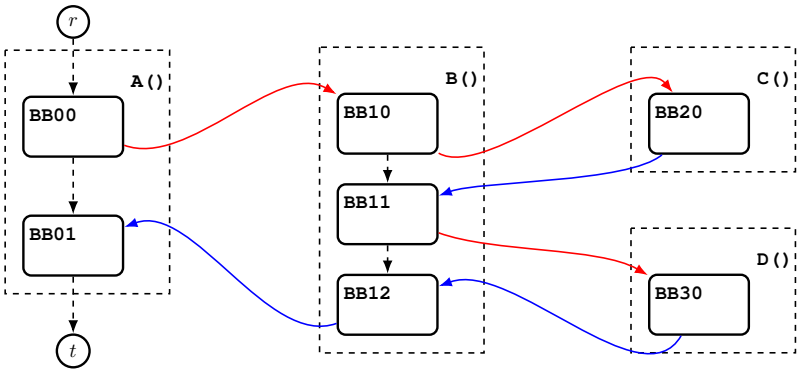
$$\mathrm{IN}_{maxOcc}(i) = \begin{cases} 0 & \text{if } i = \texttt{entry}, \\ \max_{s \in Preds(i)}(\mathrm{OUT}_{maxOcc}(s)) & \text{otherwise} \end{cases}$$

Minimum Occupancy: The data-flow equations to compute the minimum occupancy are identical. Only the max operator of the $\mathrm{IN}_{minOcc}(i)$ equation needs to be replaced by the min operator.

$$\mathrm{OUT}_{minOcc}(i) = \begin{cases} \min(\mathrm{IN}_{minOcc}(i) + k, |SC|) & \text{if } i = \texttt{sres } k \\ \max(\mathrm{IN}_{minOcc}(i), k) & \text{if } i = \texttt{sens } k \\ \max(0, \mathrm{IN}_{minOcc}(i) - k) & \text{if } i = \texttt{sfree } k \\ \mathrm{IN}_{minOcc}(i) & \text{otherwise} \end{cases}$$

$$\mathrm{IN}_{minOcc}(i) = \begin{cases} 0 & \text{if } i = \texttt{entry}, \\ \min_{s \in Preds(i)}(\mathrm{OUT}_{minOcc}(s)) & \text{otherwise} \end{cases}$$

Context sensitivity can easily be ensured either through the ICFG itself (duplication of functions) or by adding context information to the data-flow equations of the respective instructions.



(a) An ICFG representing the program from Figure 5.1. The red/blue edges represent call/return edges.

			Min. Occ		Max. Occ		Cost	
Func	**BB**	**Instr**	**IN**	**OUT**	**IN**	**OUT**	**Spill**	**Fill**
A	00	$\{A_1, A_2, A_3\}$	0	2	0	2	0	0
	01	$\{A_4, A_5\}$	0	0	0	0	0	2
B	10	$\{B_1, B_2, B_3\}$	2	3	2	3	0	0
	11	$\{B_4, B_5\}$	3	3	3	3	0	0
	12	$\{B_6, B_7\}$	0	0	0	0	0	1
C	20	$\{C_1, C_2, C_3\}$	3	4	3	4	0	0
D	30	$\{D_1, D_2, D_3\}$	3	0	3	0	3	0

(b) Occupancy bounds as computed by the IDFA approach considering the ICFG of Figure 5.3a.

Figure 5.3 – Example of occupancy analysis using the IDFA approach.

This approach is also implemented in Absint's aiT timing analyzer tool [118]. We spot, however, a mistake in the data-flow equation related to computing the minimum occupancy at `sens` instructions. The equation states that the resulting minimum occupancy is evaluated to $\min(\text{IN}_{minOcc}(i), k)$. This is incorrect as the `sens` instruction can never result in a decrease of the occupancy. The occupancy is always increased to k if the occupancy right before `sens` ($\text{IN}_{minOcc}(i)$) is less than k.

Example 2.2 *Consider again the program in Figure 5.1. We illustrate in Figure 5.3a the corresponding ICFG built similarly as in Section 3 of Chapter 3. We track the occupancy bounds according to the IDFA approach assuming a stack cache whose size is 4 blocks. We start with minimum occupancy analysis. The domain consists of a positive integer number representing the occupancy level (i.e.,* $\{0, \ldots, 4\}$*). The analysis starts off by assigning the occupancy value* 0 *at the program entry point (i.e.,* $IN_{minOcc}(\texttt{BB00}) = 0$*). The basic block* `BB00` *consists of instructions* A_1*,* A_2*, and* A_3*. Only the reserve* `sres` *at* A_2 *has an impact on the minimum occupancy which increases by* 2 *cache blocks, therefore,* $OUT_{minOcc}(\texttt{BB00}) = 2$*. Similarly, the* `sres` *increases the minimum occupancy by* 1 *cache block which results in* $OUT_{minOcc}(\texttt{BB10}) = 3$*. In* `BB20`*, the minimum occupancy is further increased by* 1 *cache block but immediately decreased due to the* `sfree` *instruction at* C_3*. This yields* $OUT_{minOcc}(\texttt{BB20}) = 3$*. The* IN_{minOcc} *information at* `BB11` *is computed by applying the join operator over*

the OUT_{minOcc} information provided by its immediate predecessors (i.e., `BB10` and `BB20`). This translates into selecting the minimum of both OUT_{minOcc} information, which yields $IN_{minOcc}(BB11) = 3$. The `sens` at B_4 does not initiate any memory transfer as the minimum occupancy is greater than `B`'s stack frame size (i.e., 1 cache block). The `sres` instruction at D_2 reserves 4 cache blocks which adds to the 3 cache blocks already existing when entering the basic block `BB30`. However, the minimum occupancy is capped by the stack cache size that is $|SC| = 4$. Those cache blocks are then freed by the `sfree` at D_3 which results in $OUT_{minOcc}(BB30) = 0$. The join operator applied at `BB12` yields $IN(BB12) = 0$, as a result the `sens` at B_6 initiates a fill of 1 memory block in the worst-case. The same applies to the `sens` at A_4 which initiates a fill of 2 cache blocks as $IN_{minOcc}(BB01) = 0$. In this example, the maximum occupancy provides the same results for IN_{maxOcc} and OUT_{maxOcc} information. However, the maximum occupancy information allows to safely estimate the worst-case spilling potentially performed by `sres` instructions. Only the `sres` at D_2 is determined to cause such memory transfers. This is due to the fact that the `sres` reserves the whole stack cache size which causes the stack cache content (i.e., $IN_{maxOcc}(BB30) = 3$) to be spilled to main memory.

3 Experiments

We evaluated both approaches using the LLVM-based compiler framework of the Patmos processor [111], which comes with a stack cache and its associated control instructions. Benchmarks of the MiBench benchmark suite [52] were compiled using optimizations (`-O2`) and subsequently analyzed using both techniques, assuming a stack cache size of 256 byte, 4 byte cache blocks, and a context string length of 0. Figure 5.4 shows the percentage of functions where the occupancy bound at function entry of SCA was either greater, equal, or smaller than that computed by IDFA.

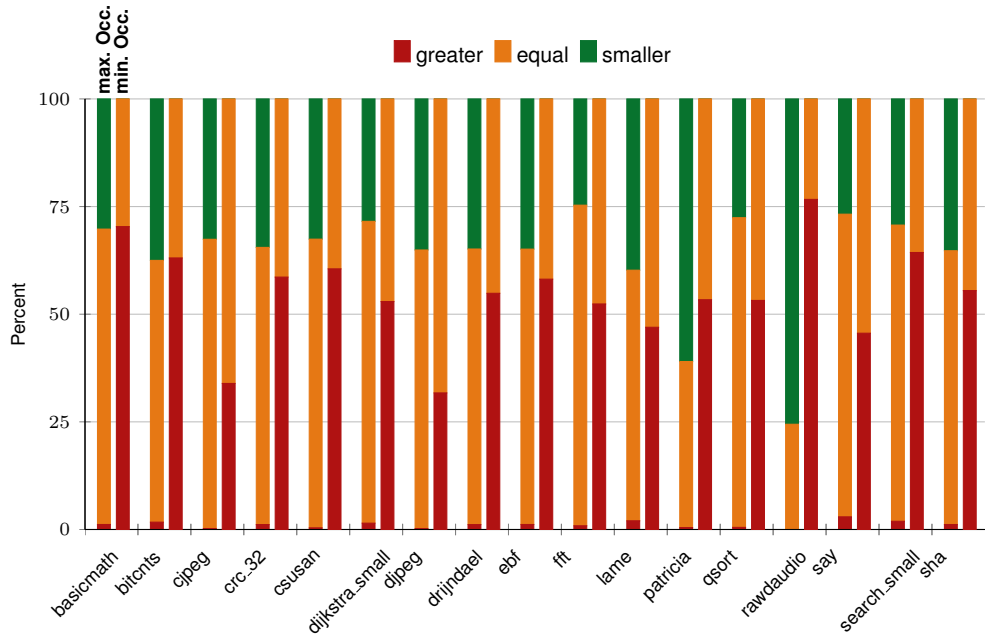


Figure 5.4 – Percentage of occupancy bounds (maximum/min) by SCA being (1) greater, (2) equal, or (3) smaller than IDFA.

Considering the maximum occupancy, SCA is less precise when the delivered bound is greater, i.e., the lower portion of the first bar of each benchmark should be as small as possible. Indeed, these cases are rare (< 3% over all benchmarks), while SCA is often more precise (34% on average).
Considering the minimum occupancy. SCA is less precise when the delivered bounds is smaller. This is represented by the upper portion of the second bar. However, this only appears for a simple function of the three benchmarks `tiff2bw`, `tiffdither`, and `tiffmedian` respectively. SCA is typically more precise (52% on average).
We repeated these experiments for IDFA with other call string lengths (1, 2, 3, 10 and 20). However, we only observed minor improvements for maximum occupancy and almost no change for minimum occupancy. The `bitcnts` benchmark, for instance, has a maximum call depth of 20, ignoring recursive functions, and still does not show relevant improvements with call strings of length 20 due to the impact of recursion elsewhere as explained later.
Overall, SCA is almost always as precise or even more precise than IDFA. The results are similar, albeit less pronounced, with longer context string lengths.

3.1 Discussion

A closer look reveals that the imprecision of IDFA is mostly due to chains of function calls, whose lengths exceed the analysis' context string length (e.g., due to recursion). Let us first examine such situations for maximum occupancy.
The problem of IDFA with long call chains is that calling contexts are no longer distinguished, i.e., all information is merged in a single calling context. The occupancy information computed for these regions is, as expected, rather pessimistic, leading to considerable overestimation of the maximum occupancy. Even worse, the overly conservative occupancy level is propagated out of these merged calling contexts along control-flow edges of function returns. Recall that the meet operator for this analysis is the max operator. This means that the conservative maximum occupancy bounds are even further propagated, way beyond the merged calling contexts that initially caused the imprecision. This particularly applies to recursive functions.

Example 3.1 *Figure 5.5 shows an example illustrating this situation. Assume that function* `A` *consists of one basic block and that function* `B` *is called before function* `D`*. Since* `B` *and* `C` *recursively call each other, their respective maximum occupancy grows until they reach the stack cache size during the fixed-point computation of IDFA (unless unbounded call strings are used). The transfer functions for the return instructions then propagate the maximum to their respective callers, which leads to a maximum occupancy that is close to the stack cache size right after the function call to* `C` *within* `B` *(and vice verse). A similarly high occupancy is propagated out of the recursion to the instruction succeeding the call from* `A` *to* `B`*. The high occupancy might actually occur within the recursion. However, the actual occupancy at this point is much lower. The overestimation is further propagated to function* `D`*. Resulting in overly conservative analysis results there, even when the context string length is not exceeded.*

Patmos' newlib C library contains (potentially) recursive functions in the start-up code of each program. IDFA thus assumes that the stack cache is filled up entirely before even reaching the program's `main` function. Since the computed maximum

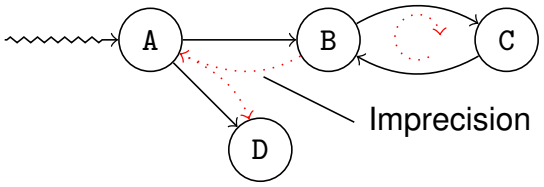


Figure 5.5 – Imprecision propagated out of recursive functions when computing maximum occupancy with IDFA.

occupancy at `main` is considerably overestimated, imprecision is propagated throughout large portions of the considered benchmarks. An important observation here is that increasing the call string length will not help fixing this problem, as the precision limit will be reached before the end of the recursion (unless infinite call strings are used). The SCA approach does not face this problem. Instead of relying on the occupancy propagated outwards by the recursive functions, it simply relies on their displacement values. A possible fix for this problem for IDFA would be to memorize the occupancy level before each call. The occupancy propagated backwards from a return then always has to be smaller than the memorized value. However, the potentially large displacement of the called function is ignored, which may still lead to considerable overestimation.
A similar problem arises for non-recursive programs with deep call chains containing two subsequent function calls that eventually invoke the same function. IDFA then behaves similar to recursive programs as shown in Figure 5.6.

Example 3.2 *Assume that, in this example, the function call from* `B` *to* `C` *appears before the call from* `B` *to* `D`*. Then, IDFA initially propagates an accurate occupancy level through the calls from* `A` *to* `B` *and finally to* `C`*. At first, even the occupancy at* `D` *is computed correctly. However, due to the deep call chains leading up to* `C`*, both calling contexts for* `D` *(originating from* `B` *or* `D`*) are merged. Due to the intermittent execution of* `D` *the occupancy is higher for this call chain. This increases the maximum occupancy of* `C`*. The increase is subsequently propagated out of* `C` *to both of its callers. This incidentally increases the occupancy after the call to* `C` *within* `B`*. Which then again increases the occupancy at the following call to* `D`*. This leads to a feedback loop similar to that seen for recursive functions in the previous example.*

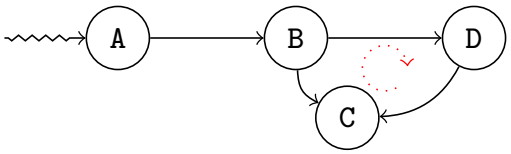


Figure 5.6 – Feedback loop enforcing imprecision of non-recursive functions for IDFA computing maximum occupancy.

Still, IDFA can be more precise than SCA (as shown by or results). This is explained by an underestimation of the minimum displacement. As mentioned before, the minimum displacement is obtained by performing a shortest path search on the program's call graph. The path here represents nested function calls and its length the minimal amount of stack space required in the stack cache by the functions

stack frames, respectively. Now, consider a case where two leaf functions[2] are called within a single basic block, i.e., when one function is called the other function is called too. In this case, the minimal path search will choose the function with the smaller stack frame to compute the minimum displacement. However, since both functions are called, the actual minimum displacement is determined by the larger stack frame. This situation can, of course, also appear in more general forms. The imprecise minimum displacement ultimately leads to an underestimation of the maximum occupancy observed in our experiments. However, this appears to be of minor importance in practice.
For minimum occupancy IDFA appears to be even more imprecise. Firstly, this is explained by the fact that the maximum displacement (in contrast to the minimum displacement) can be computed precisely. SCA's minimum occupancy thus does not suffer from inherent imprecision. In addition, IDFA spreads imprecision as before in the presence of deep call chains. This may even lead to feedback loops in non-recursive programs as described before. Two observations are particularly interesting at this point. While the IDFA approach is amenable to improvements by memorizing the maximum occupancy before calls, such a fix appears to be impossible here. The problem is that a lower bound cannot be established as easily for function calls when the minimum occupancy is computed. Secondly, it appears that the precision could be improved using very long call strings (ignoring cases incurring recursion). This, however, leads to a paradox situation: the precise computation of the minimum occupancy would then require high levels of context sensitivity in order to compute the worst-case filling at `sens` instructions. The filling, however, only depends on the nesting of functions called right before the ensure and thus is by its nature context-insensitive. The SCA approach exploits precisely this property and evidently achieves excellent results.

4 Conclusion

In this chapter, we compared the precision of stack cache occupancy bounds computed by two different approaches. On the one hand, the IDFA approach, which models the problem as a traditional inter-procedural data-flow analysis. On the other hand, the SCA approach that splits the analysis problem into several smaller ones, using context-insensitive data-flow analyses along with longest/shortest path searches on the call graph. Our experiments revealed that IDFA suffers from imprecision in nearly all benchmarks of the MiBench benchmark suite. The lack of precision is due to chains of function calls, whose lengths exceed the analysis' context string length (e.g., due to recursion). The obtained results suggest to base future analysis work (such as preemption costs that we explore in Chapter 6) on the SCA approach.

[2]Leaf functions do not call any other function.

CHAPTER 6

Analysis of Preemption Costs for the Stack Cache

In the previous chapter, we compared two analysis approaches to bound the minimum and maximal occupancy of the stack cache. We concluded that the approach used in the standard stack cache analysis (SCA) provides tighter estimates in most situations. However, the analysis was limited to individual tasks, ignoring aspects related to multitasking. A major drawback of the original stack cache design is that, due to its simplicity, it cannot hold the data of multiple tasks at the same time. Consequently, the entire *cache content needs to be* saved *and* restored *when a task is preempted. To complement the standard SCA, we propose, in this chapter, an analysis exploiting the simplicity of the stack cache to bound the overhead induced by task preemption. This chapter is structured as follows. We start with a short outline. Through a motivating example, we present in Section 2 our approach to analyze the cache-related preemption delays induced by the stack cache. A formal description of the analysis is provided in Sections 3 and 4. Then a discussion is provided in Sections 5 and 6. The analysis is evaluated in Section 7 and followed by a conclusion.*

1 Outline

The stack cache of the Patmos processor exploits the regular structure of the access patterns to stack data. Functions often operate exclusively on their local variables, resulting in spatial and temporal locality of stack accesses following the nesting of function calls. As already presented in Section 2.4 of Chapter 2, the cache can be implemented using a circular buffer using two pointers: the memory top pointer `MT` and the stack top pointer `ST`. In contrast to traditional caches, memory accesses are guaranteed hits. The time to access stack data thus is constant, simplifying the WCET analysis. The cache is managed by stack cache control instructions whose worst-case (timing) behavior only depends on the worst-case spilling and filling of `sres` and `sens`, respectively (see Chapter 5). The cache's simple design thus reduces the analysis complexity considerably.
However, the simple structure of the stack cache also has drawbacks. One problem arises when multiple tasks are executed using preemptive scheduling. The two pointers only capture the cache state of the currently running task, the state of other

(preempted) tasks is *lost* once `ST` and `MT` are overwritten. The data of preempted tasks might still be in the cache. However, the hardware *cannot* ensure that this data remains unmodified. Even worse, it cannot ensure that modified cache data, not yet written back to main memory, remains coherent. As a consequence the entire stack cache content has to be *saved* to main memory when a task is preempted. In addition, the stack cache content has to be *restored* before that task is resumed. This may induce considerable overhead that has to be accounted for during the analysis of a real-time system equipped with a stack cache.
The main contribution of this chapter is a stack cache analysis technique to bound the overhead induced by the stack cache during preemption, i.e., *cache-related preemption delays* [70].

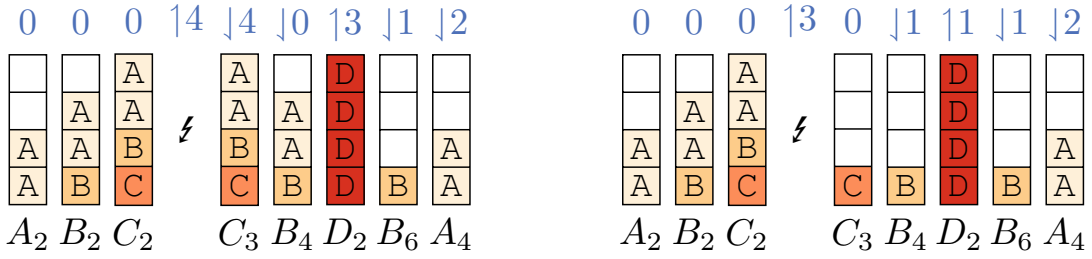
2 Analysis of Preemption Delays: Motivating Example

Preemptive multitasking provides better schedulability for real-time systems by allowing a running task to be preempted by another task having more critical timing requirements. Task preemption involves a *context switch*, which, with regard to the preempted task, consists of three steps: (1) *saving* the original task's execution context (registers, address space, device configurations, ...), (2) running another task, and finally (3) *restoring* the original task's context. Since the traditional stack cache hardware cannot be shared by several tasks, the content of the stack cache has to be considered a part of the execution context and thus needs to be saved and restored as well. This may induce some overhead that has to be accounted for during schedulability analysis. For traditional caches [70] this overhead is known as Cache-Related Preemption Delays (CRPD). We will later formally define a static program analysis that allows us to bound this overhead for the stack cache for every program point where a preemption might occur. However, we start first with a motivating example, illustrating the underlying problem:

Example 2.1 *Assume that a preemption occurs right before the* `sfree` *instruction* C_3 *(↯) of the code in Figure 6.1. The stack cache content then has to be* saved *and* restored *to/from main memory. A simple bound of the number of blocks that have to be transferred back and forth is given by the maximum occupancy provided by the SCA. In this example four blocks (of* `A`*,* `B`*, and* `C`*) need to be transferred, both, during context saving and restoration, as illustrated by Figure 6.2a. This overhead can be*

(A_1) **func** `A()`	(B_1) **func** `B()`	(C_1) **func** `C()`	(D_1) **func** `D()`
(A_2) **sres** `2` $\langle 0 \rangle$	(B_2) **sres** `1` $\langle 0 \rangle$	(C_2) **sres** `1` $\langle 0 \rangle$	(D_2) **sres** `4` $\langle 3 \rangle$
(A_3) **call** `B`	(B_3) **call** `C`	(C_3) **nop** ↯	(D_3) **sfree** `4`
(A_4) **sens** `2` $\langle 2 \rangle$	(B_4) **sens** `1` $\langle 0 \rangle$	(C_4) **sfree** `1`	
(A_5) **sfree** `2`	(B_5) **call** `D`		
	(B_6) **sens** `1` $\langle 1 \rangle$		
	(B_7) **sfree** `1`		

Figure 6.1 – Program consisting of 4 functions, reserving, freeing and ensuring space on the stack cache (cache size: 4). The annotations in angle brackets, e.g., $\langle 2 \rangle$, indicate the maximum filling/spilling behavior of stack cache control instructions.



