DEGREE PROJECT IN INFORMATION AND COMMUNICATION TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, *SWEDEN 2017*

Two-phase WCET analysis for cache-based symmetric multiprocessor systems

RODOTHEA MYRSINI TSOUPIDI

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

KTH Information and
Communication Technology

Two-phase WCET analysis for cache-based symmetric multiprocessor systems

Master's Thesis Project

RODOTHEA-MYRSINI TSOUPIDI

Master's Thesis at KTH Information and Communication Technology
Supervisor: David Broman
Examiner: Christian Schulte

TRITA-ICT-EX-2017:207

Abstract

The estimation of the *worst-case execution time* (WCET) of a task is a problem that concerns the field of embedded systems and, especially, real-time systems. Estimating a safe WCET for single-core architectures without speculative mechanisms is a challenging task and an active research topic. However, the advent of advanced hardware mechanisms, which often lack predictability, complicates the current WCET analysis methods. The field of Embedded Systems has high safety considerations and is, therefore, conservative with speculative mechanisms. However, nowadays, even safety-critical applications move to the direction of multiprocessor systems. In a multiprocessor system, each task that runs on a processing unit might affect the execution time of the tasks running on different processing units. In shared-memory symmetric multiprocessor systems, this interference occurs through the shared memory and the common bus. The presence of private caches introduces cache-coherence issues that result in further dependencies between the tasks.

The purpose of this thesis is twofold: (1) to evaluate the feasibility of an existing one-pass WCET analysis method with an integrated cache analysis and (2) to design and implement a cache-based multiprocessor WCET analysis by extending the single-core method. The single-core analysis is part of the *KTH's Timing Analysis* (KTA) tool. The WCET analysis of KTA uses *Abstract Search-based WCET Analysis*, an one-pass technique that is based on abstract interpretation. The evaluation of the feasibility of this analysis includes the integration of microarchitecture features, such as cache and pipeline, into KTA. These features are necessary for extending the analysis for hardware models of modern embedded systems. The multiprocessor analysis of this work uses the single-core analysis in two stages to estimate the WCET of a task running under the presence of temporally and spatially interfering tasks. The first phase records the memory accesses of all the temporally interfering tasks, and the second phase uses this information to perform the multiprocessor WCET analysis. The multiprocessor analysis assumes the presence of private caches and a shared communication bus and implements the MESI protocol to maintain cache coherence.

Keywords: Worst-Case Execution Time Analysis, Abstract Domain, Real-Time Systems, Low-level Analysis, Pipeline Analysis, Cache-based Analysis, Multiprocessor Analysis

Sammanfattning

Tvåsteg WCET-analys för cache-baserade symmetriska multiprocessorsystem

Uppskattning av *längsta exekveringstid* (eng. worst-case execution time eller WCET) är ett problem som angår inbyggda system och i synnerhet realtidssystem. Att uppskatta en säker WCET för enkelkärniga system utan spekulativa mekanismer är en utmanande uppgift och ett aktuellt forskningsämne. Tillkomsten av avancerade hårdvarumekanismer, som ofta saknar förutsägbarhet, komplicerar ytterligare de nuvarande analysmetoderna för WCET. Inom fältet för inbyggda system ställs höga säkerhetskrav. Således antas en konservativ inställning till nya spekulativa mekanismer. Trotts detta går säkerhetskritiska system mer och mer i riktning mot multiprocessorsystem. I multiprocessorsystem påverkas en process som exekveras på en processorenhet av processer som exekveras på andra processorenheter. I symmetriska multiprocessorsystem med delade minnen påträffas denna interferens i det delade minnet och den gemensamma bussen. Privata minnen introducerar cache-koherens problem som resulterar i ytterligare beroende mellan processerna.

Syftet med detta examensarbete är tvåfaldigt: (1) att utvärdera en befintlig analysmetod för WCET efter integrering av en lågnivå analys och (2) att designa och implementera en cache-baserad flerkärnig WCET-analys genom att utvidga denna enkelkärniga metod. Den enkelkärniga metoden är implementerad i *KTH's Timing Analysis* (KTA), ett verktyg för tidsanalys. KTA genomför en så-kallad Abstrakt Sök-baserad Metod som är baserad på *Abstrakt Interpretation*. Utvärderingen av denna analys innefattar integrering av mikroarkitektur mekanismer, såsom cache-minne och pipeline, i KTA. Dessa mekanismer är nödvändiga för att utvidga analysen till att omfatta de hårdvarumodeller som används idag inom fältet för inbyggda system. Den flerkärniga WCET-analysen genomförs i två steg och uppskattar WCET av en process som körs i närvaron av olika tids och rumsligt störande processer. Första steget registrerar minnesåtkomst för alla tids störande processer, medans andra steget använder sig av första stegets information för att utföra den flerkärniga WCET-analysen. Den flerkärniga analysen förutsätter ett system med privata cache-minnen och en gemensamm buss som implementerar MESI protokolen för att upprätthålla cache-koherens.

Nyckelord: Längsta Exekveringstid Analys, WCET, Abstrakt Domän, Realtidsystem, Låg-nivå Analys, Pipeline Analys, Cache-baserad Analys

Contents

Chapter 1

Introduction

This thesis concerns the problem of estimating the *worst-case execution time* (WCET) of a task running on a *symmetric multiprocessor* (SMP) system with private caches. That is, the longest time a task may run in the presence of temporally and spatially interfering tasks. The estimation of the WCET is necessary for applications where the correctness of the program depends on time, for example real-time applications.

The WCET problem is an active research topic with many challenges [51]. The WCET of a task depends on the machine state, i.e. the state of the hardware system, and the inputs to the task. Measuring or calculating the execution time for all possible inputs and machine states is not possible in the general case [51]. For this reason, many approaches attempt to *estimate* the WCET of a task. There are mainly two approaches for the WCET problem: static and dynamic. Dynamic approaches execute a set of instances directly on the hardware or on a simulator and extract the longest of all tested executions. This result under-estimates the WCET because it is by definition not possible to measure an execution time that is larger than the actual WCET. Static approaches intend to estimate the WCET for all possible inputs and extract an upper bound. Due to the high complexity of considering all paths separately, static approaches use approximations that are based on the program semantics. The result of the static approaches is an over-approximation of the actual WCET because the analysis makes sound approximation of the program semantics. Figure 1.1 illustrates the WCET problem and the possible approaches to estimate the WCET of a task.

The complexity of the WCET analysis depends both on the source code, and the underlying hardware architecture. The field of computer architecture focuses mostly on performance-oriented microarchitecture features that are often difficult to analyze (e.g. cache memories and parallelism) [29]. However, analyzability is an important feature for embedded-system applications. For this reason, different analyses attempt to include microarchitecture features. Many of the analysis techniques extend previous methodologies to analyze low-level mechanisms, such as pipeline and caches [15, 37]. Multiprocessing introduces even more challenges, due to the unexpected behavior and interference of independent simultaneous tasks that affect the state of the system.

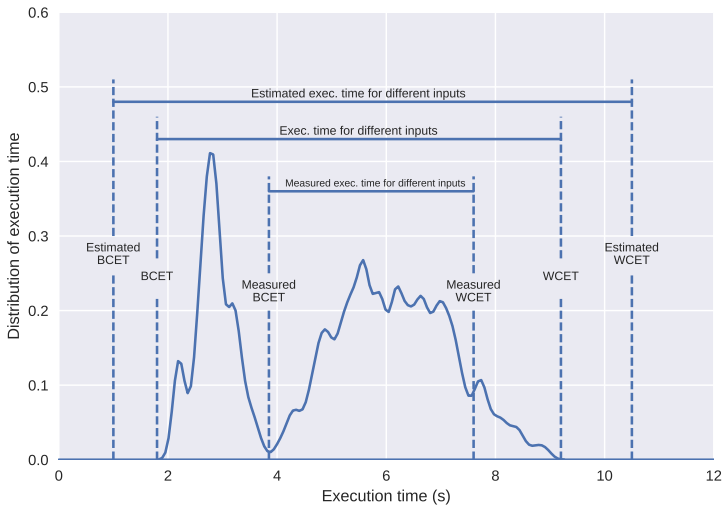


Figure 1.1: Safeness and tightness for different WCET estimation techniques. Tightness refers to the distance between the actual WCET and the over-approximated value. A safe WCET estimation guarantees that the estimated WCET is greater or equal to the actual WCET. Measurement-based techniques give an unsafe WCET because the estimated value for the WCET is always less or equal to the actual WCET. On the contrary, static program analysis provides a safe but over-approximated WCET. The tightness of the resulting WCET is a measure that may be used to evaluate the method. BCET refers to the best case execution time and is the lower bound of the execution-time graph.

The following sections provide an overview of the thesis, by presenting the problem area, the purpose, the methodology, and the contributions of this thesis. More specifically, Section 1.1 provides an overview of the problem area with regards to embedded systems and WCET estimation. Section 1.2 presents the problems and the research question of the thesis. Section 1.3 describes briefly the approach, and Section **??** the delimitations of the approach. Next, Section 1.4 describes the purpose and goals with this thesis. Section 1.5 describes the benefits, the ethical issues, and sustainability of the results of the thesis. Section 1.6 describes the Evaluation of the approach. Section 1.7 presents the contributions of this thesis and Section 4.6 presents the limitations of the implementation of the approach. Finally, Section 1.8 gives an outline of the whole thesis report.

1.1 Problem Area

WCET estimation is a concept concerning different fields, but is especially important for real-time systems, i.e., systems that are subject to time constraints. These time constraints (or deadlines) are time limits, within which the system should respond to real-time events [35]. The strictness of the timing constraints classifies real-time systems into three categories: *soft*, *firm*, and *hard* real-time systems. In soft and firm real-time systems, the

execution of a task and the result may be useful even after the deadline. In these cases, violating the deadline does not have disastrous consequences to the system or the environment. On the other hand, the deadlines of hard real-time systems are very strict, because a deadline violation may have severe consequences, such as environmental disaster or human loss. Such systems are usually safety critical, for example cars and airplanes.

A method to guarantee that all critical tasks meet their deadlines is to perform a schedulability analysis. The schedulability analysis requires as inputs, a scheduling algorithm and the WCET of the task(s) of the system. Then, this analysis attempts to prove the feasibility of the system using the specified scheduling algorithm. For this reason, there is a need for calculating upper (and lower) bounds of the execution time.

The WCET problem is undecidable in the general case because it is equivalent to the halting problem[1]. However, different techniques and algorithms can analyze common real-time applications. Such applications use simpler software structures that allow the implementation of efficient WCET analysis tools. These tools attempt to calculate the actual WCET or more often to give a *safe* and *tight* estimation. *Safe*, in a sense that the estimated value does not underestimate the actual WCET and *tight*, so that the estimated value will be as near to the actual value as possible (see Figure 1.1). The two main approaches for estimating the WCET are dynamic and static. The following paragraphs describe these two approaches.

Measurement Techniques

Measurement techniques are common methods for estimating the WCET of a program [51]. The estimation of the WCET depends on a series of executions of the program for different input values. That way, the resulting WCET is the maximum observed execution time. This technique can, however, not guarantee a safe bound for the WCET because the observed execution time can never be higher than the actual WCET (see Figure 1.1). Therefore, measurement methods cannot provide safeness guarantees for the WCET of hard real-time systems.

Measurement techniques are able to analyze soft and firm real-time systems because the deadlines of these systems are not very strict. In addition to that, measurement techniques often analyze hard real-time systems using extensive testing in combination with static analysis. Also, measurement approaches may be applied to new processors and architectures, before accurate static analysis tools for the new hardware are available [30].

Static Analysis

In hard real-time systems, safety critical tasks should always meet their deadlines. Measurement techniques cannot guarantee a safe WCET because the result usually relies on

[1] The halting problem is the problem of determining whether a program, given its input, halts [45]. If we assume that there is a solution to the WCET problem and the resulting WCET would correspond to the actual WCET or infinity in case the program is in infinite loop, then the halting problem could also be solved because it can be reduced to the WCET problem, i.e. if *WCET(p)* is less than infinity, then program *p* halts, otherwise it does not halt [39]. However, the halting problem has been proved to be undecidable [45].

a subset of all the possible input values. Instead, static approaches are able to provide formal guarantees for a safe WCET that does not underestimate the actual WCET (See Figure 1.1).

Static analysis techniques usually provide a safe bound for a wide range of real-time applications [14]. Many of these techniques are based on abstract interpretation [9], a static analysis framework that provides correctness guarantees. These methods aim at providing a safe over-approximation of the WCET based on the program semantics. Static analysis abstracts the program semantics to provides an over-approximation of the WCET. Due to the complexity of the WCET analysis, many approaches make assumptions that may restrict the expressiveness of the method, such as assuming the absence of side-effects, e.g. exceptions.

However, static analysis is able to provide a guarantee that the actual WCET of the task will not exceed the calculated WCET. The ability to provide a safe result makes static analysis techniques suitable to hard real-time applications.

1.2 Problem

The estimation of the WCET is a challenging problem. Up to the present, there is no general solution to the WCET problem, and different approaches attempt to improve the resulting upper bound (tightness), generalize the current methodologies for a wider range of applications and hardware architectures, or develop new techniques for estimating the WCET. WCET analysis for single-core architectures without speculative features is an active research topic [51] with challenges due to complexity of the WCET problem. Integrating performance-oriented microarchitecture features, such as caches and multiprocessing, which are not predictable in the general case, increases the complexity of the WCET analysis [38, 47]. A common method for dealing with new microarchitecture features is by extending the previous WCET methodologies to support these feature or to support some special analyzable cases [4, 15, 31].

The first objective of this thesis is to design and implement a multi-core WCET analysis by extending an existing one-pass single-core WCET analysis method. In addition to this, the thesis examines the expressiveness of the single-core method and implements a low-level analysis that includes a cache, cache hierarchy, and pipeline analysis. This aims at evaluating the feasibility of the one-pass single-core method with an integrated low-level analysis.

To sum up, the main question of this project can be formulated in the following way:

> Is an one-pass automatic single-core WCET estimation strategy with integrated microarchitecture features, such as cache and pipeline, feasible? How can the one-pass single-core analysis be extended for shared-memory multiprocessor systems?

1.3 Approach

The continuous development of new hardware mechanisms introduces new challenges to the WCET analysis methods. To deal with the new development, different approaches extend previous methods or develop new methods that deal with the problem from a different perspective. The first part of this thesis evaluates the feasibility of an existing one-pass WCET analysis method with the integration of low-level microarchitecture mechanisms. In particular, this part integrates a cache and a pipeline analysis to the one-pass WCET method and evaluates the feasibility of this approach. The evaluation examines the expressiveness of the approach using a benchmark suite and compares the method with another WCET analysis tool. This comparison is based on the analysis time and is time- and space-restricted.

The second part of this thesis deals with multiprocessing. The approach focuses on shared-memory systems. In such systems, the tasks that run in separate processors interfere and affect the execution of each other through the shared memory and the communication bus. The presence and the execution of one task affects the tasks running in different processing units of the system when the tasks interfere temporally, i.e. when they coexist and execute simultaneously. Hence, the execution time of temporally interfering tasks running on dedicated processing units depends on the spatial interference of the tasks, i.e. the accesses to the shared data, and the effect of the remote data access to the shared caches. In particular, the multiprocessor analysis takes as inputs a task and all the temporally interfering tasks of the system and performs a multi-staged analysis that results in the WCET of the target task. Each of the temporally interfering tasks runs on a dedicated processor. The first stage of the multiprocessor analysis analyzes each task separately and derives information about the spatial interference to the system. Using this information, the WCET analysis proceeds to the second stage that analyzes the target tasks and estimates a safe over-approximation of the WCET using upper bounds for every spatially interfering memory access.

1.4 Purpose

This thesis is part of *KTH's Timing Analysis* (KTA)[2], and more specifically, part of the *Abstract Search-based WCET Analysis* [13] of KTA. *Abstract Search-based WCET Analysis* is an annotation free methodology for calculating the worst-case timing path using abstract values. The main purpose of this thesis is to extend the WCET analysis of KTA to support simple *symmetric multiprocessor* (SMP) systems with shared and private caches. In addition to that, this project integrates some missing parts of KTA and investigates the use of a more expressive abstract domain. The target architecture of KTA is the MIPS32® *instruction set architecture* (ISA) [25].

[2]KTA tool: https://github.com/timed-c/kta

1.5 Ethics and Sustainability

WCET calculation is an important problem for embedded system software design because WCET is an input to the schedulability analysis. Tightening the WCET gives more flexibility to the schedulability analysis and may facilitate the reduction of computing resources. That is, the tasks that compose the system have tighter deadlines and the analysis may be able to allocate fewer processing units that reduces the required hardware.

Also, designing a multiprocessor analysis for hard real-time systems may lead to the use of more efficient hardware by safety-critical applications. That might reduce the required hardware for a number of applications.

With regards to ethics, this analysis describes the methodology and evaluation without making any claims that could lead to misusing of the tool. This is important for hard real-time applications, where a failure might have disastrous consequences.

Openness contributes to the replicability and reproducibility of the described methodology. Different researchers can easily verify the functionality and correctness of the proposed model and the evaluation and can therefore increase or decrease the confidence to the specific approach.

1.6 Verification and Evaluation

The evaluation of this thesis consists of three parts that intend to evaluate the three main parts of the approach of this thesis, i.e. the expressiveness of the abstract domain, the cache analysis, and the multiprocessor analysis. The first and the second parts of the evaluation use the Mälardalen benchmark suite [22]. The last part defines a set of small benchmarks that aim at verifying the multiprocessor approach.

The first part compares the implemented interval-congruence domain with the previously implemented interval abstract domain. Also, it compares the KTA tool with another tool, namely SWEET, with regards to expressiveness. The evaluation uses the analysis time to compare the implementations in both cases. The first comparison evaluates the performance and expressiveness of the implemented interval-congruence domain. The second comparison evaluates the expressiveness of the KTA methodology as a whole, which indicates the performance of the implemented parts. In addition to the expressiveness, the evaluation also measures the overhead of the low-level analysis.

The second part evaluates the cache analysis and consists of two parts. The first part measures the overhead of the cache analysis to the WCET analysis of KTA. The second part intends to evaluate the tightness of the cache analysis using Creator ci40, a hardware platform that contains a simple cache hierarchy [11].

The third part verifies the multiprocessor analysis approach by showing that the actual performance of an actual hardware platform behaves according to the approach. The evaluation of this part compares the result of the analysis with measurements on the hardware using Creator ci40 [11], using a dual-core configuration with a 2-level cache hierarchy.

1.7 Contribution

The main contribution of this thesis is (1) the evaluation of the feasibility of an existing one-pass WCET method with integrated low-level mechanisms and (2) the design and implementation of a multi-core analysis method for shared-memory symmetric multiprocessor systems by extending the existing one-pass method.

The first contribution of this thesis integrates an improved abstract value domain and a low-level analysis, including a cache hierarchy and a simple classic 5-stage pipeline, to the one-pass analysis and evaluates the feasibility of the method. The second contribution designs and implements a multi-core analysis method for shared-memory systems by extending the one-pass single-core analysis.

1.8 Outline

The rest of this thesis consists of the following chapters that introduce the background, describe the approach, and the evaluation of this thesis. Chapter 2 summarizes the related work, with respect to WCET approaches, the different abstract domain approaches, as well as low-level analyses for SMP systems. Chapter 3 describes briefly the KTA tool, presents some necessary background for the formalization that Chapter 4 uses. The latter chapter describes the approach of the abstract domain, the SMP analysis, and the pipeline analysis. Chapter 5 presents the implementation of this thesis. Chapter 6 describes the evaluation of the methodology. Based on measurements taken on a hardware platform that implements a simple SMP system with caches, the evaluation method evaluates the the approach and the results of the low-level analysis. Finally, Chapter 7 summarizes the approach and the results, and presents ideas and suggestions for future work.

Chapter 2

Related Work

WCET is specially important for real-time embedded applications. Static methods for estimating a safe WCET are the major focus of many research teams because these methods are able to provide correctness guarantees. However, the complexity of the WCET problem creates many challenges with regards to expressiveness, complexity, and automatization of these analyses. In addition to that, computer architecture continues to advance focusing mainly on performance-based criteria. More specifically, for many decades, the basic method for improving performance was based on Moore's law, i.e. the reduction of the transistor size that leads to higher frequency. However, due to physical boundaries that affect the characteristics of the transistor and among others result in increased power dissipation, the technology moves towards advanced speculative microarchitecture features, e.g. multi-tasking and multi-processing. Such techniques become common even in embedded real-time systems that have high predictability requirements that are assisting the timing analysis.

This chapter presents the related work in four categories: Static WCET Analysis, Abstract Domains, Low-level Analysis, and Multiprocessor WCET Analysis.

2.1 Static WCET Analysis

WCET estimation is an well-known problem in the field of embedded systems. While many industrial implementations use measurement techniques, the research community focuses mostly on static and formal approaches. The following parts of this section describes different static approaches that deal with the WCET problem.

A widely used technique for estimating the WCET of a program is the *implicit path enumeration technique* (IPET) [33]. IPET uses implicit paths, i.e., a set of unordered basic blocks that build a path (or a series of paths), loop bounds, and the program semantics to construct an *integer linear programming* (ILP) problem[1]. The ILP problem maximizes

[1] *Integer linear programming* (ILP) is an optimization problem that minimizes (maximizes) an objective function on a polytope. The objective function and the constraints, which specify the polytope, are linear and the solution integer [45].

an objective function that corresponds to the execution time of the program. Hence, the resulting maximal value corresponds to the WCET of the specific problem. In particular, the objective function consists of the implicit path accompanied with the maximum number of iterations (loop bounds) for each basic block. The linear constraints that IPET uses depend on the *control flow graph* (CFG) of the code. For every combination of mutually exclusive constraints, IPET generates a different constraint set and, consequently, solves a different ILP problem [33]. The WCET of the program is the maximum of the execution time results for every constraint set [33]. IPET estimates the WCET of a program without the need to perform explicit-path exploration, which can lead to high complexity. However, a draw-back of the approach is that microarchitecture techniques that introduce dependencies between different basic blocks, for example data cache analysis, require reformulation of the ILP problem [4, 32]. In particular, IPET requires reformulation of the objective function and special handling and analysis for deriving the linear constraints that form the ILP problem.

Lundqvist and Stenström [36] follow a different static approach for estimating the WCET. The technique integrates the path and the timing analysis using cycle-level simulation that models a cycle-level timing model of the hardware. The method extends the instruction-level simulation technique to handle non-concrete input or memory values (denoted as *unknown*). This notation forms the constant abstract domain [3], equipped with a top value (unknown) and ordered by inclusion. To avoid explosion in the number of simulated paths, the method merges the paths at specific points. In more detail, the simulation stops at every unknown conditional branch and continues by resuming the path that has made the minimum progress, i.e. has more steps to an exit node. When more than one paths exist, the analysis merges all the paths before resuming with the simulation [36].

Abstract execution (AE) is the methodology that KTA uses for estimating the WCET and has applications in automatic calculation of loop bounds, infeasible path identification [20], and discovery of worst-case execution inputs [14]. This method is able to integrate the path and timing analysis using abstract values and an abstract timing hardware model to calculate the WCET in an automatic way. Compared to the approach of Lundqvist and Stenström [36], the use of an abstract hardware model reduces the complexity of the analysis and allows the use of more complex abstract domains and different merging options that increase the expressiveness of the approach.

2.2 Abstract Domains

The purpose of abstract interpretation is to translate the semantics of a program from the concrete domain to an abstract domain. The concrete domain represents the actual semantics of the program, whereas the abstract domain approximates the semantics, so that a sound program analysis becomes feasible for non-trivial programs.

Many fields that implement abstract interpretation techniques, such as static analysis in compilers and error detection, use different abstract domains. The abstract domain is an important part of many static approaches to the WCET problem because it affects the expressiveness of the analysis [51]. There are two main categories of abstract domains,

i.e. *non-relational* and *relational* abstract domains. Non-relational abstract domains do not encode relations between different states directly. For example, when representing integer values, such as registers, non-relation abstract domains do not directly encode the relations and dependencies between the registers throughout the program. However, non-relational abstract domains are efficient. In order to express determining relations between variables, the analysis may use patterns that often occur in programs (as in the work of Thesing [50]). Relational abstract domains are, on the other hand, able to encode dependencies between different variables, but have often a larger analysis overhead. The rest of this section focuses on some well-known non-relational abstract domains.

Two very common numerical abstract domains are the interval [8] and the congruence domain [19]. The former represents values in the form of an interval with a lower (l) and an upper bound (h), $v_i = [l, h],\ l, h \in \mathbb{Z}$, whereas the latter represents values in the form congruences (equally distanced integers), i.e. $v_c = a + b\mathbb{Z}$. Another approach, *circular linear progressions* (CLP) [48], combines the interval domain with the congruence domain, by adding a stride parameter to the interval formalization, i.e. $v_{clp} = [l, s, h],\ l, s, h \in \mathbb{N}^+$. All parameters are non negative, $l, s, h \in [0, 2^{32} - 1)$, resulting in a circular interval that represents register wraparounds. Stride, s, is useful for representing equally dispersed values that occur after applying specific compiler optimizations (see Section 4.1.3). Källberg describes a variation of the CLP that modifies the CLP representation by replacing the upper bound with the cardinality of the set, i.e. $v_i = [l, s, n],\ l, s \in \mathbb{Z}, n \in \mathbb{N}^+$. The modified CLP represents wraparounds as the original CLP.

This project integrates the abstract domain to KTA. However, the KTA tool assumes that an overflow results in a unknown state, in accordance with the C language specification [26, 27]. Therefore, KTA does not model wraparounds. Actually, the abstract domain that this thesis implements is a combined interval and congruence abstract domain that uses the notation of the modified CLP [28]. The interval-congruence implementation contains some changes on the domain of the l, s, n parameters, i.e. $l \in \mathbb{Z}, s, n \in \mathbb{N}^+$ (see Section 4.1).

Relational abstract domains, like the polyhedron domain [10], the octagon abstract domain [42], and different weakly relational domains [41], are more expressive, but are computationally expensive. All the above-mentioned relational abstract domains have draw-backs related to non-linear operations, such as multiplication and division. These operations result in an rough over-approximation [18] when applied to relational domains. However, relational approaches might give an overall higher expressiveness and result in a tighter WCET approximation.

A recent developement improves the performance of the polyhedra domain in both time and space by two to five orders of magnitude [49]. This developement makes the polyhedra domain more attractive for static analyses and this solution may be suitable for the WCET problem.

2.3 Low-level Analysis

In this thesis, low-level analysis refers to the microarchitecture model that intends to integrate low-level microarchitecture mechanisms to the WCET analysis of KTA. Most of the static approaches use low-level models that form abstract domains. Subsequently, different analysis approaches (as in Section 2.1) integrate the low-level analysis into each approach. The following part of this section describes different methodologies that integrate low-level microarchitecture features to the WCET analysis.

Lundqvist and Stenström [36] use a cycle-level timing model of the hardware for performing WCET analysis. The pipeline and cache models use a merging policy for improving the analysis performance. In general, this approach uses traditional simulation techniques and implements a precise hardware model for both the pipeline and the cache. The pipeline analysis uses pipeline reservation tables for recording the release of each resource [36].

Integrating microarchitecture features in the IPET methodology requires reformulating the problem. Many microarchitecture features depend on the path history that IPET does not represent directly [4]. For example, Li et al. [32] integrate an instruction cache analysis in IPET requires reformulating the ILP problem. This reformulation includes redefinition of the objective function and the program constraints, so that the analysis considers the cache dependencies between the basic blocks. Li et al. [32] extract the program constraints by defining a so-called cache-conflict graph that represents the conflicts between basic blocks. Burguière and Rochange [4] integrate a bi-modal branch predictor model in IPET by adding constraints that consider the misprediction counts. The analysis modifies the execution counts of the blocks and edges in the CFG to consider the mispredicted branches [4].

In another approach based on abstract interpretation, Ferdinand and Wilhelm design a cache analysis consisting of three different analyses, i.e. the Must, the May, and the Persistence analysis [16]. Must analysis describes the conservative case that preserves the cache blocks that always remain in the cache during the execution of a specific basic block. May analysis preserves the cache blocks that might be present in the cache during the execution of a specific basic block. Finally, the Persistence analysis deals with special cases, where none of the previous analyses applies. For example, a memory access might result in a miss in the first iteration, but a hit in all iterations that follow [16]. These three combined analyses create an execution profile that results in a conservative approximation for the worst-case execution path [16]. The analysis results in a reformulated ILP problem for IPET. The Persistence analysis of Ferdinand and Wilhelm contains a correctness issue that Cullmann analyzes [12].

This thesis uses abstract execution for estimating the WCET. In a similar way as in the work of Lundqvist and Stenström [36], abstract execution models the low-level microarchitecture features in the program state and the analysis proceeds in a single pass. For modeling the pipeline, the analysis uses resource usage patterns based on a general 5-stage model [23]. The cache analysis integrates the abstract cache domain of the Must analysis of Ferdinand and Wilhelm [16] into abstract execution.