(a) Simple approach.

(b) Improved approach.

Figure 6.2 – Cache states after executing the indicated instructions (below) and number of blocks transferred (above).

reduced as illustrated by Figure 6.2b. The stack data of `C` *will be freed immediately after the preemption and thus is* dead*, i.e., the data can never be accessed after the preemption. This reduces the cost of context saving to a transfer of* 3 *cache blocks (of* `A` *and* `B`*) instead of* 4*. Also the context restoration costs are reduced. Actually, no cache block needs to be restored here. It thus suffices to re-reserve a single block on the stack cache for* `C`*'s dead stack data. The blocks of* `B` *are only accessed after returning from* `C`*. The* `sens` B_4 *will automatically restore the necessary data. According to our initial SCA this instruction does not fill any block in the worst-case without preemption, i.e., an additional overhead to transfer* `B`*'s cache block needs to be accounted for as preemption cost. The cache blocks of* `A` *are similarly restored by the corresponding* `sens` A_4*. This time, the restoration will not cause any additional cost, since the standard SCA already accounts for the filling of two cache blocks. At the same time, the occupancy before the next function call to* `D` *is reduced from* 3 *to* 1*, since only* `B`*'s stack frame was actually restored. Consequently, the spilling of* `D's` *reserve instruction* D_2 *is reduced. With preemption, actually fewer cache blocks are spilled than computed by the standard SCA – thus reducing the preemption costs. In comparison to the simple approach, transferring* 14 *cache blocks, the transfer costs only amount to* 8 *blocks.*

This example illustrates that the number of cache blocks to save/restore can be reduced depending on the future use of the cached data. Our analysis, explained in the following subsections thus, is based on the notion of *liveness* – very similar to the concept of Useful Cache-Blocks [70].

Context Saving Analysis (CSA): Clearly, data that is present in the cache, but known to be coherent with the main memory (captured by the lazy pointer `LP`, see [12]), can be excluded from context saving and thus reduce the preemption cost. Furthermore, some data might be excluded from saving depending on liveness, i.e., dead data that is not used in the future can be excluded. We will show how the analysis of dead and coherent data can be combined to reduce the number of blocks that need to be saved on a context switch.

Context Restoring Analysis (CRA): As for CSA, dead data can be excluded from context restoration. However, in many cases also live data can be excluded, e.g., when the data is spilled by an `sres` instruction before it is actually used or when an `sens` instruction would refill the data anyway. We will show that the underlying analysis problem is very similar to the liveness analysis required for CSA and, in

particular, that the placement of `sens` instructions after calls simplifies the analysis problem.

3 Context Saving Analysis

The worst-case timing of saving the stack cache's context depends on the number of cache blocks that have to be transferred to the main memory. In the simplest case, all blocks potentially holding data need to be transferred, i.e., the maximum occupancy provided by SCA is a safe bound. However, not all data in the stack cache actually needs to be considered, as illustrated by the motivating example.
The lazy pointer [12] readily allows to track coherent data that can be ignored during context saving, i.e., data known to have the same value in the cache and in main memory. Since the `LP` is implemented as a hardware register it can immediately be exploited by any context switching mechanism. Also the proposed analysis is immediately applicable and can be reused for the Context Saving Analysis. We thus do not provide details regarding the analysis here and simply assume that its results are available for the final cost computation of the CSA (see below).
Another class of data, that can be ignored during context saving, is dead data, i.e., data that will never be accessed by the program. Data in the stack cache may become dead starting from a given program point due to two reasons: (1) data that will be overwritten by an `sts` instruction (without an intermittent `lds`) in all executions or (2) data that is freed by an `sfree` (without an intermittent `lds`) in all executions. Inversely, data that is potentially used by a subsequent `lds` instruction has to be considered live.
Note that individual bytes on the stack cache might be live or dead depending on the actual usage of each individual byte, which would necessitate an analysis that is able to track individual bytes. However, due to the structure of typical stack frames generated by the compiler, we observe that dead data usually resides at the bottom of the stack, i.e., right above `ST`. The following analysis takes advantage of this fact in order to simplify the analysis complexity.
Inspired by the `LP`, we define a virtual pointer that allows us to track blocks of dead data residing right above `ST`. This virtual marker is only used by the analysis and is *not* realized as a hardware register. We call this virtual marker the *dead pointer* (`DP`):

Definition 18 *The dead pointer (`DP`) is a virtual marker tracking dead data, such that* $\texttt{ST} \leq \texttt{DP} \leq \texttt{MT}$. *Data below `DP` is considered dead, while data above `DP` is potentially live.*

The lazy pointer (`LP`) and the dead pointer (`DP`) define a partitioning of the stack cache's content into three distinct regions shown in Figure 6.3. Data above `LP` is coherent and thus can be ignored during context saving. Similarly, data below `DP` is known to be dead and can safely be ignored. Only the remaining data, between `DP` and `LP`, actually needs to be transferred to main memory. Note that this model only allows to detect dead data at the bottom of the stack cache – which we observed to be the usual case in the code generated by the compiler. The obtained results are thus a conservative approximation, i.e., more dead data might actually be present in the cache, which is not detected by the analysis and thus cannot be exploited. Likewise, more coherent data might be present below the position of the `LP` determined by

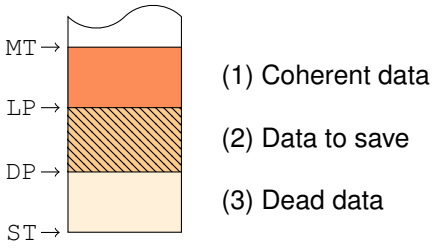


Figure 6.3 – Partitioning of the stack: (1) coherent data above LP (■), (2) data that actually needs to be saved (▨), and (3) dead data below DP (□).

the analysis. Both of these cases may lead to an over-estimation of the worst-case cost determined by the CSA, but do not compromise the analysis' correctness. Also note that this approach simplifies the actual context saving, since only a contiguous block of data needs to be transferred.

The analysis of the DP is based on a typical *backward* liveness analysis, i.e., a value is said to be live when it is used by a subsequent load (lds) and is considered dead immediately before a store (sts), or, in the case of the stack cache, an sfree. As for the traditional SCA, only the relative position of the DP with regard to ST needs to be known, which further simplifies the CSA. Our analysis is a function-local, backward data-flow analysis, conservatively tracking the lowest possible position of the DP relative to ST, i.e., for each program point the minimum value $\min(\texttt{DP} - \texttt{ST})$ is computed over all possible executions of the analyzed program. This ensures that the analysis is conservative and only considers the least amount of dead data actually in the cache for the cost computation.

As indicated above, only three kinds of instructions may modify the position of the DP. Whenever an lds is encountered, it must be ensured that DP is below its frame-relative address FA starting from ST, i.e., $\texttt{DP} \leq \texttt{FA}$, since the value loaded by the instruction is known to be live. Recall that the analysis proceeds in a backward fashion, so the loaded value is live at all program points *before* the lds, up to a preceding sts instruction potentially overwriting the same FA. An sts, on the other hand, might push the DP upward as the overwritten data is dead immediately before the store. This is only possible when the analysis is able to show that the newly discovered dead data is right above the contiguous block of dead data, such that a new contiguous block can be formed. The sts overwrites data at a given FA in the cache, the overwritten value thus can no longer be accessed and is dead at all program points before the store instruction, up to a join (conditional branch) and/or an lds instruction potentially rendering the data live. Finally, with regard to a function, all its data is dead immediately before its sfree, since none of the data in the stack frame can be accessed from this point on. The DP then is at its highest possible position, i.e., the stack frame's size k. In addition to these three instructions that may directly have an impact on the DP, the analysis also needs to consider conditional branches, i.e., instructions that may have multiple successors in the CFG. Since the analysis proceeds backward, the successor's DP values might be different. The analysis thus needs to apply a join operator (see Section 2.4 of Chapter 2), which selects a conservative approximation. In the case of CSA, the minimum, i.e., the least amount of dead data, is considered.

The following data-flow equations specify how individual instructions (Equation 6.1) and joins (Equation 6.2) may modify the relative position of the DP with regard to

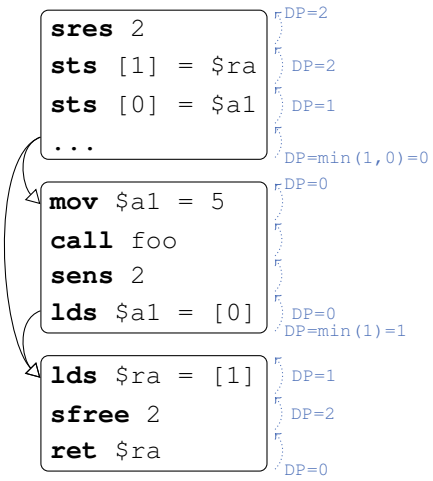


Figure 6.4 – Propagation of the DP (shown on the right in blue) within a function: stack data becomes dead right before `sfree` and `sts` instructions, while it becomes live before `lds` instructions. Other instructions do not impact the DP.

the stack frame of a function:

$$\mathrm{OUT}(i) = \begin{cases} \mathtt{k} & \text{if } i = \mathtt{sfree\ k} \\ \min(\mathrm{IN}(i), \mathtt{FA}) & \text{if } i = \mathtt{lds\ FA} \\ \mathrm{IN}(i) + 1 & \text{if } i = \mathtt{sts\ FA} \wedge \mathtt{FA} = \mathrm{IN}(i) \\ \mathrm{IN}(i) & \text{otherwise} \end{cases} \tag{6.1}$$

$$\mathrm{IN}(i) = \begin{cases} 0 & \text{if } i = t \\ \min_{s \in Succs(i)}(\mathrm{OUT}(s)) & \text{otherwise} \end{cases} \tag{6.2}$$

The position of the DP before (and after) each instruction in the function can then be computed by applying these equations iteratively until a fixed-point is reached. The initial values assigned to $\mathrm{IN}(i)$ and $\mathrm{OUT}(i)$ for each instruction i have to be chosen such that the iterative processing actually converges and delivers a safe approximation. The above data-flow equations compute the lowest position of the DP, it thus suffices to initialize the equations with the size of the stack cache $|SC|$ or the size of the current stack frame k – both are upper bounds on the maximum value of DP. The initialization of $\mathrm{IN}(t)$ to 0, where t represents the function's exit point, along with the use of the minimum as the join operator ensures that the analysis converges towards the minimum value of the DP and consequently gives a safe approximation.
Assuming a unit cost $\widehat{c}_s$ to transfer a cache block to main memory, the overhead induced by context saving before an instruction i depends on the size of the coherent area $\mathrm{CA}(i)$ (derived from the LP [12]), the size of the dead area $\mathrm{DA}(i)$ (given by Equation 6.1), and the maximum occupancy $\mathrm{Occ}(i)$:

$$savingCost(i) = \widehat{c}_s \max(0, \mathrm{Occ}(i) - \mathrm{CA}(i) - \mathrm{DA}(i)) \tag{6.3}$$

Note that the size of the coherent data as well as the maximum occupancy are potentially calling-context dependent, i.e., might change with the nesting of surrounding function calls. This is readily supported by the respective analyses and can easily be considered in the above equations. The costs would then, of course, also be context-dependent.

It would, in addition, be possible to consider the calling-context when analyzing the dead area ($DA(i)$). Whenever all data in a function's stack frame is dead, the size of its caller's dead area can be added to $DA(i)$. However, this is rarely beneficial in practice, since all functions, except leaf functions not calling other functions, store the return address on the stack. Details on inter-procedural analysis are thus omitted.

Example 3.1 *Consider the CFG of the function shown in Figure 6.4, which consists of three blocks of straight-line code. The sequence of the top most block is assumed to end with a conditional branch having the two other blocks as successors (indicated by the edges on the left). The goal of the analysis is to track the area* $\texttt{DP} - \texttt{ST}$ *of data that is known to be dead by computing the lowest possible position of* `DP` *at each program point. The analysis processes the CFG backwards, starting at the return instruction* **`ret`** *at the bottom. The computation of the analysis and its results are indicated in blue to the right of the code. Since the return is the last instruction in the function, the* `DP` *is initialized to* 0*. All stack data is potentially live here. The* `DP` *value is then propagated to its predecessor the* **`sfree`** *instruction. All stack data is known to be dead right before this instruction, the* `DP` *is thus set to* 2*, the instruction's argument (*`k`*). Next, the* **`lds`** *instruction is processed. The top most stack element of the function's stack frame is accessed (using the frame-relative address* `[1]`*) and thus becomes live, which is indicated by the new* `DP` *value of* 1*. The second load (*`[0]`*) in the block above is processed similarly. Here the* `DP` *drops to* 0*, indicating that both stack elements are live. The remaining instructions (***`mov`***, ***`call`***, ***`sens`***) in the same block have no effect on liveness. The last instruction of the top-most block has* 2 *successors with different* `DP` *values (*1 *and* 0*). The join operator conservatively takes the minimum to safely over-approximate the actually live stack data. The algorithm eventually processes the store instructions at the top. The first* **`sts`** *(*`[0]`*) overwrites the first stack element, whose value becomes dead. The* `DP` *thus is incremented to its new value* 1*. The subsequent* **`sts`** *(*`[1]`*) then overwrites the top element, rendering all stack data dead (*$\texttt{DP} = 2$*). Note, that instructions before the* **`sres`***/after the* **`sfree`** *conceptually belong to the caller.*

4 Context Restoring Analysis

Similar to context saving, the time required to restore a task's stack cache context depends on the number of cache blocks that need to be transferred from main memory to the cache. A simple solution would again be to transfer all the blocks potentially holding data, which is again bounded by the maximum occupancy.
However, as shown in Example 2.1, not all cache blocks have to be restored. We can distinguish the following cases, as illustrated by Figure 6.5: (1) cache blocks containing dead data only (given by Equations 6.1 and 6.2), (2) blocks potentially containing live data that have to be restored, and (3) blocks that are restored by a subsequent `sens`. Since only a subset of the cache blocks are restored the occupancy after a preemption is usually reduced. This may reduce the spill costs of subsequent `sres` instructions. The analysis thus has to consider another case: (4) potential gains due to reduced spilling. Case (1) and (2) can be handled by function-local analyses explained in Section 4.1, while case (3) and (4) require inter-procedural analyses covered in Section 4.2 and 4.3.

4.1 Local Restore Analyses

Dead data can simply be excluded from the memory transfer as explained in the previous subsection. However, in contrast to context saving where dead data is simply discarded, space has to be allocated on the stack cache in order to guarantee that subsequent memory accesses (stores in particular) succeed. The allocation is only needed when dead data exists, i.e., DA(i) is non-zero. Even then, the operation only requires an update of MT, which can be performed in constant time ($\widehat{c}_a$):

$$allocationCost(i) = \begin{cases} \widehat{c}_a & \text{if DA}(i) \neq 0 \\ 0 & otherwise \end{cases} \tag{6.4}$$

Blocks containing live data have to be restored and thus transferred back from main memory. This can be done explicitly during the context restoration or implicitly by an sens instruction executed later by the program. While the explicit transfer always induces additional overhead that needs to be accounted for, the implicit restoration might be for free. This happens when the maximum filling computed by the traditional SCA for the sens instruction is non-zero. The overhead associated with the explicit restoration is then, at least partially, accounted for in the program's WCET.
In order to account for the overhead of implicit and explicit transfers two quantities have to be determined: (1) the amount of data that needs to be restored explicitly and (2) the cost of implicit memory transfers performed by sens instructions. We introduce another virtual marker to model the former quantity. This pointer represents an over-approximation of the amount of *live* data in the stack cache that is *not* implicitly restored by an sens instruction before a subsequent access rendering the data live:

Definition 19 *The restore pointer (RP) is a virtual marker tracking potentially live data in the stack cache, i.e.,* ST $\leq$ RP $\leq$ MT*. Data below the* RP *is potentially live and* not *guaranteed to be restored by a subsequent* sens *instruction.*

An interesting observation is that sens instructions are placed after every function call and that functions are assumed to only access their own stack frames. This simplifies context restoration, since only stack data of the function where the preemption occurred has to be restored. The stack frames of the calling functions are then automatically restored by their respective sens. The computation of the associated overhead is explained in Section 4.2.
The analysis of the RP is a function-local, *backward* analysis that tracks the highest possible position of the pointer relative to ST (i.e., RP − ST), which means that an

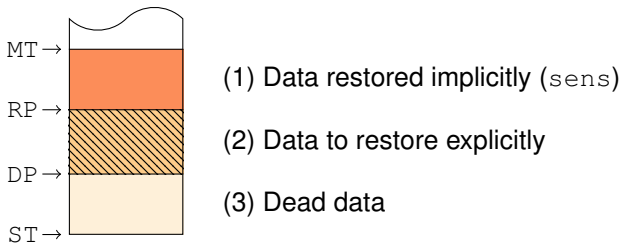


Figure 6.5 – Partitioning of the stack: (1) data restored by sens of the current as well as other functions (■), (2) data to restore (▨), and (3) dead data (□).

over-approximation needs to be computed. The position of the RP depends on the amount of data restored implicitly as well as the amount of live data. The analysis thus needs to consider the impact of sens instructions, which lower the position of the RP, as well as memory accesses, which may increment the RP. Whenever an sens instruction is encountered by the analysis the position of the RP is set to 0, which simply means that no data needs to be restored in case of a preemption that occurs immediately before that ensure (recall that the analysis proceeds backward). The sens simply reloads the entire stack frame when the task gets resumed. Data becomes live whenever it is accessed by an lds instruction, the position of the RP thus has to be larger or equal to the FA of any load instruction encountered. In order to simplify the handling of dead data, lds and sts instructions are both considered to increment the RP – despite the fact that stores do not actually render the data live. Dead data is excluded from explicit and implicit transfers anyway using the DP (as indicated above). Also note that there is no strict ordering between the DP and the RP, i.e., it might happen that DP > RP. This usually happens when dead data is present at an sens instruction, which sets the RP to 0, but has no impact on the DP. In addition to the instructions that have an immediate impact on the RP the analysis also needs to account for control-flow joins at conditional branches, which may have multiple successors with diverging RP values. The analysis always selects the maximum value and propagates this information upwards in order to ensure that the position of the RP is safely over-approximated. The following equations capture the evolution of the RP relative to ST:

$$\mathrm{OUT}_{RP}(i) = \begin{cases} 0 & \text{if } i = \texttt{sens k} \\ \max(\mathrm{IN}_{RP}(i), \texttt{FA}) & \text{if } i = \texttt{lds FA} \vee i = \texttt{sts FA} \\ \mathrm{IN}_{RP}(i) & \text{otherwise} \end{cases} \tag{6.5}$$

$$\mathrm{IN}_{RP}(i) = \begin{cases} 0 & \text{if } i = t \\ \max_{s \in Succs(i)}(\mathrm{OUT}_{RP}(s)) & \text{otherwise} \end{cases} \tag{6.6}$$

Assuming unit costs $\widehat{c}_r$ to transfer a cache block from main memory, the cost of restoring the live data of the stack cache depends on the size of the dead area ($\mathrm{DA}(i)$, Equation 6.1) and the size of the restore area ($\mathrm{RA}(i)$, Equation 6.5):

$$transferCost(i) = \widehat{c}_r \max(0, \mathrm{RA}(i) - \mathrm{DA}(i)) \tag{6.7}$$

Example 4.1 *Figure 6.6 illustrates the propagation of the RP through the CFG from the previous example. The processing again starts at the bottom of the CFG at the return instruction. At this point all data is dead (*RP = 0*) and thus does not need to be restored explicitly. This changes when the first* **lds** *instruction ([1]) is encountered, which renders the top-most element of the stack frame and all elements below it live. This is indicated by the new RP value of* 2*. The RP does not change until the* **sens** *instruction is processed. At this point all the stack frame's data is known to be live. However, the* **sens** *instruction ensures that all stack data is filled into the cache. Thus no explicit restoration is required and the new value of RP becomes* 0*. The other instructions in the block above the* **sens** *do not have an impact on the RP. As before, the last instruction of the top-most block is assumed to be a conditional branch with two successors having different RP values. This time, the maximum value* 2 *is propagated upwards in order to conservatively over-approximate the amount of potentially live data. The subsequently processed instructions do not*

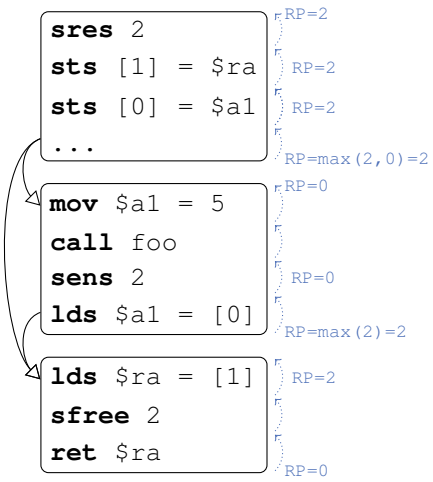


Figure 6.6 – Propagation of the RP (shown on the right in blue) within a function: only lds, sts, and sens instructions impact the RP, while other instructions do not modify its value.

have an impact on the RP since it is already at the maximum position (2), which would indicate that the entire stack frame needs to be restored explicitly. However, since the DP was shown to be non-zero (see Example 3.1), the stack frame only needs to be restored partially.