2.4 Multiprocessor WCET Analysis

Performance-oriented microarchitecture mechanisms introduce different challenges to the WCET problem, due to the weak predictability properties of these mechanisms [29, 38]. Hard embedded-system applications have certification requirements that follow strict safety standards [29, 38]. Therefore, extending the current single-core WCET estimation approaches is not always straight forward.

Predictability is an important requirement that makes it easier to attain guarantees about the behavior of a hard real-time system. Mancuso et al. present a shared-memory scheme that divides the memory in different sections that map to every processing unit. This scheme aims at utilizing an OS-level framework that isolates each processing unit by partitioning the shared memory. The purpose of this approach is to treat each core in a multi-core system independently and apply single-core analysis techniques directly [38].

Other approaches extend single-core WCET techniques to support support multiprocessing systems. Chattopadhyay et al. and Zhang and Jun [6, 52] extend the IPET methodology to analyze multi-core systems. IPET uses ILP for solving an optimization problem that models the program and the architecture. In the case of a multi-core architecture, IPET has to model the multiprocessor hardware. The analysis performs different analyses to deal with the dependencies between tasks running in dedicated processors. The interaction between the tasks occurs usually through the shared memory. More specifically, Zhang and Jun [52] extends the cache-conflict graphs, first described by Li et al. [32], for designing multi-core analysis. Chattopadhyay et al. base their analysis on the pipeline modeling of Li et al. [31] to extend the IPET methodology for multi-core platforms [6].

Pellizzoni et al. use memory traffic arrival curves to record the activity of each core in a multi-core system [47]. These arrival curves represent the upper bound of the memory traffic over time. The method performs a delay analysis that proceeds in two steps. First, the delay analysis calculates the worst-case delay of one task running in isolation and subsequently, uses the arrival curves, in a form of memory access profiling of all tasks, to estimate the worst-case delay of the analyzed task.

Chapter 3

Background

The purpose of this chapter summarizes the background that is necessary for describing the approach of this thesis. The first section, Section 3.1, describes the basic methodology of the KTA tool with focus on the parts that are relevant to the approach of this thesis. The next three sections, i.e. Sections 3.2, 3.3, and 3.4, present the background related to abstract interpretation, which is the basic method that KTA uses for deriving the WCET of a task. This theory is important for defining the approach and describing the contributions of this thesis with regards to the abstract interpretation framework. More specifically, abstract interpretation is the basic framework for the value domain, the cache, the cache hierarchy, and the pipeline states. Finally, Section 3.5 introduces the cache-coherence problem that appears in shared-memory multiprocessor systems. To address this problem, the multiprocessor analysis uses MESI [46], a bus-snooping cache-coherence protocol.

3.1 The KTA Tool

KTH's Timing Analysis (KTA)[1] tool is a static program analysis tool originally developed by David Broman [13, 17]. KTA supports source code in the C programming language and machine code in *executable and linkable format* (ELF) for the MIPS32® *instruction set architecture* (ISA) [25]. In the case of C source code, KTA uses mcb32-gcc[2], a MIPS32® gcc[3] cross-compiler, to generate the binary code. KTA implements two different types of analyses, *Interactive Timing Analysis* [17] and *Abstract Search-based WCET Analysis* [13]. The latter analysis is an ongoing project that this thesis contributes to.

The purpose of this section is to provide an overall picture of KTA that facilitates the understanding of the purpose, contribution, and implementation of this thesis. The following two subsections provide an overview of (1) the methodology and (2) the implementation of the *Abstract Search-based WCET Analysis*.

[1] KTA tool: https://github.com/timed-c/kta

[2] https://github.com/is1200-example-projects/mcb32tools

[3] *the gnu compiler collection* (gcc): https://gcc.gnu.org/

3.1.1 KTA Methodology

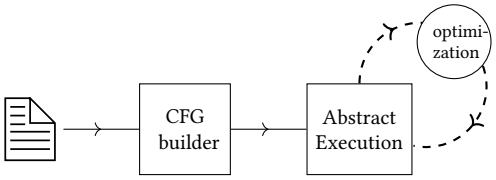


Figure 3.1: KTA tool work flow.

This subsection gives a high-level overview of the main methodology of the *Abstract Search-based WCET Analysis* of KTA. Figure 3.1 illustrates the three main phases of the methodology. These phases are the *CFG generator*, the *Abstract Execution Analysis*, and finally, the *Optimization* phase. The next paragraphs describe these phases.

CFG Generator phase

KTA has two required inputs: (1) the name of the starting point (function name) to analyze and (2) either an ELF object file that KTA parses directly or alternatively, source code written in C. The CFG-builder phase uses mcb32-gcc to compile the C code with a possibility to select the optimization level (KTA optional flag), and consequently, parses the generated binary. This parsing results into the CFG of the program. KTA does not alter the actual assembly code, so that the analyzed code is similar to the actual instructions that the microprocessor will execute. However, the CFG-builder phase adds a number of pseudo instructions that are useful for the next phase, i.e. *Runtime Analysis*. The output of this phase is an assembly-like code in *continuation passing style* (CPS) in OCaml, which is the input to the *Runtime Analysis*.

Runtime Analysis phase

The Runtime analysis is the part of KTA that estimates the WCET of a specific program. This phase receives the program code as an input from the previous phase and uses abstract execution to estimate the WCET of the specified routine. Abstract Execution [20] is a static analysis method which is based on abstract interpretation [9] and has applications in WCET analysis [14, 20]. Given an accurate hardware timing model, the abstract execution analysis of the program provides a *safe* WCET estimation. The method uses abstract values to represent the possible input values of the function.

The resulting WCET of the analysis is a safe WCET approximation. The safeness guarantee of abstract execution derives from the abstract interpretation framework. However, often, the approximated WCET is a overestimation of the actual WCET. The Optimization phase that follows attempts to estimate that actual WCET, using a search-based technique.

Optimization phase

The optimization phase attempts to calculate a tighter WCET (or the actual WCET) by locating the input that leads to the worst-case execution path. The final output is less or equal to the initially estimated WCET. This way, the optimization stage can derive the actual WCET that corresponds to a concrete input combination.

3.1.2 KTA Implementation

The purpose of this subsection is to provide an overall picture of the implementation of KTA in order to facilitate the understanding of the implementation details of the contributed parts (Section 5.1). The focus is on the parts of KTA that are closely related to the implementation of this thesis. More specifically, this section describes the output from the first phase and the second phases, i.e. the *CFG Generator output* and the *Runtime Analysis* phase (see Section 3.1). The *optimization* phase does not have a direct relation to the contributed parts, so there is no description of the this implementation.

CFG Generator output form

The shell command for executing the WCET analysis in KTA is the following:

```
kta wcet file_name.c func_name
```

The *CFG-generator* phase of KTA parses the input program and generates the *control flow graph* (CFG) for the targeted function in *continuation passing style* (CPS) in OCaml. Appendix A shows the output of the CFG generator in CPS for a simple factorial benchmark. The output of this phase includes additional information that the *Runtime Analysis* uses for estimating the WCET (see Subsection 3.1.2). This additional information includes the memory content and the global address.

The representation of the CFG consists of the basic blocks and a basic-block table that contains information about the basic blocks. Each basic block of the CFG forms an OCaml function in CPS, i.e. each basic block takes one input, the main state (mstate), and returns one output, the updated main state.

The body of the function executes abstractly the equivalent of every MIPS instruction that the basic block contains. Every operation updates the program state (pstate), which is part of the main state. Each abstract MIPS instruction corresponds to one OCaml function in the runtime library. The update of the respective abstract value uses the abstract domain implementation of the specific operation. Each basic block in the CPS representation terminates either with a *ret* or with a *next* pseudo instruction. Both instructions correspond to different functions in the runtime library that take care of resuming the execution of the analysis to the block with the highest priority. The priority of each basic block is based on the the distance of this basic block from a return node. This information is part of the basic-block table that the CFG-generator phase generates.

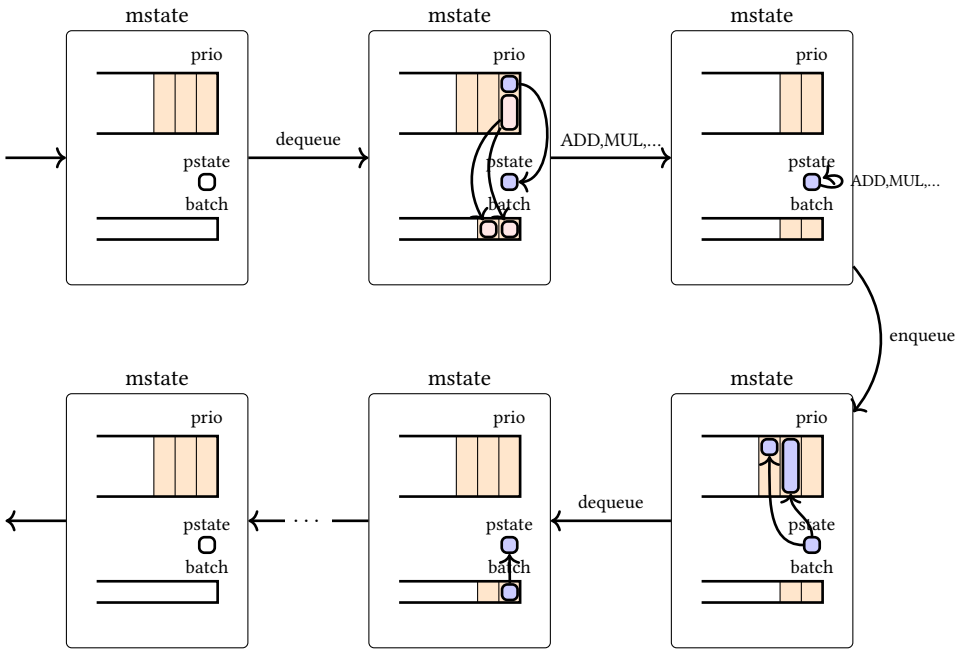


Figure 3.2: Execution sequence of the runtime analysis. The priority queue dequeues the highest priority set of program states. The analysis merges the program states if the number of program states exceeds a threshold, defined in $-bsconfig$. Then, the analysis selects the first of these program states, and finally, proceeds by executing the basic block that the selected program state corresponds to. When the execution is over, the analysis enqueues the updated program state for the destination nodes. Next, the analysis continues with the rest of the batch program states. When they are over, the same procedure continues with the next high priority program state batch.

Runtime Analysis implementation

The runtime analysis phase (see Section 3.1.1) is the part that computes the WCET using abstract execution (see Section 3.3). A basic struct that contains all the program information is the main state. The following code snippet is the definition of the main state (mstate).

```
type mstate = {
  cblock   : blockid;                (* Current basic block *)
  pc       : int;                    (* Current program counter *)
  pstate   : branchprogstate;        (* Current program state *)
  batch    : branchprogstate list;   (* Current batch of program states *)
  bbtable  : bblock_info array;      (* Basic block info table *)
  prio     : pqueue;                 (* Overall priority queue *)
  returnid : blockid;                (* Block id when returning from a call *)
  cstack   : (blockid * pqueue) list;(* Call stack *)
}
```

Figure 3.2 demonstrates how the runtime analysis works. The main state contains, among other fields, the basic-block table (bbtable) and a priority queue (prio). Initially, the priority queue contains only the first basic block, which corresponds to the initial function

(the function to analyze). Every element in the priority queue contains one or more program states. The analysis selects the highest priority element. Before proceeding, the analysis merges the program states of the selected element so that they do not exceed a threshold, which is configurable using command-line option $-bsconfig$. Then, the basic block of the first element of the merged program states starts executing and the rest of the program states are enqueued in *batch*. When the execution reaches the final instruction of the basic block, a *ret* or *next*, the resulted program state (or the two resulted program states) are enqueued to the priority queue ($prio$) and attain the priority that corresponds to their target basic block. The execution continues by executing the rest of the batch program states ($batch$). When the batch queue is empty, the analysis proceeds with the next (current) highest priority program states.

3.2 Abstract Interpretation

Abstract Interpretation is a semantics-based formal framework for static analysis [9]. The framework provides correctness guarantees, an important property for many static analysis applications.

There are various applications that use abstract interpretation to statically analyze a program [7]. Among these applications, there are approaches to the WCET problem that use techniques based on abstract interpretation to extract useful properties of a program and calculate a sound WCET for the program. For example, [5, 15, 28] use approaches based on abstract interpretation to extract information about the execution of a program.

Performing an analysis using concrete representation, i.e. without approximations, is accurate, but leads to very large analysis time, because the number of execution paths in non-trivial programs grows fast. For this reason, static analysis requires some level of abstraction or approximation. The purpose of this abstraction is to represent the program semantics in a domain that abstracts the behavior of the program in a sound way, i.e. using conservative over-approximations. By abstracting the semantics of a program, the analysis is able to extract useful information in finite time. Abstract Interpretation is a framework that defines formal properties of the abstraction for constructing sound static analyses.

This section does not provide a detailed description of the abstract interpretation framework, but an overview that is useful in the chapters that follow. Many of the definitions and structures are based on the following sources: the initial paper of Cousot and Cousot [9], the book *Principles of Program Analysis* [44], and the Introduction to the Abstract Interpretation [3]. These sources provide a more throughout description and complete definitions of the abstract interpretation framework and the related theories.

3.2.1 Definitions

Abstract Interpretation is a formal method based on Lattices, i.e. abstract mathematical structures that are based on *order theory*. A static analysis method based on abstract interpretation uses entities that form a lattice. These entities describe two domains, the

concrete and the abstract domain. The purpose of abstract interpretation is to define a sound translation from the concrete to the abstract domain, so that the analysis uses the abstract semantics to approximate the program information.

The following definitions are necessary for defining the abstract interpretation framework.

The definition of a partially ordered set is the following:

Definition 1. *A **partially ordered set** is a set L that is equipped with a relation, $\leq$, which is (1) reflective $a \leq a, \forall a \in L$, (2) transitive $a \leq b \wedge b \leq c \implies a \leq c, \forall a, b, c \in L$, and (3) antisymmetric $a \leq b \wedge b \leq a \implies a = b, \forall a, b \in L$*

A least upper bound (lub) or supremum is defined as follows:

Definition 2. ***Least upper bound** or **supremum** of a subset S of a partially ordered set $S \subset L(\leq)$, is the least element $l \in L$ that is greater or equal than all elements of S.*

Similarly the greatest lower bound (glb) or infimum is defined as follows:

Definition 3. ***Greatest lower bound** or **infimum** of a subset S in a partially ordered set $S \subset L(\leq)$, is the greatest element $l \in L$ that is lower or equal than all elements of S.*

The definition of a lattice and a complete lattice are the following:

Definition 4. *A **lattice** is a partially ordered set $L(\leq)$ such that: $\forall a, b \in L$, there is $sup = a \cup b$ and $inf = a \cap b$. $L(\leq, \cup, \cap)$ denotes a lattice.*

Definition 5. *A **complete lattice** is a lattice $L(\leq, \cup, \cap)$, such that every subset $S \subseteq L$ has a supremum, $\cup S$ and an infimum, $\cap S$. Hence, L has an infimum, $\bot = \cap\varnothing$ and a supremum $\top = \cup L$. $L(\leq, \bot, \top, \cup, \cap)$ denotes a complete lattice.*

3.2.2 Collecting Semantics

Collecting semantics aims at representing all the reachable states that a program may reach for any input state.

To do that, collecting semantics computes the set of all possible traces based on the semantics of the program. A trace is a sequence of allowed transitions, $Tr = \{s_0 \rightarrow s_1 ... \rightarrow s_n | s_0$ is the initial state, and $s_i \rightarrow s_{i+1}$ is an allowed transition$\}$, according to the program semantics. Let $final : Tr \rightarrow S$ be a function that returns the final state of a trace: $final(\{s_0 \rightarrow ... \rightarrow s_n\}) = s_n$. The set of all reachable states is $S_r = \{s | \exists t \in \mathcal{P}(Tr), s = final(t)\}$.

The set of all traces of all states ordered by inclusion forms a complete lattice [3]. This lattice, representing the concrete state of the program based on the program semantics is the concrete domain, denoted as $L_{\mathcal{C}}$. The state of a program can be a variable or a register (with the concrete domain being e.g. $L_{\mathcal{C}} = \mathcal{P}(\mathbb{Z})$, ordered by inclusion) or the pipeline and the cache state that alter based on the program semantics.

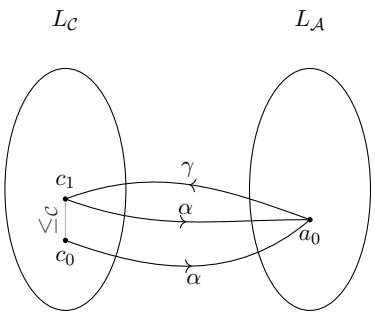


Figure 3.3: Galois Insertion.

3.2.3 Galois Connection - Galois Insertion

Abstract Interpretation performs a form of translation from a concrete domain $L_{\mathcal{C}}$ to an abstract domain $L_{\mathcal{A}}$. This translation makes it possible to apply the program semantics to the abstract domain $L_{\mathcal{A}}$ instead of the concrete domain.

Abstract interpretation defines the properties of these relations, so that the transitions to and from the abstract domain preserve the correctness of the abstraction. In particular, these transition functions form a Galois connection (or the more restricted a Galois insertion) between the two domains.

Galois Connection

Function $\alpha : L_{\mathcal{C}} \to L_{\mathcal{A}}$ is the abstraction function that takes as input a concrete value and returns the abstract representation of this concrete value. Function $\gamma : L_{\mathcal{A}} \to L_{\mathcal{C}}$ is the concretization function that does the opposite, i.e. takes as input the abstract value and returns the concrete value. These functions form a Galois connection, noted as: $(L_{\mathcal{C}}, \leq_{\mathcal{C}}) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} (L_{\mathcal{A}}, \leq_{\mathcal{A}})$, iff [3]:

$$\forall x \in L_{\mathcal{A}}, \forall y \in L_{\mathcal{C}}, x \leq_{\mathcal{A}} \alpha(y) \iff \gamma(x) \leq_{\mathcal{C}} y$$

Galois Insertion

In a Galois connection, there might be several elements, x_1, x_2 of the abstract domain, $x_1, x_2 \in L_{\mathcal{A}}$, that map to the same value in the concrete domain $y = \gamma(x_1) = \gamma(x_2), y \in L_{\mathcal{C}}$ [44]. A Galois insertion restricts the relation, so that every concrete value (e.g. set of integers) maps to one abstract value. That is, for α, γ monotone, it is:

$$\begin{aligned} &\forall x \in L_{\mathcal{A}}, \quad \alpha(\gamma(x)) = x \\ &\forall y \in L_{\mathcal{C}}, \quad \gamma(\alpha(y)) \supseteq y \end{aligned}$$

Figure 3.3 depicts a Galois Insertion.

Operators	Collecting Semantics	Abstract Semantics
$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$	$+^{\mathcal{C}} : \mathcal{P}(\mathbb{Z}) \times \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ $a +^{\mathcal{C}} b = \{x_a + x_b \mid \forall x_a \in a, \forall x_b \in b\}, a, b \in \mathcal{P}(\mathbb{Z})$	$+^{\mathcal{A}} : L_{\mathcal{A}} \times L_{\mathcal{A}} \to L_{\mathcal{A}}$ $a +^{\mathcal{A}} b = \alpha(\gamma(a) +^{\mathcal{C}} \gamma(b)), a, b \in L_{\mathcal{A}}$

Table 3.1: Abstract definition of the addition operator for an integer variable. The collective semantics are sets of integer values, and the definition of the abstract + operator uses Equation 3.2.

3.2.4 Abstract Semantics

The calculation of the abstract semantics requires translating the concrete semantics to abstract semantics.

The definition of the abstract semantics uses the Galois insertion for translating between the concrete domain (the collecting semantics) and the abstract domain. Each abstract operator should satisfy the following condition for maintaining local consistency.

$$f^{\mathcal{C}} \subseteq \gamma \circ f^{\mathcal{A}} \circ \alpha \tag{3.1}$$

Equation 3.2 presents a method to derive the abstract semantics of an operator. This definition is the best possible abstraction for an operator that satisfies the condition for local consistency (Equation 3.1) [3].

$$f^{\mathcal{A}} = \alpha \circ f^{\mathcal{C}} \circ \gamma \tag{3.2}$$

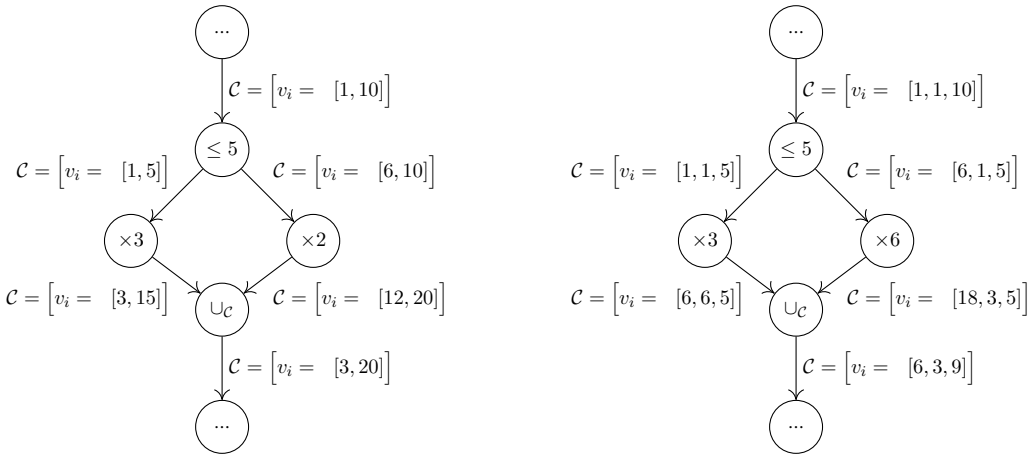
Table 3.1 shows an example of the translation of the + integer operator to the abstract semantics.

3.3 Abstract Execution

Abstract execution (AE) is a static analysis method based on abstract interpretation [43]. Abstract execution has applications in WCET-related analyses, such as the automatic calculation of loop bounds and identification of infeasible paths [20]. Another application of AE is the identification of the input values that result in the worst-case execution path [14].

According to Gustafsson et al., abstract execution is based on a static analysis method, widely used in program checking, i.e. symbolic execution [20]. However, abstract execution uses abstract values, instead of symbols, as inputs and produces abstract values as outputs. The abstract domain definitions follow the abstract interpretation framework and form complete lattices.

During AE, the abstract values represent the set of all possible inputs and values that might occur at every execution point. An abstract state represents the current program state, and every instruction typically updates this state. A branch splits the control flow to different execution paths that generate new program states. Executing all the possible paths may lead to high time and space complexity. For this reason, AE merges the states



(a) Interval Domain **(b)** Congruence Domain

Figure 3.4: Abstract Execution in the (a) Interval Domain, $x_i \in [l, h] \implies l \leq x_i \leq h$ and (b) Congruence Domain, $x_i \in [l, s, n] \implies x_i = l + s \cdot i,\ 0 \leq i < n$.

at specific program points, as for example branches or function returns. Figure 3.2 depicts the basic methodology of the AE execution approach of KTA.

Abstract execution is based on abstract interpretation. Hence, the safeness of the result depends on the abstract domain, the abstract functions, and the abstract harware model. These definitions are based on abstract interpretation, which provides soundness guarantees.

3.4 Abstract Value Domains

In abstract interpretation, an abstract domain forms a complete lattice $L(\leq, \bot, \top, \cup, \cap)$ equipped with a set of monotone functions with type: $L \times L \Rightarrow L$. The interval and the congruence abstract domains are two well-known and widely-used abstract domains. The interval domain represents a variable as an interval and the congruence domain represents a variable as a set of equally dispersed integers. Section 2.2 describes the two domains in more detail. The combination of the two abstract domains can result in a hybrid abstract domain that benefits from both representations. That way, the IC abstract domain represents a value as an interval with a constant stride parameter in the following way: $x =_{\mathcal{A}} [l, s, n] \Rightarrow \gamma(x) \in \{l, l + s, l + 2s, ..., l + (n - 1)s\}, l \in \mathbb{Z}, s, n \in \mathbb{N}^+$.

In addition to the value representation, all the MIPS32® operations are mapped to abstract functions that are sound, i.e. satisfy the local consistency condition. So, the concrete result of the concrete function is a subset of the concrete value of the result of the abstract function applied to the abstract representation of the concrete input: $f_c(x_c) \subset_C \gamma(f_A(\alpha(x_c)))$

3.4.1 Interval Domain

The interval domain [9] is a commonly-used value abstract domain in static analysis. The main advantages of the interval domain are its simplicity and efficiency.

The definition of the interval domain is the following:

$$S_I = \{[l, h] | l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leq h\}$$

The partial ordering on Interval, $\subseteq_I$ is as follows:

$$[l_1, h_1] \subseteq_I [l_2, h_2] \Leftrightarrow l_1 \geq l_2 \wedge h_1 \leq h_2$$

The least upper bound, $[l_1, h_1] \cup_I [l_2, h_2]$, of two intervals, $[l_1, h_1]$ and $[l_2, h_2]$, are:

$$[l_1, h_1] \cup_I [l_2, h_2] = [min(l_1, l_2), max(h_1, h_2)]$$

The greatest lower bound, $[l_1, h_1] \cap_I [l_2, h_2]$, of two intervals, $[l_1, h_1]$ and $[l_2, h_2]$, are:

$$[l_1, h_1] \cap_I [l_2, h_2] = \begin{cases} [max(l_1, l_2), min(h_1, h_2)] & , max(l_1, l_2) < min(h_1, h_2) \\ \varnothing & , otherwise \end{cases}$$

The concretization function is the following:

$$\gamma_I(s_i \in S_I) = \begin{cases} \mathbb{Z} \cup \{-\infty, +\infty\} & , s_i = \top \\ \varnothing & , s_i = \bot \\ \{x \in \mathbb{Z} | l \leq x \leq h\} & , s_i = [l, h] \end{cases}$$

The abstraction function is the following:

$$\alpha_I(s_a \in \mathcal{P}(\mathbb{Z})) = \begin{cases} \bot & , s_a = \varnothing \\ [min(s_a), max(s_a)] & , otherwise \end{cases}$$

3.4.2 Congruence Domain

The congruence domain [19] is a value abstract domain that represents all the equally dispersed integer values. It is also often used in conjunction with interval domain to form more accurate value abstract domains.

The congruence domain is denoted as $a + b\mathbb{Z}$, with $a \in \mathbb{Z}$ and $b \in \mathbb{N}$ and represents the set of all number that are a modulus b, i.e. $\{x = a \pmod b | x \in \mathbb{Z}\}$. The set represents all numbers with distance b having an offset a:

$$S_C = a + b\mathbb{Z}$$

The partial ordering on the congruence domain, $\subseteq_C$, is the following:

$$a_1 + b_1\mathbb{Z} \subseteq_C a_2 + b_2\mathbb{Z} \Leftrightarrow (a_1 - a_2 \in b_2\mathbb{Z}) \wedge (b_1\mathbb{Z} \subseteq b_2\mathbb{Z})$$

The least upper bound $a_1 + b_1\mathbb{Z} \cup_C a_2 + b_2\mathbb{Z}$ and the greatest lower bound $a_1 + b_1\mathbb{Z} \cap_C a_2 + b_2\mathbb{Z}$ of two congruences are:

$$a_1 + b_1\mathbb{Z} \cup_C a_2 + b_2\mathbb{Z} = a_1 + gcd(a_1 - a_2, b_1, b_2)$$

$$a_1 + b_1\mathbb{Z} \cap_C a_2 + b_2\mathbb{Z} = \begin{cases} a + lcm(b_1, b_2)k & , \exists a \in a_1 + b_1\mathbb{Z} \cap a_2 + b_2\mathbb{Z}, k \in \mathbb{Z} \\ \varnothing & , otherwise \end{cases}$$

In the previous formulas, gcd is the *greatest common divisor*, and lcm is the *least common multiple*.

The concretization function is the following:

$$\gamma_C(a + b\mathbb{Z}) = \begin{cases} \{a + kb | k \in \mathbb{Z}\} & , a, b \neq 0 \\ \varnothing & , b = 0 (a + 0\mathbb{Z} = \bot) \\ \mathbb{Z} & , a = 0 (0 + 1\mathbb{Z} = \top) \end{cases}$$

The abstraction function is the following:

$$\alpha_C(s_c \in \mathcal{P}(\mathbb{Z})) = \begin{cases} a + 0\mathbb{Z} = \bot & , s_c = \varnothing \\ x_0 + gcd(\{|x_i - x_j|, \forall x_i, x_j \in s_c\}), x_0 \in s_c & , otherwise \end{cases}$$

3.5 Cache Coherence in Multiprocessor Systems

This section introduces the cache-coherence problem that appears in shared-memory systems and describes MESI, the protocol that the multiprocessor analysis uses to address the cache-coherence problem. MESI is a widely-used bus-snooping protocol that addresses the cache-coherence problem and has a low implementation overhead. The multiprocessor analysis of this thesis considers *symmetric multiprocessor* (SMP) systems, a subclass of shared-memory multiprocessor systems. In an SMP system, all processing units of the multiprocessor system are identical, and a shared bus connects these identical units. This means that all processing units in an SMP system contain identical private cache hierarchy and an identical processing unit. There are multiple applications of SMP architectures in small-scale multiprocessor systems, such as PCs and embedded platforms, for example the Creator ci40 platform [11]. MESI is a very common protocol in small-scale systems because the protocol implementation is simple and reduces the bus traffic compared with other shared-bus snooping solutions. The MESI protocol has four states, but many of the implementation details differ in different hardware implementations. This chapter uses the original protocol description of Papamarcos and Patel [46].

The next parts of this section present some basic background and definitions, introduce the cache-coherence problem and describe the MESI protocol that the multiprocessor analysis of this thesis implements.

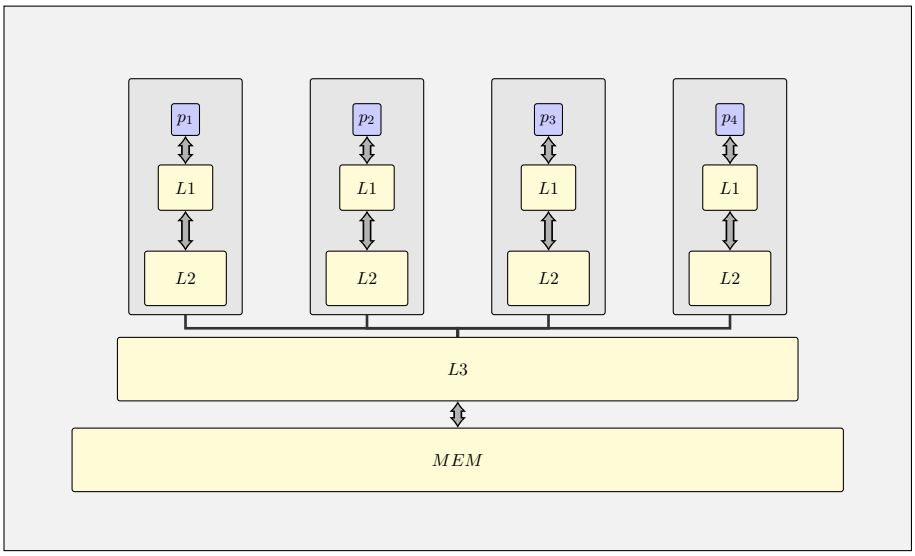


Figure 3.5: Symmetric Multiprocessor System with 2 levels of private caches and one shared cache.

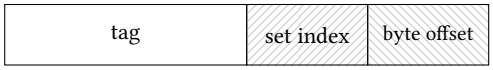


Figure 3.6: An Address consisting of three parts. The least significant part of the address, i.e. *byte offset*, indexes a byte within a block. One block comprises a cache line. The *set index* part indexes a set in a cache. The *tag* part identifies the address block in the cache.

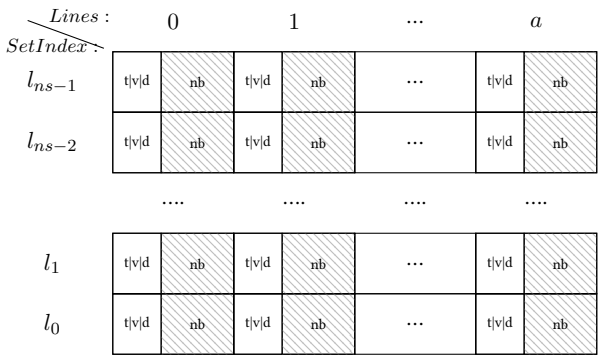


Figure 3.7: An a-way set associative cache with ns sets and nb bytes per block. Each line contains the data (block) and additional fields (denoted as $t|v|d$). These fields are the tag, the valid bit, and the dirty bit. The total size of the cache is $cs = n \cdot a \cdot nb$.

3.5.1 Basic Cache Notation

A cache is a fast memory that often resides near the CPU and contains copies of memory subsets. The purpose of the cache is to exploit the temporal and spatial locality, i.e. the reuse of the same and adjacent data by a program. This section provides only a brief overview of caches and their functionality. More information about caches is available in the book of Hennessy and Patterson. [23, Appendix B].

A cache contains the notion of a *block*, i.e. the smallest memory subset that may appear in the cache. Every memory block maps to specific position(s) in the cache. Figure 3.6 depicts the partition of a memory address in a cache. The least significant bits of an address, namely the *byte offset*, decides on the position of the byte in the block. The *set index* part of the address decides on the position of the memory block in the cache. The first part of the address, i.e. the *tag*, identifies the memory address of the block that resides in the cache. A block together with the tag, and some additional flags, i.e. the valid bit and the dirty bit, constitute a *cache line*. The *valid* bit indicates that the content of a cache is valid and is an low-overhead way to invalidate the a cache line and the total cache. The *dirty* bit indicates that the content of the cache is not consistent with the memory. An important parameter of a cache is the *associativity*. Associativity refers to the number of equivalent cache lines that one memory block maps to. Figure 3.7 depicts an A-way set assotiative cache. Another important parameter in a cache is the replacement policy. The replacement policy dictates which block to replace when a set is full.

3.5.2 Cache-Coherence Problem

Multiprocessor systems with private and shared memory exhibit cache-coherence problems, due to the presence of multiple copies of the same memory locations in the memory hierarchy. The problem appears when multiple private caches share the same memory block these blocks do not always contain the same updated copy of the data. More precisely, the cache-coherence problem has two aspects, coherence and consistency. Coherence concerns the behavior of reads and writes to the same memory location, whereas consistency defines the behavior of writes (and reads) to different memory locations by the same processor [23, Chapter 5].

There are different solutions for maintaining coherence in a shared-memory multiprocessor system. Two of the main techniques are bus-snooping protocols and directory-based cache-coherence protocols. Directory-based cache-coherence protocols maintain the information about the status of a block in one location. Bus-snooping protocols, such as the MESI protocol, use the bus to maintain the status of each cache, coherent [23, Chapter 5]. The next subsection describes the MESI protocol.

3.5.3 MESI Protocol

MESI is a four-state cache-coherence protocol. This definition of the MESI protocol is based on the original description of the protocol by Papamarcos and Patel [46]. The MESI

State	Description
Invalid	The block is not valid.
Exclusive	No other cache contains this block. It is consistent with the shared memory.
Shared	There may be a cache that contains this block. It is consistent with the shared memory.
Modified	No other cache contains this block. The block is modified and inconsistent with the shared memory.

Table 3.2: Description of the four states of the MESI protocol: *Invalid*, *Exclusive*, *Shared*, and *Modified*.

protocol uses a shared bus for communication between the different cores of the SMP. In this analysis, each core accesses the shared memory and at least one level of private cache.

In SMP systems with one bus that connects the independent processing units with the shared memory, this bus is the bottleneck. The reason is that the bus serializes all shared-memory requests. Therefore, these systems cannot scale to a large number of cores. However, MESI is relatively easy and inexpensive to implement, and many embedded multi-core systems implement MESI to maintain coherence.

States

MESI is an extension of MSI that is a three-state write-invalidate bus-snooping protocol with many extensions [23, Chapter 5]. The states of the MSI protocol are: Modified (M), Shared (S), and Invalid (I). Every write to a shared-memory block invalidates (I) all remote copies of this block. A read miss or a read hit of a shared (S) or modified (M) block results in a transition to a shared state (S). MSI allows cache-to-cache transactions when the requested memory block is present in another a remote cache.

MESI extends MSI by introducing an Exclusive (E) state, which indicates that the specific memory block is not present in any of the remote caches. The introduction of the exclusive state (E) improves the performance of the MSI protocol by reducing the traffic on the bus. That is, a write to an exclusive block (E) does not broadcast an invalidate message to the bus. The cache state of a memory block acquires the exclusive (E) state, when the shared memory (and not one of the private remote caches) satisfies the requested memory. If the private cache of a different processing unit contains the requested block, it replies to the request and delivers the data to the requesting cache. Then, the state of the cache block of the requesting cache becomes Shared (S). Table 3.2 describes the definition of the states of the MESI protocol.

Figure 3.8 illustrates all the possible MESI state transitions. On a local read or write re-

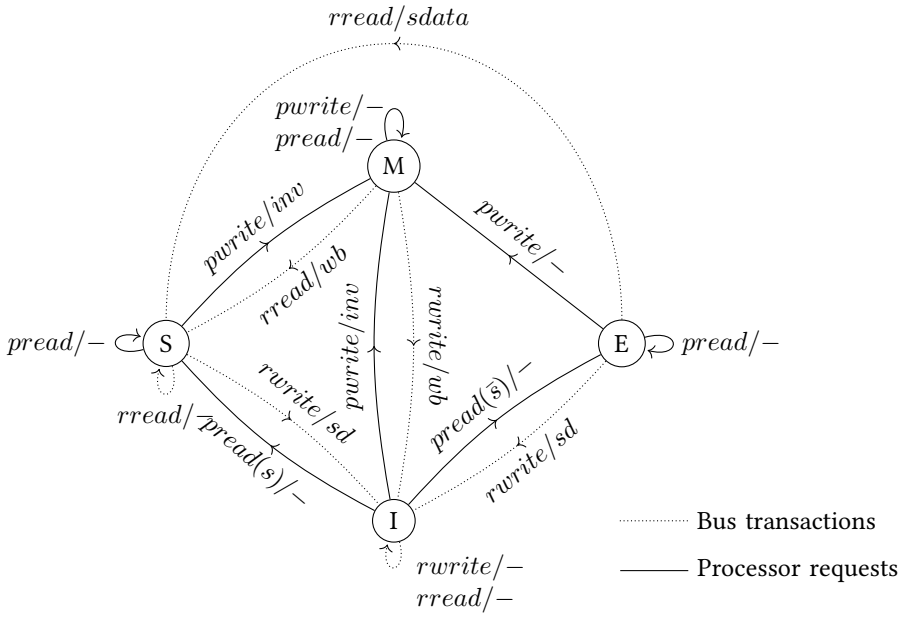


Figure 3.8: MESI protocol, Processor and Bus initiated operations. M, E, S, and I stand for the states of the MESI protocol. The dotted lines correspond to the bus-initiated transactions, whereas the solid lines correspond to the CPU-initiated transactions.

quest, the requesting processor initiates the transition. On a read request, if the core receives a block by another core, the resulting state is shared (S). If the shared memory delivers the requested block the cache block acquires the exclusive state (E). In this figure, all local requests appear in solid lines. The dotted lines represent the remote requests, i.e. requests or invalidates of other cores. On these remote processor requests, the bus snooper initiates the state transition.

Chapter 4

Approach

This chapter describes the main contribution of this thesis in five parts: (1) the value abstract domain, (2) the cache state, (3) the cache hierarchy state, (4) the pipeline state, and (5) the multi-core cache-based analysis.

First, Section 4.1 describes the implementation of a non-relational value abstract domain that replaces the previously implemented value abstract domain in KTA. The implemented abstract domain is based on the *circular linear progressions* (CLP) abstract domain that Sen and Srikant describe in [48] and the modified version of CLP of Källberg in [28].

The following three sections constitute the low-level analysis. The first two sections, namely Section 4.2 and Section 4.3, define the cache and the cache hierarchy states. The definition of the *cache state* is based on the Must analysis of Ferdinand and Wilhelm [16]. The *cache hierarchy state* combines the different cache levels to an abstract cache hierarchy state that represents the cache hierarchy. Subsequently, Section 4.4 describes the pipeline abstract state, which models a classic RISC 5-stage pipeline.

Up to this point, the analysis concerns a single-core cache-based system. The single-core cache-based analysis integrates the low-level analysis to the *program state* of KTA during the *Runtime Analysis* phase. Section 4.5 describes the multi-core cache-based analysis approach that models a symmetric multiprocessor system with shared and private caches. This analysis implements the MESI bus snooping protocol [46] to maintain cache coherence. The analysis aims at estimating the WCET of a task under the presence of spatially and temporally interfering tasks. The multiprocessor analysis makes use of the single-core cache-based analysis in two phases. The first phase gathers necessary information for each of the contributing tasks, and the second phase performs the WCET analysis using the information collected in the first phase.

4.1 Abstract Value Domain

This section focuses on the definition and the description of the IC abstract domain, i.e. the abstract domain that this thesis integrates into KTA. IC is a simplified version of the modified *circular linear progressions* (CLP) domain that Källberg describes in his thesis

[28]. The Modified CLP originates from the work of Sen and Srikant that describe the CLP abstract domain [48]. CLP combines two well-known abstract domains, i.e. the interval [9] and the congruence domain [19]. The CLP domain is more expressive than the interval domain because it is able to express non-continuous intervals. The latter results often in a decreased number of infeasible paths. CLP uses circular representation to represent wraparound [28, 48]. However, KTA, in accordance with the C language specification [26, 27], considers overflow an unknown state. For this reason, in case of an overflow, the register or variable gets the top value, denoted as $\top$, that corresponds to any possible value, i.e. $[-2^{31},\ 2^{31}-1]$.

The IC abstract domain uses the notation of the modified CLP, but is actually a combination of the interval and the congruence domain. The IC domain follows the modified formalization of Källberg [28]. The next parts of this section define the IC domain, present some of the operations that correspond to the MIPS32® instructions, and, finally, present an example to motivate the selection of the IC domain.

4.1.1 Interval-Congruence Domain Definition

The representation of an number in the IC domain is the following:

$$S_{IC} = \{[l, s, n] | l \in \mathbb{Z} \cup \{\top\}, s, n \in \mathbb{N}^+\}$$

The least upper bound $[l_1, s_1, n_1] \cup_{IC} [l_2, s_2, n_2]$ and greatest lower bound $[l_1, s_1, n_1] \cup_{IC} [l_2, s_2, n_2]$ of two sets is:

$$[l_1, s_1, n_1] \cup_{IC} [l_2, s_2, n_2] = [l, s, n], \quad \text{where}$$

$$\begin{aligned} l &= \min[l_1, l_2], \\ s &= \gcd(|l_2 - l_1|, s_1, s_2), \\ n &= \tfrac{h-l}{s}, \text{ where} \\ h &= \max\left(l_1 + s_1(n_i - 1), l_2 + s_2(n_i - 1)\right) \end{aligned}$$

$$[l_1, s_1, n_1] \cap_{IC} [l_2, s_2, n_2] = [l, s, n], \quad \text{where}$$

$$\begin{aligned} l &= \min[l), \\ s &= \operatorname{lcm}(s_1, s_2], \\ n &= \tfrac{h-l}{s}, \text{ where} \\ h &= \min\left(l_1 + s_1(n_1 - 1), l_2 + s_2(n_2 - 1)\right) \end{aligned}$$

The abstraction function is the following function:

$$\alpha_{IC}(\{k_0, k_1, ..., k_n\} | \forall i, k_{i+1} > k_i) = \begin{cases} (k_0, 0, 1), & n = 0 \\ \cup_{IC}(\alpha(\{k_0, ..., k_{n-1}\}), (k_n, 0, 1)), & otherwise \end{cases}$$

Similar to the work of Källberg [28], the concretization function for the signed and unsigned cases are the following:

$$\gamma_{IC_S}([l, s, n]) = [l, l + s, l + 2s, ..., l + (n - 1)s]$$

The unsigned concretization function is:

$$\gamma_{IC_u}([l, s, n]) = \begin{cases} [\{l\}], & l > 0, s = 0 \\ [\{2^w + l\}], & l \leq 0, s = 0 \\ [\{l, l + s, l + 2s, ..., l + (n - 1)s\}], & l > 0, s > 0 \\ [\{l + sn', l + sn' + s, ..., l + sn' + (n - n' - 1)s\}, & \\ \{2^w + l, 2^w + l + s, ..., 2^w + l + (n' - 1)s\}] \mid n' = \left\lceil \left| \frac{l}{s} \right| \right\rceil, & otherwise \end{cases}$$

For modeling of MIPS32® architecture for KTA, the domain is restricted based on the register size. Hence, instead of $(-\infty, +\infty)$, the top element is as follows:

$$\top = [l, s, n], \quad \text{where}$$

$$\begin{aligned} l &\in [-2^{31}, 2^{31} - 1], \\ s &\in [0, 2^{32} - 1], \\ n &\in [1, 2^{32}] \end{aligned}$$

4.1.2 MIPS Operations

The following parts describe the definitions of some of the MIPS32® instructions and their implementation in the IC abstract domain.

Addition/Subtraction

The abstract functions in IC correspond to the correspond to the MIPS32® instructions [25].

The addition corresponds to the MIPS32® *ADD* instruction:

```
ADD rd, rs, rt
```

This operation adds the contents of two abstract values as follows:

$$[l_1, s_1, n_1] +^{IC} [l_2, s_2, n_2] =^{IC} [l, s, n], \text{ where,}$$

$$\begin{aligned} l &= l_1 + l_2 \\ s &= \gcd(s_1, s_2) \\ n &= \tfrac{h-l}{s} + 1) \\ h &= \text{high}(l_1, s_1, n_1) + \text{high}(l_2, s_2, n_2) \end{aligned}$$

The subtraction `SUB` is similar to the addition operation:

$$[l_1, s_1, n_1] -^{IC} [l_2, s_2, n_2] =^{IC} [l_1, s_1, n_1] +^{IC} [-l_2, s_2, n_2]$$

Multiplication

There are two different operations that correspond to the multiplication of two numbers. Both treat the operands as signed numbers. Instruction `MUL` returns the 32 least significant bits of the multiplication of *rs* and *rt* to register *rd*. `MULT` returns the 32 least significant bits of the result to the internal low *LO* and the 32 most significant bits to internal hi, *HI*.

```
MUL rd, rs, rt
```

and

```
MULT rs, rt
```

The multiplication considers only the least significant word. The most significant word takes the $\top$ value. The definition of multiplication in the interval-congruence domain is:

$$\begin{aligned}
[l_1, s_1, n_1] \quad & *^{IC}[l_2, s_2, n_2] =^{IC} [l, s, n], \text{ where,} \\
\\
l &= \min[l_1 * l_2, l_1 * h_2, l_2 * h_1, h_1 * h_2] \\
s &= \gcd(|s_1 * l_2|, |s_2 * l_1|, s_2 * s_1] \\
n &= \tfrac{(h'-l)}{s} + 1 \\
h_1 &= \text{high}(l_1, s_1, n_1) \\
h_2 &= \text{high}(l_2, s_2, n_2)
\end{aligned}$$

4.1.3 Motivation for the IC domain

The IC domain combines the interval and the congruence domains to express sets that consist of disperse integers. The main advantage of the IC domain compared to the interval domain appears in the multiply and the left-shift operations. That is, the IC domain is able to project these operations, so that the cardinality of the resulting set remains the same or increases. On the contrary, the interval domain often results in over-approximations that include infeasible values that may result in infeasible paths. This property improves the expressiveness of the abstract domain in specific cases.

A common case where the IC domain improves the expressiveness of the analysis is code resulting from a common compiler optimization strategy, namely *strength reduction transformation* [1, Chapter 9]. The compiler applies this optimization in loops that operate on arrays. Listing 4.1 shows an example of such code. In every loop iteration, the value of the loop index increases by a constant number. This index accesses the array elements by adding the array element size multiplied by the index to the array pointer. The IC domain is able to project the multiplied value and preserve the number of elements in the abstract

value of the induction variable. On the contrary, the interval domain results in a more conservative domain that contains infeasible paths that the analysis can never satisfy.

Code snippets 4.1 and 4.2 show the KTA analysis execution steps for the C code snippet in Listing 4.1 for the interval and the IC domain, respectively. In this case, the IC domain is able to terminate the analysis, whereas the interval domain fails.

```
1    int a[100];                  // Global array of integers
2    void map_2(int n) {
3      int i;                     // The step of the for loop is 1
4      for(i=0; i<n; i++) {       // and the loop bound is n.
5        a[i] = 2*a[i];           // The strength reduction transformation
6      }                          // multiplies the index by 4
7    }
```

Listing 4.1: A simple code example that illustrates the effect of the IC abstract domain on the expressiveness of the analysis. A *for* loop accesses every array element using the index variable *i*. The *strength reduction transformation* optimization uses the array element size as an index step. To do that, the termination variable is multiplied by the array element size. The interval domain is not able to project this multiplication, resulting in a more conservative set that can never be satisfied (see example Executions 4.1 and 4.2).

			#		#		#		#	
	...		#		#		#		#	
	bleqz	$a0,ex1	#		#		#		#	
	sll	$a0,$a0,2	#	$a0=[1,10]*4=[4,40]	#		#		#	
	lui	$v0,64	#	$v0=64	#		#		#	
	addiu	$v0,$v0,a	#	$v0=a	#		#		#	
	addu	$a0,$v0,$a0	#	$a0=[a+4,a+40]	#		#		#	
	lw	$v1,0($v0)	#	$v1=a[0]	#		#		#	
l0:			#	loop 1	#	loop 2	#	...	#	loop 10	#	...
	sll	$v1,$v1,1	#	$v1=a[0]«1	#	$v1=a[1]«1	#		#	$v1=a[9]«1	#	
	sw	$v1,0($v0)	#	a[0]=$v1	#	a[1]$v1	#		#	a[9]=$v1	#	
	addiu	$v0,$v0,4	#	$v0=a+4	#	$v0=a+8	#		#	$v0=a+40	#	
	beq	$v0,$a0,ex1	#	ex1: $a0=a+4	#	ex1: $a0=a+8	#		#	ex1: $a0 = a+40	#	
			#	l0: $a0=[a+5,a+40]	#	l0: $a0=([a+5,a+7],	#		#	l0: $a0=([a+5,a+7],	#	
			#		#	[a+9,a+40])	#		#	[a+9,a+11],	#	
			#		#		#		#	,...,[a+37,a+39])	#	
	lw	$v1,0($v0)	#	$v1=a[1]	#	$v1=a[2]	#		#	$v1=a[10]	#	
ex1:			#		#		#		#		#	
	...		#	($a0=a+4)	#	($a0=a+8)	#		#	($a0=a+40)	#	

Execution 4.1: Abstract execution of the code snippet shown in Listing 4.1 using the interval domain. The first column shows the MIPS assembly code, compiled with the mcb32 gcc crosscompiler with the "-O1" flag. The following columns show the result of the abstract execution iterations, when using the interval domain. In this case, the WCET analysis of KTA does not terminate.

Limitations The current implementation of the IC domain cannot handle some of the cases of the *strength reduction transformation* optimization [1, Chapter 9]. More specifically, the IC domain cannot handle cases when the array element size is not a power of two because the compiler splits the multiplication to a sum of left-shift operations, for example $100 = 64 + 32 + 4$. An example of such a case is a *for* loop accessing the rows of a 2-dimensional array, when row size is not a power of two (see Listing 4.2). The `ADD` abstract operation treats each IC set as independent and reduces the stride, $s_1 \neq s_2$, to the $\gcd(s_1, s_2)$ of the two abstract values. For example, an initial interval $[1, 10]$ is $[1, 1, 10]$

	...		#		#		#		#	
	bleqz	$a0,ex1	#		#		#		#	
	sll	$a0,$a0,2	#	$a0=[1,1,10]*4=[4,4,10]	#		#		#	
	lui	$v0,64	#	$v0=64	#		#		#	
	addiu	$v0,$v0,a	#	$v0=a	#		#		#	
	addu	$a0,$v0,$a0	#	$a0=[a+4,4,10]	#		#		#	
	lw	$v1,0($v0)	#	$v1=a[0]	#		#		#	
l0:			#	loop 1	#	loop 2	#	...	#	loop 10
	sll	$v1,$v1,1	#	$v1=a[0]«1	#	$v1=a[1]«1	#		#	$v1=a[9]«1
	sw	$v1,0($v0)	#	a[0]=$v1	#	a[1]=$v1	#		#	a[9]=$v1
	addiu	$v0,$v0,4	#	$v0=a+4	#	$v0=a+8	#		#	$v0=a+40
	beq	$v0,$a0,ex1	#	ex1:$a0=a+4	#	ex1:$a0=a+8	#		#	ex1:$a0=a+40
	lw	$v1,0($v0)	#	l0: $a0=[a+8,4,9])	#	l0: $a0=[a+12,4,8]	#		#	l0: $a0=[]
			#	$v1=a[1]	#	$v1=a[2]	#		#	$v1=a[10]
ex1:			#		#		#		#	
	...		#	($a0=a+4)	#	($a0=a+8)	#		#	($a0=a+40)

Execution 4.2: Abstract execution of the code snippet shown in Listing 4.1 using the IC domain. The first column shows the MIPS assembly code, compiled with the mcb32 gcc crosscompiler with the "-O1" flag. The following columns show the result of the abstract execution iterations, when using the interval domain. In this case, the WCET analysis of KTA terminates.

in the IC domain; multiplying it with $100 = 64 + 32 + 4$ results in: $([1, 1, 10] << 6) +^{IC} ([1, 1, 10] << 5) +^{IC} ([1, 1, 10] << 2) = [64, 64, 10] +^{IC} [32, 32, 10] +^{IC} [4, 4, 10] = [96, 32, 28] +^{IC} [4, 4, 10] = [100, 4, 226]$, instead of $[1, 1, 10] *^{IC} 100 = [100, 100, 10]$ The result leads to a more conservative IC set that contains infeasible paths. Hence, the analysis does not terminate. Section 7.2 discusses enhancements to the abstract domain or the runtime analysis that may improve the expressiveness of the abstract domain.

```
int a[100][100];            // Global array of integers
void map_2(int n) {
  int i;                    // The step of the for loop is 1
  for(i=0; i<n; i++) {      // and the loop bound is n
    a[i][0] = 2*a[i][0];    // The array pointer increases by 4*100.
  }
}
```

Listing 4.2: A simple code example that illustrates the limitations of the IC abstract domain with regards to expressiveness. A *for* loop accesses every array element using the index variable *i*. The *strength reduction transformation* optimization uses the array element size as the index step. In this case, number $4 \cdot 100$ is not a power of two and the result of the multiplication contains infeasible paths. Hence, the analysis cannot terminate.

4.2 Cache abstract state

The abstract cache model for the KTA tool models a cache with *least recently used* (LRU) replacement policy. LRU is a widely-used replacement policy that has deterministic behavior. The LRU policy replaces the block in the cache set that has been requested by the CPU less recently. Many determining parameters, such as the associativity, the block size, and the cache size are configurable. The cache model is based on the Must analysis of Ferdinand and Wilhelm [16]. KTA integrates the cache abstract state as part of the program state (see Section 3.1.2) in the Runtime Analysis (see Section 3.1.1).

The rest parts of this section describe the cache *state semantics* and the cache operations, i.e. *update*, *join*, and *execution time*.

4.2.1 Semantics

Let nb be the number of bytes in a block, a, the associativity, ns, the number of sets in cache, and cs, the cache size. The relation between these parameters is: $cs = n \cdot a \cdot nb$ Figure 3.7 shows an a-way set associative cache, with ns sets. The block size is nb, and the total size is cs. Figure 3.6 shows how the cache indexes an address. The less significant bits index the byte within a block. The following field indexes the set that the address maps to. The last field identifies the memory block in the cache.

The cache analysis models the cache state, C, as an ordered set of ns ordered cache sets, S. Each cache set, $s \in S$ contains a cache lines, with a being the associativity of the cache. L represents a set of cache lines (cache blocks) with the same LRU priority. The LRU priority decides which cache line(s), $l \in L$, to replace in case of a conflict. That is, in case of a conflict, the cache replaces the cache line(s) with the highest priority. According to LRU, the priority of the elements of L is based on the memory accesses by the CPU. When the cache requests a new block, this new block acquires the lowest priority (most recently used). The cache analysis updates the rest of the blocks by increasing their priority, accordingly. In case that the cache line is full, i.e. contains a blocks, then the LRU policy replaces the least recently used block. If the replaced block contains modified data, the cache writes back the data to the memory. The abstract state does not duplicate the data, but the execution time contribution of the cache considers the write back.