It remains to account for the implicit transfer costs at sens instructions in the current function that are not already included in the program's WCET. This situation arises whenever the RP pointer is *not* at its maximum position (the size of the function's stack frame k). The function's stack frame is thus only partially restored by the explicit transfer after a preemption and some additional cache blocks need to be filled from main memory implicitly by the next sens instruction. The additional cost of this transfer depends on the size of the function's stack frame, the size of the restore area (RA (i) from above), and the number of cache blocks that need to be transferred by the sens instruction for a regular execution without preemption, which is provided by the standard SCA in the form of an annotation to the instruction ($\langle b \rangle$). The overhead is trivially upper-bounded by the function's stack frame size k. A more precise bound would be $\mathtt{k} - \mathrm{RA}(i)$, which reflects the reduction of the cost of the implicit transfer cost by deducting the explicitly transferred blocks. Another bound can be derived from the maximum filling bound b associated with an sens instruction. The additional costs due to the implicit restoration cannot exceed $\mathtt{k} - b$, which represents the maximum number of cache blocks whose transfer costs are not accounted for in the program's WCET.
The following cost function combines both of the above approaches. However, before the cost function can be defined, an intermediate step has to be performed, which propagates the maximum filling bounds associated with individual sens instructions to all program points throughout the function. This allows to determine for each instruction, also those that are not an sens, the number of cache blocks that are potentially filled by any subsequent sens instruction. This intermediate step can be implemented using a function-local, *backward* DFA, propagating the difference between the sens's argument k and its filling bound $\langle b \rangle$ (obtained from the standard

SCA) upwards through the CFG:

$$\mathrm{OUT}_{FL}(i) = \begin{cases} \texttt{k} - \texttt{b} & \text{if } i = \texttt{sens k } \langle \texttt{b} \rangle \\ \mathrm{IN}_{FL}(i) & \text{otherwise} \end{cases} \tag{6.8}$$

$$\mathrm{IN}_{FL}(i) = \begin{cases} 0 & \text{if } i = t \\ \max_{s \in Succs(i)}(\mathrm{OUT}_{FL}(s)) & \text{otherwise} \end{cases} \tag{6.9}$$

The overhead caused by implicit memory transfers of `sens` instructions can then be computed from the number of cache blocks that are filled implicitly ($\mathrm{FL}(i)$, Equation 6.8) and the number of blocks that were explicitly restored, i.e., the size of the restore area ($\mathrm{RA}(i)$, Equation 6.5):

$$ensureCostLocal(i) = \widehat{c}_r \max(0, \mathrm{FL}(i) - \mathrm{RA}(i)) \tag{6.10}$$

As for the analysis of the `DP` before, the above data-flow equations for the `RP` and the local filling need to be initialized properly in order to ensure that the fixed-point computation converges. Since both analyses define the join operator as the maximum over all successors, the equations have to be initialized to 0 before the resolution process starts. In the case of the `RP` analysis this indicates that no data needs to be restored explicitly after the function's `sfree`. The fist access to stack data encountered by the analysis will then increment the `RP` value accordingly. The iterative processing then ensures that the analysis converges towards a safe upper bound. A similar argument applies to the propagation of the local filling bounds. The equations also contain an explicit initialization to 0 for the function's exit point t. This initialization is, in fact, redundant, given the fact that t cannot have any successors and that all data-flow equations are initialized to 0 anyway.

4.2 Global Ensure Analysis

The analyses in the previous subsections exclusively focus on the state of the stack frame of a single function and account for additional costs related to the restoration of the stack frame of the function whose execution was interrupted by a preemption. The stack frames of other functions that are currently on the call stack are not explicitly restored. This is done via implicit memory transfers, which are performed by the `sens` instructions that are placed after every function call. The underlying idea is very similar to the local reserve analysis discussed before, with the main difference that no explicit memory transfer is performed whatsoever.
To analyze the costs associated with these implicit memory transfers, an over-approximation needs to be computed that considers all possible states of the call stack, i.e., all possible chains of nested function calls leading up to a call to the function under analysis. This is sufficient, since the additional overhead is only induced by the `sens` instructions that are executed upon returning from functions along the call stack. The program's call graph (CG) is a well-known representation capturing all chains of nested function calls that may occur during the execution of the program (see Section 2.4 of Chapter 2). Each such chain observed during the execution of the program corresponds to a path in the call graph starting at the program's entry point (typically the `main` function) and leading to the graph node representing the current function. In order to compute the desired over-approximation, the analysis thus needs to consider all paths leading to the currently considered function and associate a cost with each path.

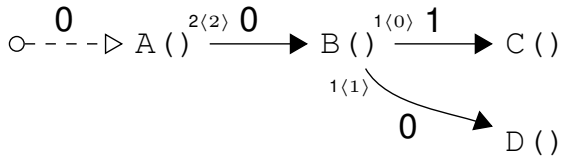


Figure 6.7 – Weighted CG of the code in Figure 6.1 used to bound the additional transfer costs at `sens` instructions of other functions.

We model this problem as a longest path search on a weighted CG, considering all paths from the program's entry node to the current function. The edge weights in the graph are the number of blocks that are *not* filled by the `sens` associated with the corresponding call site, which is given by $\mathrm{FL}(i)$ (Equation 6.9) of the site's `call` instruction. Note that this problem is very similar to the computation of the maximum displacement of the original SCA [11]. However, the length of the path is bounded: (1) by the maximum occupancy at the call site (which is itself bounded by the stack cache size) and (2) by the minimum amount of stack data remaining in the stack cache after returning from the function, i.e., $\max(0, |SC| - D(f))$, where $|SC|$ denotes the stack cache size and $D(f)$ the function's maximum displacement. The latter case is particularly interesting, since no computation is required when the function's displacement is larger than or equal to the stack cache size. The length of the path and the restoration costs then simply become 0. Also note that the global ensure costs are always the same, independent of where the program is interrupted in the function. It is thus sufficient to pre-compute the costs only once for each function.
Our algorithm thus pre-computes the global ensure costs as follows. A weighted call graph is constructed beforehand, where the edge weights are provided by the local ensure analysis (Equation 6.9). The algorithm then processes each function f separately. First, it is verified whether the maximum displacement $D(f)$ of f exceeds the stack cache size. If this is the case, the global ensure costs are bounded by 0, and the algorithm simply proceeds with the next function. Otherwise, an integer linear program (ILP) is constructed, which is similarly structured as the traditional IPET approach [77]. The ILP encodes all paths originating at the root node of the CG leading to the current function, such that the objective function represents the length of the path. An ILP solver then computes the longest such path, by maximizing the objective function. Note that this approach allows to handle any kind of program, including those with recursion. The original work on the SCA includes a detailed description on the handling of recursion [11]. Note that for programs without recursion the longest path for *all* functions can be computed in linear time using dynamic programming [29]. Given the length $\mathrm{FLG}(f)$ of such a path for function f, the costs induced at other functions is:

$$ensureCost(f) = \widehat{c}_r \mathrm{FLG}(f) \tag{6.11}$$

Example 4.2 *Consider the code from the initial example in Figure 6.1. The algorithm begins by constructing a weighted call graph as shown in Figure 6.7. Apart from the edge weight that is shown in the middle of each edge, the figure also indicates the information provided by the local ensure analysis at the respective call site. The numbers at the origin of each edge represent the argument* `k` *of the next ensure*

instruction following the call site as well as its filling bound in angle brackets. The edge weight simply correspond to the difference between these two values.
The edge weight for the call from `B` *to* `C`*, for instance, evaluates to* 1*, since the corresponding ensure instruction may transfer an additional block, which is not accounted for by its original bound* $\langle 0 \rangle$ *(*$1 - 0 = 1$*). Similarly, the edge weight of the call from* `A` *to* `B` *evaluates to* 0*. The corresponding* `sens` *transfers up to* 2 *blocks, of which both are already accounted for by the bound* $\langle 2 \rangle$ *(*$2 - 2 = 0$*).*
For `C` *the longest path has a length of* 1*, since an additional block needs to be transferred if a preemption were to happen during the execution of* `C`*. The longest path from the program's entry to function* `D` *has length* 0*, i.e., all cache blocks of calling functions are restored for free as they are accounted for by the original bounds. The same result could have been computed from* `D`*'s maximum displacement (*4*), which is equal to the cache size (*4*).*

4.3 Global Reserve Analysis

Lazily restoring the stack cache content not only allows us to avoid explicit memory transfers during context switches, but it may also turn out to be profitable. Even in the worst-case only the stack frame of the current function is restored, which leaves the remaining space in the stack cache free and thus effectively reduces the stack cache's occupancy. This may be beneficial for subsequent `sres` instructions, since the reduced occupancy may also reduce the maximum spilling. This, consequently, may reduce the running time of the program under analysis. There are two scenarios where such a gain might be observed: (1) at an `sres` of another function that is called from the current function and (2) at an `sres` of another function that is called after returning from the current function. It is important to note here, that multiple `sres` instructions may profit from the reduced occupancy, i.e., when several calls are nested or are performed in sequence with increasing displacement values. The analysis thus needs to be able to accumulate gains of multiple function calls, while providing a conservative under-estimation of the actual gains. We will initially focus on the first scenario and limit our attention to a single function call, and later extend this solution in order to handle the accumulation of gains as well as gains from the second scenario.
Recall that the WCET of the program under analysis already includes an estimate of the maximum spilling at `sres` instructions, which is computed for each function individually from the maximum occupancy before entering the function and the amount of stack space (`k`) reserved by the function's `sres`. This can be generalized to several nested function calls by considering the displacement at the outer-most function call. The maximum spilling performed by all called functions can then be bounded by considering the maximum occupancy along with the maximum displacement of the nested function calls. While the minimum spilling can be bounded by considering the minimum occupancy along with the minimum displacement. Consequently, a conservative estimation of the minimum gain can be computed by comparing the minimum spilling of a normal execution with the minimum spilling after a preemption. More formally, given a call instruction i with a minimum occupancy $\mathrm{mOcc}(i)$ and a minimum displacement $d(i)$ the minimum spilling during a normal execution is given by:

$$minSpill(i) = \max(0, \mathrm{mOcc}(i) + d(i) - |SC|) \tag{6.12}$$

The minimum spilling with preemption is computed in a very similar way. However, the minimum occupancy is lower due to the lazy restoration of the stack cache's content. A simple bound of the minimum occupancy, that is sufficiently precise in practice, is the size of the current function's stack frame, i.e., the argument of the current function's stack cache control instructions `k`:

$$minSpillPr(i) = \max(0, \mathtt{k} + d(i) - |SC|) \tag{6.13}$$

The minimum gain from the reduced spilling at a call site i is then given by:

$$siteGain(i) = \max(0, minSpill(i) - minSpillPr(i)) \tag{6.14}$$

A simple solution to account for the impact of the *next* function call is to propagate the gain at call sites backward through the CFG. The following equations determine the minimal gain that is guaranteed to occur for only one of the *subsequent* function calls. At joins, the equations select the minimum, while the maximum is selected on straight line code:

$$\mathrm{OUT}_{GN}(i) = \begin{cases} \max(\mathrm{IN}_{GN}(i), siteGain(i)) & \text{if } i = \mathtt{call} \\ \mathrm{IN}_{GN}(i) & \text{otherwise} \end{cases} \tag{6.15}$$

$$\mathrm{IN}_{GN}(i) = \begin{cases} 0 & \text{if } i = t, \\ \min_{s \in Succs(i)}(\mathrm{OUT}_{GN}(s)) & \text{otherwise} \end{cases} \tag{6.16}$$

As before, the data-flow equations have to be initialized in order to ensure that the analysis converges. The equations have to be initialized to the maximum possible gain, i.e., $|SC|$, the size of the stack cache, except for the function's exit node t. Since, at this moment, the analysis only considers local gains due to calls from within the current function, the gain at the end of the function evaluates to 0 for t (Equation 6.16). The analysis converges towards a minimum gain, despite the fact that on straight-line code the maximum value is propagated (which initially indeed is $|SC|$). This is ensured by the initialization of t (0) and the fact that the minimum value is selected at joins (Equation 6.16). The analysis thus will eventually reevaluate the data-flow equations of all program points reachable in the reversed CFG from the function's exit node t and converge towards a minimum.

Since we initially did not expect considerable returns from this analysis, our initial publication [13] adopted this simple approach only without developing it further. Though simple to compute, this solution is conservative. The gain of subsequent calls can, in fact, be accumulated since the occupancy remains lower than in a regular execution even after returning from the called functions. Unfortunately, this accumulation of costs cannot directly be encoded using data-flow equations. The accumulated costs in cyclic regions of the CFG would grow infinitely and thus yield wrong results.

Since then, we noticed that the underlying problem can, in fact, be seen as a shortest path problem on a weighted CFG. The edge weights in the graph represent the gain associated with individual call sites (Equation 6.14), while the length of the shortest path from an instruction i to the CFG's sink node t represents the accumulation of gains for all of the visited call sites. This is possible since every call site is uniquely identified even if some function is called many times in some execution path. However, the analysis has to make sure that the gain of visited call sites is not accumulated more than once. Fortunately, this cannot occur since the only way to revisit the

same call site again would be in a loop. Such a scenario is naturally avoided by the shortest path search algorithm, since revisiting the same call site would increase the path length. Given the length LSG (i) of the shortest path for instruction i, it is now possible to account for the actual gain associated with all function calls possibly executed within the current function after a preemption at i:

$$reserveGainLocal(i) = \widehat{c}_s \mathrm{LSG}(i) \tag{6.17}$$

Note that the length of the path, and thus the local gain, is bounded. The gain can never exceed $|SC| - \mathtt{k}$, since the lazy restoration may in the worst-case only reload the local stack frame, whose size is given by `k`. This can also be seen by assuming that the minimum occupancy ($\mathrm{mOcc}(i)$ in Equation 6.12) evaluates to $|SC|$. Simplifying the formulas (cf. Equation 6.13 and 6.14) yields the same result. This bound holds for nested function calls and sequences of function calls. The nesting of function calls is conservatively modeled using the minimum displacement ($d(i)$) in the formulas. The effects of function calls that are performed in sequence are conservatively modeled by considering the minimum occupancy, provided by the standard SCA. Recall that the minimum occupancy can be bounded locally by considering the impact of function calls through their maximum displacement (see Chapter 2.Section 2.4). The gain of each function call in a sequence thus is reduced by preceding calls due to the reduced minimum occupancy, which immediately depends on the calls' maximum displacements. The accumulated local gain thus cannot exceed the aforementioned bound since the gain gradually approaches 0 due to the interplay between minimum occupancy and maximum displacement of intermittent calls.

Example 4.3 *The gain due to the reduced spilling at the function entry of* `B` *needs to be analyzed, right after its reserve instruction* B_2*. The function first calls* `C`*, whose minimum occupancy, provided by the standard SCA, evaluates to* 3*, while its displacement evaluates to* 1*. Spilling will never occur while executing* `C`*, since the stack frames of* `A`*,* `B`*, and* `C` *fit into the stack cache (*$2 + 1 + 1 = 3 + 1 <= 4$*). The local reserve gain associated with the respective call site thus is* 0*. Likewise, the minimum occupancy before the call to* `D` *is* 3*, its displacement, however, is* 4*. The* `sres` *instruction* D_1 *consequently spills* 3 *blocks (*$3+4-4$*) during a regular execution (Equation 6.12), while only a single block (*$1 + 4 - 4$*) is spilled after a preemption (Equation 6.13). The call site is thus associated with a weight of* 2 *(*$3-1$*). Since both calls are executed in any case, the length of the shortest path from the preemption point to the end of* `B` *evaluates to* 2 *(cf. Equation 6.17).*

The previous analysis only accounts for the gain due to function calls within the current function. In addition, it is also possible to account for potential gains after returning from the current function. Recall that the stack frames of all functions currently on the call stack are lazily restored. The occupancy after a preemption may thus also be lower for these functions compared to a regular execution without preemption. Similar to the computation of the global ensure costs, we can account for this gain through a path search on a weighted CG. The edge weights for this graph are given by the local reserve gain (Equation 6.17) at the respective call sites. The minimal gain that is guaranteed to occur for all executions has to be computed. The algorithm thus has to search for the shortest path in the CG instead of the longest. Given the length $\mathrm{GSG}(f)$ of the path for function f, the global gain due to `sres` instructions is given by:

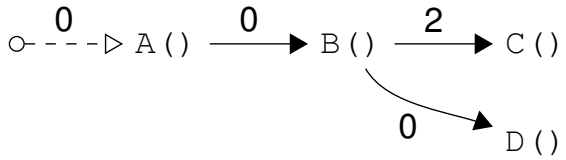


Figure 6.8 – Weighted CG of the code in Figure 6.1 used to bound the global gain due to `sres` instructions of other functions.

$$reserveGain(f) = \widehat{c}_s \mathrm{GSG}(f) \tag{6.18}$$

An interesting observation at this point is that the global reserve gain is bounded just as the local reserve gain before. As the analysis climbs upwards through the call graph (towards the program's entry function), the displacement of the functions increases. With this increase the potential gain of subsequent function calls diminishes (as before), limiting the global reserve gain to the minimum of either the minimum occupancy at the function's entry or $|SC| - \mathtt{k}$, where `k` represents the current function's stack frame size.

Example 4.4 *Consider the code from the initial example in Figure 6.1. The algorithm begins by constructing a weighted call graph as shown in Figure 6.8. The edge weights correspond to the local gain associated with each call site, given by Equation 6.17. For the call from* `B` *to* `C`*, for instance, the edge weight evaluates to* 2*. This is because a preemption that occurs in* `C` *will eventually return to its caller* `B` *with a reduced occupancy. This will lead to reduced spilling during the subsequent call to* `D`*, as explained in more detail in Example 4.3. The local reserve gain after the call to* `C` *at instruction* B_4 *thus gives the above edge weight. The same applies for the call from* `B` *to* `D`*. Here, the local reserve gain at instruction* B_6 *yields the edge weight* 0*, since no additional function calls appear after that instruction. The global reserve gain at* `C` *thus evaluates to* 2 *(*$0+0+2$*), which correspond to the length of the path from the call graph's root to the node representing the function. For all other functions the global gain simply evaluates to* 0*.*

4.4 Context Restore Costs

The total context restoration costs are then bounded by accumulating the individual costs for space allocation, the explicit and implicit transfer of cache blocks at `sens` instructions (locally and globally). In addition, the preemption costs are partially amortized by the reduced spilling at `sres` instructions (locally and globally). Note that $f(i)$ denotes the function containing instruction i:

$$\begin{aligned} restoreCost(i) = \; & allocationCost(i) && \text{(Eq.6.4)} \\ & + transferCost(i) && \text{(Eq.6.7)} \\ & + ensureCostLocal(i) && \text{(Eq.6.10)} \\ & + ensureCost(f(i)) && \text{(Eq.6.11)} \\ & - reserveGainLocal(i) && \text{(Eq.6.17)} \\ & - reserveGain(f(i)) && \text{(Eq.6.18)} \end{aligned} \tag{6.19}$$

Example 4.5 *Consider again the preemption point at instruction* C_3 *in the code shown in Figure 6.1. The context restoring analysis first determines the minimal/maximal offset of the* `DP` *and* `RP` *with regard to* `ST` *respectively (Equations 6.1 and 6.5).*

The `RP` *offset is* 0*, due to the absence of* `lds` *and* `sts` *instructions in the code, which are omitted for brevity.* `DP` *on the other hand, is equal to* 1*, since all data is dead right before the* `sfree` *instruction* C_4*. Therefore a single block has to be allocated to properly rebuild* `C`*'s stack frame, i.e.,* $allocationCost(C_3) = 1$*. Since all data of the current stack frame is dead, neither an explicit memory transfer nor an implicit restoration by an* `sens` *instruction is necessary, i.e.,* $transferCost(C_3) = 0$ *and* $ensureCostLocal(C_3) = 0$*.*
After returning to its caller, the ensure instruction B_5 *has to restore the entire stack frame of* `B`*. The additional cost has not been considered by the bound provided by the standard SCA (*$\langle 0 \rangle$*). The instruction thus fills an additional cache block. The* `sens` *instruction* A_4*, on the other hand, restores all of* `A`*'s cache blocks for free (*$\langle 2 \rangle$*). Therefore, the global ensure cost accounts for the transfer of an additional cache block (*$ensureCost(\texttt{C}) = 1$*), as illustrated before by Example 4.2.*
As `C` *does not call any other function, it cannot profit from a local reserve gain (*$reserveGainLocal(C_3) = 0$*). However, the analysis determines the potential gain for function calls after returning from* `C`*. The occupancy before the call to* `D` *is reduced by* 2 *blocks compared to a regular execution. The local gain associated with the corresponding call site is thus* 2*. Since there is no other subsequent call in* `B` *nor in* `A`*, no further gain can be considered. The global reserve gain for function* `C` *thus amounts to* 2 *cache blocks (*$reserveGain(\texttt{C}) = 2$*), as shown in Example 4.4.*
The total cost, associated with context restoration after a preemption at the indicated program point C_3*, is thus given by* $1 + 0 + 0 + 1 - 0 - 2 = 0$*, assuming unit costs of* $\widehat{c}_a = \widehat{c}_s = \widehat{c}_r = 1$*.*