Abstract Block

Let nb be the number of bytes in a block. Let $byte(m) = (m)\%(log_2(nb))$ be a function that given the memory address, returns the byte index within a block. Let M be the set of all possible addresses. Equation 4.1 defines the set of cache blocks, $B \subseteq M$, as the set of the addresses that corresponds to the first byte of a cache block.

$$B = \{b \,|\, b \in M \wedge byte(b) = 0\} \tag{4.1}$$

Abstract Cache Line

Let $set : M \rightarrow \mathbb{N}$ be a function that returns the cache set index of the input address. It is: $set(m) = (m/log_2(nb))\%log_2(L)$. Equation 4.2 defines a cache line as a set of blocks. The abstract line, L, represents all cache lines with a specific LRU priority. More that one lines can have the same priority, because of the join operation that merges two cache states. Under a join, in case the two states contain the same cache block, the block acquires the highest of the two states (less recently used). So, two cache lines may acquire the same priority when both cache lines are present in the two caches and each has the

resulted priority in one of the two cache states (see Section 4.2.3). A cache line, $l \in L$, cannot contain more elements than the associativity, a.

$$L = \{l \,|\, l \in (\mathcal{P}(B)) \wedge (|l| \leq a) \wedge (\forall b_i, b_j \in l, set(b_i) = set(b_j))\} \tag{4.2}$$

Abstract Cache Set

Let $lset : L \rightarrow \mathbb{N}$ be a function that takes one input, $l \in L$, and returns the cache set index that corresponds to l, i.e. $lset(l) = set(b) | b \in l$. Equation 4.3 defines the cache set type, S. A cache set, $s \in S$, is an ordered set of lines, $l_i \in L, i \in [0, a-1)$. All cache lines in a cache set map to the same cache set. The number of elements of each cache set corresponds to the cache associativity. The position of each element in the cache set corresponds to the priority of this element. The priority depends on the access order of the lines, i.e. the most recently used set has the lowest priority, e.g. l_0 corresponds to the lowest priority and l_{a-1} to the highest.

$$S = \left\{ \begin{array}{ll} (l_0, l_1, ..., l_{a-1}) \,|\, \forall i, j \in [0, a), & l_i, l_j \in B \wedge \\ & lset(l_i) = lset(l_j) \wedge \\ & \left(\sum_{k=0}^{i} |l_k| \leq i + 1 \right) \end{array} \right\} \tag{4.3}$$

The last part of Equation 4.3, i.e. $\forall i \in [0, a), \sum_{k=0}^{i} |l_k| \leq i + 1$, expresses an additional requirement. That is, the total number of lines in a cache set should not exceed the associativity, and a cache set with priority i cannot contain more elements than the free lower priority cache sets.

Abstract Cache State

Equation 4.4 defines the cache state, C. Let ns be the number of sets in a cache, a cache state, $c \in C$, consists of ns caches sets. Each cache set, $s \in S$, corresponds to a combination of cache lines that map to the same set.

$$C = \{(s_0, ..., s_{ns-1}) \,|\, \forall i \in [0, ns) \,, (s_i \in S) \wedge (\forall l \in s_i, \, \forall bs \in l, \, \forall b \in bs, \, set(b) = i)\} \tag{4.4}$$

4.2.2 Update

The definition of the update operation is similar to the respective definition of the Must analysis [16]. The cache state is part of the program state and each memory operation, i.e. *read* or *write*, affects the cache state. Diagrams 4.1 show the actions that follow a *read* and a *write* transaction. These transactions may result in a cache update operation both upon a cache HIT and a cache MISS. A cache HIT results in an *update* of the replacement priorities. A cache MISS results in an *insert* followed by an *update* of the replacement

priorities. A cache MISS may result in a *write-back* when the cache set is full and the replaced block is dirty, i.e. it contains modified data.

Cache Set Update

Let $sline : S \times \mathbb{N} \to \mathcal{P}(L)$ be a function that takes two inputs, a cache set, $s \in S$, and a priority, $i \in [0, a)$, and returns the set of blocks with priority i in the cache set s, i.e. $sline(s, i) = l_i, l_i \in s$. The *cache set update* function updates the replacement priorities and inserts a new block if it is not present in the cache. Figures 4.2a and 4.2b show the state update of a cache on a hit and a miss, respectively. Equation 4.5 defines the cache set update operation, $\mathcal{U}_S : S \times B \to S$.

$$\mathcal{U}_S(s, b) = \begin{cases} (l_0 \leftarrow \{b\}, \\ \ l_i \leftarrow s[i-1], \mid i \in [1, j] \\ \ l_j \leftarrow s[j-1] \cup (s[j] - \{b\}) \\ \ l_i \leftarrow s[i] \mid i \in [j+1, a-1]), & \exists j \in [0, a-1], b \in s[j] \\ \\ (l_0 \leftarrow \{b\}, \\ \ l_i \leftarrow s[i-1] \mid i \in [1, a-1]), & otherwise \end{cases} \tag{4.5}$$

Cache State Update

The *cache update* function, $\mathcal{U}_C : C \times B \to C$, updates the cache state, where the requested address $m \in M$ might reside. Equation 4.6 defines the cache update function.

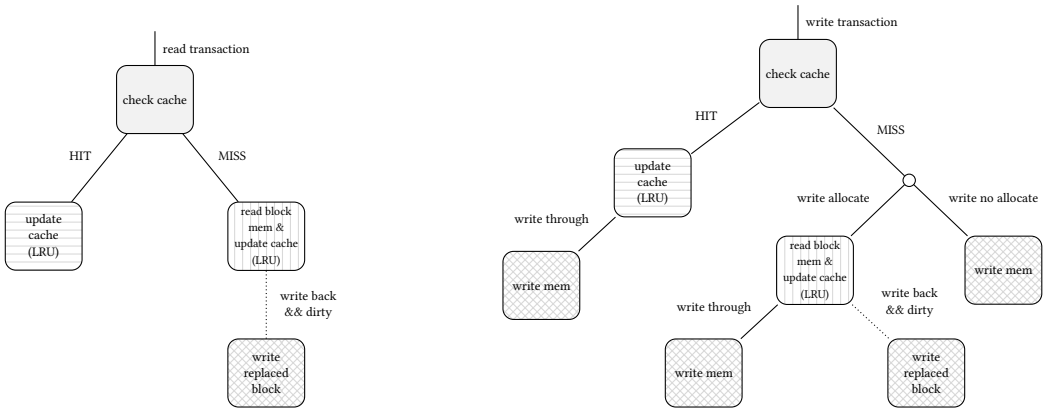
$$\mathcal{U}_C(c, b) = \begin{array}{ll} (c_j \leftarrow \mathcal{U}_S(c[j], b), & \mid set(b) = j, \\ \ c_i \leftarrow c[i], & \mid otherwise) \end{array} \tag{4.6}$$

The actions following a *read* and *write* operation depend on the cache policies. Figure 4.1 demonstrates the possible result upon a cache transaction. The main configuration parameters used for this model are *write allocate* (WA) and *write no allocate* (WNA), and *write back* and *write through*. The first two, allow and, respectively, prohibit loading the block to the cache in case of a write miss. The last two write policies are complementary; *Write back* delays and *write through* forces the immediate write of a modified block to the memory.

Read Update

The *read update* function is equivalent to a *cache update* under both a MISS and a HIT. Equation 4.7 defines the cache update function under read, $\mathcal{U}_{C_R} : C \times B \to C$.

$$\mathcal{U}_{C_R}(c, b) = \mathcal{U}_C(c, b) \tag{4.7}$$



(a) Read Cache **(b)** Write Cache

Figure 4.1: Cache read and write transaction diagrams for different write policies. The horizontal-patterned boxes represent the cache update case, when the update operation reorders the priorities after a HIT. The vertical-patterned boxes correspond to a cache update that inserts a new block to the cache and reorders the priorities, respectively. The latter case is a result of a cache read MISS, and cache write MISS for write-allocate policies. In these cases, inserting a new block can result in replacing a dirty block. The crosshatch-patterned boxes represent all write-to-memory cases, both due to a write-through/write-no-allocate policies and due to the replacement of a dirty block.

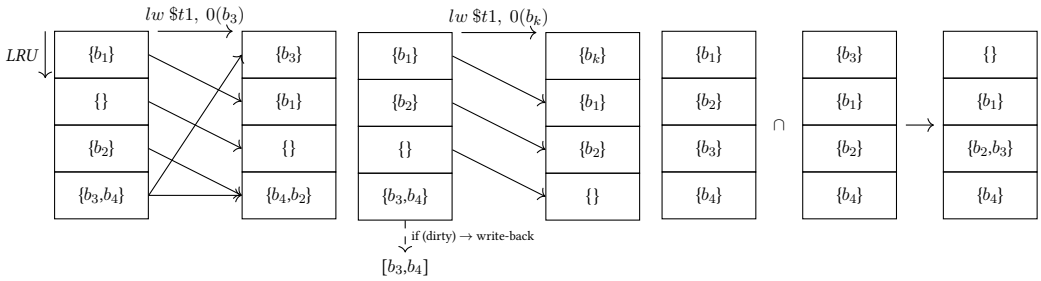
Write Update

The main difference between a read and a write update function is in the case of WNA policy under a MISS. In this case, the cache state remains unchanged. Equation 4.8 defines the *write update* function.

$$\mathcal{U}_{C_W}(c,b) = \begin{cases} c, & WNA \wedge \nexists h \in [0,a), b \in s[h] \\ \mathcal{U}_C(c,b), & otherwise \end{cases} \tag{4.8}$$

4.2.3 Join

The definition of the *cache join* function is similar to the respective definition of the Must analysis [16]. When a merge between two program states occurs, the join operation merges the two cache states to one cache state. The cache join function applies the set join function to the equivalent cache sets of the two cache states.



(a) Update LRU - Cache Hit

(b) Update LRU - Cache Miss (read or write (with write-allocate policy))

(c) Join LRU

Figure 4.2: The abstract model of a 4-way set-associative cache with LRU replacement policy. The LRU replacement policy replaces the block that has been least recently accessed by the cpu, i.e. the age it was accessed. The figure depicts the sets order by the access order, i.e. the top set of lines has been accessed more recently, whereas the last line is the least recently used line. When the cache line is full, the last set will be replaced.

Cache Set Join

Figure 4.2c illustrates the cache set join operation of an LRU cache. Equation 4.9 defines the cache set join operation, $\mathfrak{J}_{\mathfrak{S}} : S \times S \to S$.

$$\mathfrak{J}_S(s_1, s_2) = (l_k \leftarrow \{b \,|\, \exists i, j \in [0, a)\,,\, b \in s_1[i] \wedge b \in s_2[j]\,,\, k = max(i, j)\} | k \in [0, a)) \quad (4.9)$$

Cache State Join

The *cache state join* operation applies the *cache set join* operation to the equivalent cache sets of the two cache states. Equation 4.10 defines the cache join operation, $\mathfrak{J}_C : C \times C \to C$.

$$\mathfrak{J}_C(c_1, c_2) = (\mathfrak{J}_S(c_1[i], c_2[i]) | \, \forall i \in [0, ns)) \quad (4.10)$$

4.2.4 Execution Time

The contribution of a cache to the execution time depends on the cache characteristics, such as the size and associativity. In this model, the contribution of the cache to the execution time is the access time (or hit time) and is configurable for every cache model (see Section 5.1.3). Let $\mathrm{access\_time}$ be a function, $\mathrm{access\_time} : C \to \mathbb{N}$, which returns the cache access time. Equation 4.11 defines the *execution time* operation, $\mathcal{T}_C : C \to \mathbb{N}$.

$$\mathcal{T}_C(c) = \mathrm{access\_time}(c) \quad (4.11)$$

4.3 Cache Hierarchy abstract state

In more complex systems, one cache level is not sufficient to hide the memory access latency. Therefore, most modern systems use multiple levels of caches. The higher the level of the cache, the larger the cache. In case of a cache MISS in one level, the request is propagated to the next cache level[1]. In case no cache contains the requested memory address, then the main memory services the request. Every cache level contributes to the total memory access latency with an access time. Typically, the higher the cache hierarchy level, the larger the access time to the cache.

The cache hierarchy analysis assumes that the block size of every level is the same. The next sections describe the *cache semantics* and the *update*, *join*, and *execution time* operations.

4.3.1 Semantics

Let nl be the number of cache levels in the cache hierarchy, the hierarchy consists of different levels of caches. A cache level can be unified or separate, i.e. the instruction and the data caches share the same cache or different, respectively. The representation of a cache level in the cache hierarchy analysis is the following:

$$C' = (c_i, c_d) \vee c_u, \ with \ c_i, c_d, c_u \in C$$

Equation 4.12 defines the cache hierarchy for unified or separate caches. Equation 4.13 defines a simplified version, that does not distinguish between unified and separate caches.

$$H = \{(c_1, c_2, ..., c_{nl}) \,|\, \forall i \in [1, nl], c_i \in C'\} \tag{4.12}$$

$$H = \{(c_1, c_2, ..., c_{nl}) \,|\, \forall i \in [1, nl], c_i \in C\} \tag{4.13}$$

For simplicity reasons, the rest parts of this section and chapter use the simplified cache hierarchy type of Equation 4.13. This definition does not consider the case of non-unified instruction and data caches. This simplification does not affect the actual operations because the instruction and the data operations are independent.

4.3.2 Update

The *cache hierarchy* update operation updates all caches until the level where the requested memory address resides. All the subsequent caches remain unchanged. The *cache hierarchy update* updates the caches by applying the *cache update* operation (see Equation

[1]Often the memory request is propagated simultaneously to all cache levels and the memory to hide the memory access latency.

4.6). In case that the requested memory address is not present in any cache, the *cache hierarchy* update function updates all caches. Equation 4.14 defines the *cache hierarchy* update operation, $\mathcal{U}_h : H \times B \to H$.

$$\mathcal{U}_h(h,b) = \begin{cases} (c_i \leftarrow \mathcal{U}_c(h[i],b), & | \, i \leq j, \\ \ c_i \leftarrow h[i] & | \, i > j), \quad (\exists j \leq nl, b \in cset(h[j],b)) \wedge \\ & \quad\ (\forall k < j, b \notin cset(h[k],b)) \\ (c_i \leftarrow \mathcal{U}_c(h[i],b)), & otherwise \end{cases} \tag{4.14}$$

4.3.3 Join

The *join* operation of a cache hierarchy abstract state applies the cache join operation to each cache level. Equation 4.15 defines the join operation $\mathfrak{J}_h : H \times H \to H$.

$$\mathfrak{J}_h(h_1,h_2) = (c_i \leftarrow \mathfrak{J}_c(h_1[i],h_2[i]) | \forall i \in [0,l)) \tag{4.15}$$

4.3.4 Execution Time

Each access to a cache level contributes to the total WCET. The total *execution time* is the sum of all cache access times. This definition assumes that the access time of each cache is independent. Let $cache\_set : C \times B \to \mathcal{P}(L)$ be a function that returns a of cache lines, $l_s \in \mathcal{P}(L)$, that corresponds to the requested block, b. Equation 4.16 defines the *execution time* contribution of a memory operation that requests address m, given a cache hierarchy, h.

$$\mathcal{T}_h(h,b) = \begin{cases} \sum_{i=1}^{j} \mathcal{T}(h[i]), & \exists j \leq nl, b \in cache\_set(h[j],b) \\ \sum_{i=1}^{nl} \mathcal{T}(h[i]) + N_m, & otherwise \end{cases} \tag{4.16}$$

4.4 Pipeline abstract state

The purpose of this section is to describe the pipeline analysis by describing the pipeline properties and defining the operations of the abstract pipeline state. The first subsection gives an overall description of the pipeline model, whereas the second subsection describes the abstract pipeline semantics.

4.4.1 Pipeline Definition

Most modern processors use instruction pipelining to exploit instruction level parallelism. A pipeline aims at reducing the latency of each instruction by dividing the instruction execution path into shorter units, called pipeline stages. Different instructions reside in consecutive pipeline stages at every cycle. For optimal performance, each pipeline stage

should be completely independent. However, most pipelines exhibit dependencies between different pipeline stages. These dependencies, called hazards, reduce the full capability of the pipeline.

The purpose of this pipeline analysis is to model a simple 5-stage RISC pipeline. The abstract pipeline model is not hardware specific and uses resource-usage patterns for modeling the pipeline dependencies. The model of the pipeline is a 5-stage RISC-based pipeline, with the following stages: Instruction Fetch (F), Instruction Decode (D), Execute (E), Memory (M), and Write Back (W).

Hazards

There are three main types of hazards in a classic RISC pipeline: *data hazards*, *control hazards*, and *structural hazards* [23]. Data hazards occur when an instruction uses the result of a preceding instruction. The waiting instruction stalls the pipeline while waiting for the result of the previous instruction. Control hazards happen when the address of the next instruction depends on the result of a previous instruction. This hazard appears in branch instructions. For dealing with control hazards, processors implement techniques, such as branch prediction or the so-called delay slot. Branch prediction selects one of the two branches based on different methodologies and continues the execution using the predicted destination. In case the prediction was wrong, the pipeline has to flushes all executed instruction and fetch the correct destination. The branch delay slot refers to the introduction of an instruction that follows the branch instruction. The execution and completion of this instruction does not depend on the branch destination. The MIPS32® ISA uses delay slot. So, this analysis implements delay slot. Finally, structural hazards occur when different pipeline stages use the same resources. The following paragraphs describe the hazards of this pipeline analysis.

Data Hazards: Data hazards happen when an instruction depends on the outputs from previous instructions. Usually, the outputs of an instruction are already available at the end of either the Execute (E) stage or the Memory (M) stage. The dependent instruction requires the input data at the beginning of the Execute (E) stage or, in special cases, at the beginning of the Decode (D) stage.

The number of cycles of the Execute (E) stage differs for different instructions. In the cases that the number of cycles in the execution stage is larger than one, the model assumes that the instruction occupies the Execute (E) stage until the completion of the execution stage. That assumption is conservative because some instructions, such as DIV and MULT, execute in independent units that include separate pipelines. However, this is the behavior of the pipeline in the worst case, i.e. when the next instruction depends on the output of one of the instructions that use an independent execution unit.

This pipeline analysis implements operand forwarding, i.e. the delivery of the dependent operands to the requesting stage when the dependent operands are available. Operand forwarding eliminates all data hazards except for two cases. The first case occurs when an instruction requires as input the result of a preceding memory load operation, e.g. a

lw instruction (see Figure 4.3b). In this case, the result of the load instruction is not ready before the end of the Memory (M) stage. Therefore, the instruction that depends on this result cannot proceed to the Execute (E) stage, but instead stalls until the result is available. The second case occurs in branch instructions, i.e. the branch decision depends on the result of a preceding instruction. Depending on the instruction, the result is ready at the end of the Execute (E) stage or at the end of the Memory (M) stage. However, the operand comparison and the branch destination calculation is performed at the Instruction Decode (D) stage, so that the branch destination instruction is known and ready for execution after the delay slot. To resolve this case, the branch stalls until the dependent operator is ready[2]. Figure 4.3c shows the effect of such a case on the pipeline.

Control Hazards: Control hazards occur when the next instruction address is not known. The second stage of the pipeline, i.e. Instruction Decode (D) stage, implements both the calculation of the target address and the branch decision. So, the pipeline knows the destination address and, hence, the next instruction to fetch at the end of the D stage. This means that the next instruction is available at the end of the D stage, introducing an one-cycle delay. However, the delay-slot hides this one-cycle delay. Hence, the pipeline does not display any control hazard.

Structural Hazards: Structural hazards occur when two different pipeline stages share the same resource. This analysis assumes the presence of one bus for both data and instructions. For example, when a miss occurs in the Instruction Fetch (F) stage of an instruction and the previous instruction accesses the memory at the M stage, the memory transactions are serialized, and, thus, the pipeline stalls. Figure 4.3d illustrates this hazard.

4.4.2 Pipeline Abstract Semantics

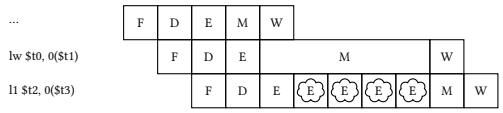
A pipeline state, ps, consists of the five values representing the five pipeline stages. Each value represents the stage-release cycles, i.e. the number of relative cycles after which each stage is free. The stage-release cycles are normalized because the pipeline uses stalls and is not speculative. Additionally, the pipeline contains the destination register of the previous instruction, rd (or rt), and the time its result is ready t_{rd}.

The representation of the pipeline is the following:

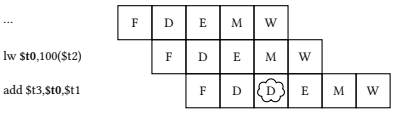
$$ps = ([t_F, t_D, t_E, t_M, t_W], (rd, t_{rd})), \text{ with } t_W \geq t_M \geq t_E \geq t_D \geq t_F \tag{4.17}$$

The update state includes the destination register, rd, in the case that the instruction is a memory read operation. The instruction following the memory operation stalls when the some of the source registers is equal to the destination register of the previous read

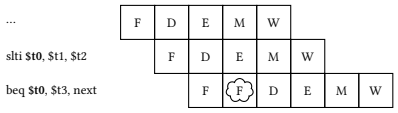
[2] This case does not occur in some hardware implementations that merge the Decode (D) and the Execute (E) stage, for example the PIC32MX3XX/4XX family [40].



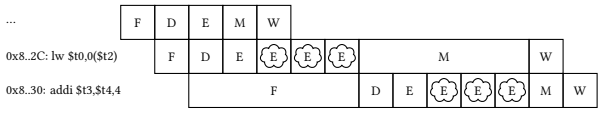
(a) Pipeline without dependencies or hazards. In case a stage is not able to complete in one cycle, the pipeline stalls and the next instruction cannot enter the delayed stage.



(b) Pipeline RAW dependency. The instruction following a memory load requires the memory result at the beginning of the execution stage (E). However, this result is available at the end of the memory stage of the load instruction. This results in a data RAW hazard.



(c) Pipeline RAW dependency. The instruction preceding a branch instruction calculates one of the operands. This operand is available at the end of the execution staget (E). The branch instruction needs this result at the beginning of the instruction decode (D) stage because that is when the branch result is calculated.



(d) Structural hazard for memory accesses. This model assumed serialized memory accesses and an instruciton miss (address 0x8...30) causes all the shared-memory misses that overlap the instruction miss penalty to stall. In this example, the memory operation results in a miss as well.

Figure 4.3: Pipeline hazards that are implemented for the WCET analysis for MIPS32® ISA. *F*, *D*, *E*, *M*, and *W* correspond to the five pipeline stages, i.e. instruction fetch, instruction decode, execute, memory, and write back. The bubbles (clouds) represent the cycles during which the processor stalls.

memory instruction. The update state includes also one or two source registers, rt and rs, depending on the instruction type. The update state has the following form:

$$dps = ([dt_F, dt_D, dt_E, dt_M, dt_W], (rd, rt, rs)) \tag{4.18}$$

4.4.3 Update

The *pipeline update* function updates a pipeline state ps using the update state (see Equation 4.18) that corresponds to the new instruction. The order of the calculation is important because each pipeline stage depends on the previous pipeline stage. Equation 4.19

defines the *pipeline update* function.

$$\mathcal{U}_{ps}(ps, dps) = \begin{cases} (t_F \leftarrow max(ps(t_D), ps(t_F) + dps(t_F)), & \\ t_D \leftarrow max(ps(t_E), t_F + dps(t_D)), & \\ t_E \leftarrow max(ps(t_M), t_D + dps(t_E)), & \\ t_M \leftarrow max(ps(t_W), t_E + dps(t_M)), & \\ t_W \leftarrow t_M + dps(t_W), & \\ rd \leftarrow (dps(rd), t_M)), & dps(rt) \neq ps(rd) \wedge dps(rs) \neq ps(rd) \\ & \\ (t_F \leftarrow max(ps(t_D), ps(t_F) + dps(t_F)), & \\ t_D \leftarrow max(ps(t_E), t_F + dps(t_D), ps(t_{rd})), & \\ t_E \leftarrow max(ps(t_M), t_D + dps(t_E)), & \\ t_M \leftarrow max(ps(t_W), t_E + dps(t_M)), & \\ t_W \leftarrow t_M + dps(t_W), & \\ rd \leftarrow (dps(rd), t_M)), & dps(rt) = ps(rd) \vee dps(rs) = ps(rd) \end{cases} \tag{4.19}$$

Finally, the resulted pipeline state is normalized as shown in Equation 4.20.

$$\mathcal{U}(ps, dps) = \mathcal{U}_{ps}(ps, dps) - ps(t_F) \tag{4.20}$$

4.4.4 Join

Equation 4.21 defines the *pipeline join* function of is the following:

$$\mathcal{J}_{ps}(ps_1, ps_2) = \begin{cases} (t_F \leftarrow max(ps_1(t_F), ps_2(t_F)), \\ t_D \leftarrow max(ps_1(t_D), ps_2(t_D)), \\ t_E \leftarrow max(ps_1(t_E), ps_2(t_E)), \\ t_M \leftarrow max(ps_1(t_M), ps_2(t_M)), \\ t_W \leftarrow max(ps_1(t_W), ps_2(t_W)), \\ rd \leftarrow ps_1(rd) \vee ps_2(rd)) \end{cases} \tag{4.21}$$

4.4.5 Execution Time

Each instruction contributes to the total WCET. Equation 4.22 defines the number of contributed cycles. The analysis calculates the contribution of the pipeline to the *execution time* before the normalization in Equation 4.20.

$$\mathcal{T}_p(ps, dps) = \mathcal{U}_{ps}(ps, dps)(t_W) - ps(t_W) \tag{4.22}$$

4.5 Multiprocessor Analysis

This section describes the multiprocessor approach that estimates the WCET of a task running in a multiprocessor system. This analysis estimates the WCET of a tasks that runs in a system under the presence of temporally and spatially interfering tasks. The analysis concerns *symmetric multiprocessor* (SMP) systems and models the MESI protocol (see Section 3.5) for maintaining cache coherency. Also, the analysis assumes that the

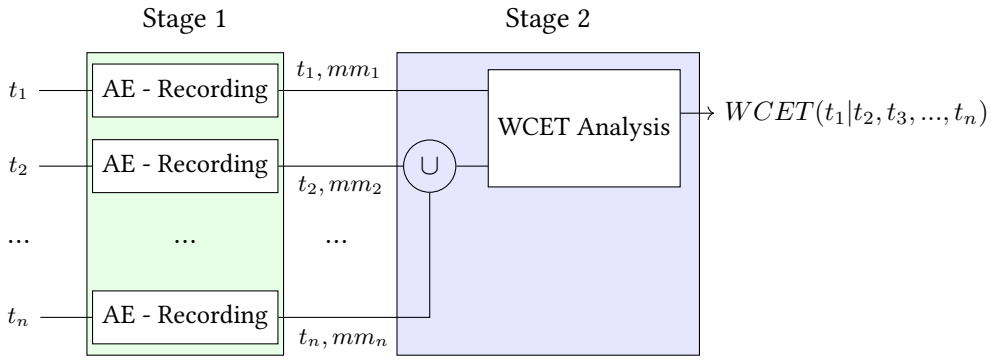


Figure 4.4: WCET Analysis for task t_1, when n tasks are temporally interfering on a symmetric shared-memory multiprocessing system. The analysis consists of two phases. The first phase uses abstract execution to record and extract the memory-access footprint of each task. The second phase estimates the WCET using the memory-access footprints of all the temporal interfering tasks.

cache hierarchy is inclusive, i.e. that each cache level is a subset of all higher cache levels. The cache block size is equal for all cache levels.

The multiprocessor analysis takes as inputs the task to analyze and a set of interfering tasks and proceeds in two stages. Both stages make use of the single-core cache-based analysis. The first stage analyzes each temporally interfering task, separately using the single-core analysis of KTA. In this stage, the analysis is not complete, as it only records the memory transactions of each task. The next stage uses these recordings to perform the WCET analysis on the target task.

The following parts describe the multiprocessor analysis methodology and the enhancements to the single-core analysis. The first subsection describes the methodology and assumptions, and the second redefines some of the single-core analysis operations for estimating the WCET.

4.5.1 Methodology

The multiprocessor analysis uses the single-core analysis for estimating the WCET of a task running on a SMP system. Figure 4.4 illustrates the main methodology of the multiprocessor analysis that consists of two stages.

In the first stage of the multiprocessor analysis, KTA analyzes all the spatially and temporally interfering tasks and performs a modified analysis on each of them. The modified analysis is a slightly modified version of the runtime analysis of KTA (see Section 3.1.1). This stage of the multiprocessor analysis performs a complete abstract execution analysis that records the memory accesses of each of the interfering tasks. The memory recordings consist of the block addresses that the task accesses throughout its execution. For each of the accessed block addresses, the analysis records the number of read and write operations, separately. This stage passes the recorded information to the second stage.

The second stage uses the memory recordings of all the interfering tasks to modify the analysis of the target task. First, the analysis merges all the accesses of the tasks excluding the target task. This merge adds the number of the reads and the writes of all tasks for each memory address. Together with the merge, the analysis calculates the overhead of each memory read or write to a shared block, i.e. a block accessed by other temporally interfering tasks. The calculation of the overheads is based on the MESI protocol. Table 4.1 shows the modeled behavior of the analysis under a read and a write. In particular, there are three different cases for each memory block. These cases are: no remote access, only remote read, and remote write. The latter, i.e. remote write, includes the case of both read and write accesses. In addition to the adjusting penalties for memory accesses, the analysis takes some additional actions for the memory blocks accessed by remote tasks. More specifically, the remote tasks affect the replacement priority of each block in the shared caches. For this reason, the analysis assumes that these blocks are locked to the lowest priority (most recently used). The actual accesses to these blocks follows the rules of Table 4.1, however, the analysis assumes that these blocks cover the most recently used blocks in the worst case. In other words, the analysis excludes these cache lines completely. During the cache analysis of the task, each remotely accessed block is treated according to Table 4.1, and the cache analysis ignores the locked blocks that represent the worst-case most recently used blocks by the remote tasks.