5 Computational Complexity

The overall complexity of the CSA and CRA depends on the various analysis steps, which consist of four classes of analysis problems: (1) function-local data-flow analyses, (2) longest path searches on the CG, (3) shortest path searches on the CG, and finally (4) shortest path searches on the CFG of individual functions.
We assume that the data-flow equations of the various DFAs are solved using a traditional worklist algorithm, which iterates until a fixed point is reached. Various complexity bounds can be considered for different classes of DFAs, depending on the size of the CFG as well as on characteristics of the analysis domain.
In our case, the domains are essentially natural numbers in the range $[0, \texttt{k}]$ (cf. the analyses of the `DP`, `RP`, and local filling) or $[0, |SC|]$ (local gain), where `k` in turn is also bounded by $|SC|$. The considered analyses are monotone, i.e., the analyzed values steadily increase or decrease until either the minimum or maximum value of the domain is reached. This is called the height of the domain, which can be bounded by $|SC|$ for all considered DFAs.
The iterative worklist algorithm then propagates the domain values along the control-flow edges in the CFG. This leads to a first, conservative, complexity bound for the previously described analyses, which is in $O(|E||SC|)$ considering the height of the domain $|SC|$ and a CFG with $|E|$ edges. Another bound can be derived using the *loop connectedness* of the reversed CFG (since all considered problems are backward problems). This parameter characterizes the nesting of loops in a CFG G with respect to a spanning tree T of G [56, 64] and usually is denoted as $d(G, T)$. The number of iterations performed by the worklist algorithm can be bounded by this

parameter when the *order* in which the CFG edges are processed is well chosen. The iterative processing may then process each CFG edge at most $d(G,T)+3$ times, resulting in a complexity bound of $O(|E|(d(G,T)+3))$, where $|E|$ again denotes the number of CFG edges in the CFG G. The loop connectedness can be considered constant in practice (since loops tend to have a simple structure). Similarly, the size of the stack cache $|SC|$ can be considered constant with regard to the analysis problems. The overall complexity of all the previously described DFAs is thus linear in the size of the CFGs of the individual functions in the program under analysis.
The global ensure analysis relies on longest path searches on the CG in order to bound the cost induced by implicit memory transfers of `sens` instructions. For programs with recursion (which are often forbidden in the context of real-time systems) this requires the construction of an ILP (similar to the well-known IPET approach by [77]) for each node of the CG. The ILPs are subsequently solved by an external solver (such as CPLEX or LPSolve). While it is possible to bound the complexity of constructing an ILP in linear time with respect to the size of the CG, it is difficult to bound the solving times. Integer linear programming in general is NP-hard. However, it appears that today's solvers are able to handle the problem instances we encountered in our experiments quite well. The solvers almost instantly provide an optimal solution – even open-source solvers that do not apply sophisticated heuristics. For programs without recursion, the longest path search for all functions can be performed in $O(|F|+|A|)$ [29], where $|F|$ represents the number of functions in the CG and $|A|$ the number of call sites. Note, furthermore, that the two approaches can be combined, i.e., dynamic programming is applied to a reduced CG where cyclic regions are collapsed. The potentially expensive ILP solving can then be limited to the recursive functions only [63].
The shortest path searches on the CG (global reserve gain) and the CFGs of individual functions (local reserve gain) can be performed in quadratic time in the size of the respective graphs using simple algorithms. More advanced algorithms allow to reduce the complexity to almost linear time, e.g., the algorithm of [121] yields a complexity in $O(|E|+|V|\log\log|V|)$. The local reserve gain can thus be computed in almost linear time with regard to the size of the CFGs of individual functions. The same applies to the global reserve gain, which can be computed in almost linear time with regard to the number of functions and call sites in the program. These bounds are independent from the graphs' shape, which may well contain cycles, i.e., loops in the case of CFGs or recursion in the CG.
The complexity of the proposed CSA and CRA analyses thus is dominated by the longest/shortest path searches, whose complexity depends on the size of the program under analysis (both in terms of function size as well as the size of the CG). Lastly, the complexity of the standard SCA needs to be taken into consideration, since intermediate results of this analysis are reused in various analysis steps of the CSA and CRA. The SCA is similarly based on longest/shortest path searches that are combined with function-local DFAs. The complexity analysis for the SCA is almost identical to the discussion from above [63]. The overall complexity to compute the preemption costs is thus not impacted and is also dominated by the longest/shortest path searches.

6 Discussion

The analysis proposed here mostly operates locally on individual functions. This reduces the computational complexity (context-sensitivity is avoided) and simplifies the efficient analysis of large programs (e.g., through parallel analysis). Inter-procedural information is modeled through longest path problems on the CG, which is much smaller than a corresponding inter-procedural CFG. As real-time software usually avoids recursion, these computations are very efficient (linear in the size of the CG). Also note that the function-local data-flow analyses usually ignore `sres` and `sfree` instructions. Consequently, the computed results do not apply for code before an `sres` as well as after an `sfree`. This is not an issue, since the correct information can be derived from the calling functions, i.e., code before/after the first/last stack cache control instruction in a function is logically considered to be part of the immediate caller.

7 Experiments

This section presents an evaluation of the preemption costs associated with the stack cache. We cover full results from the static analysis described in Sections 3 and 4. The benchmarks are taken from the MiBench benchmark suite [52], which covers a large variety of small- and medium-sized programs typically found in embedded systems. The programs were compiled with optimizations enabled (`-O2`) using the LLVM[1] compiler for the Patmos processor [111]. The instruction set of the processor follows the *Very Long Instruction Word* (VLIW) paradigm and may execute up to two instructions that are grouped into parallel bundles at the same time. All instructions explicitly take a predicate operand, which allows to conditionally nullify instructions depending on the predicate value that is evaluated at runtime. The hardware of the platform is configured with a 64KB, 4-way set-associative data cache using LRU replacement, and a write-through policy (recommended for real-time systems, see [125]). Code is cached by a 64KB method cache [111] with LRU replacement and 32 code block entries. The stack cache is 256b small and uses a lazy pointer [12]. Note that varying the stack cache size between 256b and 1KB showed little impact on the results obtained. The global memory is assumed to have a moderate latency of 21 cycles. Memory transfers are performed in bursts of 32b. The cache line size of all caches matches the memory's burst size. Note, the stack cache control instructions still operate in words, while memory transfers are performed in bursts.
The analysis is implemented in the Patmos backend of the LLVM compiler, and operates on the final machine-level code, right before code emission. The reported numbers represent a simplified cost model, consisting of the number of bytes that have to be saved or restored during context switching at the beginning of basic blocks, i.e., sequences of straight-line code that are typically terminated by a (conditional) branch instruction.

7.1 Context Restoring Analysis

The context restoring analysis shows remarkable results over all benchmark programs considered. The main benefit stems from the fact that the `sens` instructions are

[1] `http://www.llvm.org/`

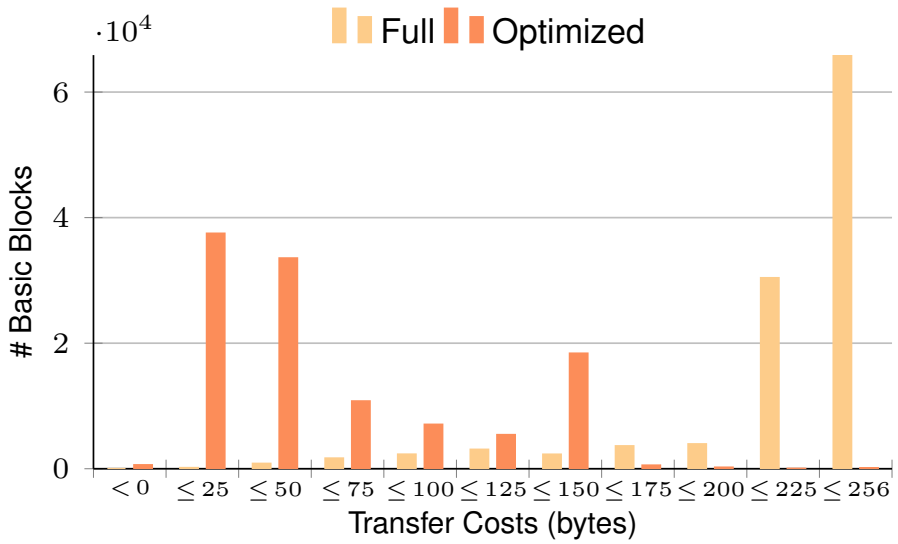


Figure 6.9 – Histogram of transfer sizes (in bytes) for context restoration at basic blocks using max. occupancy (Full) and our approach (Optimized). Lower is better.

placed after each function call. Many of these instructions restore a part of the stack cache context for free, leading to considerable reductions in comparison to a full restoration based on maximum occupancy. In total, the benchmarks consist of 114257 basic blocks of which, 113596 (99.4%) show an improvement. In the mean, over all benchmarks, the improvement is 4.1 fold (min. 3x, max. 7x per benchmark). Figure 6.9 nicely illustrates these improvements. An unoptimized, full restoration typically reloads 250b or more (50281 basic blocks or 44%), while our optimized approach typically only reloads up to 50b (71658 or 67%) with another peak between 126b and 150b.

In many cases no explicit memory transfer is needed at all (49494 or 43.4%), i.e., Equation 6.7 evaluates to 0, while for virtually all other cases the entire local stack frame is explicitly restored (62934 or 56%). Out of the 49494 cases, where no explicit memory transfer is required, 39558 will eventually have their entire stack frame reloaded by a subsequent `sens`. Consequently, in 93.9% of the cases the preemption costs will have to account for the restoration of the current function's entire stack frame (either by a subsequent `sens` or by an explicit memory transfer).

Furthermore, a close look at the minimum and maximum restore costs reveals that there often is no variation with regard to the restoration costs within individual functions. Out of the 8588 functions, 6575 (77%) show no variation at all. The variation for the remaining 2013 functions is illustrated by Figure 6.10, which relates the maximum restoration costs against the minimum. In addition, we show the identity function $f(x) = x$ as a reference. Values close to the shown line indicate low variation. In our measurements, 1287 functions (64%) show an absolute variation below $32b$ and only 218 functions (10%) have a large variation above $64b$. Due to the fact that the minimum restoration costs often evaluates to 0 (1436 functions or 71%), a relative comparison is difficult.

As can be seen in Figure 6.9 and 6.10, even a few cases can be observed where the total restoration cost becomes negative (609 basic blocks or 0.5%), i.e., the program runs faster since the total transfer size to restore the cache content is smaller than the gain due to reduced spilling (cf. Equation 6.15). For 3488 basic blocks a non-zero

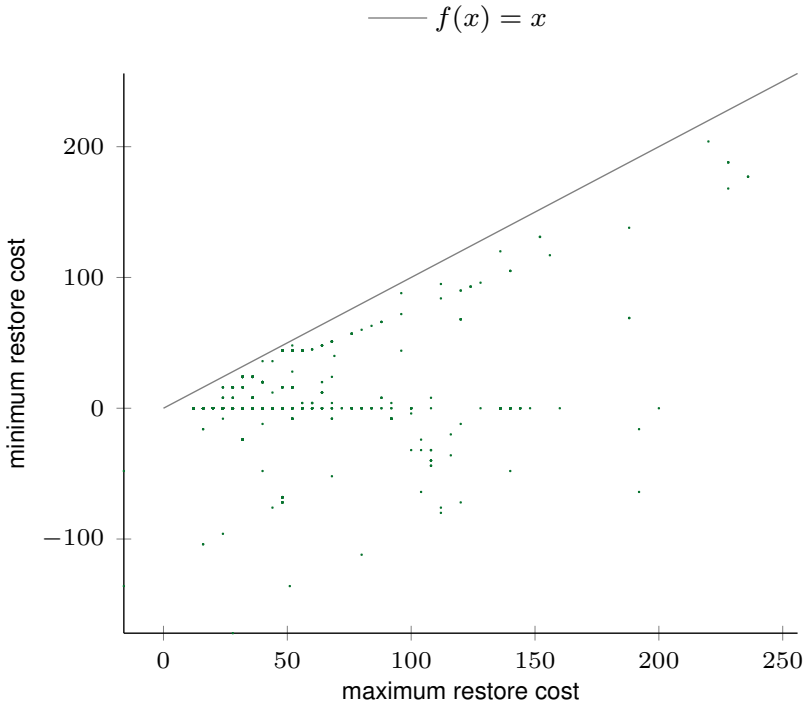


Figure 6.10 – Minimum vs Maximum cost reduction (in bytes) for context restoration at functions. Smaller distance to the reference line is better.

gain due to reduced spilling was found (3%). Our new analysis algorithm improves upon the previous version [13] by 77% with regard to the average local reserve gain and by a factor of 3.72 when considering the local and global gain combined.
Due to the fact that the analysis operates on a very simple domain (integers) and usually only considers individual functions, the analysis time itself is negligible. Also the inter-procedural aspects of the analysis appear to scale well. This particularly applies to the longest path search on the CG required to determine the worst-case restoration cost of `sens` instructions of other functions (Section 4.2 and 4.3). Over all benchmarks only 7 functions out of 1428 require a potentially time-consuming longest path search in a strongly connected component of the CG. For 771 the length of the path is known to be 0 due to the maximum displacement provided by the standard SCA. All other functions are in non-cyclic regions of the CG, which allows us to apply dynamic programming to compute the longest path.

7.2 Context Saving Analysis

Despite the fact that the context saving analysis does not account for inter-procedural effects, it shows consistent improvements over all benchmark programs considered. From 114257 basic blocks in the benchmarks 11618 (10.1%) show a reduction in the context saving overhead. However, the reductions are moderate, as can be seen in the histogram of Figure 6.11. For the basic blocks with lower transfer size, the reduction amounts to 8.9% on average over all benchmarks (minimum 5.9%, maximum 20.7% on average, per benchmark), resulting in a moderate shift in the histogram (from the right to the left). These results are hardly surprising, since the data of all functions currently holding data in the stack cache has to be saved. The reductions are thus much smaller than for the context restoring analysis. It is evidently much harder to

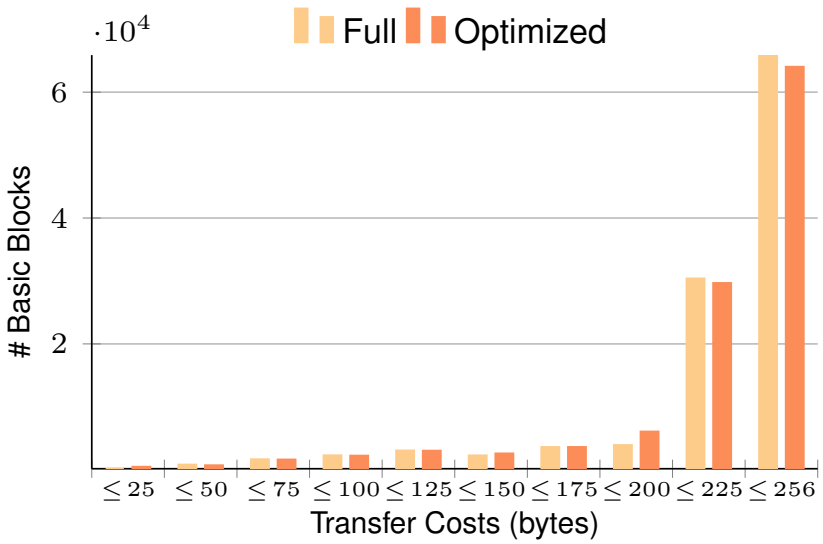


Figure 6.11 – Histogram of transfer sizes (in bytes) for context saving at basic blocks using max. occupancy (Full) and our approach (Optimized) from Section 3. Lower is better.

eliminate the context saving overhead, which unfortunately can have an immediate impact on the Worst-Case Response Time (WCRT) of other tasks.

8 Conclusion

The stack cache exploits the access patterns to stack data, which results in simpler hardware and analysis. Due to its simplicity, the stack cache cannot hold the stack data of different tasks at the same time. The stack cache content thus becomes part of the task's execution context, which has to be saved/restored explicitly during context switching.

We presented a static program analysis to determine the worst-case preemption costs associated with the stack cache during context switching. The analysis is composed of several smaller, function-local data-flow analyses. Inter-procedural effects are handled through variants of the longest path problem. Experiments showed that the analysis complexity is low and that the restoration costs can be reduced heavily, since ensure instructions (`sens`), placed after function calls, often restore the cache context for free. On the other hand, context saving costs appear difficult to eliminate. To mitigate this issue, we present in Chapter 9 an idea to *virtualize* the stack cache. Via the use of the scratchpad memory and TDM bus arbitration, the goal is to ultimately hide the cost of context switch operations while dynamically managing *multiple* stack caches.

CHAPTER 7

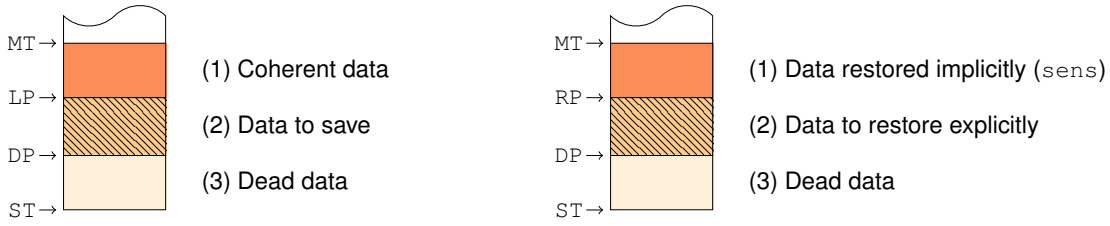
Preemption Mechanisms for the Stack Cache

In the previous chapter, we provided analyses to bound the preemption cost for every instruction in the program. The information generated by the analyses is rich and includes various parameters involved in the preemption cost depending on the precise location of the preemption. However, the potentially large quantity of information, although precise, is of little use if the scheduler cannot use it without inducing considerable overhead. We thus propose in this chapter preemption mechanisms. This chapter is organized as follows. We start with a short introduction, then we present our approach in Section 2. Section 3 provides solutions as to how the mechanisms can be integrated to support different preemption schemes. The experiments are presented in Section 5 before concluding.

1 Outline

The preemption mechanism describes a set of operations that the real-time scheduler has to perform in order to execute a context switch. This includes saving and restoring the processor's register values, resetting the memory management unit (if one exists), as well as re-configuring other shared hardware resources. If a stack cache is present its content needs to be saved and restored explicitly.
Under Patmos, a simple solution has been chosen and implemented in RTEMS [1] [2] – a popular RTOS in avionics. For the context saving, the approach consists first of computing the occupancy of the preempted task `Occ = MT - ST`. Then, the entire content of the stack cache is saved by performing an **`sres`** `|SC|`, which temporarily allocates a *stack frame* of the total size of the stack cache. This causes the entire content of the stack cache (i.e., occupancy) to be spilled to main memory and thus be saved. Finally, the occupancy and special registers are saved including the `MT` pointer which now holds the stack top address of the preempted task. On the other hand, the context restoration consists first of restoring the special registers which include the `MT` pointer. The stack cache content is restored using **`sens`** `Occ`, which

[1] `https://www.rtems.org`
[2] `https://github.com/t-crest/rtems`



(a) Context saving. (b) Context restoring.

Figure 7.1 – Partitioning of the stack cache for context switch operations.

fills the cache starting from the address in the MT pointer.
This approach, although simple, does not benefit from the context switch optimizations introduced in the previous chapter. In fact, this is equivalent to the *simple approach* against which we compared our *optimized approach* which showed a significant improvement of preemption costs (see Section 7 of Chapter 6). The question therefore is how to make preemption mechanisms benefit from these improvements without inducing unreasonable overhead? Moreover, how can these mechanisms profit from different preemption schemes (i.e., fixed and non-fixed preemptions)? This chapter provides our answer to these questions.

2 Preemption Mechanisms

Before introducing the preemption mechanisms, let us first recall how our approach optimizes the context switch operation. The stack cache occupancy is split into different regions depending on the context switch operation. The idea is to save/restore only the *useful* data at program locations. We thus define two partitionings depending on the operation:

Context Saving: Two pointers are used for the partitioning. The LP pointer, (see [12]), keeps track of data that is coherent with the main memory, and the DP pointer tracks data that is known to be dead. The LP and DP define a partitioning of the stack cache's content into three distinct regions shown in Figure 7.1a. Data above LP is coherent and thus can be ignored during context saving. Similarly, data below DP is known to be dead and can safely be ignored. Only the remaining data, between DP and LP, actually needs to be transferred to main memory.

Context Restoring: Two pointers are involved as well. The DP pointer tracks certainly dead data, while the RP pointer tracks potentially live data in the stack cache. Three regions can thus be defined as illustrated in Figure 7.1. Data below RP is potentially live and not guaranteed to be restored by a subsequent sens instruction. Similarly, data below DP contains data that is certainly dead. The remaining data to be transfered from main memory is then located between DP and RP pointers.