<table>
<tr><th>$t_i, i < N, i \neq j$ \ t_j</th><th>R</th><th>W</th></tr>
<tr><td>$\varnothing$</td><td>$h' = h$
$m' = m$</td><td>$h' = h$
$m' = m$</td></tr>
<tr><td rowspan="2">$n_r = \sum_{i=0}^{k} R_i, k < N$</td><td rowspan="2">$h'_{pr} = h_{pr}$
$h'_{sh} = max(N_c, h_{sh})$
$m' = max(N_c, N_m)$</td><td>$h'_{pr} = h_{pr} + N_i$
$h'_{sh} = max(N_c + N_i, h_{sh})$
$m' = max(N_c + N_i, N_m)$</td></tr>
<tr><td>$oh = n_r \cdot N_{wb}$</td></tr>
<tr><td rowspan="2">$n_w = \sum_{i=0}^{m} W_i, m < N$</td><td rowspan="2">$\left.\begin{matrix} h' \\ m' \end{matrix}\right\} max(N_c, N_m)$</td><td>$\left.\begin{matrix} h' \\ m' \end{matrix}\right\} max(N_c + N_i, N_m)$</td></tr>
<tr><td>$oh = n_w \cdot N_{wb}$</td></tr>
</table>

Table 4.1: Cache access miss and hit overhead due to the MESI coherence protocol. h and m denote the number of cycles due to a hit and miss, respectively, of the cache for a uni-processor system. h' and m' denote the number of cycles due to hits and misses, respectively, in a multiprocessor system using MESI. h_{pr} denotes the number of cycles for a hit in a private cache. h_{sh} denotes the number of cycles for a hit in a shared cache. oh is the total overhead caused by the bus transactions that force the processor to stall while writing modified data back to the memory, when another CPU requests it. n_r and n_w are the total number of *reads* and *writes* that the remote tasks perform on the specific block.

System assumptions

This analysis makes some assumptions regarding the MESI protocol implementation. First, the SMP architecture includes private caches, shared caches, and memory. Cache to cache transactions are possible when the requested memory address is available in one of the private cache of the other cores. In the cases that the available cache block is in either E or M states, the cache that contains it will send the data to the requesting cache. When the available cache block is in S state, this means that other caches might contain the same copy. So, in that case, every cache that contains the cache block might send the data to the requesting core. Which cache will actually send the data depends on the bus priority scheme [46]. In addition to that, a cache that contains the requested block in M state and receives a remote request for the specific memory block will *write back* the block to the shared memory. Finally, this analysis assumes that each task runs on a dedicated core.

4.5.2 Semantics

The multiprocessor analysis needs to consider the remote tasks. For this reason, there are two main modifications to the single-core cache-based analysis. The first modification regards the cache hierarchy. The remotely accessed blocks reserve the most recently used blocks of all the shared caches. The second modification affects the execution under a read and a write. The first subsection defines the required mathematical notations. The subsequent subsections define the modified cache structure and the execution time for the multiprocessor analysis.

Definitions

Parameter	Description
N_c	The number of cycles for receiving a block from another cache. The cache that includes the block in E (exclusive) or M (modified) state sends the data.
N_i	The number of cycles for invalidating all the caches that contain an S (shared) block in their caches.
N_m	The number of cycles for receiving the data from the shared memory/cache.
N_{wb}	The number of cycles for writing back a block when an exclusive read is issued on the bus. That may happen in case when the cache contains a modified block.

Table 4.2: Different parameters that define the different delays due to MESI [23].

The WCET estimation of a task that shares resources with other temporally interfering tasks consists of two phases. The first phase records the memory block accesses of each of the time-sharing tasks and provides these recordings to the second phase. The second phase uses these recordings to handle each memory request according to the protocol.

During the first phase, KTA records reads and writes, separately. Table 4.3 shows the possible combinations of memory requests, together with the subsequent possible state

transitions of the MESI protocol. Table 4.1 shows the overhead of a potential cache hit or miss due to the cache coherence protocol.

Let nt be the total number of spatially and temporally interfering tasks. Let $nm = |b|, b \in B$ be the memory size. Let $R = \{(r, w)|r, w \in \mathbb{N}^*\}$ be the set of the recorded accesses that consists of reads and writes. These recondings may be zero or greater. Let $Re = \{(r_0, ..., r_i, ..., r_{nm})|i \in [0, nm), r_i \in R\}$ be an ordered set of memory accesses that corresponds to the memory recording of each task. Let T be the set of the spatially and temporally interfering tasks (see Equation 4.23).

$$T = \{t_i|i \in [0, nt), t_i \in Re\} \tag{4.23}$$

Let $rec_r : T \times B \rightarrow \mathbb{N}$ and $rec_w : T \times B \rightarrow \mathbb{N}$ be two functions that take a task and a memory address as inputs and return the number of recorded reads and writes, respectively. Function $private : H \rightarrow H$ takes a cache hierarchy as input and returns a subset of the input that consists of solely the private caches. Let $hblocks : H \rightarrow \mathcal{P}(B)$ be a function that takes a cache hierarchy as an input and returns the set of all block addresses that the cache hierarchy contains.

4.5.3 Cache Hierarchy

The cache hierarchy takes into consideration the case when the activity of remote tasks can replace blocks that only the target task accesses. To achieve that, the cache hierarchy locks or reserves the most recently used lines of each set of all shared caches. The number of reserved cache lines is equal to the non-shared blocks that the remote tasks access.

Functions $update\_set$, $update\_cache$, and $update\_hcache$ update the lines in a set, a cache, and a cache hierarchy. Function $update\_hcache : H \times Re \rightarrow H$ updates the hierarchy for the multiprocessor analysis by applying function $update\_cache : C \times Re \rightarrow C$, which, in its turn, calls $update\_set : S \times \mathcal{P}(R) \rightarrow S$. Let $rec\_set : Re \times \mathbb{N} \rightarrow \mathcal{P}(R)$ be a function that takes as inputs a recording and a set number, and returns the set of recordings that contain the recordings that correspond to the set number. It is: $rec\_set(r, i) = \{r[b]|\forall b \in [0, nm), (b = set(i)) \wedge (rec_r(r, b) + rec_w(r, b) \neq 0)\}$. Equations 4.24, 4.25, and 4.26 define the update functions.

$$update\_set(s, r_s) = \begin{cases} (l_i \leftarrow \perp & |i < j, \\ \ l_i \leftarrow s[i] & |i \geq j), \quad j = |r_s| \end{cases} \tag{4.24}$$

$$update\_cache(c, r) = \Big\{ \ (s_i \leftarrow update\_set(c[i], r_s)| \quad r_s = rec\_set(r, i)) \tag{4.25}$$

$$update\_hcache(h, r) = \begin{cases} (c_i \leftarrow update\_cache(h[i], r) & |\forall i \in [1, nl], h[i] \notin private(h), \\ \ c_i \leftarrow h[i] & |\forall i \in [1, nl], h[i] \in private(h)) \end{cases} \tag{4.26}$$

4.5.4 Execution Time

Equation 4.27 defines the execution time $\mathcal{T}_{h_r} : H \times B \times T \times Re \rightarrow \mathbb{N}$ under a read operation. Similarly, equation 4.28 defines the execution time $\mathcal{T}_{h_w} : H \times B \times T \times Re \rightarrow \mathbb{N}$ under a write operation.

$$\mathcal{T}_{h_r}(h,b,t,t_j) = \begin{cases} \mathcal{T}_h(h,b), & \forall t_i \in t - \{t_j\}, rec_r(t_i,b) = 0, rec_w(t_i,b) = 0 \\ \mathcal{T}_h(h_{pr},b), & (\forall t_i \in t - \{t_j\}, rec_w(t_i,b) = 0) \wedge \\ & (\exists t_i \in t - \{t_j\}, rec_r(t_i,b) \neq 0) \wedge \\ & (h_{pr} = private(h)) \wedge (b \in hblocks(h_{pr})) \\ max(\mathcal{T}_h(h,b), N_C), & otherwise \end{cases} \tag{4.27}$$

$$\mathcal{T}_{h_w}(h,b,t,t_j) = \begin{cases} \mathcal{T}_h(h,b), & \forall t_i \in t - \{t_j\}, rec_r(t_i,b) = 0, rec_w(t_i,b) = 0 \\ \mathcal{T}_h(h_{pr},b)+N_i, & (\forall t_i \in t - \{t_j\}, rec_w(t_i,b) = 0) \wedge \\ & (\exists t_i \in t - \{t_j\}, rec_r(t_i,b) \neq 0) \wedge \\ & (h_{pr} = private(h)) \wedge (b \in hblocks(h_{pr})) \\ max(\mathcal{T}_h(h,b), N_C+N_i), & otherwise \end{cases} \tag{4.28}$$

Write-back overhead The multi-core WCET analysis treats every memory access differently. For every memory access, the analysis checks whether other temporally interfering tasks access the same address during their execution. In particular, if the target task writes to a shared address, then a remote task might request it. Hence, at some point in time, the target task has to write back the modified data to the memory. Two quantities limit the number of *write-backs*. These quantities are (1) the number of the *remote accesses* to that block and (2) the number of *writes* of the target task. Equation 4.29 defines this additional overhead.

$$\mathcal{T}_{oh}(rec,t,j) = \sum_{b \in B} \left(N_{wb} \cdot min \left(\sum_{i \in [0,j) \cap (j,nt)} \left(rec_w(t_j,b) + rec_r(t_j,b) \right), rec_w(t[j],b) \right) \right) \tag{4.29}$$

4.6 Limitations

The following points describe briefly the limitations of this approach.

- The hardware analysis is targeting specific hardware implementing the MIPS32® architecture and extending it to support other architectures and platforms might be cumbersome.
- The current analysis does not consider timing anomalies. These phenomena occur when a local worst case does not necessarily imply a global worst case, e.g. out-of-order execution. In the presence of timing anomalies, the analysis cannot involve greedy methods.
- The analysis focuses on integer arithmetic and does therefore not support floating point operations.

- There are limitations in the supported instructions. As for now, the analysis does not support indirect jumps.
- There is a limitation in the coding paradigms that the current implementation of KTA is able to analyze. This abstract domain uses a non-relational representation of the program variables and memory content, so that the analysis does not encode complex information about the relations between variables. KTA encodes only specific relations using patters.
- The multiprocessor analysis assumes that the bus is ideal and that there are no read and write queues. That means that the overhead of a memory operation is constant regardless of the number of tasks and the current bus traffic. To deal with that, the analysis uses conservative approximations of the overheads.

t_i \ t	R	R/W	W
$\varnothing$	-	-	-
$k \cdot R, k \leq i$	$I \to E: N_m$ $I \to S: N_c \mid N_m$ $E \to S: -$	$I \to E: N_m$ $I \to M: N_m$ $I \to S: N_c \mid N_m$ $M \to S: N_{wb}$ $E \to S: -$ $E \to M: -$ $S \to M: N_i$	$I \to M: N_m$ $S \to M: N_i$ $M \to S: N_{wb}$
$k \cdot R, k \leq i$ $m \cdot W, m \leq i$	$I \to S: N_c$ $I \to E: N_m$ $S \to I: -$ $E \to I: -$ $E \to S: -$	$I \to S: N_c$ $I \to E: N_m$ $I \to M: N_c + N_i \mid N_m$ $M \to I: N_{wb}$ $M \to S: N_{wb}$ $E \to I: -$ $E \to S: N_{wb}$ $E \to M: -$ $S \to I: -$ $S \to M: N_i$	$I \to M: N_c + N_i \mid N_m$ $M \to I: N_{wb}$ $M \to S: N_{wb}$ $S \to I: -$ $S \to M: N_i$
$m \cdot W, m \leq i$	$I \to S: N_c$ $I \to E: N_m$ $S \to I: -$ $E \to I: -$	$I \to S: N_c$ $I \to E: N_m$ $I \to M: N_c + N_i \mid N_m$ $M \to I: N_{wb}$ $E \to I: -$ $E \to M: -$ $S \to I: -$ $S \to M: N_i$	$I \to M: N_c + N_i \mid N_m$ $M \to I: N_{wb}$

Table 4.3: Cache access miss and hit overhead due to the MESI snooping coherence protocol. h and m denote the hit and miss time of the cache for a uni-processor system. oh denotes the overhead of the processor for *writing back* the data that the same processor modifies (*writes*) due to memory *read* or *write* requests by other processors. The number of the remote *reads* and *writes* and the number of the local *writes* limit that overhead oh.

- The analysis supports only *symmetric multiprocessor* (SMP) systems that use MESI to maintain cache coherence.
- The approach uses the assumption that the cache hierarchy is inclusive, i.e. every cache is a subset of all higher level caches. Also, all cache levels have equal block size.

Chapter 5

Implementation

This chapter describes the implementation of the approach of this thesis. The implementation includes the cache, the cache hierarchy, and the pipeline states that constitute the low-level analysis, the IC abstract domain, and finally, the multiprocessor analysis. In addition to these, the last section presents some additional implementations that complete a few missing parts of KTA. All parts of the implementation are part of the runtime analysis of KTA. This chapter presents only the basic parts of the KTA implementation, therefore, the description of the KTA implementation in Section 3.1 might be necessary for comprehending the following sections. The following sections describes the implementation of this thesis that incorporates the IC abstract domain, the low-level analysis, and the multiprocessor analysis into KTA.

5.1 Implementation

This section describes the implementation of this approach that integrates different parts to KTA. First, the integrated parts to the single-core analysis and second, the multi-core analysis.

5.1.1 Single-Core Analysis

Some parts of the implementation integrate some additional functionality to the single-core analysis. The implementation of the IC abstract domain currently replaces the interval domain by implementing the abstract domain interface of KTA. Figure 5.1 depicts a high level diagram of the single-core analysis of KTA. The dotted area denotes the part of the contribution of this thesis. This includes minor contributions to the AE implementation, such as modifications to the program state, adding missing instruction, and changes to the branch pattern.

The low-level analysis incorporates the cache state, the cache hierarchy state, and the pipeline state to the program state. Listing 5.1 defines the pstate. It consist of the register file *reg*, the memory hierarchy *hmem* that includes the cache hierarchy and the mem-

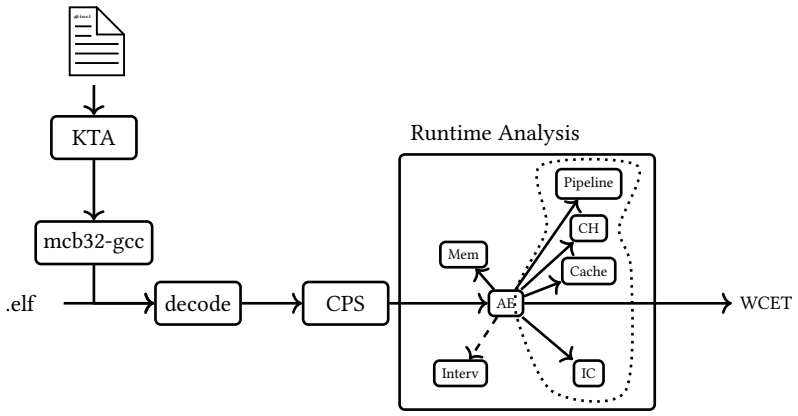


Figure 5.1: A high level diagram of the KTA tool methodology for WCET estimation. The diagram does not include the optimization step. The dotted curve denotes the contributions of this thesis. The changes to the AE are minor and include modifying the program state, the implementation of missing instructions, and some changes to the branch patterns.

ory, the pipeline *pipeline*, the *bcet* and the *wcet* of the *spliter_set* that is necessary for the optimization step.

```
  type progstate = {
  reg    : aregister;
  hmem   : amemhierarchy;   (* Memory hierarchy state *)
  pipeline : apipeline;     (* Pipeline state *)
  bcet   : int;
  wcet   : int;
  input_set : spliter list;
}
```

Listing 5.1: The program state of KTA.

5.1.2 IC Abstract Domain

The code that implements the IC abstract domain resides in "runtime/aint32congruence.ml". The KTA value domain template defines an interface for manipulating the abstract values. The interface defines a type for the abstract domain, the abstract function, and the join function. Branch operations are special because they result in new splitted abstract values.
The following code snippet illustrates the function that corresponds to the addition of two IC values.

```
type interval = (low * step * num)
....
let aint32_add v1 v2 =
  aint32_binop (fun (l1,s1,n1) (l2,s2,n2) ->
```

```
      let h1 = high l1 s1 n1 in
      let h2 = high l2 s2 n2 in
      let l = l1 + l2 in
      let s = gcd s1 s2 in
      let h = h1 + h2 in
      let n = number h l s in
      let s = if n = 1 then 0 else s in
    if l<lowval || h>highval then raise AnyException
    else (l,s,n)
  ) v1 v2
```

5.1.3 Cache State

The cache state is part of the memory hierarchy state. The initial configuration for the cache is part of the CPU model that resides in "runtime/cpumodel.ml". Listing 5.2 shows an example that models the memory model of Creator ci40 [11].

```
let cache_model =
  [S (
       (*Icache*)
       { assoc = 4; size = 32768;
         word_size = 4; block_size = 32;
         write_allocate = true; write_back = true;
         hit_time = 1;
         shared = false;
       },
       (*Dcache*)
       { assoc = 4; size = 32768; (*1 lsl 15 *)
         word_size = 4; block_size = 32;
         write_allocate = true; write_back = true;
         hit_time = 2;
         shared = false;});
   (*L2*)
   U ({ assoc = 8; size = 1 lsl 19;
        word_size = 4; block_size = 32;
        write_allocate = true; write_back = true;
        hit_time = 10-2;
        shared = true;
      });
  ]
```

Listing 5.2: Memory model of the Creator ci40 [11] board.

Listing 5.4 shows the definition of the cache state. *Set* and *Cache* are OCaml *Map* modules.

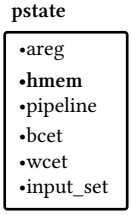
```
type aset_t = (lru * dirty * invalid) Set.t

type acache = {
    cache : aset_t Cache.t;
```

```
    cacheinfo: acache_info_t;

    setinfo : info_t;
    wordinfo : info_t;
    byteinfo : info_t;

    disabled : bool;

    cache_stats : cache_stats_t;
  }
```

Listing 5.3: Cache state definition.

5.1.4 Cache Hierarchy State

The *cache hierarchy* state is part of the *memory hierarchy* state of the program state. The memory hierarchy consists of the cache hierarchy and the memory.



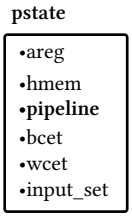
Listing 5.4 shows the data structures of the cache hierarchy. The cache hierarchy state models the cache hierarchy, as a list of caches (Line 5). Each cache level, `cache_t`, consists of either a unified cache or a separate cache consisting of a instruction and a data cache (Lines 1-3). Finally, Lines 7-11 define the memory hierarchy, `amemhierarchy`, contains the memory, `amemory` and the cache hierarchy `cache_hierarchy_t`.

```
1   type cache_t =
2   | Uni of acache
3   | Sep of acache * acache
4
5   type cache_hierarchy_t = cache_t list
6
7   type amemhierarchy = {
8     mem : amemory;
9     cache : cache_hierarchy_t;
10  }
```

Listing 5.4: Cache hierarchy state definition.

5.1.5 Pipeline State

The *pipeline state* is part of the program state.



The pipeline state consists of the stage release times for each stage and the dependencies. Listing 5.5 shows the data structures of the pipeline analysis. Line 11 defines the abstract pipeline state that is part of the program state. Line 1 defines the pipeline state as a tuple of 5 stages. The pipeline analysis uses dependency patterns to model the hazards. The hazards correspond to branch instructions and memory read (Lines 4 and 5). Line 9 defines the dependency type that contains the instruction that precedes the current state and the stage after which the specific stage is ready.

```
1   type apipeline_t = f_stage * d_stage * e_stage * m_stage * w_stage
2
3   type instruction_type =
4   | Mem of registers option * registers option * registers option
5   | Br of registers option * registers option
6   | ND of registers option * registers option
7   * registers option * registers option
8
9   type dependency_t = instruction_type * stage_ready
10
11  type apipeline = apipeline_t * dependency_t option
```

Listing 5.5: The abstract pipeline type definitions. The abstract pipeline state `apipeline` is part of the `pstate`.

5.1.6 Multiprocessor Analysis

Figure 5.2 depicts the multiprocessor analysis method. For the multiprocessor analysis, KTA receives as inputs the target task, i.e. the task to analyze, and a number of temporally interfering tasks. Each task uses one dedicated core. The first stage (Stage I) records the memory accesses for all tasks, and the second stage (Stage II) uses the recordings of the first stage to derive the WCET of the target task.

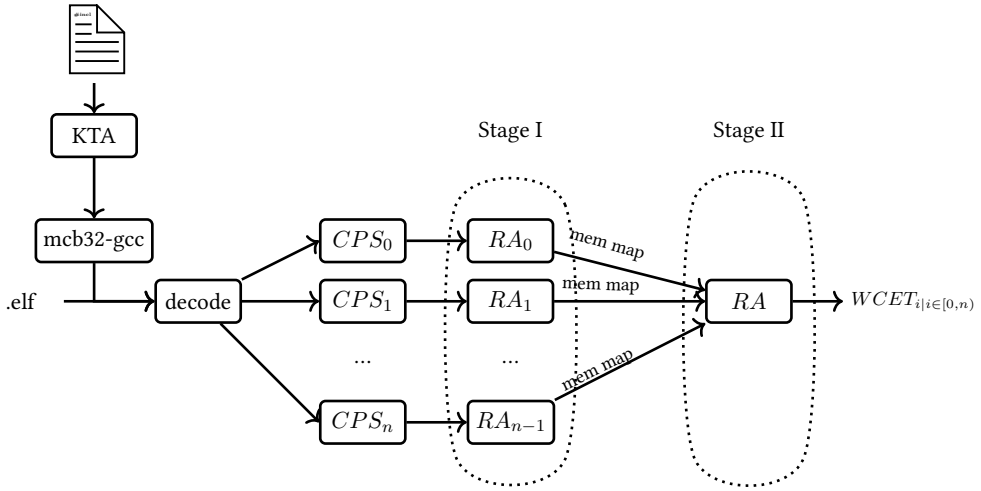


Figure 5.2: The multiprocessor analysis method consists of two phases; the first records the memory behavior of each of the tasks and the second analyzes the target task, using the information of the previous stage.

Chapter 6

Evaluation

The evaluation of the approach of this thesis consists of three parts. The first part of the evaluation concerns the expressiveness of the KTA tool, which gives an indication of the expressiveness of the abstract domain. The second part focuses on the evaluation of the single-core cache-based analysis, and the third part concerns the evaluation of the multi-core analysis methodology.
This evaluation aims at examining the research question that consists of two parts: (1) evaluating the feasibility of the KTA tool using a low-level hardware timing model and (2) designing and implementing of the multiprocessor analysis approach by extending the single-core approach. The two first parts of this evaluation deal with the first research question by comparing KTA with SWEET, a well-known timing analysis tool, with regards to coverage and analysis performance. Further comparisons attempt to evaluate the implemented low-analysis. The last part of the evaluation deals with the second research question and attempts to verify the approach and evaluate the result of the multiprocessor analysis.
The first section of this chapter provides a description of the Experimental Setup parameters that apply to all the Evaluation parts. The following sections describe the three evaluation parts: (1) Expressiveness Evaluation, (2) Single-Core Cache Analysis Evaluation, and (3) Multi-Core Cache-based Analysis Evaluation.

6.1 General Experimental Setup

The compilation of all benchmarks and tools, as well as KTA, is performed on an Intel® Core™ i5-6500T CPU, at 2.50GHz, with 8GB RAM, running Ubuntu GNU/Linux, Release 16.04, with Linux Kernel, 4.4.0-91-generic. KTA is compiled with the OCaml compiler[1], version 4.02.3. For the first part of the evaluation, KTA invokes mcb32-gcc[2], a MIPS32® cross-compiler. However, in the two last parts of the evaluation, KTA uses the mips-

[1]OCaml compiler framework: https://ocaml.org/
[2]MCB32 Toolchain: https://github.com/is1200-example-projects/mcb32tools

mti-elf toolchain[3] because mbc32-gcc does currently not support the third revision of the MIPS32® architecture that the testing hardware, i.e. Creator ci40, implements.

6.1.1 Benchmarks

The evaluation of the first two parts, namely the *Expressiveness* and *Single-core cache evaluation*, uses the Mälardalen benchmark suite[4]. The Mälardalen benchmark suite is a collection of benchmarks targeting embedded real-time applications and is widely used for evaluating WCET analysis tools and applications. Table B.1 lists all the benchmarks of the Mälardalen benchmark suite, together with additional information, such as whether the benchmark uses floating-point numbers. Also, Table B.1 describes some known problems that appear during the compilation or analysis of SWEET and KTA. *SWEdish Execution Time analysis tool* (SWEET) is an open source timing analysis tool that derives flow information (flow facts) from a program [43]. SWEET uses an intermediate representation, namely the *ARTIST2 Language for WCET Flow Analysis* (ALF) language [21]. The Expressiveness Evaluation section discusses the content of this table in more detail (see Section 6.2).
A more recent selection of benchmarks for embedded real-time applications is the TACLe benchmark suite[5]. The TACLe suite contains a wider range of selected benchmarks for real-time systems. However, due to time limitations, this part has not been investigated yet. In the future, there is a plan to replacing the Mälardalen benchmarks to the TACLe benchmarks for a more throughout evaluation (see Future Work, Section 7.2).

6.1.2 Execution on hardware

Two parts of the evaluation, namely the Single-core Cache-based Analysis Evaluation in Section 6.3 and the Multi-core Cache-based Analysis Evaluation in Section 6.4, compare the result of the WCET analysis with actual hardware execution cycles. These parts use the Creator Ci40 IoT Hub board and compare the CPU cycles with the KTA analysis result [11]. This board has a cXT200 *system-on-chip* (SoC) [11], implementing a dual core multithreading MIPS32® 550Mhz interAptiv™ processor interAptiv core [24] with level 1 and level 2 caches.
The cache-based analysis evaluation activates only one core and one thread, so that the system behaves as a single-core machine with two levels of caches. The aim of this part is to evaluate the tightness of the cache-based analysis.
The multi-core analysis evaluation activates two threads, one on each core, so that the system behaves as a dual core SMP with a level 1 private cache and a level 2 shared cache.

[3] MIPS32 Baremetal Toolchain: https://community.imgtec.com/developers/mips/tools/codescape-mips-sdk/download-codescape-mips-sdk-essentials/
[4] WCET benchmarks: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[5] TACLe: http://www.tacle.eu/index.php/activities/taclebench

6.1.3 Analysis Termination Methods

The WCET analysis of KTA cannot guarantee finite analysis time, nor finite memory utilization. Therefore, there is a requirement to limit either of them, or both. The evaluation uses three methods for restricting the time and memory resources of the analysis.

KTA parameter *-max-cycles*:

KTA provides a command line option *-max-cycles* that sets the maximum number of cycles that a path may reach. The default value is 100,000,000. In the following example, KTA runs the analysis of the *binary_search* function of the *bs.c* benchmark with maximum number of cycles set to 100,000.

```
$ kta wcet bs.c binary_search -max_cycles 100000
```

However, the restriction of the longest path does not automatically limit the memory utilization, which can grow fast. For example, there are cases, where the analysis splits continuously to new paths with no path reaching the *maximum cycles* limit. Then, the analysis might exhaust the RAM, and, on a linux system, KTA starts utilizing the swap memory, which is much slower. That way, the time-based measurements are not uniform, and the analysis can take long time to terminate (when it reaches the *maximum cycles* limit). Setting a uniform memory limit that is less than the size of the RAM for all benchmarks, gives a uniform solution for the evaluation.

ulimit command:

In a Unix system, the *setrlimit* system call can limit the resources that a process is allowed use. The *ulimit* shell command is a wrapper around the *setrlimit* and *getrlimit* system calls that can set the limit of the virtual address space that a process and its children may utilize. In the code below, *ulimit* sets the memory soft limit to approximately 500MB. This command precedes the execution of the benchmarks and applies to each and every benchmark.

```
$ ulimit -Sv 500000
```

timeout command:

The *maximum cycles* parameter is not available for the other tools, e.g. SWEET. So, in order to limit the analysis time in a uniform way for all tools, all benchmarks run under the *timeout* GNU/Linux command. This command sends a *TERM* signal (the default option) to the process, when the execution time of the process exceeds the specified time limit. In the following command, KTA analyzes the *main* function of *test.elf* with a maximum analysis time of 5 minutes. In case the analysis has not completed after this time limit, KTA receives a *TERM* signal and the analysis terminates.

```
$ timeout 5m kta test.elf main
```

Both KTA and SWEET terminate upon receiving a *TERM* signal.

6.1.4 Measuring Time

Analysis time - *time* command

The expressiveness evaluation uses the *analysis time* as an evaluation measure, whereas the single-core cache analysis and the multi-core cache-based analysis compare the KTA with the actual number of cycles run on the hardware.
The expressiveness evaluation uses the GNU/Linux *time* command. In that way, the measurement of the analysis time of different tools can be performed in a uniform way. The time command returns the system utilization time of a command, i.e. the time that a command occupies the system. In order to reduce the interference of the host system, most parts of the evaluation repeat the analysis for a number of iterations and calculate the average time.

```
$ time timeout 5m kta test.elf main
```

Analysis time - OCaml Sys.time() function

Measuring the impact of the cache analysis on the KTA tool, does not require considering the compilation time or other parts of the analysis that are similar in the two cases, but rather the actual analysis time. Hence, the *time* command is not appropriate, because it measures the total analysis time, including compiling the source, and is affected by the host system. For this reason, this part of the evaluation uses the *gettimeofday()* function of the *Unix* OCaml library. The *Unix.gettimeofday()* function returns the current time in seconds and has resolution better that 1 second. The current time returns the time since the 1st of January, 1970. Compared to the *Sys.time()* OCaml function, *Unix.gettimeofday()* has better resolution.
The following OCaml code snippet shows the *Unix.gettimeofday()* invocation. This method returns only the abstract-execution-based analysis time.

```
let _st_time = Unix.gettimeofday() in
analyze funcname_ bblocks gp_addr mem [] [] 0;
printf "Time Elapsed: %fs\n" (Sys.gettimeofday()-_st_time)
```

Hardware execution cycles - *rdhwr* instruction

The parts of the evaluation that compare the result of the WCET analysis with actual hardware execution cycles make use of the *rdhwr* instruction. Instruction, "*rdhwr rd, $2*", provides read access to the coprocessor 0 (C0) high resolution counter [24]. The resolution of the C0 counter is two processor cycles.
The following C preprocessor macro snippet uses *rdhwr* for measuring the number of elapsed cycles when calling function `func` that takes one input, `input`, and returns a result in a variable `res`. When measuring the number of execution cycles of a function, the measurement code should not measure any instruction or data cache misses or other

delays that are not related to the execution time of the function in question. The `.align x` directive aligns the instructions to a 2^x-byte boundary. The cache block is 32 bytes. So, aligning the measurement code ensures that none of the instructions between the first and the second *rdhwr* instructions will result in an instruction miss. Also, the code uses the saved register, *$16* to save the result of the processor cycles before calling `func` to avoid a memory data access within the measurement code. Because this function call is not taken care by gcc, all registers that might change during the call are saved to the stack by `PUSH_ALL` and retrieved back by `POP_ALL`.

```
#define MEASURE_1_RET(func,input,res) {      \
    asm volatile (  PUSH_ALL "\n\t           \
                    .align 5\n\t             \
                    nop\n\t                  \
                    rdhwr $16, $2\n\t        \
                    li $4," # input "\n\t    \
                    nop\n\t                  \
                    jal " # func "\n\t       \
                    move %[res], $2\n\t      \
                    rdhwr %[end], $2\n\t     \
                    sw $16,%[start]\n\t"     \
                    POP_ALL "\n\t"           \
                    :[res]"=r"(res),         \
                     [end]"=r"(end)          \
                    :[start]"m"(start));     \
}
```

6.2 Expressiveness Evaluation

Evaluating the expressiveness of the abstract domain requires a comparison measure that can evaluate the performance of the KTA tool with regards to expressiveness, i.e. the ability of the tool to perform the analysis and provide a WCET estimation for different programs. This part of the evaluation concerns the abstract domain design, including all the enhancements and optimizations. The first part of the expressiveness evaluation compares the hybrid interval and congruence domain with the previous implementation of the interval domain. The second part compares the current KTA implementation with another WCET tool, namely SWEET [34, 43]. SWEET is a well-known WCET analysis tools [51], with active contribution to the WCET field. In addition to that, SWEET uses abstract execution for calculating the WCET, which makes the two tools easier comparable. There was an initial plan to include more tools to the comparison (such as OTAWA [2]), but due to both technical reasons and unexpected delays, this was not possible.
This evaluation uses the Mälardalen WCET benchmark suite. Table B.1 presents the benchmarks that are part of the Mälardalen benchmark suite. Among these benchmarks, some use floating-point numbers, a feature that KTA currently does not support (see Table B.1). Therefore, the evaluation does not consider these benchmarks at all. In addition to that, among the benchmarks, there are known issues for both tools. KTA does not implement register jump (excluding return to the caller function), so all benchmarks producing

code containing the MIPS32® *jr* instruction, fail at an early stage. SWEET fails in some examples due to compiler errors (see Table B.1).
The ability of the three tools to analyze the benchmarks indicates the expressiveness of the respective tool. In addition to the expressiveness evaluation, i.e. whether a tool is able to analyze a given benchmark, another evaluation measure is the timing performance, i.e. how much time the analysis takes to finish. This performance evaluation measures the time required for the tool to provide the WCET estimation. Neither of the expressiveness evaluation parts considers the actual WCET result. The reason for that is that the purpose of this comparison is to measure the expressiveness, i.e. whether each tool is able to return a result, and to compare the two tools using the best response time for each tool. However, the selection of the configuration parameters, for example the "merge" parameter in SWEET and the "batch-size" parameter in KTA, has an impact on the tightness of the WCET result of the respective tool.

6.2.1 IC Domain - Interval Domain

This part of the evaluation compares the new implemented IC abstract domain, with the previously implemented abstract domain, i.e. the Interval domain. The comparison of the Interval and the IC domains uses the Mälardalen benchmark suite.
The next section describes the experimental setup and the following section presents the results of the evaluation.

Experimental Setup

This part of the evaluation concerns solely KTA. The configuration of KTA is the default, with the Cache and the Pipeline enabled. The configurable parameters include the maximum batch size, i.e. the maximum number of program states with the same priority. KTA can configure the batch size parameter using the "-bs_config" command line optional parameter. The default value is 4, and the evaluation of the IC Domain uses the default value. The selection of the batch size for KTA is based on the expressiveness for both domains. The selection procedure examines only three different values, i.e. 4, 30, and 100. Among these, batch size 4 does not succeed analyzing the *binary_search* function of *bs.c*. However, 30 and 100 timed out when analyzing *fir.c*. Because the analysis time increases when increasing the batch size, the final selected batch size for this evaluation is the default value, i.e. 4. The input arguments are small intervals and are the same in the case of both domains.
Restricting the time and memory resources is essential because KTA does not guarantee termination. The limit of the longest execution path is set to the default value: 100,000,000. The *ulimit* command limits the memory utilization to approximately 1GB. In addition to that, the *time* command limits the absolute time of for each benchmark to 10 minutes.
The measurement of the elapsed time uses the *time* command, which returns the total time during which the analysis occupies the system. The time limit is set to 10 minutes.

```
ulimit -Sv 1000000
time timeout 10m kta wcet $fname $func \
```

```
                    -bsconfig 30 -optimize $optimization
```

Results and Discussion

Figures 6.1, 6.2, 6.3, and 6.4 present the results of the comparison between the interval and the IC domains. Each figure corresponds to a different optimization flag. Different optimization flags result into different code. The following paragraphs discuss the results.

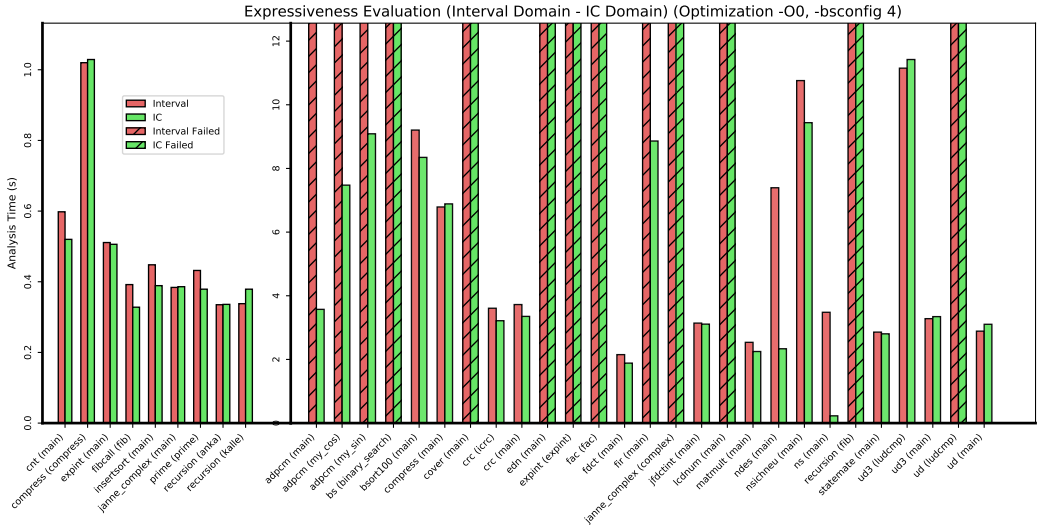


Figure 6.1: IC and Interval Domain for optimization level: -O0. The red bars represent the Interval and the green bars the IC domain. The height of the bars represents the analysis time in seconds. The bars with the "/" pattern, represent the benchmarks the failed with an out-of-memory error, timeout, or compilation error. The plot is divided in two time ranges, i.e. [0,2] and (2, 300].

Optimization Level -O0: In the case of *-O0* (see Figure 6.1), the IC domain advance slightly with regards to expressiveness. The analysis of two benchmarks terminates only for the IC domain whereas the analysis with the interval domain does not terminate. These benchmarks are all functions that belong to the "adpcm.c" benchmark and the main function of "fir.c".
However, there is a number of benchmarks that fail for both domains. In particular, 9 benchmarks fail for both the Interval and the IC domain. In the case of the *binary_search* function of *bs.c*, the analysis does not terminate when the batch size is 4, but terminates with larger batch sizes, e.g. *bs_config* = 30. This happens because *binary_search* generates new paths and merging them produces a more conservative interval. This merged interval affects the control flow, and the analysis cannot terminate because some subset of the new conservative interval never terminates. In addition to that, *cover.c* and *lcdnum.c*, generate code that contains a jump register instruction ("jr"), which KTA does not support right now. Also, the "expint" function of the "expint.c" benchmark produces a division-by-zero

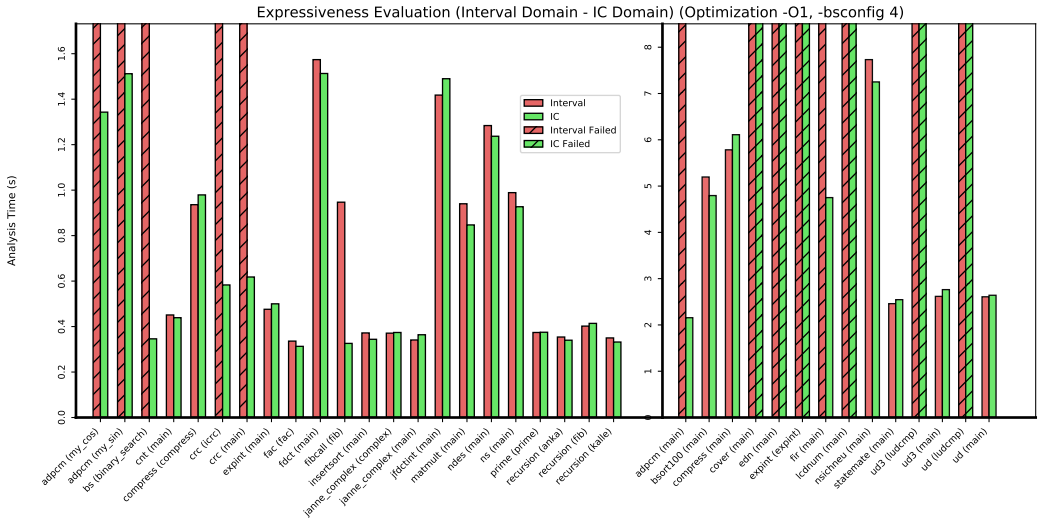


Figure 6.2: IC and Interval Domain for optimization level: -O1. The red bars represent the Interval and the green bars the IC domain. The height of the bars represents the analysis time in seconds. The bars with the "/" pattern, represent the benchmarks the failed with an out-of-memory error, timeout, or compilation error. The plot is divided in two time ranges, i.e. [0,2] and (2, 300].

error that is caused by either the conservative merging of the abstract domains, or by some other error or bug.
With regards to analysis time, there are many cases, where the IC Domain exhibits lower analysis time than the Interval Domain. The reason for that is that the interval domain results in more conservative intervals after merging. For that reason, the analysis takes longer time to terminate.

Optimization Level -O1: When applying the level 1 optimizations, the benchmarks demonstrate slightly different behavior (see Figure 6.2). Six benchmarks fail for both domains. Among them, "cover.c" and "lcdnum.c" fail, because they use the unsupported jump register instruction ("jr").
The IC domain still performs better. In addition to "fir.c" and "adpcm.c", it also analyzes the "main" and "icrc" function of the "crc.c" benchmark, whereas the interval domain fails. With regards to the analysis time, the analysis time of the IC domain is still better in many cases. That depends on the batch size, but it also indicates that the actual overhead of the IC domain, does not outweigh the advantages that the IC domain has on improving the abstract domain. Increasing the batch size, however, reduces that difference, and in most cases, the Interval domain performs better with regards to timing.
It is also worth noting that the "fir.c" benchmark terminates for the Interval domain, when the timeout value is larger. However, the analysis time is larger than the IC domain.

Optimization Level -O2: Figure 6.3 illustrates the results for the benchmarks with the "-O2" optimization enabled. A major difference between the level 2 and level 1 optimiza-

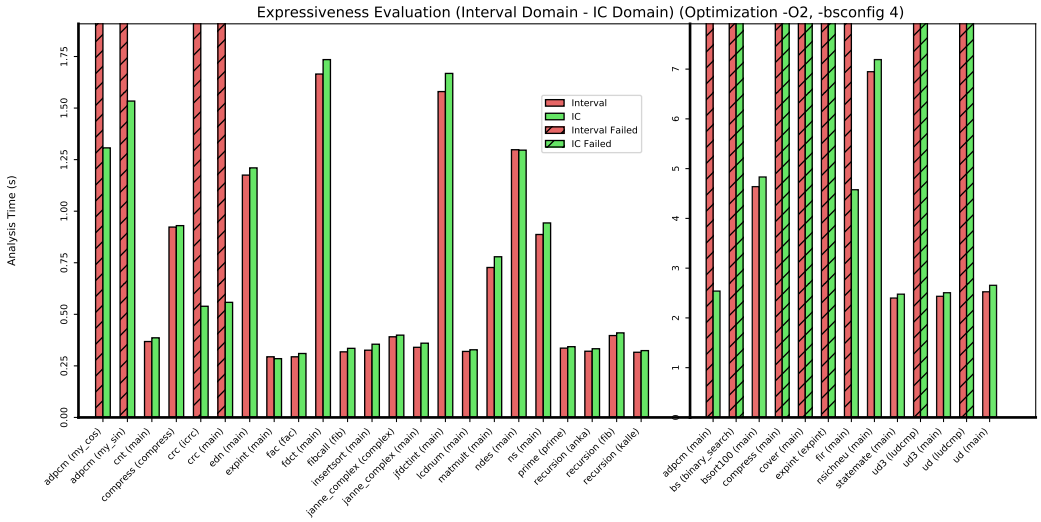


Figure 6.3: IC and Interval Domain for optimization level: -O2. The red bars represent the Interval and the green bars the IC domain. The height of the bars represents the analysis time in seconds. The bars with the "/" pattern, represent the benchmarks the failed with an out-of-memory error, timeout, or compilation error. The plot is divided in two time ranges, i.e. [0,2] and (2, 300].

tions is that the "lcdnum.c" benchmark does not produce a "jr" instruction, and KTA can, therefore, analyze it. However, the total number of benchmarks that terminate is the same as the previous case because the "compress.c" benchmark fails with an out-of-memory error, i.e. exceeds the ulimit virtual memory limit.
The rest of the results and conclusions are similar to the *-O1* case.

Optimization Level -O3: Figure 6.4 shows the results for the benchmarks with the "-O3" optimizations enabled. The "bs.c" and "cover.c" do not terminate for any of the domains. The IC domain remains better with regards to expressiveness. However, the Interval domain performs better in most of the benchmarks with regards to the analysis time.

6.2.2 Tool Expressiveness Comparison

This part of the evaluation compares KTA with another tool, namely SWEET. The comparison is based on the analysis time and evaluates the two tools with regards to expressiveness and their response time. For the comparison to be valid, both tools require setting up some parameters.
The next two subsections, describe the configuration of each tool and the way they were executed.

SWEET

SWEET is a research prototype tool for flow analysis that can produce a safe and tight WCET. The main functionality of SWEET is flow analysis, which calculates information

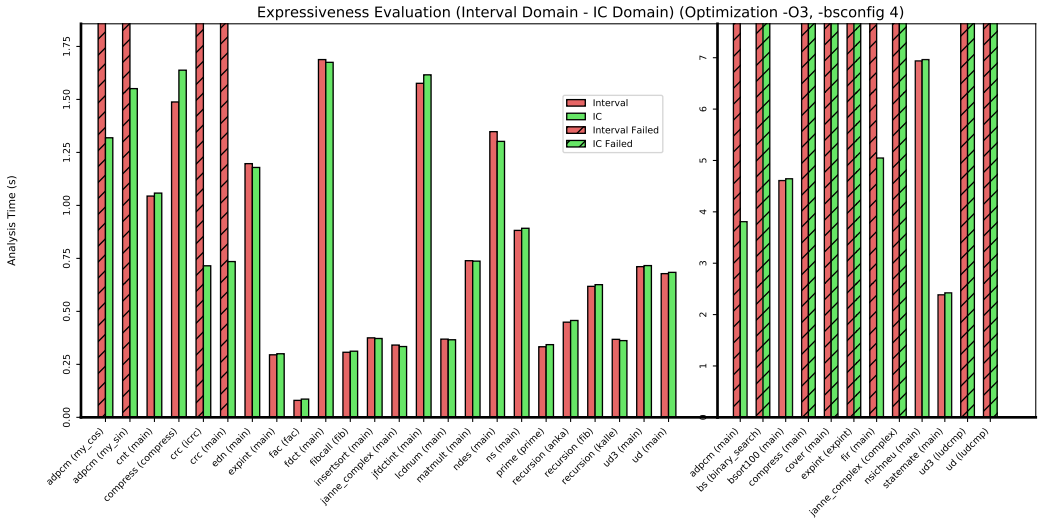


Figure 6.4: IC and Interval Domain for optimization level: -O3. The red bars represent the Interval and the green bars the IC domain. The height of the bars represents the analysis time in seconds. The bars with the "/" pattern, represent the benchmarks the failed with an out-of-memory error, timeout, or compilation error. The plot is divided in two time ranges, i.e. [0,2] and (2, 300].

about a program, e.g loop bounds and infeasible paths. WCET analysis uses this information to produce a tight and safe WCET [43].

SWEET implements different abstract domains for the WCET analysis, including both relational and non-relational abstract domains. More specifically, SWEET can perform the analysis using intervals, circular linear progressions, and different polyhedral domains [43]. The two former domains are non-relational, whereas the latter are relational. The evaluation considers only the two relational domains, i.e. interval and CLP for technical reasons.

SWEET uses abstract execution as the main flow analysis technique. In AE, the abstract states can be merged at different points. In SWEET the selection of merging points is configurable. Table 6.1 shows the different merging policies that SWEET allows. SWEET allows the selection of any combination of these merging points, which leads to $2^5 = 32$ different options.

Ideally, the evaluation should consider all possible combinations of the merging points. However, running all possible combinations leads to very high execution time for the evaluation, especially when the evaluation examines other parameters in conjunction with the *merge_point* parameter. Therefore, a two-step procedure, which precedes the final evaluation, selects the merge combination parameters that perform best on the Mälardalen benchmark suite. The selection procedure consists of the following two steps: (1) Benchmark selection, (2) Parameter selection.

Both steps use memory and time limits to restrict the resources that the analysis can utilize. Command *ulimit* limits the maximum memory utilization to 500MB, and *timeout* limits the maximum analysis time to 5 minutes. The same resource restrictions apply to

Option	Description
all	All merge points are enabled.
none	No merging enabled.
fe	Every function entry point.
fr	Every function return point.
le	Every loop exit edge.
be	Every back edge.
je	Every joining edge.

Table 6.1: SWEET: Abstract execution merging points.

the final experiment that compares SWEET with KTA. The following paragraphs describe the two selection steps.

Benchmark Selection The first step runs all the benchmarks for a limited number of merging points. These merging points are the ones shown in Table 6.1, i.e. each merging point independently, the combination of all the merge points, i.e. "all", and no merging point, i.e. "none".
This step selects a subset of the benchmarks that is the input of the next step. The selected benchmarks exhibit the largest analysis time deviation among the Mälardalen benchmarks. For this purpose, the select process uses the standard deviation of the analysis time to assess the benchmarks.
The calculation of the standard deviation, s, which is based on the analysis time measurements, is the following:

$$\bar{x} = \frac{1}{N}\sum_{i=0}^{N} x_i, \quad s = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(x_i - \bar{x})^2}$$

The results lead to the selection of 7 benchmarks that exhibit the highest standard deviation. These are: "bsort100.c", "ndes.c", "janne_complex.c", "ud.c", "expint.c", "nsichneu.c", and "bs.c".

Parameter Selection The second step uses the 7 benchmarks selected in the previous step. Each benchmark runs for all the 32 different merge parameter combinations, with the same resource limitations as the previous step. For each benchmark, the evaluation runs only for two optimization levels, i.e. 0 and 3 because these optimization levels exhibit the highest diversity. The selection of the best merge parameter combination considers mainly the number of failed benchmarks. There are 5 combinations that perform the same. In addition to them, the "none" merging option is selected. All the selected *merge_point* parameters together result in minimal failed examples. The 6 selected merging combinations are: "all", "le,be,je", "fe,fr,le,je", "le,je", "fr,le,be,je", and "none".

Execution The execution of SWEET consists of two steps. First, the c code is converted to the alf format, using ALF-llvm[6]. This evaluation uses the *c_to_alf* script that SWEET provides. The optimization level of the *c_to_alf* script is not set, but it uses *opt* to apply specific optimizations. The following code is the original *c_to_alf* script:

```
echo "Build $1.ll"
clang -Wall -emit-llvm -S -o - $1.c | opt -mem2reg -instcombine \
                    -instsimplify -instnamer | llvm-dis -o $1.ll
echo "Build $1.alf"
llc $1.ll -march=alf -o=$1.alf
rm $1.ll
```

The modified *c_to_alf* script for handling the four optimization levels is the following:

```
echo "Build $1.ll"
clang -Wall -emit-llvm -O0 -S -o - $1.c | opt -mem2reg -instcombine \
                      -instsimplify -instnamer | llvm-dis -o $1.ll
echo "Build $1.alf"
llc $1.ll -march=alf -o=$1.alf
rm $1.ll
```

The evaluation script that compares SWEET with KTA modifies the optimization flag as follows:

```
sed -i -E "s/-O[0-3]/-O"${optimization}"/g" c_to_alf
timeout $timeout bash c_to_alf ${f}
```

Finally, the evaluation of SWEET uses the *ALF AST cost lookup table* method for automatically calculating the WCET, and runs the benchmark in the following way:

```
time timeout $timeout sweet -i=${f}.alf,std_hll.alf    \
  func=${func} annot=${f}.ann                          \
    -ae vola=t aac=${f}.clt tc=st,op merge=$merge      \
      -do type=$domain
```

File *$f.ann* is the annotation file that can set the range of different variables at different program points. These program points and variables refer to the generated *alf* code, i.e. *$f.alf*. Because the process is automatic, the annotation file initializes only the input variables for each function. The example below illustrates the content of the annotation file, *bs.ann*, that corresponds to the *bs.c* benchmark. Input variable *x* of function *binary_search* is initialized to *TOP_INT*, i.e. all possible integers. This values is set at *FUNC_ENTRY*, i.e. the function entry point.

```
FUNC_ENTRY binary_search ASSIGN "%x" TOP_INT;
```

KTA Configuration

The configuration of KTA uses the default value for the *max-cycles* parameter, i.e. 100,000,000. This part of the evaluation concerns only the IC domain. Hence, the evaluation runs only for the IC domain.

[6]ALF-backend: https://github.com/visq/ALF-llvm

Another parameter that can affect the expressiveness of KTA is the batch size, i.e. the maximum number of program states that may exist before merging. A low number for the *bs_config* parameter can lead to lower analysis time, but at the same time it leads to a more conservative WCET and higher possibility of failure, namely lower expressiveness. The evaluation uses only three values for *bs_config*, i.e. 4, 30, and 100.

6.2.3 Experiment

The final experiment uses the selected merging combinations for SWEET and three different values for KTA. The memory limit is approximately 500MB and the timeout 5 minutes. The following script is the core configuration for the benchmarks. This code snippet runs for all the different optimization levels, i.e. 0-3.

```
ulimit -Sv 500000
timeout=5m
merge_combinations="all le,be,je fe,fr,le,je le,je fr,le,be,je none"
bsconfigs="4 30 100"
domain=clp
timeout $timeout bash c_to_alf ${f}
for merge in $merge_combinations
do
  time timeout $timeout sweet -i=${f}.alf,std_hll.alf    \
    func=${func} annot=${f}.ann                          \
      -ae vola=t aac=${f}.clt tc=st,op merge=$merge      \
        -do type=$domain
done
for bsconfig in $bsconfigs
do
  time timeout $timeout kta wcet ${f}.c ${func} -bsconfig \
    $bsconfig -optimize $optimization
done
```

6.2.4 Results and Discussion

Figures 6.5, 6.6, 6.7, and 6.8 illustrate the results of the comparison of SWEET with KTA for the different optimization levels. The results show the best analysis time for each tool. The analysis time for SWEET does not include the time required for compiling the benchmark with ALF-backend. The configuration of the KTA tool includes the cache analysis, whereas SWEET does not include a cache analysis.

Optimization Level -O0: Figure 6.5 shows the results of the execution of KTA and SWEET. Three of the benchmarks, namely, “fac.c”, the “fib” function of “recursion.c”, and the “ludcmp” function of “ud.c” fail on both tools.
SWEET fails on two more benchmarks, namely all functions of “adpcm.c” and “recursion.c”. The “adpcm.c” functions timeout, whereas “recursion.c” fails on an assertion: assert(exit.nodes.size>0). The latter error is the cause of failure for the “fac.c” benchmark. “ud.c” fail due to a floating-point error. All other benchmarks fail due to a timeout.

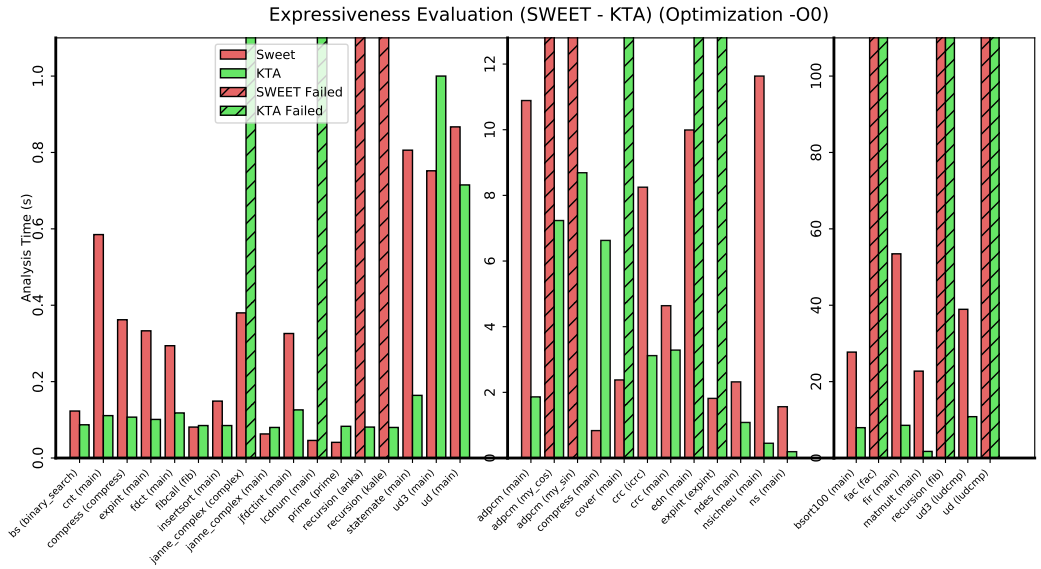


Figure 6.5: SWEET and KTA (Optimization level: -O0). The results are the best for both tools. The red bars represent the analysis time of SWEET and the green the analysis time of KTA. The plot consists of three parts for three ranges of the analysis time, [0,1], (1,20], and (20,300] seconds.

In KTA, the "expint" function of the "expint.c" benchmark fails on a division-by-zero error. The reason for this error is most probably the conservative merge of the abstract domain which includes the zero value. However, this error needs to be further investigated. The "lcdnum.c" and the "cover.c" benchmarks fail because the generated code contains an indirect jump ("jr"). The rest of the benchmark fail with an out-of-memory error.
In all cases, except for the "compress.c", "janne_complex.c", "fibcall.c", and the "ud3.c" benchmarks, KTA, completes the analysis faster than SWEET. With regards to expressiveness, SWEET performs slightly better because in addition to the commonly failed benchmarks, SWEET fails on three additional benchmarks, whereas KTA on five.