From this, we notice that the context saving operation relies on the LP and DP pointers while the context restoring depends on RP and DP pointers. We now introduce in Figure 7.2 assembly code allowing to explicitly save and restore the content of the stack cache:

```
func SCSave(DP)
 sub rx = |SC| - DP
 sresr rx
 stw MT
```

(a) SCSave.

```
func SCRestore(RP,DP)
 ldw MT
 mov ST = MT
 mov LP = MT
 sub rx = RP - DP
 sensr rx
 mov ry = DP
 sresr ry
```

(b) SCRestore.

Figure 7.2 – Low-level functions to save/restore the stack cache content.

SCSave: The function saves the content of the stack cache to main memory, depending on the size of the stack cache and the amount of dead data. We assume that the stack cache size is known statically ($|SC|$), while the amount of dead data is assumed to be a parameter of the function (DP). The entire content of the stack cache can be saved by performing an **sres** |SC|. This automatically avoids the spilling of coherent data (LP). However, dead data would be saved as well. The SCSave function thus subtracts the size of the dead data from the stack cache size (**sub**) and stores the difference in a register (rx). The value of this register is then used by an **sresr** instruction, which is equivalent to a regular reserve with the only difference that the instruction's argument is a register. Finally, we save the MT pointer (**stw**), which points to an address in main memory where all of the current task's stack data is now saved (excluding dead data at the bottom). This address is later needed during context restoration.

SCRestore: The function restores the stack cache content after a preemption and, for this, requires the RP and DP pointers as arguments. Before any stack data can be transferred to the cache, the previously saved MT pointer needs to be reloaded first (**ldw**). The ST and LP pointers are set to the same address, which represents an empty stack cache. Then we proceed to the restoration by explicitly filling cache blocks between RP and DP using an **sensr** instruction. As before, an **sensr** takes its argument from a register (rx). The value of the register is computed from the difference between the two arguments RP and DP (**sub**). Note that we assume here that RP $\geq$ DP in order to simplify the assembly code. The ensure transfers live data from main memory and updates the MT pointer. Neither the LP nor the ST is modified since the reloaded data is known to be coherent. Finally, we take care of dead data, which was excluded from context saving before. It suffices to allocate a matching amount of cache blocks on the stack cache using an **sresr** instruction. This ensures that subsequent accesses to the stack cache succeed, while avoiding a useless memory transfer.

With the help of the SCSave and SCRestore functions any desired context saving mechanism can be implemented. One only has to ensure that the functions are called at the right moment and do not cause any side-effects, e.g., on registers of the involved tasks. However, it remains to resolve one issue: how do the functions obtain the parameter values for DP and RP?
One possible, but impractical solution, would be to store these parameter values in a look-up table. It would then be possible to retrieve the precise values of both pointers, as determined by the analysis, during context switching. The memory

footprint of the table as well as the costs associated with the table look-up disqualify this solution. Therefore, we need means to exploit the rich analysis information without impacting the predictability or inducing excessive overhead. This, however, highly depends on the underlying preemption policy.

3 Handling Preemption Schemes

In combination with a scheduling algorithm, the preemption policy determines the circumstances under which a running task is allowed to be preempted. For instance, the fully-preemptive policy allows preemptions to occur freely at any time and at any position in a task's program. Whereas the non-preemptive policy does not allow any preemption to occur, and a task is started only after a running task is terminated. Although fully-preemptive approaches may offer better schedulability, they make it difficult to provide tight WCET estimates using cache analyses, since the preemption point is not known in advance. Hybrid approaches have been introduced to tackle this problem, either by statically fixing preemption points or limiting the number of preemptions that tasks may suffer. Examples of such approaches include the deferred preemption model [28], the floating non-preemptive region model [19], or the preemption threshold [124]. So, in order to provide a preemption mechanism that best matches a preemption policy, it is of major importance to consider whether the policy relies on statically fixed or non-fixed preemption points.

3.1 Fixed Preemption Schemes

The main advantage of fixed preemption points is that it gives precise control over the execution of real-time programs. It is a powerful approach allowing the preemption mechanism to take full advantage of all the capabilities of a cache analysis. This allows to choose interesting preemption points within a single task depending on the overhead determined by a cost analysis (considering, among others, cache analyses such as the CSA and CRA). These candidate preemption points are then considered globally by schedulability tests to ensure that the constraints of the entire system are respected. This may help to reduce the overhead due to preemption with regard to the global system utilization.
Once preemption points are chosen the full results of the previously described analyses (CSA and CRA) can be exploited easily, since dedicated code triggering a context switch can be inserted. This code may simply invoke the `SCSave` function before yielding the processor to the operating system kernel. Once the task is reactivated it suffices to call `SCRestore`. In both cases the function's parameters are immediately available and can be considered by the inserted code. This, furthermore, allows the WCET analysis to include the respective code and its overhead. A downside of this approach is, however, that the stack cache content is potentially transferred to/from main memory even when the operating system decides not to preempt the running task. Alternatively, the two functions could be implemented in the operating system, which then invokes them as needed. The interface between the task and the operating system then needs to be revised such that the task can communicate the `DP` and `RP` parameters when yielding the processor, e.g., by explicitly setting predefined registers.

3.2 Non-Fixed Preemption Schemes

In contrast to the previous strategy, handling non-fixed preemption approaches is quite challenging as preemption locations are not known in advance. This means that the preemption mechanism cannot pass, as described above, predetermined parameter values for DP and RP to the SCSave and SCRestore functions respectively. One solution, already mentioned before, is to store the parameter values in a look-up table. The operating system would then simply retrieve the parameter values considering the precise location of the preemption, e.g., by using the task's program counter as an index. The size of the look-up table inevitably disqualifies this solution. However, it might be possible to compress the table or reduce its size. For example, the table size could be reduced by storing only a single value for each of the two pointers for each function in the program, instead of storing the pointers for *all* possible program points. This would drastically decrease the table size at the expense of a slight increase of look-up costs. The values stored for each function have to be safe approximations. For the RP the maximum value over all program points in the function has to be chosen, while for the DP the minimum value has to be selected. This solution still appears impractical. However, the idea to attach approximations to limited regions within a program can be generalized.
In the following, we will present two solutions to this problem, based on lightweight extensions to the hardware and/or instruction set. The first solution relies on conservative approximations at the granularity of whole functions using an additional stack cache control register. The second solution requires a modification of the instruction set, which allows to embed analysis information in the standard stack cache control instructions.

Stack Cache Control Register

This solution is motivated by our experiments, which are explained in detail in Section 7 of Chapter 6. Our measurements indicate, not too surprisingly, that in most of the cases the stack frame of the current function needs to be restored entirely, either by an explicit memory transfer (RP − DP) or an implicit memory transfer (through a subsequent sens). The preemption mechanism may thus simply restore the entire stack frame of the function where the preemption occurred, since the overhead for this operation already has been accounted for in any case. The parameters DP and RP for the SCSave and SCRestore functions are then simply approximated by 0 and k, respectively, where k is the size of the function's stack frame.
The problem is that the standard stack cache does not track the size of the current stack frame. It only *knows* the ST and MT pointers representing the occupancy. The occupancy may reflect three different situations. Firstly, the stack cache only holds a subset of the frame. The occupancy thus is smaller than k. Secondly, the stack cache contains only the frame. The occupancy here matches k. Finally, the stack cache may hold data of other functions in addition to the frame. The occupancy then is larger than k. It is obviously not possible to derive the size of the current frame from the state of a standard stack cache.
We thus propose to introduce an additional stack cache control register FP that keeps track of the size of the current stack frame. The stack cache control instructions are then required to keep this register up-to-date. The sres and sens instructions

simply copy the value of their respective arguments into this register. On the other hand, `sfree` instructions merely reset the register to 0, since they destroy the stack frame and no stack cache access may occur until the next ensure.
The preemption mechanism may then retrieve the value of the `FP` register and pass it as the parameter `RP` to the `SCRestore` function. The `DP` parameter is conservatively set to 0 for both, the `SCSave` and the `SCRestore` functions.
This solution only requires minimal modifications to the stack cache hardware. The additional `FP` register and the logic needed to update it is negligible and thus has virtually no impact on the hardware cost and clock frequency. The timing behavior of instructions is not modified as the register update can be performed in parallel with other operations in a single cycle. The time-predictable behavior of the stack cache is thus preserved. Finally, the solution does not incur any overhead whatsoever with respect to the program's memory footprint or execution time. A shortcoming of this approach is that the value of the `RP` parameter is frequently overestimated, while the `DP` parameter is not exploited at all. The approach thus effectively discards all information regarding function-local analyses.

Instruction Set Extension

An alternative approach is to modify the stack cache control instructions, such that they can be used to piggy-back the analysis information. The basic idea is to add two additional operands to the `sres` and `sens` instructions that explicitly specify the values of the `DP` and `RP` parameters. The values of these operands are copied into two dedicated stack cache control registers, which then can be consulted by the preemption mechanism to invoke the `SCSave` and `SCRestore` functions. The `sfree` instruction does not receive additional operands and instead simply resets the two control registers to 0. This allows to express changing values of the parameters at a much finer level of granularity, independently from the size of the stack frame. More specifically, the operand values apply to all program points between two successive stack cache control instructions. The operand values can easily be computed by considering the maximum value for the `RP` and the minimum value for the `DP` in the corresponding region of the program. Function calls can be ignored in this computation, as will be explained below.
In order to illustrate the approach we consider two scenarios of successive instructions: an `sres` followed by an `sres` of another function (callee) and an `sfree` followed by an `sens` instruction of another function (caller).
In the first case, the operand values of the first `sres` instruction apply to all program points up to the execution of the second `sres` instruction, which automatically overrides the corresponding control registers. Note, in particular, that this includes all program points in the called function before its `sres` (see Section 6 of Chapter 6). This is safe since these instructions cannot have any impact on the stack cache. Consequently, the operand values of the first `sres` can be computed by considering local program points belonging to the same function as the reserve, i.e., calls can be ignored. Furthermore, all instructions between a function call and its `sens` instruction cannot have an impact on the stack cache either. It is thus safe to consider all function-local program points between any two subsequent stack cache control instructions in order to compute the operands. Note that this reasoning also applies to other pairs of instructions, i.e., `sens-sens`, et cetera.

In the latter case, the sfree instruction destroys the stack frame of the current function. Implicitly, the stack frame of the caller now becomes active. However, at the moment when the sfree instruction is executed, the characteristics of the caller's stack frame are not known and thus cannot be embedded as operands in the free instruction. We solve this issue by simply resetting both control registers for the DP and the RP to 0. This means that, from the perspective of the preemption mechanism, the instructions between an sfree and the subsequent sens belong to the callee, which slightly differs from the model explained in Section 6 of Chapter 6. The preemption cost bound remains safe under this interpretation, since the implicit filling at the sens instruction is correctly accounted for by the global ensure costs. The hardware overhead of this solution, again, is marginal, since only two additional registers as well as logic to update them are needed. The impact on the hardware cost and clock frequency remains negligible. Also, the timing behavior of the stack cache control instructions does not change, preserving the time-predictability of the stack cache. However, the additional operands require space in the instruction encoding, which may either increase the instruction size or otherwise impose limits on the maximum stack frame size – depending on the characteristics of the instruction set architectures and the number of free bits in the original instruction encoding of the stack cache control instructions. Assuming that the operands can be encoded using otherwise unused bits, this solution does not impose any overhead w.r.t. the program's memory footprint or execution time.

4 Experiments

Now we evaluate the impact of the preemption mechanisms, as described in Section 2, on the analysis of preemption costs we covered in Chapter 6. Recall that the preemption mechanism invokes the SCSave and SCRestore functions, which require two input parameters: the amount of dead data and the amount of data that needs to be restored explicitly, which are denoted as DP and RP respectively. For the following experiments we consider three different implementation variants: (1) ISA-full, which is based on an instruction set extension that allows to specify two additional operands for both, the DP and the RP parameters, (2) ISA-RP, an instruction set extension covering only the RP parameter, and (3) FP, which represents a solution based on a single stack cache control register (FP) holding the current stack frame size. These configurations are compared against the optimized configuration from the previous experiments, representing the most precise preemption costs provided by an optimized analysis. Note that the results of the optimized analysis can be used in the setting of fixed preemption points.
An obvious difference between these preemption mechanisms is the level of granularity. The optimized analysis is able to compute precise results for each instruction in the program. In the previous experiments the analysis results were, however, limited to the beginning of basic blocks. Over all benchmark programs the number of basic blocks amounts to 114257 (each potentially consisting of multiple instructions). As a reference, the ISA-full and ISA-RP variants operate at a coarser level of granularity, only considering stack cache control instructions. The number of these instructions depends on the number of defined functions and call sites in the program. In the considered benchmarks 8588 functions are defined, which are referenced by 10475 call sites. In total 27651 stack cache control instructions can be found in all benchmark

programs combined (two for each function and one for each call site). The number of locations that may reflect changes in the underlying analysis information is thus reduced to about a quarter (24%). The FP variant essentially operates at the level of whole functions. This reduces the level of granularity even further to 8588 (8%) different locations.

In order to evaluate the transfer costs induced by the various mechanisms, a common level of granularity has to be chosen. The basic block level seems to be a reasonable choice, as it allows to demonstrate the performance of the underlying preemption mechanisms at a tight granularity and allows us to easily compare them against the optimized approach. However, it is important to note that all three preemption mechanisms have a diverging interpretation of the transfer costs when returning from a function. More precisely, the costs associated with the program points between an `sfree` of the callee and the corresponding `sens` of the caller differ from the optimized analysis. These program points rarely coincide with the beginning of basic blocks, since call instructions are not considered terminators for basic blocks in LLVM. The impact of this design choice is thus not explicitly captured by the presented numbers. At the same time, the concerned regions at most contain two program points, one before and one after the corresponding return instruction. The compiler also often manages to put the `sfree` instruction in the return's delay slot, which only leaves a single program point between the `sfree` and the `sens` executed immediately afterward.

Starting with the context restoration, we first observe that the proposed preemption mechanisms overall follow a similar pattern as the optimized approach, as illustrated by Figure 7.3. The ISA-full and ISA-RP variants perform slightly better, showing only a moderate degradation in terms of precision. The difference between these two variants is insignificant, which indicates that the `RP` pointer is more profitable than the `DP`. This is not surprising, as the `RP` is regularly reset to 0 at every `sens` instruction, which limits the propagation of high `RP` values throughout large parts of a function. This is different from the `DP`, whose value evolves depending on stack cache accesses only and thus might be propagated throughout large parts of a function. A closer look reveals that, for these two approaches, the context restoration

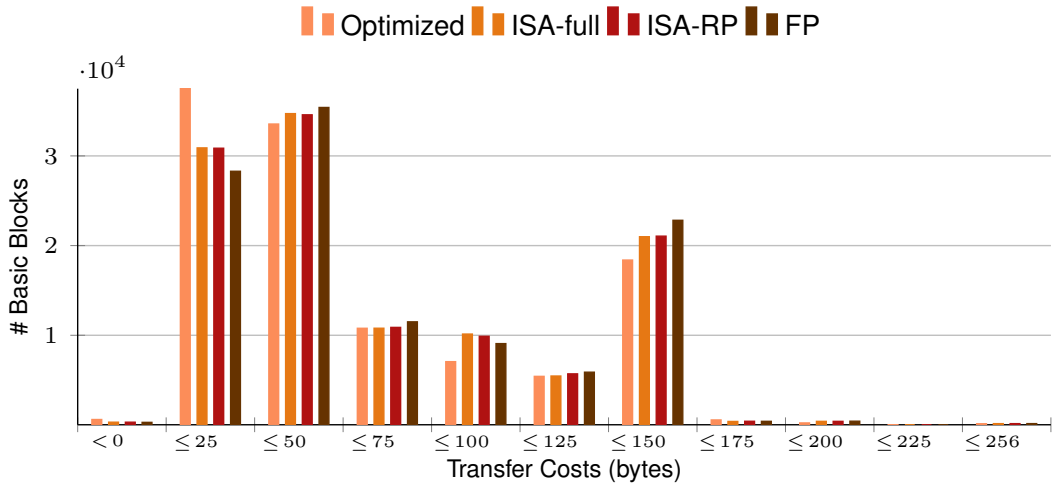


Figure 7.3 – Histogram comparing the transfer sizes (in bytes) for context restoration at basic blocks using the ISA-full, ISA-RP, and FP preemption mechanisms to the optimized analysis. Lower is better.

	<0	≤ 25	≤ 50	≤ 75	≤ 100	≤ 125	≤ 150	≤ 175	≤ 200	≤ 225	≤ 256
<0	-318	5	53	103	56	75	25	0	1	0	0
≤ 25	–	-9211	4566	1251	1177	188	2027	1	1	0	0
≤ 50	–	–	-2773	706	1618	45	404	0	0	0	0
≤ 75	–	–	–	-1342	461	111	759	0	11	0	0
≤ 100	–	–	–	–	-1301	772	529	0	0	0	0
≤ 125	–	–	–	–	–	-726	698	28	0	0	0
≤ 150	–	–	–	–	–	–	-9	3	6	0	0
≤ 175	–	–	–	–	–	–	–	-185	182	0	3
≤ 200	–	–	–	–	–	–	–	–	-7	0	7
≤ 225	–	–	–	–	–	–	–	–	–	-6	6
≤ 256	–	–	–	–	–	–	–	–	–	–	0

Table 7.1 – Increase of restoration cost for the FP preemption mechanism in comparison to the optimized analysis, illustrating the movement of basic blocks to the right side of the histogram in Figure 7.3. Smaller numbers are better.

cost remains below 50b in the majority of the cases (57%). In only 0.8% of the basic blocks the context restoration exceeds 150b – this almost matches the optimized analysis. A noticeable drop is, however, observed for cases with very low transfer costs between 0b and 25b. The drop amounts to 6600 basic blocks, which represents about 21% of the 30894 basic blocks in that cost range for the optimized analysis. The transfer costs of the respective basic blocks slightly increase, which corresponds to a slight shift to the right and explains the increased bar heights nearby. For instance, the restoration costs for 53% of these 6600 basic blocks now fall into the next higher cost range (26b to 50b) and another 24% of the blocks fall into the cost ranges after that (51b to 125b). The costs of the remaining cases then fall into the range from 126b to 150b. This indicates a moderate loss of precision, which is mainly due to the coarser granularity of these two approaches. The two approaches also succeed to conserve nearly 50% of the cases with negative restoration costs, i.e., reflecting potential runtime gains.

The FP approach generally follows the same trends. The shift in the diagram from the left side to the right is albeit more pronounced. The drop for the cost range from 0b to 25b amounts to 9211 basic blocks (30%). About half of the basic blocks appear in the next higher cost range (26b to 50b), while 28% of the cases can be found in the cost range from 51b to 125b. The remaining basic blocks (22%) move into the cost range above 125b, with two cases falling into the range from 151*b* to 200b. Despite the fact that the relative numbers appear to be close to the instruction-set-based preemption mechanisms, the absolute numbers are considerably more pronounced. This explains, for instance, the noticeable peak for the cost range from 126*b* to 150b for this preemption mechanism. A detailed overview of the movements between the different cost ranges is illustrated by Table 7.1. The negative numbers on the diagonal indicate the number of basic blocks whose restoration costs were increased, while the positive numbers indicate to which cost range these basic blocks moved.

As for context saving, we can observe a very slight shift to the right for the ISA-full approach, as can be seen in Figure 7.4. This mainly concerns 1141 basic blocks, whose transfer costs already were high (between 176b and 200b) even for the optimized analysis. The precision loss here mostly stems from the `DP` pointer, which suffers

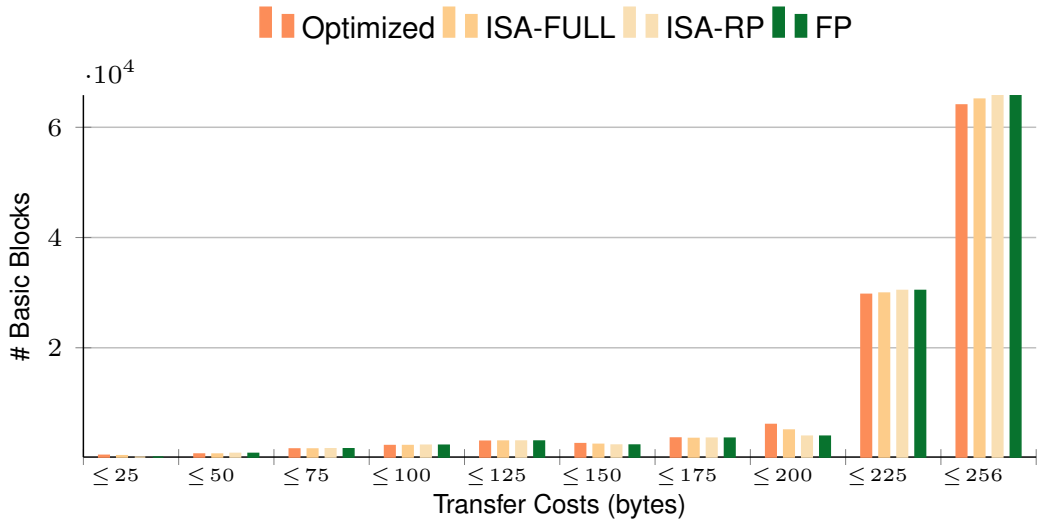


Figure 7.4 – Histogram comparing the transfer sizes (in bytes) for context saving at basic blocks using the ISA-full, ISA-RP, and FP preemption mechanisms to the optimized analysis. Lower is better.

from the previously mentioned propagation of unfavorable values throughout the coarser regions. This shift is more pronounced for the ISA-RP and FP variants, since both do not exploit the `DP` parameter (which is conservatively set to 0). The number of basic blocks impacted almost doubles (2180). This also applies to other cost ranges and here in particular the range from 201*b* to 225b. A detailed breakdown of the movements in the histogram is given by Table 7.2.
From the results above, one can conclude that ISA-full provides some advantage over the other two mechanisms as it allows to exploit (at least to some degree) the analysis information concerning dead data. On the downside, the instruction-set-based approaches require additional changes to the hardware, the instruction set, as well as the compiler. In particular, the changes to the encoding of the stack cache control instructions might be problematic in practice. We will explore this issue using the Patmos processor and its instruction set as an example. Patmos instructions are encoded either using 64 bits or 32 bits depending on the corresponding instruction formats. The 64-bit format is dedicated to simple arithmetic instructions with a full 32-bit *long* immediate. All stack cache control instructions are encoded according to the 32-bit-wide Stack Control format (STC), which reserves 1 bit to indicate bundled (VLIW) instructions, 4 bits for the predicate operand, 5 bits to indicate the instruction format, and additional 4 bits for the instruction opcode. Consequently, 18 of the 32 bits are used to encode the instruction's operand (either an immediate or register index for the standard stack cache). Assuming a cache block size of 4b, this allows the stack cache to manage stack frame sizes of up to 1MB, which appears generous for most embedded applications. In order to implement the proposed instruction set extensions for the ISA-full and ISA-RP preemption mechanisms these 18 bits need to be split between either 3 (`RP`, `DP`, `k`) or 2 (`RP`, `k`) operands, respectively. An even distribution would then either leave 6 or 9 bits for each operand. This would reduce the maximum stack frame size, but not the total stack cache size, to 256b or 2KB. The limit of 256b is sufficient for the experiments conducted here. Even when the stack cache size is essentially unbounded, most of the considered benchmarks exhibit a maximum stack frame size on the stack cache of 140b, while

	≤ 25	≤ 50	≤ 75	≤ 100	≤ 125	≤ 150	≤ 175	≤ 200	≤ 225	≤ 256
≤ 25	-318	145	156	17	0	0	0	0	0	0
≤ 50	–	-34	4	8	19	3	0	0	0	0
≤ 75	–	–	-122	113	4	0	0	1	0	4
≤ 100	–	–	–	-77	51	11	3	0	10	2
≤ 125	–	–	–	–	-41	38	2	1	0	0
≤ 150	–	–	–	–	–	-307	296	1	2	8
≤ 175	–	–	–	–	–	–	-332	51	281	0
≤ 200	–	–	–	–	–	–	–	-2180	2082	98
≤ 225	–	–	–	–	–	–	–	–	-1668	1668
≤ 256	–	–	–	–	–	–	–	–	–	0

Table 7.2 – Increase of saving cost for the FP preemption mechanism in comparison to the optimized analysis, illustrating the movement of basic blocks to the right side of the histogram in Figure 7.4. Smaller numbers are better.

the largest stack frame encountered is merely $240b$ large. In a general setting, this restriction might, however, become limiting. Increasing the cache block size might remedy this problem. The limit of 2KB, on the other hand, appears to be practical even for large embedded systems. The FP variant, based only on an additional internal stack cache control register, does not face such restrictions and might thus be easier to use in such larger systems. Overall, all of the three proposed preemption mechanisms appear to be practical.

4.1 Hardware Implementation

We implemented all of the aforementioned hardware extensions in a Patmos hardware model. From the original model, specified in Scala, hardware is synthesized using the Altera Quartus II 13.1 tool suite for an an Altera DE2-115 board.
The implementation of the FP preemption mechanism requires an additional special register as well as some logic circuits that are needed to keep track of the current stack frame size. In particular, `sres` and `sens` instructions have to copy their argument to this special register, while `sfree` reset the register to zero. Only a dozen of code lines were needed to extend the stack cache model (originally about 500 code lines). Ignoring other components of the processor core, the hardware overhead in comparison to the original stack cache design is very minimal and is evaluated to $2,2\%$ and $3,5\%$ in logic cells and logic registers, respectively. We also looked at the resulting overhead at the core level, which includes, among others, the computational units, a stack cache, a data cache, an instruction cache, and a local scratchpad memory. Once again, the hardware overhead is negligible and costs around $0,6\%$ and $0,1\%$ in logic cells and logic registers, respectively. The impact on the maximum clock frequency, on the other hand, is surprisingly positive. We observed a slight improvement from 82 MHz to 83 MHz. Note that this improvement may be caused by some slight change in the complex heuristics employed by the synthesis software. The overhead of the ISA-full and ISA-RP approaches is comparable to that of the FP approach and only require some additional logic registers. The required changes are in fact virtually identical, since the number of instruction operands has no relevant impact on the hardware level.

In conclusion, the implementation of the proposed hardware extensions is very simple and only incurs insignificant additional hardware costs.

5 Conclusion

We proposed three different preemption mechanisms that allow the task scheduler to exploit the analysis information during context switching. Two of these mechanisms are based on an instruction set extension that attaches analysis information as operands to the stack cache control instructions. In comparison to the full static analysis these mechanisms may reflect changes in the analysis information at a much coarser level. Our experiments nevertheless showed that only a moderate loss in precision is incurred. A downside of these approaches is, however, that, in addition to support from the operating system, modifications to the compiler are required. An alternative solution relies solely on an additional stack cache control register that is updated by the stack cache control instructions in a transparent manner. Apart from the modifications to the task scheduler no additional tool support is required. This, however, comes at a cost: the granularity at which changes in the underlying analysis information can be reflected is limited to whole functions. This incurs an additional loss in precision.

CHAPTER 8

Eager Stack Cache Memory Transfers – A Prefetching-Like Technique for the Stack Cache

In Chapter 6, we observed a non-intuitive situation, where a preemption can potentially reduce spilling costs associated with subsequent `sres` *instructions. The benefit comes essentially from an early spilling due to context saving. This observation has set us on the following direction: What if we could anticipate and eagerly perform the spilling/filling before they are actually needed? This idea reassembles prefetching in conventional caches. Except that, by contrast, the stack cache does not operate on addresses and all its accesses are hits. Clearly, the stack cache structure has some opportunities to offer, but one needs also to determine possible side effects. What's more, how do eager memory transfers affect the stack cache analysis? We, thus, explore means to perform prefetching-like operations for the stack cache, and study their outcome from the perspective of WCET analysis. After a short introduction, the chapter is structured as follows: The general idea as well as a detailed description of the technique are provided in Section 2. In Section 3, we present some arbitration strategies to which we can associate the eager transfers to. Experiments are presented in Section 4 followed by conclusions.*

1 Outline

In order to improve predictability and ensure composability, the original stack cache design [11] stalls the processor while performing spilling or filling, even when the stack cache would not be used by any of the subsequent instructions. This allows to analyze the stack cache's timing behavior in isolation from other components of the Patmos computer architecture [111] at the expense of average-case performance. In this work, we explore the use of eager – or anticipatory – memory transfers in order to alleviate this shortcoming. The goal is to improve average-case performance by performing memory transfers in the background alongside with other instructions that are executed by the processor. The eager transfers are, however, not allowed to interfere with the worst-case behavior of the stack cache (or any other hardware component in the system). Most notably, the timing bounds computed for a regular

stack cache without our optimizations, should not be invalidated in the presence of our optimizations. This is ensured by exploiting features of a recently proposed stack cache extension [12] to track data that are coherent between the cache and main memory.

2 Eager Memory Transfers

Prefetching is a well-known technique used in conventional caches, which aims to hide memory access latencies caused by cache misses. Instead of waiting for a cache miss to initiate a memory transfer, prefetching anticipates such misses and fetches data from memory in advance of the actual memory reference. The idea, though simple, raises two important problems: (1) the addresses of future memory references need to be predicted and (2) side effects may arise due to the eviction of data from the cache in order to make space. Both of these issues are difficult to solve in general settings and pose even more problems in the context of real-time systems requiring predictability.
We explore the use of *eager memory transfers* – combining prefetching and eager eviction [71] – in order to reduce the latency of the stack cache control instructions. We introduce two kinds of eager memory transfers: (1) *eager spilling* transfers data from the stack cache to main memory, while (2) *eager filling* transfers data from main memory to the stack cache. The stack cache, in contrast to conventional caches, tracks its content using simple pointers and thus can only cache a contiguous memory region between the `ST` and `MT` pointers. In the following we will exploit this feature in order to realize "prefetching-like" functionality for the stack cache and address the two aforementioned problems faced in standard caches.

Address Prediction: Due to the use of pointers to track the stack cache content, it is trivial to predict the address of any future memory transfers that might be initiated by any stack cache control instruction. Data is either read from memory at the address starting at `MT` or written to memory at the address up to `LP`, depending on whether the (effective) occupancy will grow too large (`sres` spilling up to `LP`) or will become too small (`sens` filling from `MT`). It thus suffices to predict whether data needs to be spilled or filled with regard to the future stack cache control instructions.

Side effects: We rely on a recently proposed stack cache extension [12] that allows to track coherent data between the stack cache and main memory in order to avoid side effects when performing eager memory transfers. A first observation is that eager spilling only needs to consider incoherent data (just like regular spilling). The eagerly spilled data is, however, not evicted from the stack cache. Instead, it simply becomes coherent. Since no data was evicted from the cache, side effects on future `sens` instructions are excluded. Similarly, since the amount of incoherent data was reduced, the spilling at future `sres` instructions is potentially reduced. A second observation is that eagerly filled data is known to be coherent. Side effects on future `sres` instructions are consequently excluded after eager filling since the amount of incoherent data did not change. The filling at future `sens` instructions, on the other hand, is reduced due to the newly loaded data.

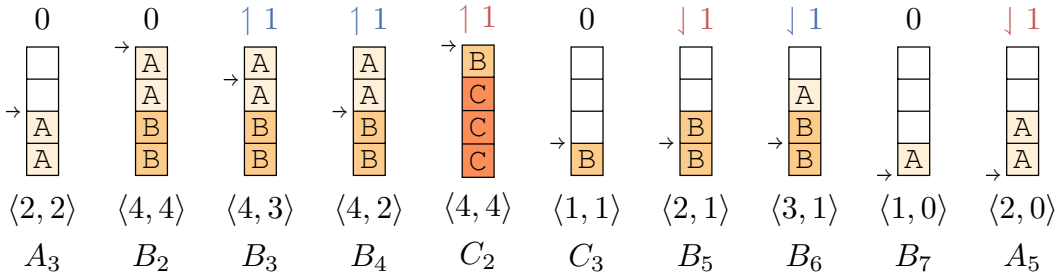
Example 2.1 *Consider functions* `A`*,* `B`*, and* `C` *shown in Figure 8.1a and a stack cache whose size is 4 blocks, i.e.,* $|SC| = 4$*. Figure 8.1b depicts the evolution of the*

```
(A1) func A()
(A2)   sres 2 ⟨0⟩
(A3)   sws 1 = r9
(A4)   call B
(A5)   sens 2 ⟨2⟩
(A6)   sfree 2
```

```
(B1) func B()
(B2)   sres 2 ⟨0⟩
(B3)   ...
(B4)   call C
(B5)   sens 2 ⟨1⟩
(B6)   ...
(B7)   sfree 2
```

```
(C1) func C()
(C2)   sres 3 ⟨3⟩
(C3)   sfree 3
```

(a) Program consisting of 3 functions, reserving, freeing and ensuring space on the stack cache. The annotations in angle brackets, e.g., ⟨2⟩, indicate the maximum filling/spilling behavior of stack cache control instructions.



(b) The evolution of the stack cache state through the program. Assumed cache size: 4. ST is fixed at the bottom of the cache state. MT points to the top stack elements in main memory (not shown). The arrow → indicates the position of LP in the cache space. Memory transfers in cache blocks are represented at the top of the cache state: Fill (↓), Spill (↑). Eager memory transfers are blue colored. Transfers initiated by stack cache control instructions are red colored.

Figure 8.1 – Example of eager memory transfers.

stack cache state at particular program points. For this example, we assume that an eager memory transfer can be performed in 1 *cycle and that no other hardware component interferes with the stack cache (e.g., data cache, method cache). We also assume an empty stack cache at the program entry point (i.e.,* MT = ST = LP*). The* `sws` *at* A_3 *stores some value in the stack cache such that the cache content is not coherent with main memory. At this point the effective occupancy is equal to the cache occupancy. An eager spill operation is initiated at program point* B_3*. This causes* LP *to decrement, which reduces the effective occupancy to* 3*, but not the cache occupancy (i.e.,* 4*). Then, another eager spill operation is performed at program point* B_4*, which further drops the effective occupancy to* 2*. Without the eager spilling, the* `sres` *instruction at* C_2 *would have spilled* 3 *cache blocks. However, as the effective occupancy was reduced only* 1 *cache block needs to be spilled. Note that the eager spilling has no side effect on the* `sens` *instruction as it does not alter the cache occupancy. The* `sens` *fills* 1 *cache block just as predicted by the stack cache analysis. In contrast to eager spilling, the eager filling increases the cache occupancy but not the effective occupancy. The eagerly filled cache block in* B_6 *only increases the* MT *pointer which results in increasing the cache occupancy to* 3*. As a result, instead of filling* A*'s the whole stack frame (*2 *cache blocks), only* 1 *cache block is needed to be filled. Similarly to eager spilling, the eager filling does not cause any side effects for subsequent* `sres` *instructions as the effective occupancy remains unchanged.*

The eager memory transfers are guaranteed to have no side effects on the stack cache itself. However, side effects on other hardware components, and here in particular

```
if (MT - ST < |SC|) {
  start = MT;
  end = ⌊(MT+BS)/BS⌋ × BS;
  fill(start, end);
  MT = end;
}
```

(a) The eager fill operation.

```
if (LP − ST < k) {
  end = LP;
  start = ⌈(LP−BS)/BS⌉ × BS;
  spill(start, end);
  LP = start;
}
```

(b) The eager spill operation.

Figure 8.2 – Pseudo code illustrating the operation of the eager filling and eager spilling.

the bus and main memory, may arise. For instance, a cache for regular data might be blocked by an eager memory transfer upon a cache miss. Such interferences may, of course, impact the program's worst-case performance and compromise predictability as well as composability.
An elegant solution is to exploit the arbitration scheme that mitigates between competing memory accesses [49, 13]. In the context of this work, we use the Patmos multi-core architecture, which relies on time-division multiplexing (TDM) to arbitrate main memory accesses. In the following we assume that each processor core may transfer a single memory burst from/to main memory in a dedicated *TDM slot*. Transfers may only be initiated at the beginning of a TDM slot, which are periodically scheduled in a *TDM period*. The duration of a period then depends on the number of cores n and the duration of a TDM slot k and is given by $n \cdot k$ cycles. We assume that the memory controller is able to process transfers with arbitrary start addresses and lengths. The actual memory transfer is, however, performed at the granularity of bursts, i.e., the start address and length are aligned accordingly to the burst size (excess data is either masked or discarded). In such a setting it is easy to detect TDM slots that are not used by any other hardware component. It suffices to check that no other memory request is pending at the beginning of the processor core's TDM slot. The free TDM slots of a processor can then be used to perform the eager memory transfers and avoid any side effects on either the stack cache itself nor any other hardware component.

2.1 Eager Fill

The *eager fill* operation aims to reduce the latency of a future ensure instruction `sens` k. Recall that filling is required only when the occupancy is too small, i.e., $\texttt{MT} - \texttt{ST} < k$. The occupancy has to be increased in order to reduce the latency. This can be achieved by loading, i.e., filling, data from main memory such that `MT` can be pushed upwards until the occupancy reaches the stack cache size. The corresponding memory transfer, however, has to be limited to a single burst transfer in order to guarantee that only a single TDM slot is occupied. Assuming a burst size `BS`, an eager fill operation thus proceeds as depicted by the algorithm in Figure 8.2a. The eager fill operation can be initiated whenever a TDM slot is free and is then guaranteed to be free of any interference with other hardware components that might wish to access main memory. It remains to be shown that the worst-case timing of subsequent stack cache operations is not affected. Three cases have to be considered, depending on the kind of the next stack cache control instruction:

sres k May only initiate a memory transfer when incoherent data has to be evicted from the cache. The address range of the transfer ($[\texttt{ST} + |SC|, \texttt{LP}]$) only depends on the position of ST and LP. Eager filling does not modify either of those pointers (effective occupancy) and thus cannot impact spill costs.

sfree k Free instructions do not access memory and exhibit constant latency.

sens k May only initiate a memory transfer when the occupancy is too low. The address range of the transfer ($[\texttt{MT}, \texttt{ST} + k]$) only depends on ST and MT. The former is not impacted by eager filling, while the address of MT is incremented, i.e., the occupancy was previously increased. Fill costs thus may only be reduced.

Eager fill operations, consequently, may only improve the latency of future sens instructions. Note, however, that some side effects may still arise. This may appear when all filling of an sens instruction is eliminated. In this case, the sens instruction no longer synchronizes with the TDM period and may change the alignment of subsequent memory accesses. This may incidentally increase the number of stall cycles of these memory accesses. The number of additional stall cycles can, however, never exceed the gain induced by eager filling. WCET estimates computed without considering eager filling thus remain valid.

2.2 Eager Spill

The aim of the eager spill operation is to anticipate and reduce future spill costs associated with subsequent sres instructions. A spill is initiated by an sres if the effective occupancy would exceed the size of the stack cache, i.e., $\texttt{LP} - \texttt{ST} > |SC|$. The effective occupancy thus has to be lowered in order to reduce the spill latency. One possible solution is to copy incoherent stack data to main memory without evicting them from the cache. This allows to decrement LP and thus reduce the effective occupancy.
As for eager filling, the corresponding memory transfer size must not exceed the burst size so that at most one TDM slot is used. Assuming a burst size BS, an eager spill operation then proceeds as depicted by Figure 8.2b. The eager spill operation can be performed during free TDM slots as soon as the effective occupancy is non null. We will, nonetheless, prevent the spilling of data from the stack frame of the current function. This is because it may happen that data about to be eagerly spilled is modified by a stack store instruction. This would require additional checks to ensure that incoherent data is correctly tracked and increase hardware costs as well as complexity. As before, only free TDM slots are used, which guarantees that eager spill operations cannot interfere with other memory accesses. The worst-case timing of subsequent stack cache control operations is also not affected:

sres k: May only initiate a memory transfer when the effective occupancy becomes too large. The covered address range ($[\texttt{ST} + |SC|, \texttt{LP}]$) only involves the ST and LP pointers. The latter is lowered by eager spilling, while the former is not modified, i.e., effective occupancy was previously decreased. The spill costs experienced by an sres instruction thus may only be reduced.

sfree k: Free instructions do not access memory and exhibit constant latency.

sens k: May only initiate a memory transfer when the occupancy is too low. The address range of the transfer ($[\texttt{MT}, \texttt{ST} + k]$) only depends on `ST` and `MT`. Both are not impacted by eager spilling. Fill costs thus cannot be impacted by eager spilling.

Eager spill operations, consequently, may only improve the latency of future `sres` instructions. Similarly to eager filling, the alignment of memory accesses with regard to the TDM period may change. The worst-case timing behavior of the program is not impacted.

3 Spill/Fill Arbitration

The eager fill and spill operations can be executed asynchronously alongside other instructions that are executed by the processor whenever a free TDM slot is encountered and the respective conditions necessary to perform a transfer are met. The two operations naturally compete for the available TDM slots, we thus define several simple arbitration policies.

Spill/Fill-Only: As the names indicate, in these two configuration schemes only one of the two eager operations is performed throughout program execution, subject to the respective conditions as described above. This allows us to quantify the attainable profit of either operation, ignoring the potential overhead induced by unprofitable eager transfers.

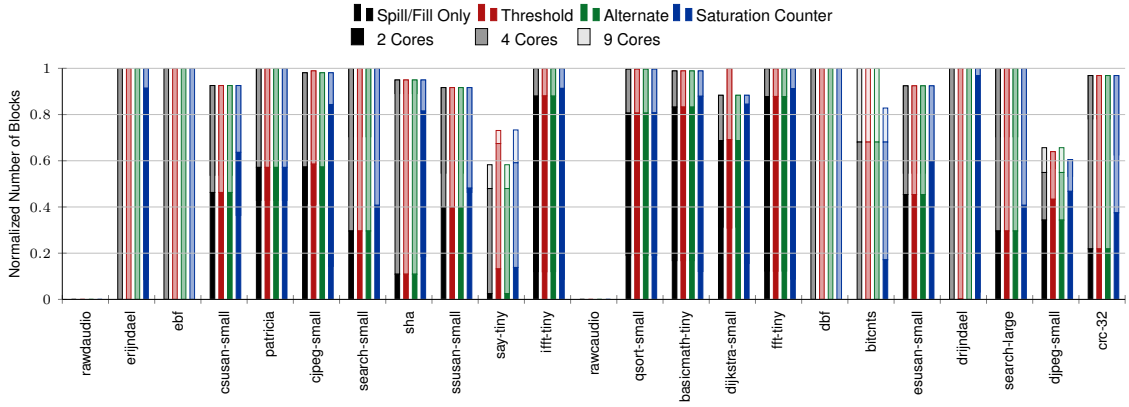
Alternate: Eager spill and fill are performed alternatingly in order to attain the maximum profit by applying both operations whenever this is possible on a fair arbitration policy.