Optimization Level -O1: Figure 6.6 shows the results that correspond to the experiment with the level 1 optimizations enabled. In this case, SWEET does not complete the analysis for a different benchmark, namely, "bsort100 (main)", which times out. However, SWEET successfully analyzes the "main" function of the "adpcm.c" benchmark.
KTA has better performance in this optimization level than the previous. It successfully analyzes "janne_complex (complex)", which failed in the previous case.
In this level, the expressiveness of KTA is the same as SWEET, since both tools fail on two benchmarks and on four additional each.
The analysis time for KTA is in this case shorter than SWEET for most of the benchmarks, as well. However, SWEET performs better for various benchmarks, i.e. "expint (main)", "fac (fac)", "fibcall (fib)", "janne_complex (main)", "prime (prime)", "ud3 (main)", and "compress (main)".

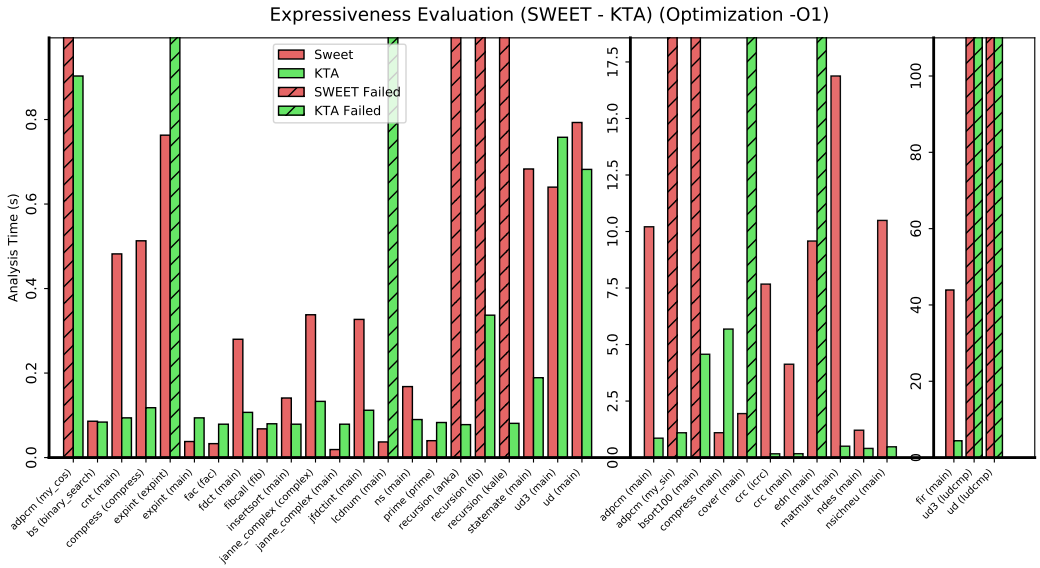


Figure 6.6: SWEET and KTA (Optimization level: -O1). The red bars represent the analysis time of SWEET and the green the analysis time of KTA. The plot consists of three parts for three ranges of the analysis time, [0,1], (1,20], and (20,300] seconds.

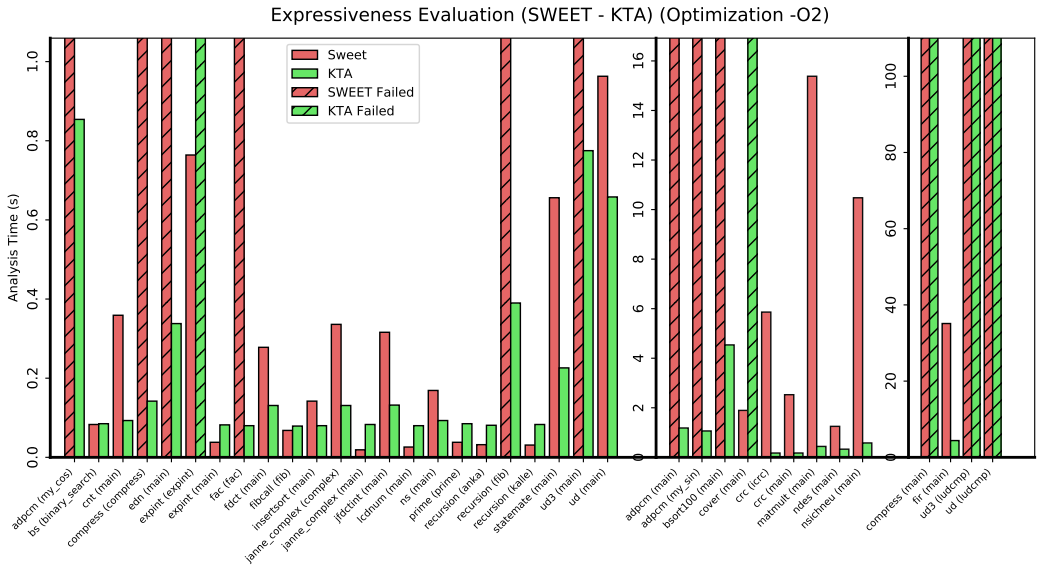


Figure 6.7: SWEET and KTA (Optimization level: -O2). The red bars represent the analysis time of SWEET and the green of KTA. The plot consists of three parts for three ranges of the analysis time, [0,1], (1,20], and (20,300] seconds.

Optimization Levels -O2 and -O3: Figures 6.7 and 6.8 show the results for the higher optimization levels, i.e. -O2 and -O3.
Both tools fail on “compress (main)”, “ud3 (ludcmp)”, and “ud (ludcmp)”. KTA fails on two

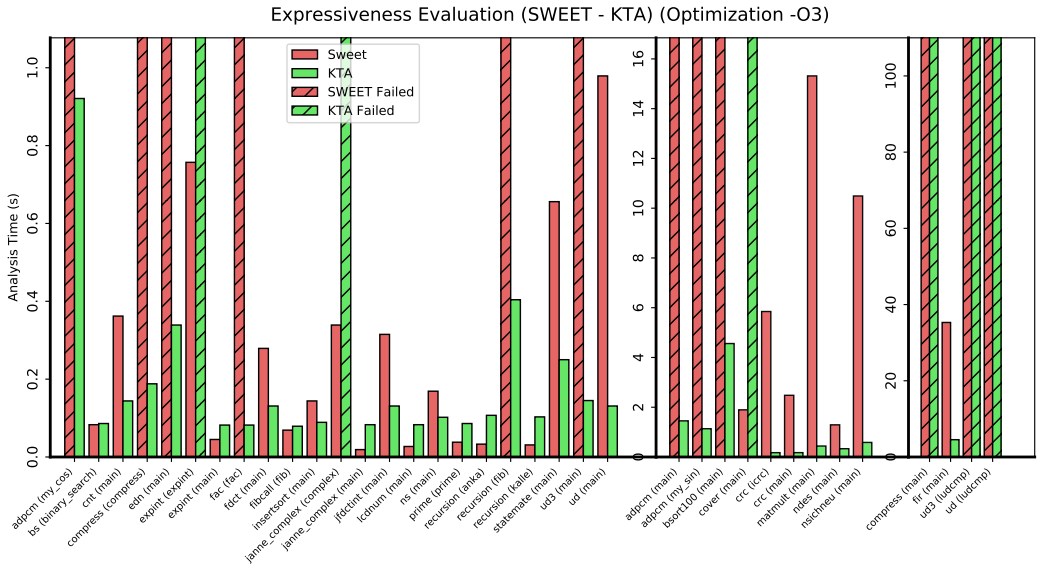


Figure 6.8: SWEET and KTA (Optimization level: -O3). The red bars represent the analysis time for SWEET and the green of KTA. The plot consists of three parts for three ranges of the analysis time, [0,1], (1,20], and (20,300] seconds.

additional benchmarks, i.e. “expint (expint)” and “cover (main)”. With -O3, KTA fails also on “janne_complex (complex)”.
SWEET fails on more benchmarks. Benchmark “compress(compress)” fails with an LLVM error: “Error: emitConstant()”. Benchmark “edn (main)” fails also with an LLVM error: “Error: unsupported: visitInsertElementInst”. The rest of the benchmarks show the same behavior as in the previous optimization levels.
So, when -O2 and -O3 are enabled, KTA seems to perform better with regards to expressiveness, because the number of failed benchmarks is lower.
Also, the analysis time of KTA is in the general case lower. Especially the benchmarks that appear on the second and the third parts of Figures 6.7 and 6.8, demonstrate very high time difference between the two tools.

6.3 Single-core Cache-based Analysis Evaluation

The evaluation of the cache-based analysis consists of two parts. The first part measures the time overhead of the cache analysis on the KTA methodology. The second part attempts to evaluate the tightness of cache-based analysis using a hardware platform, i.e. the Creator Ci40 IoT Hub board.

6.3.1 Analysis Time Overhead

The cache analysis has an overhead on the analysis time. The purpose of this section is to quantify this overhead by comparing the analysis time with and without the cache

analysis.

Experimental Setup

To measure this overhead of the cache, the evaluation uses an optional parameter, i.e. *-nocache*. This parameter deactivates the cache analysis and uses the total memory access time for every access. The cache configuration, e.g. the cache size, associativity, is the same as the next Section, i.e. Hardware-base Evaluation.
The KTA tool parameters are the default, i.e. *bsconfig=4* and *-max_cycles=1000000000*. This evaluation part considers all the different optimization levels, separately. The benchmark configuration of this part is as in Table B.1.
The time measurement uses the *Unix.gettimeofday* function that measures the analysis time, without including the compilation and disassembling stages. The measured time is only the WCET analysis that uses abstract execution.

Results and Discussion

Figures 6.9, 6.10, 6.11, and 6.12 show the results of this comparison. Table 6.2 shows the overhead of the cache analysis. The calculation of the overhead uses Equation 6.1. Also, the pipeline analysis is not deactivated, but the overhead is linear.

$$overhead = \frac{1}{N}\sum_{i=1}^{N}\frac{(t_{i_{cache}} - t_{i_{nocache}})}{t_{i_{nocache}}} \tag{6.1}$$

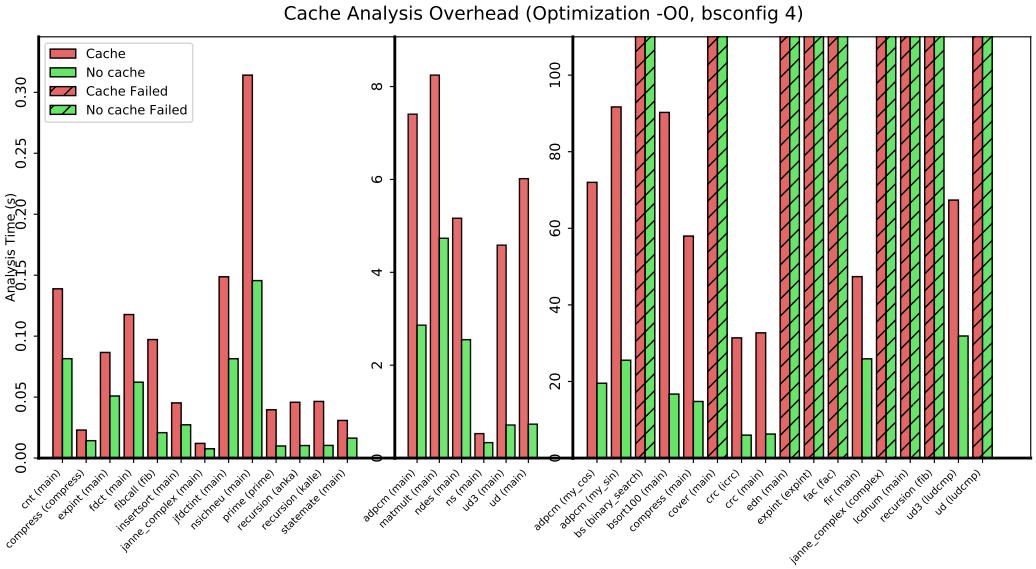


Figure 6.9: Cache analysis overhead for optimization level 0.

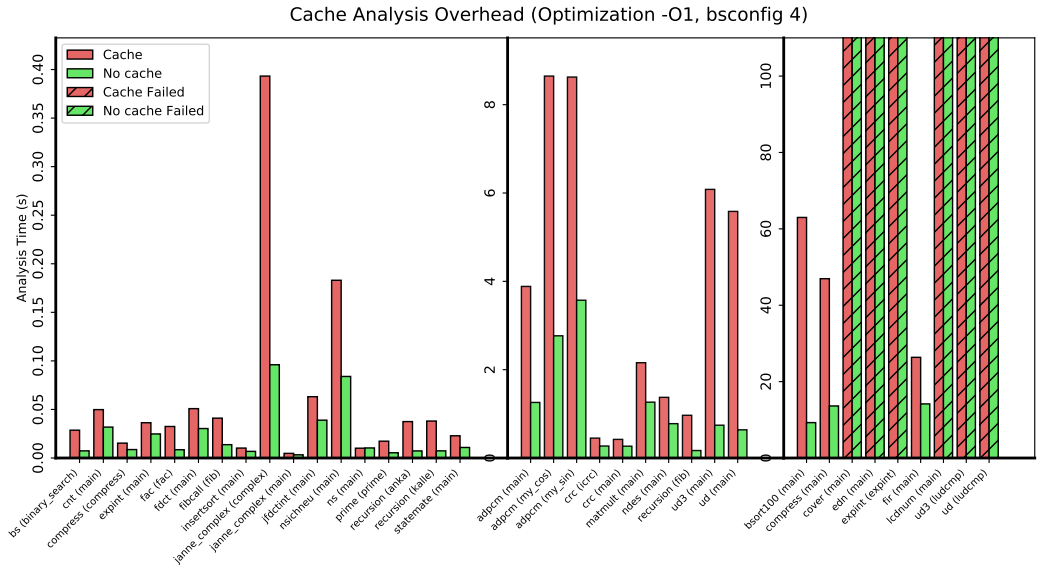


Figure 6.10: Cache analysis overhead for optimization level 1.

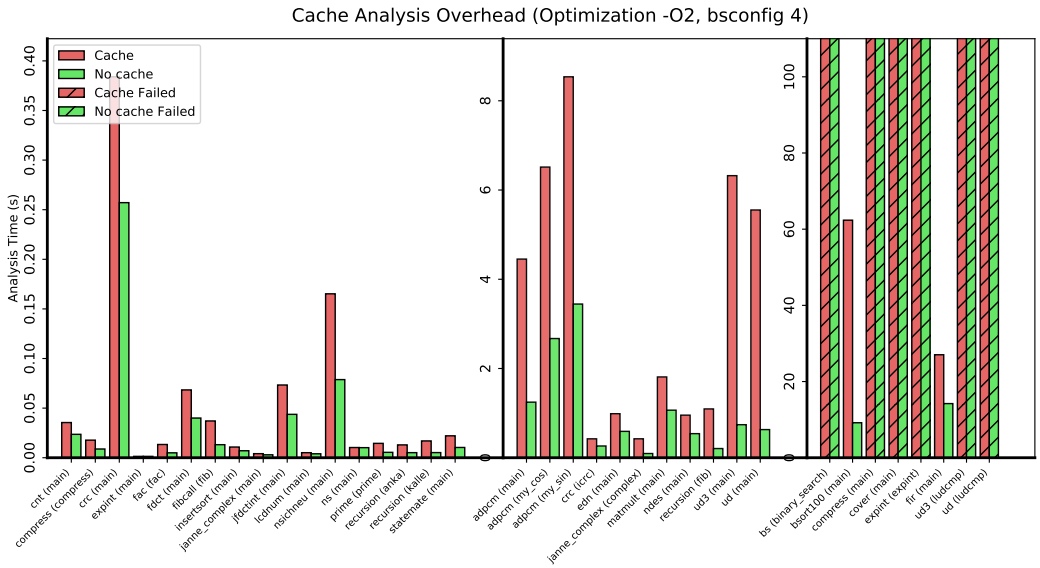


Figure 6.11: Cache analysis overhead for optimization level 2.

Figures 6.9, 6.10, 6.11, and 6.12 show that the cache analysis clearly introduces an overhead to the WCET analysis. Lower optimization levels display larger overhead because in unoptimized code the register allocation is not optimal resulting in more memory accesses. In all cases, the instruction cache adds an overhead. Benchmarks such as “bsort100(main)” introduces higher overhead, because they include operations on arrays, which results in more memory accesses.

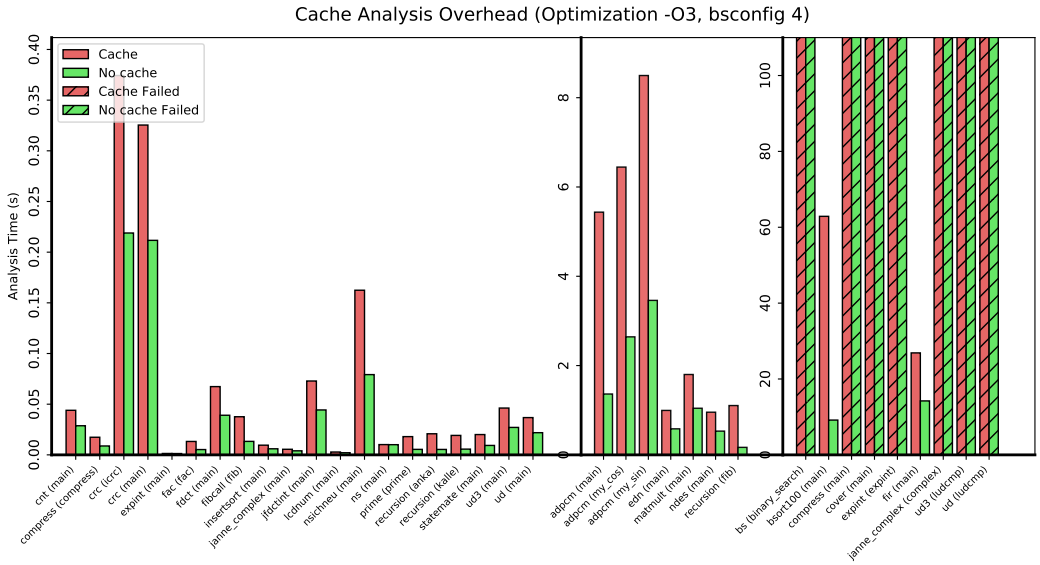


Figure 6.12: Cache analysis overhead for optimization level 3.

Opt Level	Number of benchmarks	Overhead in %
0	27	223.08
1	30	215.43
2	30	179.75
3	29	139.70

Table 6.2: The overhead of the cache analysis for each optimization level using Equation 6.1. The total overhead for lower optimization levels is larger because unoptimized code contains more memory accesses (unoptimized register allocation).

6.3.2 Hardware-based Evaluation

The Creator ci40 board contains a multi-core processing unit with two cores, and each core sees both an L_1 cache and an L_2 cache. The replacement policies of L_1 and L_2 are LRU and pseudo LRU, respectively.
The memory system of cXT200, including the MIPS32® interAptiv™ processor, includes a two level cache hierarchy with a private Level 1 cache, a shared Level 2 cache, and 256MB of RAM. Table 6.3 shows the cache characteristics, such as the cache size, the cache line size, and the associativity.

Cache analysis configuration parameters

The low-level analysis uses configuration parameters that quantify the access times to the memory and the caches, as well as, the execution time of every instruction in the pipeline. These parameters depend on the specific hardware implementation, such as the cache

	Instruction L1 Cache	Data L1 Cache	Unified L2 Cache
Sets per way	256	256	2048
Bytes per line	32	32	32
Ways	4	4	8
Size	32KB	32KB	512KB

Table 6.3: Cache hierarchy characteristics for the Creator Ci40 board, implementing a multi-core MIPS32® architecture.

characteristics, e.g. size and associativity. Therefore, these parameters differ significantly in every hardware implementation.
A way to acquire these parameters is through the documentation provided by the manufacturer. However, this documentation does not provide the precise latency for accessing the L_1 and L_2 caches. The interAptiv™ multiprocessing system user's manual mentions the approximate main memory (DRAM) access latency to be between 50 and 200 processor cycles [24]. However, this number is very approximate.
Because the timing information provided in the documentation is limited, measuring these parameters directly on hardware can provide an estimation of the worst-case latencies. Table 6.5 shows the measurement results. The selection of the measurement parameters is based on the cache characteristics. Different *stride* values emulate cache hit and miss penalties for all the cache levels by loading addresses that are mapped to the same cache-block. When the number of the cache conflicting memory accesses is greater that the number of ways, these consecutive accesses result in continues cache misses. However, the result is not accurate in the case of the Level 1 cache, which is 4-way set associative because there are 8 32-byte write-buffers that hide the cache miss penalty. For this reason, the number of consecutive loads and stores has to be greater than eight.

```
asm volatile ("rdhwr %0, $2":"=r"(start));
asm volatile ("lw %0, %1":"=r"(dst):"r"(src));
...
asm volatile ("rdhwr %0, $2":"=r"(end));
```

Using dynamically measured values for configuring the analysis parameters, makes the analysis vulnerable with regards to safeness. The reason for that is that the measured values depend highly on the quality of the measurements and their extend. It is also well known that dynamic techniques cannot guarantee the worst-case parameters because they provide the worst observed parameters.

Cache-based analysis Evaluation

The low-level cache-based analysis evaluation is based on tightness. In particular, the evaluation measures how tight the estimated WCET of the KTA tool is. The evaluation of the low-level analysis uses the Mälardalen WCET benchmark suite[7] [22].

[7] WCET benchmarks: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

# of sw	Stride	Avg (cycles)	Min (cycles)	Max (cycles)
9	1	1.7	1.6	1.8
10	1	1.9	1.8	2.0
11	1	2.2	2.2	2.2
12	1	2.2	2.2	2.3
9	256	2.9	2.9	2.9
10	256	3.0	3.0	3.0
11	256	3.2	3.1	3.3
12	256	3.2	3.2	3.2
9	2048	12.4	12.4	12.4
10	2048	13.1	13.0	13.2
11	2048	13.7	13.6	13.8
12	2048	14.1	14.0	14.2
9	32768	69.5	55.8	178.0
10	32768	149.1	141.4	220.6
11	32768	194.9	184.9	263.8
12	32768	213.7	202.5	275.7

Table 6.4: Estimation of the average miss latencies for L_1 and L_2 caches for 9-12 consecutive conflicting stores (sw). L_1 cache is 4-way set associative, however, there are $8 \times 32 - byte$ write buffers that compensate for the cache misses.

# of lw	Stride	Avg (cycles)	Min (cycles)	Max (cycles)
9	1	3.2	3.1	3.3
10	1	3.1	3.0	3.2
11	1	3.0	2.9	3.1
12	1	3.2	3.2	3.2
9	256	3.2	3.1	3.3
10	256	3.1	3.0	3.2
11	256	3.0	2.9	3.1
12	256	3.2	3.2	3.2
9	2048	19.0	18.9	19.3
10	2048	19.0	19.0	19.0
11	2048	19.0	18.9	19.3
12	2048	19.4	19.3	19.5
9	32768	93.3	88.9	175.3
10	32768	155.8	148.0	236.2
11	32768	206.6	196.5	276.0
12	32768	195.6	186.2	255.5

Table 6.5: Average miss latencies for L_1 and L_2 caches for 9-12 consecutive conflicting loads (lw).

In this evaluation, KTA analyzes each of the Mälardalen benchmarks (as discussed in 6.2) with concrete values as inputs. The non-concrete inputs are intervals within which the

WCET is known. The input to KTA is a concrete value that is also used in the hardware.

KTA-tool and Hardware setup: The configuration of KTA includes the cache penalty parameters and different optimization levels for the each benchmark.
The code for measuring the execution time of a task on the hardware is the following:

```
#define MEASURE_1_RET(func,input,res) {      \
    asm volatile (  PUSH_ALL "\n\t           \
                    .align 5\n\t             \
                    nop\n\t                  \
                    rdhwr $16, $2\n\t        \
                    li $4," # input "\n\t    \
                    nop\n\t                  \
                    jal " # func "\n\t       \
                    move %[res], $2\n\t      \
                    rdhwr %[end], $2\n\t     \
                    sw $16,%[start]\n\t"     \
                    POP_ALL "\n\t"           \
                    :[res]"=r"(res),         \
                     [end]"=r"(end)          \
                    :[start]"m"(start));     \
}
```

For the evaluation, the measurement of the execution time of the benchmark on the hardware runs multiple times in a loop. However, only during the first execution the caches are uninitialized resulting in compulsory misses. As a result, the first execution is in all measurements the worst case execution time that we are able to measure. In order for the result not to depend on one sole measurement (the first), during the evaluation of this part, each benchmark execution is repeated 10 times after a reset. For a result to be as near to the WCET as possible, the hardware measurements are taken on a cold start and the variables used for measuring the time (e.g. start and end) are loaded before the first call, so that they will be cached. In addition to that, as discussed in Section 6.1.4, the measurement code is aligned to the instruction cache line size in order to avoid the measurement of additional instruction cache misses.
MIPS® interAptiv™ processor contains a number of advanced techniques for accelerating the hardware. These include: out-of-order load and store completion, instruction cache way prediction, branch prediction, non-blocking loads, and return address branch prediction. However KTA does not implement these mechanisms, so the evaluation should not include them. Configuration register 7 can disable these mechanisms. Table 6.6 shows the configuration options, as well as, the respective bits of configuration register 7 that were set [24].

Results and Discussion

Figures 6.13, 6.14, 6.15, and 6.16 show the results of the evaluation. The ideal result would be that the WCET result and the hardware execution are the same. However, the abstract model that KTA models and the actual hardware are not identical. So, the results of the

Option	Description	Config7 bit
ICWP	Instruction cache way prediction	12
CPOOO	Out-of-order Data return	6
ULB	Uncached Load Blocking	4
BP	Branch Prediction	3
RPS	Return Prediction Stack	2
SL	Non Blocking Loads.	0

Table 6.6: Configuration options for disabling speculative mechanism of the interAptiv™ processor.

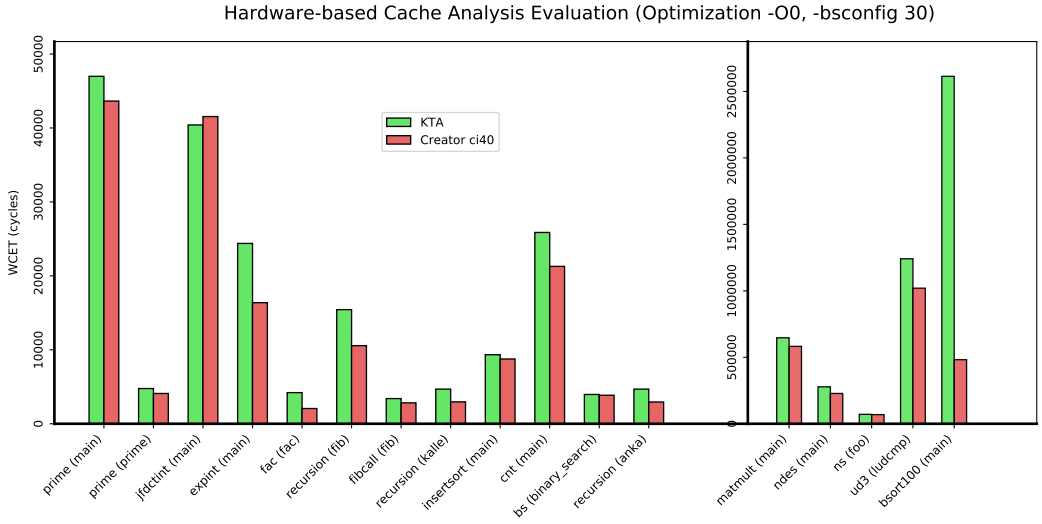


Figure 6.13: Single-core cache analysis evaluation for bsconfig=30.

hardware and KTA differ. More importantly, in some of the benchmarks, e.g. "fdct(main)", the WCET result is not the expected, because of incompatibilities with the hardware timing model. The Creator ci40 board implements complex mechanisms that KTA abstracts. One example is the replacement policy of the L_2 cache, which is pseudo LRU. However, KTA models LRU replacement policy for both the L_1 and the L_2 caches.

6.4 Multi-core Cache-based Analysis Evaluation

The evaluation of the multi-core analysis is more difficult than the single-core case, because a multi-core execution time depends on coherence cache misses and the bus traffic that is difficult to reproduce on the hardware. Also, the traffic affects the memory access time in a non-predictable way. The procedure for measuring the execution time for the multi-core analysis is similar to the single-core cache evaluation. The analysis measures the execution time dynamically on the hardware and subsequently, compares the worst of the measured cases to the KTA analysis result.