Threshold: This approach aims to reduce the amount of unprofitable eager operations, e.g., eagerly spilling data that is never evicted. Eager operations are performed alternatingly until a preset (effective) occupancy level (threshold) is reached. In the experiments, eager spilling stops when the effective occupancy is half of the stack cache size. Likewise, eager filling stops when the occupancy reaches half of the stack cache size.

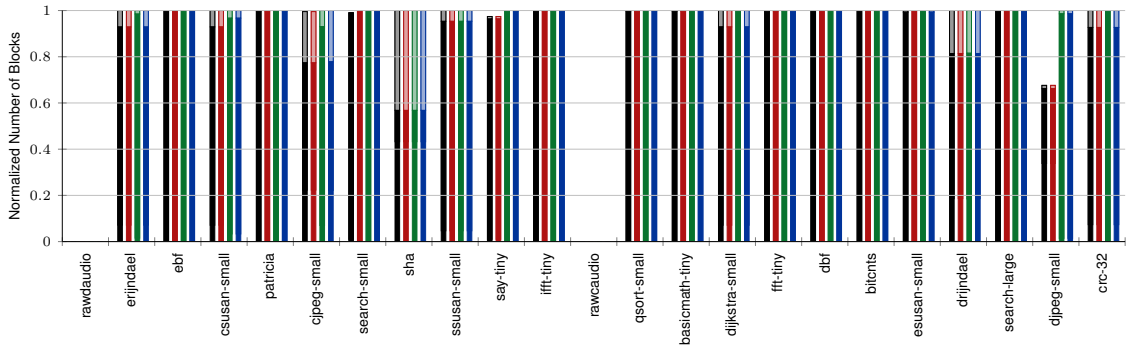
Saturation Counter: In this approach, the kind of the next stack control instruction is predicted and eager operations are chosen such that their transfer costs are reduced. The hypothesis is that `sres` and `sens` instructions are performed in sequences when descending/ascending the call chain. The prediction uses a saturation counter, similar to branch prediction [94], that is in-/decremented up to prespecified maximum levels whenever an `sres`/`sens` instruction is encountered. The eager spill/fill operations are then only permitted when the counter value lies within predefined ranges. We use a simple 1-bit saturation counter in the experiments.

4 Experiments

We evaluated eager memory transfers using the cycle-accurate simulator of the Patmos processor [111], which implements a stack cache and its associated control instructions. It also allows to simulate several processor cores in parallel that access



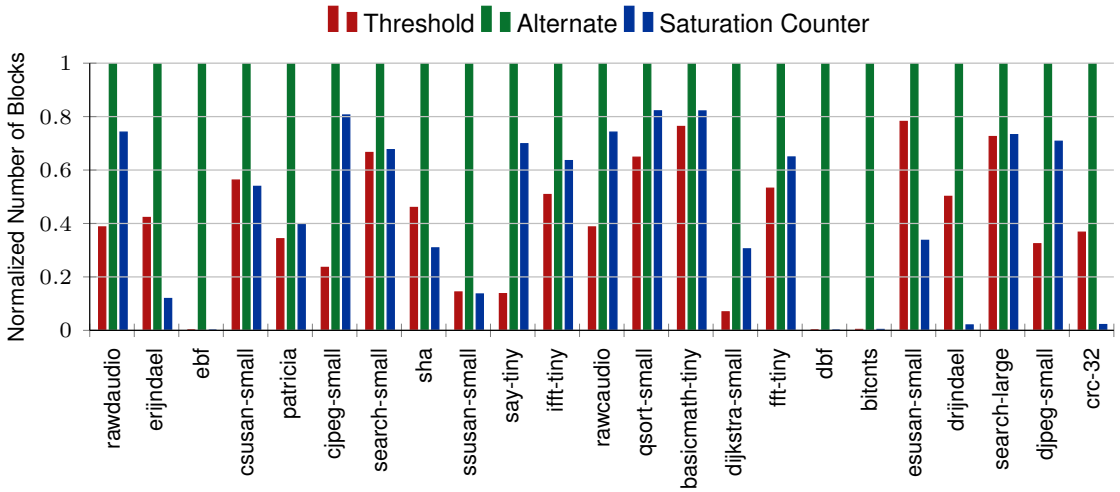
(a) Spill. The black bar represents the Spill-Only configuration.



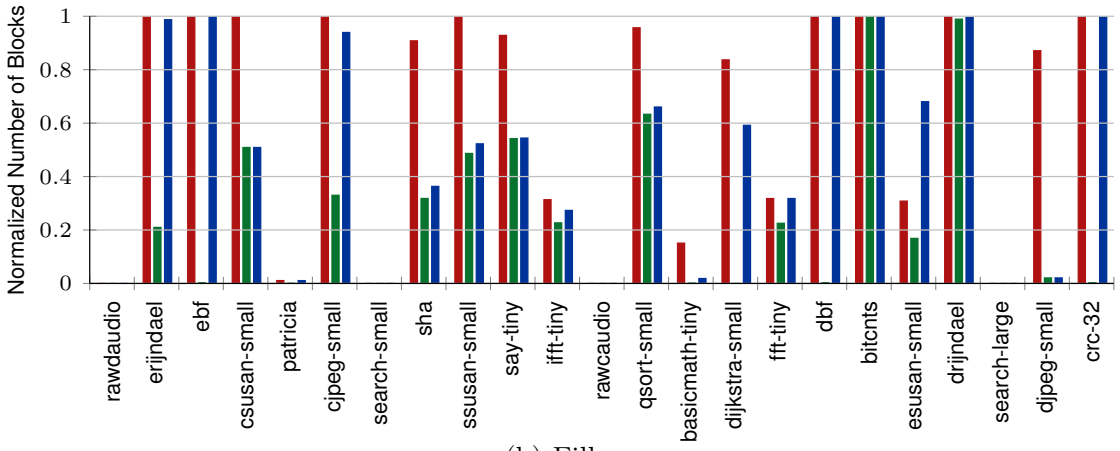
(b) Fill. The black bar represents the Fill-Only configuration.

Figure 8.3 – Normalized number of total cache blocks regularly spilled/filled with respect to standard stack cache implementation supporting lazy pointer. (Lower is better)

a shared main memory using bursts of 32 B. Memory arbitration is then performed using a TDM policy. We furthermore extended the stack cache implementation to support eager memory transfers using the arbitration strategies described above. Benchmarks of the MiBench benchmark suite [52] were compiled using optimizations (`-O2`) and subsequently executed on multi-core configurations with 2, 4 (2×2), and 9 (3×3) cores. Each core is equipped with a 256 byte stack cache, a 64 KB, 4-way set-associative data cache using a *least-recently used* replacement and write-through policy, as well as a 64 KB, 64-entry method cache using *first in, first out* replacement. The stack cache operates on 4 byte blocks, while the block size of the other caches matches the burst size of the main memory. Memory accesses take 21 cycles.
Figure 8.3 shows the normalized reduction in the number of blocks spilled and filled by `sres` and `sens` instructions in comparison to regular program execution without eager memory transfers. For eager spilling, results show a considerable reduction of spill costs by 62% over all benchmarks for the dual-core platform. For several benchmarks all spilling is performed by the eager operation (`erijndael`, `ebf`, `dbf`, `bitcnts`, `drijndael`). The total stack size of `rawcaudio` and `rawdaudio` fits into the stack cache. So, no spilling is ever performed for these benchmarks. The results for 4 and 9 cores are very close and give reductions of 6% and 1% respectively. Notable differences can be observed for `say-tiny`, `bitcnts`, and `djpeg-small`. This can be explained by the increased TDM period, which reduces the number of free TDM slots and the potential to perform eager memory transfers. All arbitration strategies were able to reduce the number of blocks spilled by `sres` instructions. The



(a) Spill.



(b) Fill.

Figure 8.4 – Efficiency of the various eager spill/fill arbitration policies relative to the Spill- and Fill-Only configurations on a dual-core platform (Lower is better).

Alternate and Threshold configurations performed best and almost always reached the best possible result represented by the Spill-Only strategy.
The results for eager filling are less pronounced, resulting in reductions of only 7.4%, 1.7%, and 0.1% for the platforms with 2, 4, and 9 cores respectively. The large difference with eager spilling is surprising. Investigations showed that our hypothesis that `sres`/`sens` instructions often appear in sequences appears to hold. However, the average distance between `sres` instructions is typically much larger than the distance between `sens` instructions. The probability to encounter free TDM slots thus is much smaller between consecutive `sens` instructions, thus reducing the amount of eager filling that can be performed. Again, all strategies are able to achieve reductions. However, the Threshold configuration clearly performs best. This is once more surprising, since the theoretical bandwidth available for filling in the Alternate approach should at least reach 50% of the bandwidth of the Threshold configuration. It appears that the limited number of TDM slots available in between `sens` instructions aggravates the competition with eager spilling, explaining this bias. Further investigations are, however, needed to confirm this hypothesis.
We also performed measurements on a single-core configuration, where the processor performs memory accesses using a private bus (without TDM). Eager operations were initiated following the Alternate arbitration scheme immediately when no other

bus requests were pending. Note that in this case interferences with other memory accesses frequently occur. We observed that spilling *and* filling of the stack control instructions was completely eliminated for almost all benchmarks, i.e., all cache transfers were carried out by eager operations. This indicates that eager transfers are effectively limited by the number of free TDM slots. An interesting idea would thus be to investigate means to explicitly allocate non-free TDM slots to eager operations. This could allow to entirely eliminate stalls at the stack control instructions in an analyzable and predictable manner.
In addition to the effective reduction by the various configurations in the number of memory transfers suffered by `sres` and `sens` instructions, we also compared the relative efficiency of the approaches. Figure 8.4 shows the normalized number of blocks eagerly spilled/filled with respect to the aggressive Spill-Only and Fill-Only configurations respectively. The Threshold configuration appears to provide the best trade-off between efficiency and the actual reduction of memory transfers by the stack control instructions. On the dual-core platform and over all benchmarks, it eagerly spills 60% and eagerly fills 30% fewer cache blocks than the Spill-/Fill-Only configurations respectively. Still the amount of excess spilling (and to a lesser degree filling) is considerable. On average, over all benchmarks 75 times the number of cache blocks are spilled compared to the number of cache blocks spilled by the program when eager memory transfers are deactivated.
However, excess spilling is not necessarily a waste. The reduced effective occupancy may reduce the cost of context switching [13]. The Threshold configuration on a dual-core platform decreases the average effective occupancy over the benchmarks' entire execution time by about 25%. For `ssusan_small`, for instance, the reduction amounts to 68%, thus considerably reducing the context switch cost related to the stack cache.

5 Conclusion

We presented an elegant and simple extension of the stack cache that allows to perform memory transfers eagerly in order to reduce the latency of future stack cache control instructions. We exploit the capability to track coherent data in the stack cache using the lazy pointer (`LP`), which allows us to distinguish between the effective occupancy and the total cache occupancy. Eager filling increases the occupancy and thus may benefit future `sens` instructions, while eager spilling decreases the effective occupancy and thus may profit `sres` instructions. The interplay between effective occupancy and occupancy guarantees that the worst-case timing is not impacted. In addition, we propose to perform these eager operations in free TDM slots to avoid any interference with concurrent memory accesses.

CHAPTER 9

Conclusion and Future Work

1 Contributions

Throughout this thesis we investigated timing analysis for time-predictable architectures. We based our work on the Patmos processor proposed by the T-CREST project. The Patmos approach relies on the interplay between the hardware, the compiler, and the timing analysis to achieve better worst-case performance. Patmos is a VLIW architecture that combines well-known hardware techniques with newly designed hardware components (e.g., stack cache and method cache). The compiler support for many of its hardware components is already provided. The limitation, however, is the lack of proper timing analysis support. Without this, it is difficult to assess the effectiveness of the time-predictable hardware in enhancing the worst-case performance. We thus proposed:

- A lightweight approach to the handling of predicated execution in WCET analysis. Predicated code is problematic for precise WCET analysis and has an impact on virtually all analysis steps. Existing WCET analysis tools simply ignore predicates and conservatively consider the effect of both predicate values. Our solution consists of recovering the hidden control-flow early in the WCET analysis process. Subsequent instructions are then duplicated once assuming the predicate is `true`, once assuming it is `false`. The resulting unfolded CFG can then be analyzed by subsequent high-level and low-level analyses in the WCET analysis process as if predicated code did not exist. The CFG unfolding keeps track of branch delay slots, nested delayed branches, function calls, and parallel instruction bundles. The induced overhead in terms of code duplication is moderate according to our experiments.
- A comparison of occupancy analyses for the stack cache. So far, two timing analyses for the stack cache have been proposed, each relying on different approaches to compute the cache occupancy. On the one hand, an analysis based exclusively on the traditional inter-procedural data-flow analysis framework (IDFA). On the other hand, a tailored analysis (SCA) that decomposes the problem into smaller function-local data-flow analyses along with analyses on

the call graph to capture inter-procedural effects. Experiments showed the limitations of the IDFA approach, which suffers imprecisions as soon as context strings become too long, due to recursions for instance.

- An analysis of preemption costs for the stack cache. Preemptions are often used as means to increase reactiveness and schedulability of real-time task systems. However, the simple structure of the stack cache does not allow it to be shared among multiple tasks. The context of preempted tasks thus needs to be saved and restored. We optimized the context switch operation based on a simple partitioning of the stack cache. Instead of saving/restoring the stack cache content, the idea is to save/restore only *useful* stack data at program points. We, furthermore, extended the tailored SCA analysis to determine the worst-case costs associated with context switch operations. The analysis consists of a combination of function-local data-flow analyses and inter-procedural analyses to capture call/return effects. Experiments showed a significant reduction of costs related to the context restoration. This is due to the implicit restoration carried-out for free by subsequent `sens` instructions. The context saving, on the other hand, showed limited reductions. Possible solutions would be the use of virtual stack caches as highlighted in the next section.

- Preemption mechanisms for the stack cache. The rich information provided by our analysis of preemption costs may be difficult to exploit without proper preemption mechanisms. These mechanisms define the set of operations the scheduler needs to perform in order to realize the context switch operations. They must thus induce minimal overhead. We proposed possible implementations of these mechanisms, which do not require the full analysis information as input. Two of these implementations are based on an extension of the ISA. The idea is to attach analysis information as parameters to the stack cache control instructions. The side effect of this is potentially reduced precision due to increased granularity. The other possible implementation is merely based on a register that tracks the stack frame size of the current function. The granularity is thus even higher. However, the required implementation effort is minimal. Experiments show that the precision loss for the approaches based on the ISA extension is not significant. Precision loss is more noticeable on the latter approach. However, the overall trend in cost reduction remains unchanged. Both approaches induce insignificant hardware overhead.

- A prefetching-like technique for the stack cache. In conventional caches, prefetching allows to anticipate and hide memory transfers before they are actually referenced. A downside of this is the induced complexity in timing analysis due to (1) imprecise addresses and (2) side effects on either the cache itself or any other component accessing main memory. The stack cache's simple design exploits the access patterns to stack data and thus does not rely on addresses. We explored a technique that allows to *eagerly* perform spilling/filling operations ahead of stack cache control instructions. The proposed approach is guaranteed to have no side effects on the stack cache itself. To avoid possible interferences on the memory bus, we propose to perform these eager operations only during unused TDM slots. This ensures both timing-composability and

the soundness of results provided by the stack cache timing analysis. A possible extension to this work is an analysis that determines unused TDM slots in order to provide tighter timing guarantees.

We implemented most of the aforementioned timing analyses in Odyssey – our WCET analysis tool for Patmos. Odyssey is fully integrated into the LLVM compiler toolchain. The tool is located right before code emission, i.e., after all code transformations have been already performed. It has, therefore, an accurate representation of the program being analyzed. In contrast to existing WCET tools, Odyssey supports the dual-issue execution as well as predicated execution. It, moreover, implements analyses for the method cache and the conventional instruction cache. Odyssey could be used as a platform for to further explore timing analysis techniques as well as hardware and compiler optimizations.
Indeed, the results of this work alone do not allow to conclusively determine whether time-predictable architectures are a better fit for high-performance safety-critical systems. More efforts are needed to ultimately compare (1) the achievable performance and (2) the relative performance with respect to conventional architectures. However, this work takes us one step forward towards understanding tailored architectures and exploring their potential in enhancing worst-case performance.

2 Extension and Future Work

Important efforts were spent on the exploration of new ideas to improve worst-case performance in time-predictable hardware. Some of these ideas need to be refined, while others already led to some interesting results. We summarize here below some topics that we already started investigating as well as perspectives for future work.

2.1 Virtual Stack Caches

The Chapters 6 and 7 presented the timing analysis of preemption costs and mechanisms for context switching assuming a single stack cache. However, an important question arises regarding the integration of this timing analysis into a schedulability test. We evoke here below some of the issues that may emerge during this process and propose *virtual stack caches* as a possible solution.
When a preemption occurs, the content of classical data caches will be updated as the preempting task performs memory accesses, i.e., when misses occur. When the preempted task is resumed, an additional CRPD must be accounted for in its WCET due to data blocks that were evicted by the preempting task. A response time analysis integrating CRPDs can then be performed, as shown in [17] for instance.
When considering a stack cache, the stack data of the preempted task must be saved before the preempting task can set up its own stack space. The preempting task is thus delayed. While the cost for saving the stack cache content of the preempted task can be bounded using the CSA, this delay may come with undesirable side-effects. Apart from an increased WCRT of high-priority tasks, this delay can also vary heavily and cause undesirable jitter, depending on the preempted tasks and their respective CSA results. A similar issue exists in caches with a write-back policy only, which are still under research [32, 21]. While the stack cache simplifies the WCET analysis of a

single task, this additional CRPD, that depends on the preempted tasks, complicates the WCRT analysis when using preemptive schedulers.
The virtual stack cache (VSC) design described in Abbaspour's thesis [103] represents an interesting approach to mitigate this issue. The idea consists of dedicating to each task its own VSC. These caches are then mapped to a fast local scratchpad memory, shared among all these tasks (i.e., running on the same physical core). Assuming that all the VSCs of a system fit into the underlying memory, the context saving and restoration costs are then completely eliminated. It suffices to retrieve the location where the VSC of the preempting task is mapped, which, in the simplest case, means fetching two pointers. The scheduling issue pointed out above, simply disappears along with the preemption overhead.
Scratchpad memories are typically small and expensive, which limits the number of VSCs that can be stored simultaneously under a static partitioning. It also appears to be a waste of resources to keep inactive stack data in the scratchpad. This suggests a dynamic handling by the system's task scheduler to save and restore the VSCs of inactive tasks to/from main memory. The dynamic restoration of the VSCs under the control of the task scheduler, opens new research perspectives that may be explored. The task scheduler clearly requires a task model that allows to express constraints related to the VSCs (size, preemption costs, ...). The task scheduler, in addition, has to reason about the bandwidth requirements of the necessary memory transfers associated with preemptions and needs a means to ensure that sufficient bandwidth ultimately is available to perform the transfers in time. Alternatively, the schedulability test may account for additional stall time that may occur when memory transfers cannot be guaranteed to be completed. This also requires associated analyses that allow to determine a lower bound on the bandwidth that can be guaranteed by the memory in parallel with the execution of a given task. These problems consequently touch several research domains, including operating system design, schedulability tests, computer architecture, as well as WCET analyses.

2.2 Unused TDM slots

In multi-core platforms with shared memory, TDM arbitration is often used to avoid inter-core interferences in a predictable fashion. Under Patmos, the arbiter dedicates one TDM slot to each core, which are scheduled in a TDM period. Memory transfers requested by some core may only be initiated at the beginning of its dedicated TDM slot. This greatly simplifies the WCET analysis and enforces timing-composability. A downside of TDM arbitration, however, is the under-utilized memory bandwidth. This is often due to situations where there is no memory requests to be served at the beginning of the core's TDM slot. Requests that come later are delayed to the next TDM period and the core's TDM slot remains *unused*. During this time, the CPU stalls for at most 2 TDM periods depending on when the request is issued. This may have a severe impact on both worst-case and average-case performance. We thus explored means to enhance the utilization of memory bandwidth under TDM arbitration.
A first step was to get quantitative insights as to the availability of these unused or free TDM slots. Using the cycle-accurate Patmos simulator we collected statistics on the number of unused TDM (or bus) slots during the execution of a benchmark (results are published in [13]). Statistics are collected depending on the number of

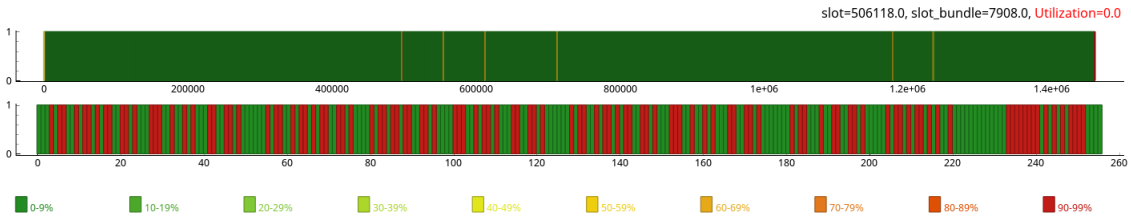


Figure 9.1 – Visualization of the TDM slots utilization for the `bitcount` benchmark assuming 2 cores configuration. Top plot shows a general overview of the TDM slots utilization during a complete execution. Bottom plot shows the utilization of individual TDM slots within a specific time frame. Colors represent the TDM slots utilization, ranging from green (low utilization) to red (high utilization).

cores in different multi-core configurations. The measurements are interesting, in particular when the number of cores is small. The average percentage of free TDM slots is found to be above 36% in a dual-core configuration. The percentage drops as the number of cores increases. However, if we consider their number a large amount of slots are available, as the total execution time increases.
An interesting question then is: *where* and *how* these free TDM slots occur in the program? The Patmos simulator tracing infrastructure allows to track free TDM slots during the execution of a program, which then can be visualized graphically. Depending on the benchmark, the collected results revealed different regions where TDM slots are likely (or less likely) to be found. For instance, free TDM slots are less likely to be found at the beginning of the program due to cold cache misses (caches are still empty). However, even in such regions we were able to notice some regular pattern in the occurrence of free TDM slots (see Figure 9.1). Other regions showed more pronounced repetitive patterns, mostly due to large loops as the whole data cannot fit into different cache structures. On the other hand, regions with high percentage of free TDM slots are often found in small loops.
The obtained results are promising and suggest means to enhance the memory bandwidth utilization under TDM arbitration. One possible direction would be to propose a static analysis for free TDM slots. Ongoing work investigates data-flow analysis to conservatively determine at which program points free TDM slots might occur. If free TDM slots could be guaranteed, one could combine them with optimizations such as prefetching. In such a case, the results found in Chapter 8 could be transformed into timing guarantees to reduce the WCET estimates. Another direction that already led to interesting results is to dynamically reschedule free TDM slots in mixed-critical systems [55]. The goal consists of improving resource utilization by executing non-critical tasks as long as critical tasks meet their deadlines.

2.3 Method Cache

The method cache is an important time-predictable component of the Patmos processor. Although this thesis did not cover the method cache as part of its main contributions, efforts were still spent to optimize its behavior and the corresponding timing analysis.
The scope-based method cache analysis proposed by Huber et. al. is pessimistic (see Section 6 of Chapter 3). We thus tried to explore the traditional age-based analysis

of Ferdinand [44, 45] to support the method cache. The method cache can be seen as a fully-associative cache that holds variable cache block sizes (see Section 2.4 of Chapter 2). A cache block can be evicted upon an access either when (1) the cache size is exceeded, or (2) the number of cache blocks exceeds the cache's associativity. Our analysis therefore tracks the minimum and maximum *ages* of code blocks based on their sizes as well as cardinalities, i.e., the analysis keeps track of the number of code blocks present in the cache.

A weakness of this approach appears at loops. Typically, in loops the first iterations load code blocks into the cache, while subsequent ones profit from their presence in the cache. The access classification analysis encounters a problem when merging abstract cache states at back edges reaching the loop header. The join operator of the must analysis only keeps the maximum ages at that program point, which results in *resetting* the ages of the loop's code blocks (i.e, accessed within the loop). This has the side-effect that the age of other code blocks keeps growing during the data-flow analysis until it gets higher than the initial age of the loop's code blocks. This means that if the loop's code blocks were initially not present in the cache, all other code blocks will be evicted, leading to pessimistic results.

For loops that fit into the cache, a simple fix consists of considering all conflicting code blocks in the loop and adding their contribution to the age of other code blocks. For large loops, we started exploring means to provide hardware support by marking code blocks as *disposable*. This hardware optimization allows to implicitly evict code blocks that are marked as disposable once they are left. The marking is done at compile-time and could be based on different criteria. Examples include dominance relations inside the loop, the size of code blocks, or execution frequency. Code blocks that appear to be profitable candidates for the disposable marking include those with a small size or that are not frequently reused. It would be interesting to formalize these heuristics in order to guide the disposable marking and evaluate its impact on the worst-case performance.

We started this work as part of a collaboration with Stefan Hepp, a PhD student from Vienna University. Some of the heuristics have been explored, but not yet published. The work is, however, ongoing in other forms such as master projects at Telecom ParisTech.

Bibliography

[1] Copper Development Association transportation. http://copperalliance.org.uk/applications/transportation. Accessed: 2018-01-14.

[2] *Cortex-R4 and Cortex-R4F Technical Reference Manual.*

[3] Data-flow analysis. http://www.cs.colostate.edu/~mstrout/CS553/slides/lecture03.pdf. Accessed: 2019-02-18.

[4] *GR740 Preliminary Data Sheet and User's Manual.*

[5] *MicroBlaze Processor Reference Guide.*

[6] NVIDIA Announces World's First AI Computer to Make Robotaxis a Reality. https://nvidianews.nvidia.com/news/nvidia-announces-world-s-first-ai-computer-to-make-robotaxis-a-reality. Accessed: 2018-04-17.

[7] *Patmos Reference Handbook.*

[8] Software Bugs. https://www5.in.tum.de/~huckle/bugse.html. Accessed: 2018-04-17.

[9] Sweet (swedish execution time tool). http://www.mrtc.mdh.se/projects/wcet/. Accessed: 2019-02-18.

[10] S. Abbaspour and F. Brandner. Alignment of memory transfers of a time-predictable stack cache. In *Proceedings of the Junior Researcher Workshop on Real-Time Computing*. 2014.

[11] S. Abbaspour, F. Brandner, and M. Schoeberl. A time-predictable stack cache. In *Proceedings of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. 2013.

[12] S. Abbaspour, A. Jordan, and F. Brandner. Lazy spilling for a time-predictable stack cache: Implementation and analysis. In *Proceedings of the Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OASICS*, pages 83–92, 2014.

[13] Sahar Abbaspour, Florian Brandner, Amine Naji, and Mathieu Jan. Efficient context switching for the stack cache: Implementation and analysis. In *Proceedings of the International Conference on Real Time and Networks Systems*, RTNS '15, pages 119–128. ACM, 2015.

[14] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[15] B. Akesson, K. Goossens, and M. Ringhofer. Predator: A predictable sdram memory controller. In *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 251–256, Sept 2007.

[16] S. Altmeyer and C. Burguiere. A new notion of useful cache block to improve the bounds of cache-related preemption delay. In *Euromicro Conference on Real-Time Systems*, ECRTS '09, pages 109–118, 2009.

[17] Sebastian Altmeyer, RobertI. Davis, and Claire Maiza. Improved cache related pre-emption delay aware response time analysis for fixed priority pre emptive systems. *Real-Time Systems*, 48(5):499–526, 2012.

[18] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[19] Sanjoy Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 137–144, July 2005.

[20] I. Bate, P. Conmy, T. Kelly, and J. McDermid. Use of modern processors in safety-critical applications. *The Computer Journal*, 44(6):531–543, Jan 2001.

[21] Tobias Blaß, Sebastian Hahn, and Jan Reineke. Write-back caches in WCET analysis. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, pages 26:1–26:22, 2017.

[22] Armelle Bonenfant, Marianne De Michiel, and Pascal Sainrat. orange : A tool for static loop bound analysis. 2008.

[23] F. Brandner, S. Hepp, and D. Prokesch. D5.2 - Initial compiler version, 2012. Report of T-CREST Deliverable D5.2, `http://www.t-crest.org/page/results`.

[24] Florian Brandner, Stefan Hepp, and Alexander Jordan. Criticality: static profiling for real-time programs. *Real-Time Systems*, 50(3):377–410, May 2014.

[25] Florian Brandner and Amine Naji. Worst-Case Execution Time Analysis of Predicated Architectures. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of *OpenAccess Series in Informatics (OASIcs)*, pages 6:1–6:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[26] D. Broman, M. Zimmer, Y. Kim, H. Kim, J. Cai, A. Shrivastava, S. A. Edwards, and E. A. Lee. Precision timed infrastructure: Design challenges. In *Proceedings of the 2013 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6, May 2013.

[27] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions. In Niklas Holsti, editor, *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6.

[28] Alan Burns. Advances in real-time systems. chapter Preemptive Priority-based Scheduling: An Appropriate Engineering Approach, pages 225–248. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1995.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[30] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[31] Patterson David and Hennessy John. *Computer Organization and Design*. Elsevier, 2013.

[32] Robert I. Davis, Sebastian Altmeyer, and Jan Reineke. Response-time analysis for fixed-priority systems with a write-back cache. *Real-Time Syst.*, 54(4):912–963, October 2018.

[33] P. Degasperi, S. Hepp, W. Puffitsch, and M. Schoeberl. A method cache for Patmos. In *Proc. of the Symposium on Object/Component/Service-oriented Real-time Distributed Computing*. IEEE, 2014.

[34] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability - the SPEAR design example. In *Proc. of the Euromicro Conference on Real-Time Systems*, pages 169–176. IEEE, 2003.

[35] Arnoldo Díaz-Ramírez, Pedro Mejía-Alvarez, and Luis E. Leyva del Foyo. Comprehensive comparison of schedulability tests for uniprocessor rate-monotonic scheduling. *Journal of Applied Research and Technology*, 11(3):408 – 436, 2013.

[36] Daniel L. Dvorak (editor). Nasa study on flight software complexity. Technical report, NASA Office of Chief Engineer, 2009.

[37] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *2009 IEEE International Conference on Computer Design*, pages 54–59, Oct 2009.

[38] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 264–265, New York, NY, USA, 2007. ACM.

[39] A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. In *Proc. of the Int. Symposium on Microarchitecture*, pages 180–191. IEEE, 1995.

[40] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, pages 152–159, May 2003.

[41] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *Proc. of the Int. Workshop on Worst-Case Execution Time Analysis*, volume 55 of *OASIcs*, pages 1–10. Schloss Dagstuhl, 2016.

[42] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, Oct 2010.

[43] Christian Ferdinand and Reinhold Heckmann. ait: Worst-case execution time prediction by static program analysis. In Renè Jacquart, editor, *Building the Information Society*, pages 377–383, Boston, MA, 2004. Springer US.

[44] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. Cache behavior prediction by abstract interpretation. *Science of Computer Programming*, 35(2):163 – 189, 1999.

[45] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, Nov 1999.

[46] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. Assessing the suitability of the ngmp multi-core processor in the space domain. In *Proceedings of the Tenth ACM International Conference on Embedded Software*, EMSOFT '12, pages 175–184, New York, NY, USA, 2012. ACM.

[47] J. A. Fisher, P. Faraboschi, and Y. Cliff. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann (Elsevier), 2005.

[48] Nielson Flemming, Hankin Hanne, R, and Hankin Chris. *Principles of Program Analysis*. Springer, 1999.

[49] J. Garside and N. C. Audsley. WCET preserving hardware prefetch for many-core real-time systems. In *Proc. of the Int. Conf. on Real-Time Networks and Systems*, RTNS '14. ACM, 2014.

[50] C. B. Geyer, B. Huber, D. Prokesch, and P. Puschner. Time-predictable code execution – instruction-set support for the single-path approach. In *Proc. of the Int. Symposium on Object/component/service-oriented Real-time distributed Computing*, pages 1–8, 2013.

[51] Daniel Grund and Jan Reineke. Abstract interpretation of fifo replacement. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, pages 120–136, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[52] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workshop on Workload Characterization*, WWC '01, 2001.

[53] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. Compsoc: A template for composable and predictable multi-processor system on chips. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):2:1–2:24, January 2009.

[54] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *2008 Real-Time Systems Symposium*, pages 456–466, Nov 2008.

[55] F. Hebbache, M. Jan, F. Brandner, and L. Pautet. Shedding the shackles of time-division multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468, Dec 2018.

[56] Matthew S. Hecht and Jeffrey D. Ullman. Analysis of a simple algorithm for global data flow problems. In *Symposium on Principles of Programming Languages (POPL'73)*, pages 207–217. ACM, 1973.

[57] Stefan Hepp and Florian Brandner. Splitting functions into single-entry regions. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 17:1–17:10, New York, NY, USA, 2014. ACM.

[58] P. Hu. Static analysis for guarded code. In *Proc. of the Int. Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 44–56. Springer, 2000.

[59] Benedikt Huber, Stefan Hepp, and Martin Schoeberl. Scope-Based Method Cache Analysis. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 73–82, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[60] Benedikt Huber, Daniel Prokesch, and Peter Puschner. Combined wcet analysis of bitcode and machine code using control-flow relation graphs. volume 48, pages 163–172, 06 2013.

[61] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 203–212, April 2011.

[62] R. Johnson and M. Schlansker. Analysis techniques for predicated code. In *Proc. of the Int. Symposium on Microarchitecture*, pages 100–113. IEEE, 1996.

[63] Alexander Jordan, Florian Brandner, and Martin Schoeberl. Static analysis of worst-case stack cache behavior. In *Proceedings of the Conference on Real-Time Networks and Systems*, RTNS'13, pages 55–64, 2013.

[64] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[65] E. Kasapaki, M. Schoeberl, R. B. Sørensen, C. Müller, K. Goossens, and J. Sparsø. Argo: A real-time network-on-chip architecture with an efficient GALS implementation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(2):479–492, Feb 2016.

[66] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[67] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1):72–105, Jun 2010.

[68] E. Lakis and M. Schoeberl. An sdram controller for real-time systems. In *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, pages 1–8, June 2013.

[69] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, March 2004.

[70] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[71] H.-H. S. Lee, G. S. Tyson, and M. K. Farrens. Eager writeback - a technique for improving bandwidth utilization. In *Proc. of the Int. Symp. on Microarchitecture*, MICRO 33, pages 11–21. ACM, 2000.

[72] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Wcet analysis of multi-level set-associative data caches. *9th Intl. Workshop on Worst-Case Execution Time WCET Analysis*, 10, 06 2009.

[73] Joseph Leung and Hairong Zhao. Real-time scheduling analysis. Technical report, Office of Aviation Research and Development, 2005.

[74] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1):56 – 67, 2007. Special issue on Experimental Software and Toolkits.

[75] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1):27–58, Jan 2005.

[76] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for wcet analysis. *Real-Time Syst.*, 34(3):195–227, November 2006.

[77] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the Design Automation Conference*, DAC '95, pages 456–461. ACM, 1995.

[78] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM.

[79] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[80] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A pret microarchitecture implementation with repeatable timing and competitive performance. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, pages 87–93, Sept 2012.

[81] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of loops. In *Proceedings of the 7th International Conference on Compiler Construction*, CC '98, pages 80–94, London, UK, UK, 1998. Springer-Verlag.

[82] Stefan Metzlaff. Analysable instruction memories for hard real-time systems, 2012.

[83] Stefan Metzlaff. Isptap – instruction scratchpad timing analysis program: Features and usage. Technical report, University of Augsburg, 2013.

[84] Stefan Metzlaff, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. A dynamic instruction scratchpad memory for embedded processors managed by hardware. In Mladen Berekovic, William Fornaciari, Uwe Brinkschulte, and Cristina Silvano, editors, *Architecture of Computing Systems - ARCS 2011*, pages 122–134, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[85] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Using smt to hide context switch times of large real-time tasksets. In *Proceedings of Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA'10, pages 255–264, 2010.

[86] Jörg Mische, Irakli Guliashvili, Sascha Uhrig, and Theo Ungerer. How to enhance a superscalar processor to provide hard real-time capable in-order smt. In Christian Müller-Schloer, Wolfgang Karl, and Sami Yehia, editors, *Architecture of Computing Systems - ARCS 2010*, pages 2–14, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[87] Amine Naji, Sahar Abbaspour, Florian Brandner, and Mathieu Jan. Analysis of preemption costs for the stack cache. *Real-Time Syst.*, 54(3):700–744, July 2018.

[88] Amine Naji and Florian Brandner. A Comparative Study of the Precision of Stack Cache Occupancy Analyses. In Benjamin Lesage, editor, *9th Junior Researcher Workshop on Real-Time Computing*, page 4, Lille, France, November 2015. Julien Forget.

[89] Amine Naji and Florian Brandner. Eager stack cache memory transfers. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 5:1–5:11, 2016.

[90] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero. An analyzable memory controller for hard real-time cmps. *IEEE Embedded Systems Letters*, 1(4):86–90, Dec 2009.

[91] Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla. A hard real-time capable multi-core smt processor. *ACM Trans. Embed. Comput. Syst.*, 12(3):79:1–79:26, April 2013.

[92] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. *SIGARCH Comput. Archit. News*, 37(3):57–68, June 2009.

[93] J. C. H. Park and M. Schlansker. On predicated execution. Technical report HPL-91-58, HP Laboratories, 1991.

[94] D. A. Patterson and J. L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface*. Morgan Kaufmann, 4rd edition, 2012.

[95] P. Puschner. Experiments with wcet-oriented programming and the single-path architecture. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 205–210, Feb 2005.

[96] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard. The t-crest approach of compiler and wcet-analysis integration. In *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, pages 1–8, June 2013.

[97] Peter Puschner. *Transforming Execution-Time Boundable Code into Temporally Predictable Code*, pages 163–172. Springer US, Boston, MA, 2002.

[98] G. Ramalingam. On loops, dominators, and dominance frontier. *SIGPLAN Not.*, 35(5):233–241, May 2000.

[99] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 99–108, Oct 2011.

[100] Jan Reineke and Daniel Grund. Relative competitive analysis of cache replacement policies. *SIGPLAN Not.*, 43(7):51–60, June 2008.

[101] National Research Council, on Engineering, Division Physical Sciences, Science, Computer Telecommunications Board, Committee on Sustaining Growth in Computing Performance, S.H. Fuller, and L.I. Millett. *The future of computing performance: Game over or next level?* 04 2011.

[102] Benjamin Rouxel, Damien Hardy, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. 06 2017.

[103] Abbaspour Sahar. Time-predictable stack caching, 2016.

[104] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 35–44, New York, NY, USA, 1999. ACM.

[105] Martin Schoeberl. Jop: A java optimized processor. In Robert Meersman and Zahir Tari, editors, *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, pages 346–359, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[106] Martin Schoeberl. Time-predictable computer architecture. *EURASIP J. Embedded Syst.*, 2009:2:1–2:17, January 2009.

[107] Martin Schoeberl. Is time predictability quantifiable? In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 333–338, July 2012.

[108] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449 – 471, 2015.

[109] Martin Schoeberl, David Vh Chong, Wolfgang Puffitsch, and Jens Sparsø. A Time-Predictable Memory Network-on-Chip. In Heiko Falk, editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 53–62, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[110] Martin Schoeberl, Wolfgang Puffitsch, Stefan Hepp, Benedikt Huber, and Daniel Prokesch. Patmos: a time-predictable microprocessor. *Real-Time Systems*, 54(2):389–423, Apr 2018.

[111] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. Towards a time-predictable dual-issue microprocessor: The patmos approach. In *Proceedings of Bringing Theory to Practice: Predictability and Performance in Embedded Systems*, volume 18, pages 11–21. OASICS, 2011.

[112] J. W. Sias, W.-M. W. Hwu, and D. I. August. Accurate and efficient predicate analysis with binary decision diagrams. In *Proc. of the Int. Symposium on Microarchitecture*, pages 112–123. ACM, 2000.

[113] M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H.-H. S. Lee. Predicate-aware scheduling: A technique for reducing resource constraints. In *Proc. of the Int. Symposium on Code Generation and Optimization*, pages 169–178. IEEE, 2003.

[114] Vijayaraghavan Soundararajan and Anant Agarwal. Dribbling registers: A mechanism for reducing context switch latency in large-scale multiprocessors. Technical report, 1992.

[115] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Guillaume Borios, Victor Jégu, and Reinhold Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. 01 2005.

[116] Thesing Stephan. Safe and precise wcet determination by abstract interpretation of pipeline models, 2004.

[117] A. Stoutchinin and G. Gao. If-conversion in SSA form. In *Proc. of the Int. Euro-Par Conference*, pages 336–345. Springer, 2004.

[118] T-CREST. Report on architecture evaluation and WCET analysis. Technical report, 2013.

[119] Henrik Theiling. Ilp-based interprocedural path analysis. In Alberto Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Embedded Software*, pages 349–363, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[120] Lothar Thiele and Reinhard Wilhelm. 03471 abstracts collection – design of systems with predictable behaviour. In Lothar Thiele and Reinhard Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[121] Mikkel Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *Journal of Computer and System Sciences*, 69(3):330–353, November 2004.

[122] Eric Tune, Rakesh Kumar, Dean M. Tullsen, and Brad Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the Symp. on Microarchitecture*, MICRO'04, pages 183–194, 2004.

[123] X. Vera, B. Lisper, and Jingling Xue. Data caches in multitasking hard real-time systems. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 154–165, Dec 2003.

[124] Yun Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, pages 328–335, 1999.

[125] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, 2009.

[126] M. Zimmer, D. Broman, C. Shaver, and E. A. Lee. Flexpret: A processor platform for mixed-criticality systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 101–110, April 2014.

Personal Publications

Here below is a list of publications on which I contributed during my thesis.

- Sahar Abbaspour, Florian Brandner, Amine Naji, and Mathieu Jan. Efficient context switching for the stack cache: *Implementation and analysis. In Proceedings of the International Conference on Real Time and Networks Systems*, RTNS '15, pages 119–128. ACM, 2015.

- Amine Naji and Florian Brandner. A Comparative Study of the Precision of Stack Cache Occupancy Analyses. In Benjamin Lesage, editor, *9th Junior Researcher Workshop on Real-Time Computing*, page 4, Lille, France, November 2015. Julien Forget.

- Amine Naji and Florian Brandner. Eager stack cache memory transfers. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 5:1–5:11, 2016.

- Amine Naji, Sahar Abbaspour, Florian Brandner, and Mathieu Jan. Analysis of preemption costs for the stack cache. *Real-Time Systems*, 54(3):700–744, July 2018.

- Florian Brandner and Amine Naji. Worst-Case Execution Time Analysis of Predicated Architectures. In Jan Reineke, editor, *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 57 of OpenAccess Series in Informatics (OASIcs), pages 6:1–6:13, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.