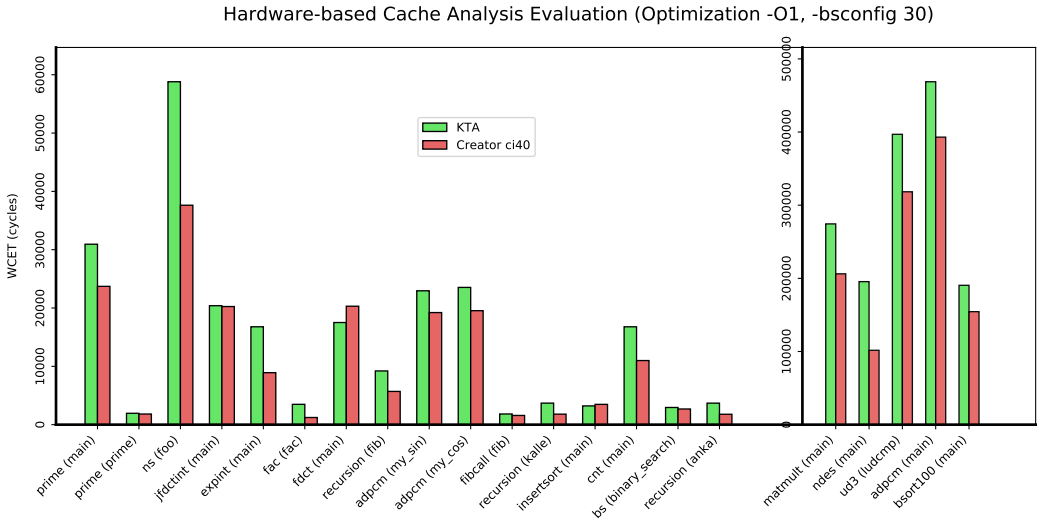


Figure 6.14: Single-core cache analysis evaluation for bsconfig=30.

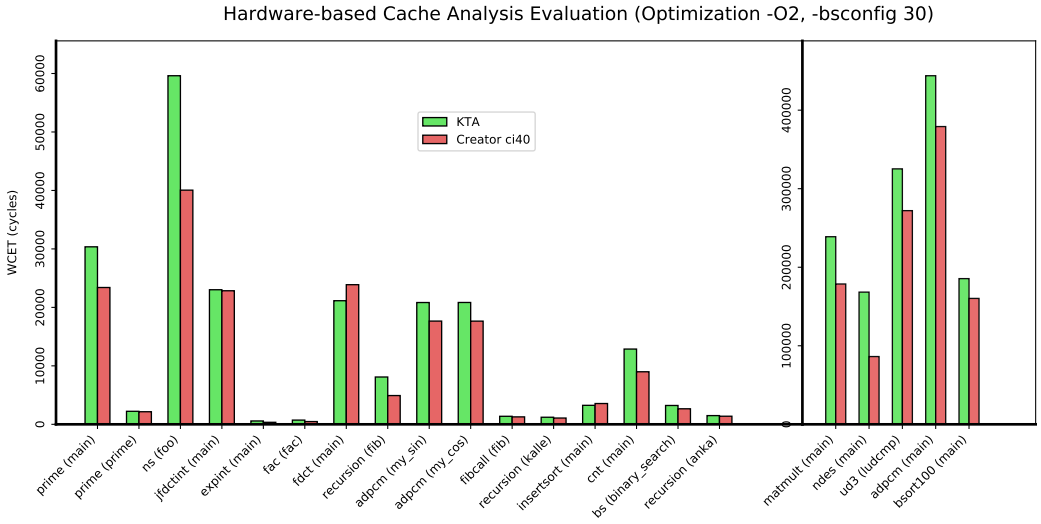


Figure 6.15: Single-core cache analysis evaluation for bsconfig=30.

6.4.1 Experimental Setup

The following subsections describe the experimental setup, with regards to (1) the benchmarks, (2) the hardware configuration, and (3) the KTA tool configuration.

Multicore Analysis Evaluation Benchmarks

This part of the evaluation does not use the Mälardalen benchmarks because they are sequential. Instead, the evaluation uses a number of small benchmarks that are able to

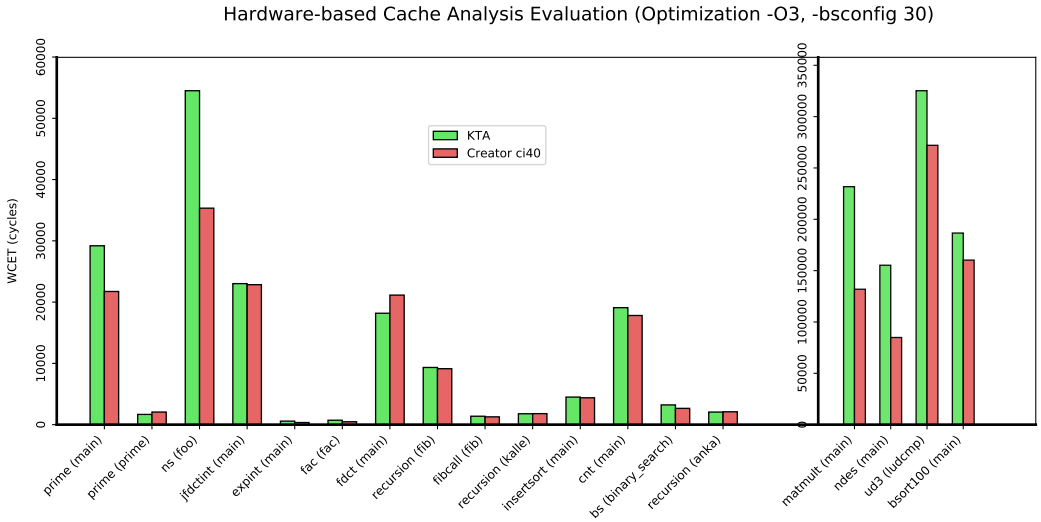


Figure 6.16: Single-core cache analysis evaluation for bsconfig=30.

support the multi-core analysis. The evaluation runs the benchmarks on the two-core Creator ci40 board and subsequently, compare the worst-case result against KTA.
The benchmarks consist of 6 pairs of tasks and use arrays (vectors) with the size of one level-1 data cache line, i.e. 32 bytes. Each task is aligned to the instruction cache line size using the compiler flag *-falign-functions=32* (mips-mti-elf). Hence, the instructions of the two temporally interfering tasks do not reside at the same instruction cache line that would cause false sharing. If false sharing was possible, a cache line that has been loaded to the shared level-2 cache would result in a hit for the other processor. This condition makes the execution time dependent on the order of the execution of the two cores and therefore, the evaluation becomes more complex. The following paragraphs describe briefly the six task pairs.

Task pair 1: Read-Read The two tasks of this pair, *task1_0* and *task1_1*, read the same global array *vectors.v*, and share no further addresses. This is an R/R case, namely the case when the temporally interfering tasks do not modify (write) any shared address, but only read shared addresses. The model of the multi-core analysis assumes that reading from a shared address increases the traffic, but it actually does not result into invalidates that would affect the cache hit rate.

```
// R - R
void task1_0(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v0[i] = vectors.v[i] + 1;
  }
}
void task1_1(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v1[i] = vectors.v[i] + 1;
  }
}
```

Task pair 2: Read-Write One of the tasks, *task2_0*, reads the global array *vectors.v*, and the other task, *task2_1*, writes to the same array. This is an R/W case, which means that one of the temporally interfering tasks reads from a shared address and the other writes to the same address. The multi-core model expects that the latter task will invalidate the shared cache block, resulting in L_1 cache misses for the task that reads the shared block (*task2_1*).

```
// R - W
void task2_0(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v0[i] = vectors.v[i] + 1;
  }
}
void task2_1(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v1[i] + 1;
  }
}
```

Task pair 3: Write-Write Both tasks, i.e. *task3_0* and *task3_1*, write to the same global array *vectors.v*, and share no further addresses. This is a W/W case, namely the case when both the temporally interfering tasks modify the same cache block. According to the model, every task will broadcast an invalidate to all remote caches and subsequently, the cache will write back the data to the shared memory, here the L_2 cache. The cache write-allocate policy causes the next task to load the block, and based on the model, a cache-to-cache transaction will occur.

```
// W - W
void task3_0(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v0[i] + 1;
  }
}
void task3_1(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v1[i] + 1;
  }
}
```

Task pair 4: Read/Write-Write One of the tasks, *task4_0*, reads and writes to the global array *vectors.v* that covers one cache line. The other task, i.e. *task4_1*, only writes to the same array. This is a RW/W case, namely one of the tasks reads and writes to a shared address, whereas the other only writes to the same array. According to the model, the reading (and writing) task will have to reload the shared block, if the writing task writes to it before the other task.

```
// RW - W
void task4_0(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v[i] + 1;
  }
}
void task4_1(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v1[i] + 1;
  }
}
```

Task pair 5: Read/Write-Read/Write Both tasks, i.e. *task5_0* and *task5_1*, read and write to the same global array *vectors.v*, and share no further addresses. This is a RW/RW case, i.e. both tasks read and write to the same shared addresses. Every write, invalidates the copies in the other caches. Reads may result to a HIT or a MISS, depending on the which *core* modified the memory block last.

```
// RW - RW
void task5_0(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v[i] + 1;
  }
}
void task5_1(void) {
  int i;
  for (i=0; i<N; i++) {
    vectors.v[i] = vectors.v[i] + 1;
  }
}
```

Task pair 6: Read and Read/Write-Read/Write The two tasks of this benchmark are *task6_0* and *task6_1*. They are similar to the previous case, i.e. task pair 5, with the difference that task 6 pair involves reading from a non-shared memory buffer, i.e. `vectors.v0` and `vectors.v1`.

```
// RW - RW
void task6_0(void) {
  int i;
  for (i=0; i<N; i++)
    vectors.v[i] = vectors.v[i]
                    + vectors.v0[i] + 1;
}
void task6_1(void) {
  int i;
  for (i=0; i<N; i++)
    vectors.v[i] = vectors.v[i]
                    + vectors.v1[i] + 1;
}
```

Hardware Configuration

The measurements on the hardware use the same configurations as in the case of the single-core cache analysis, but also activates the second core. The interAptiv™ processor implements also hardware threads, which are inactive during this evaluation. The benchmark tasks take no inputs, so the code in the following Listing applies to all benchmarks:

```
#define MEASURE_0_RET(func,res,start,end) {     \
    asm volatile (  PUSH_ALL "\n\t              \
                    .align 5\n\t                \
                    nop\n\t                     \
                    rdhwr $16, $2\n\t           \
                    nop\n\t                     \
                    jal " # func "\n\t          \
                    move %[res], $2\n\t         \
                    rdhwr %[end], $2\n\t        \
                    sw $16,%[start]\n\t"        \
                    POP_ALL "\n\t"              \
                    :[res]"=r"(res),            \
                     [end]"=r"(end),            \
                    :[start]"m"(start));  }
```

KTA Configuration

The configuration of KTA corresponds to analyzing temporal and spatial interfering tasks. This evaluation considers only the 0 optimization level (-O0), because the code is simple and the different optimizations levels do not alter the code significantly. All the other parameters are set to their default options.
For each benchmark, i.e. pair of tasks, KTA performs the analysis for 4 cases, (1-2) each of the two tasks run independently with no interference and (3-4) each of the two tasks run under the presence of the other task.

```
#!/bin/bash
for i in {1..6}
do
    kta wcet ${MBENCH_PATH}/tasks.elf task${i}_0
    kta wcet ${MBENCH_PATH}/tasks.elf task${i}_1
    kta wcet ${MBENCH_PATH}/tasks.elf task${i}_0 -tasks task${i}_1
    kta wcet ${MBENCH_PATH}/tasks.elf task${i}_1 -tasks task${i}_0
done
```

6.4.2 Results and Discussion

The following subsections present (1) the hardware measurements for the average case and (2) the final WCET results of KTA. The average case measurements illustrate the hardware behavior when there is temporal and spatial interference between the cores. The second part compares the result of KTA with actual worst-case measurements on the hardware.

Average Case

Figures 6.17-6.22 shows the average case execution time, excluding the initial cold (uncached) executions. Every subfigure corresponds to each of the task pairs. This result shows that the correlation of the cores when they interfere matches with the expected. In all six examples, the non-interfering measurements are the same, which means that there is no execution-time assymetry between the pair tasks. However, when running simultaneously, it is possible that there is assymetry in the execution order of the tasks.

Task pair 1: Read-Read This case consists of two small tasks called *task1_0* and *task1_1*. These tasks are interfering temporally, i.e. running at the same time. The spatial interferenece includes reading the same data vector with no other spatial interference. This represents the R/R case, i.e. both tasks that are temporally interfering solely read from the same data, but do not display any further spatial interference. The presence of the core has small impact on the actual execution time. Figure 6.17 shows the effect of this type of interference.

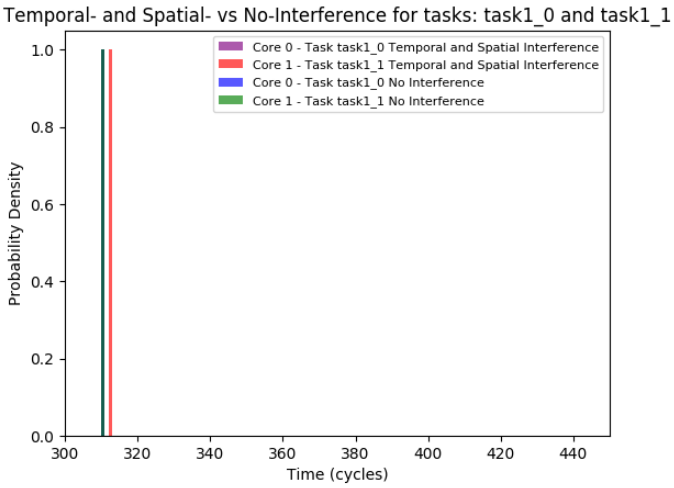


Figure 6.17: Execution time on Creator ci40 without the initial cold misses of the cache. Each core executes on of the tasks. One hardware thread runs on each core.

Task pair 2: Read-Write The second pair, i.e. *task2_0* and *task2_1*, consists of one task reading and the other task writing to a shared address. In this case, the writing task, i.e. task2_1, forces the reading task, i.e. task2_0, to invalidate its L_1 cache copy. That will cause a new cache miss, when task2_0 attempts to load one of the words in the cache block (vector). Figure 6.18 shows this effect, with core 0 delaying almost 100 cycles, when core 1 writes to the shared address. Also, core 1 is also affected by core 0 because every write takes more time waiting to send invalidates to the shared caches.

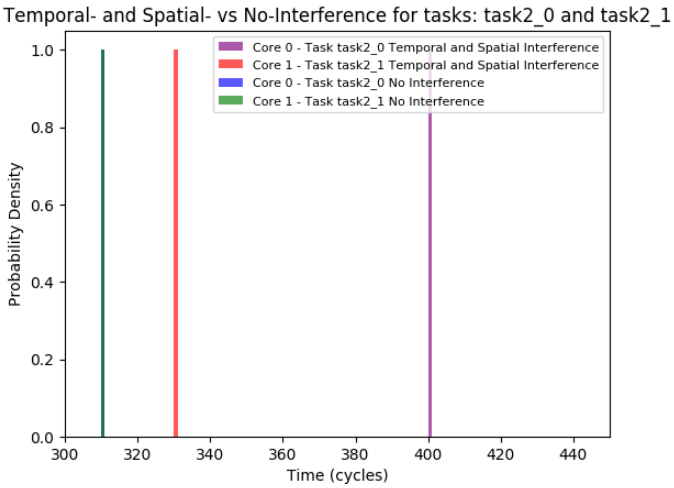


Figure 6.18: Execution time on Creator ci40 without the initial cold misses of the cache. Each core executes on of the tasks. One hardware thread runs on each core.

Task pair 3: Write-Write The third task pair consists of two tasks, i.e. *task3_0* and *task3_1*, that write to the same address, while temporally intefering. The two tasks display no further spatial interference. This case displays the same effect as in the previous case, i.e. the write to a shared data vector causes an additional delay to invalidate and write back. The presence of write buffers hides the miss latency. Figure 6.19 shows the effect of this type of spatial inteference to the average execution time.

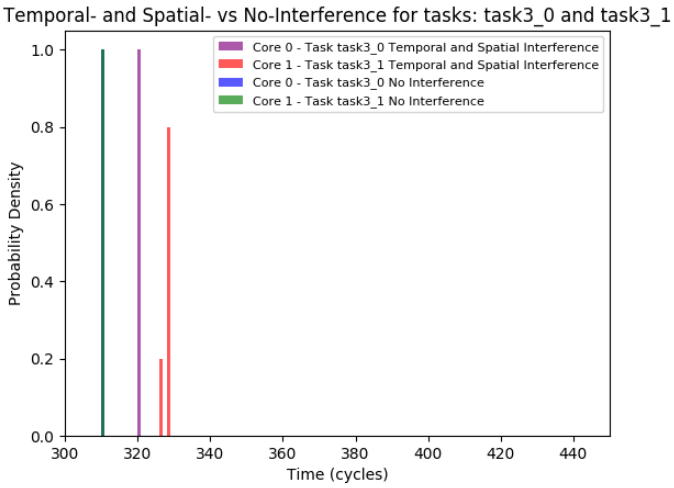


Figure 6.19: Execution time on Creator ci40 without the initial cold misses of the cache. Each core executes on of the tasks. One hardware thread runs on each core.

Task pair 4: Read/Write-Write The forth task pair consists of two tasks, i.e. *task4_0* and *task4_1*, that interfere spatially while running on one core each. Core 0 reads and writes from the shared memory block, whereas core 1 only writes. Core 1 seems to have a delay due to the invalidation of the shared data. The delay of core 0 is not so large, for similar reasons as with task2. In particular, core 0 performs some of the reads before the writes of core 1 to the shared address. Figure 6.20 shows the effect of the R/W-W type of spatial inteference to the average execution time.

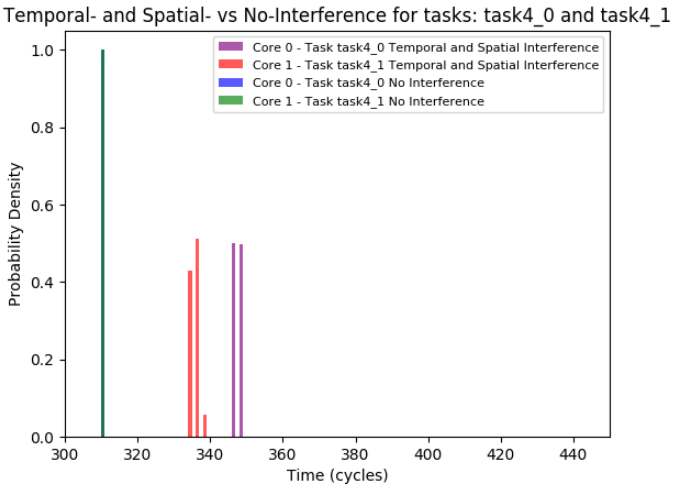


Figure 6.20: Execution time on Creator ci40 without the initial cold misses of the cache. Each core executes on of the tasks. One hardware thread runs on each core.

Task pair 5: Read/Write-Read/Write In the case when both tasks, *task5_0* and *task5_1*, read and write to the shared address the delay is analogous to the task2 example. Both tasks need to invalidate the data when they write, but also wait for reading the modified data. The asymmetry is probably due to the order that the two processes access the data. Figure 6.21 shows the effect of this type of spatial inteference to the average execution time.

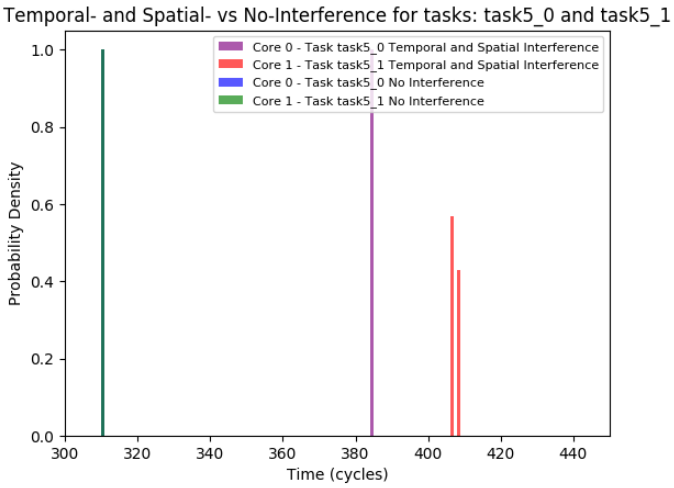


Figure 6.21: Execution time on Creator ci40 without the initial cold misses of the cache. Each core executes on of the tasks. One hardware thread runs on each core.

Task pair 6: Read and Read/Write-Read/Write This task pair, i.e. *task6_0* and *task6_1*, is similar to task5, with the difference that this pair performs one additional read to a non-shared address. Figure 6.22 shows the effect of this type of spatial inteference to the average execution time. The average execution time in this case is symmetric, i.e. the red and purple lines that correspond to the measurements of the interfering tasks are very near, indicating that the measurements are symmetric with regards to the order of the two tasks.

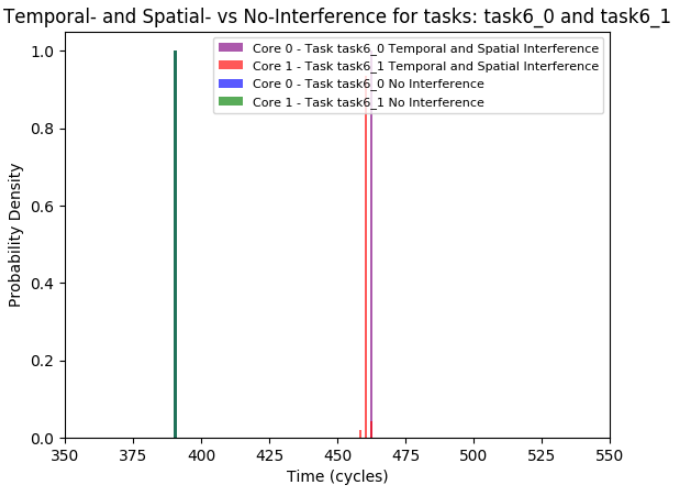


Figure 6.22: Execution time on Creator ci40 without the initial cold misses of the cache. Each core executes on of the tasks. Only one hardware thread runs on each core.

Worst-Case Execution Time

task number	task_0		task_1		task_0/task_1		task_1/task_0	
	ci40	KTA	ci40	KTA	ci40	KTA	ci40	KTA
1	1994	3064	2048	3064	2046	3064	2038	3064
2	1994	3064	2056	3064	2038	5083	2040	3177
3	2048	3064	2054	3064	2048	3429	2040	3429
4	2002	2776	2056	3064	2048	5448	2040	3429
5	1986	2776	2054	2776	2048	5448	2038	5448
6	2516	3468	2530	3468	2508	6137	2500	6137

Table 6.7: The table contains the worst measured execution time on the hardware (Creator ci40) and the result of the analysis for tasks that interfere temporally and spatially with tasks running on other cores. Every line corresponds to each benchmark. Every row corresponds to different analysis; the first two columns correspond to the single-core analysis of the first (respectively second) task of the pair that comprises the benchmark, the last two columns correspond to the multi-core analysis of one task when the other interferes temporally and spatially.

The previous case concerns the average execution time on the hardware, which derives by excluding the very high measurements. These very high measurements occur only during the first execution of the task on the hardware, when the data and the instructions are not loaded to the caches. The latter measurements correspond to the worst-case measured execution time. Table 6.7 demonstrates the measurements on the hardware together with the results of KTA. The results of KTA model the worst possible execution time when each task executes independently and when it runs under the presence of another temporally and spatially interfering task, i.e. the target task's pair task. However, Table 6.7 shows that the presence of interfering tasks does not affect the worst-case hardware measurements, as it affects the average-case hardware measurements. This result indicates that the multi-core analysis is conservative and might be possible to tighten it. However, the worst-case hardware measurement do not necessarily correspond to the actual worst-case execution time and in a multi-core system, it is difficult to emulate the worst case multi-core interference, e.g. a sequence of requests that result in continuous cache misses. An identified limitation of the approach is that the memory-access recordings do not provide information about the order of the remote memory accesses. Providing such information will add an addition overhead to the analysis, but it might tighten the resulting WCET.

Chapter 7

Conclusion and Future Work

This chapter summarizes the work of this thesis (Section 7.1) and discusses the future work (Section 7.2) that may improve, generalize, and extend the current work.

7.1 Conclusion

This dissertation concerns the problem of estimating the WCET of a task. The WCET problem is important in the field of embedded systems, where safety-critical systems have time constraints. The following subsections summarize the work of this thesis project and discuss the results of the evaluation.

7.1.1 Feasibility of KTA

The first objective of this thesis is to evaluate the feasibility of the one-pass method of KTA [13] after integrating a low-level analysis. The low-level analysis consists of a cache, cache hierarchy, and pipeline analysis, mechanisms present in most modern processing systems. The evaluation examines the expressiveness of the one-pass method including the overhead of the low-level analysis. In particular, the evaluation compares the analysis time of KTA against SWEET, a widely used WCET tool. This comparison applies different optimization levels, i.e. -O0, -O1, -O2, and -O3 to both tools and examines the coverage and performance of the two methods. The results differ based on the optimization level. However, the default configuration for SWEET uses the default optimization flag (-O0). For this reason the conclusion summarizes the results of the default configuration that corresponds to -O0. The benchmarks that did not return a result form three categories: *timeout*, *out-of-memory*, and *failed*. The last category represents any type of failure that can range between compile-time errors and analysis failures. Table 7.1 shows the results of the evaluation with regards to the number of the unsuccessfully terminating benchmarks for each tool.
Regarding the performance, excluding all non-terminating benchmarks, 5 benchmarks perform better in SWEET than KTA, and 18 benchmarks perform better in KTA than

failure cause	KTA	SWEET
timeout	0	1
out-of-memory	5	0
failed	3	4

Table 7.1: Number of benchmarks that do not return a WCET for KTA and SWEET. These results correspond to -O0 optimization flag.

SWEET. This result supports the feasibility of the WCET method of KTA that includes a low-level cache analysis.
The next section, Section 7.2, discusses the future work that can improve the coverage of KTA by implementing the missing instructions and including a more expressive relative abstract domain, i.e. the polyhedra domain [49]. In addition to that, the failed *out-of-memory* benchmarks indicate that there might be possibilities in reducing the memory footprint of KTA.

7.1.2 Multiprocessor analysis

The second objective of this thesis project is to estimate the WCET on shared-memory multiprocessor systems. The contribution of this thesis is to design and implement a multi-core analysis by extending the WCET analysis method of KTA. The multi-core analysis estimates the WCET of a task running on an *symmetric multiprocessor* (SMP) system under the presence of temporally and spatially interfering tasks. The analysis uses the low-level analysis of the single-core approach that models an LRU cache and a 5-stage RISC-based pipeline, features that are common in embedded-system applications. Furthermore, the multi-core analysis implements MESI, a cache-coherence protocol commonly used in small-scale multiprocessor systems used in the field of embedded systems.
The evaluation of the multiprocessor analysis reveals some difficulties in evaluating a multi-core approach using a hardware platform. These difficulties depend on the complexity of the effects that the temporal interference has on parallel tasks and the difficulty in controlling this interference without intervening in the execution.

7.2 Future Work

The project focuses on the low-level WCET of symmetric multiprocessor systems with private caches as well as the abstract domain representation. The following points summarize the improvements and extensions to the current implementation that are part of the future work.

7.2.1 Implementation

- Implementation of unimplemented instructions, such as indirect jumps and frequently used *Special* MIPS32® instructions.

- Evaluation and validation of the analysis using the most recent and complete TACLe benchmark suite.
- Replace the value abstract domain with the polyhedra abstract domain [49]. The polyhedra domain is relational and is expected to increase the coverage of the method.
- Extension of the multiprocessor analysis to support other coherence protocols.
- Improve the memory footprint of KTA by optimizing the current implementation or selecting appropriate data structures for the cache and memory.

7.2.2 Future Research

- Redesign the pipeline to make it less dependent on specific dependency patterns. The purpose is to design a configurable state for different pipeline configurations.
- Analysis and integration of code patterns into the analysis in order to improve the expressiveness. These patterns include common code sequences that occur as a result of compiler optimizations.

Appendix A

CPS code

```
open AbstractMIPS

open Printf

open Unix

let gp_addr=(4231280)
let mem = []

(* -- Basic Block Identifiers -- *)

let final_      = 0
let fact_       = 1
let l2_         = 2
let l1_         = 3
let ex1_        = 4

(* -- Program Code -- *)

let final ms = ms

(* Function: fact *)

let fact ms = ms                        |>
  addiu    sp sp (-16)                  |>
  sw       fp 12(sp)                    |>
  addu     fp sp zero                   |>
  sw       a0 16(fp)                    |>
  addiu    v0 zero 1                    |>
  sw       v0 0(fp)                     |>
  jds      l1_                          |>
  sll      zero zero 0                  |>
  next

let l2 ms = ms                          |>
  lw       v1 0(fp)                     |>
  lw       v0 16(fp)                    |>
```

```
  mult     v1 v0                          |>
  mflo     v0                             |>
  sw       v0 0(fp)                       |>
  lw       v0 16(fp)                      |>
  addiu    v0 v0 (-1)                     |>
  sw       v0 16(fp)                      |>
  next

let l1 ms = ms                            |>
  lw       v0 16(fp)                      |>
  sltiu    v0 v0 2                        |>
  beqds    v0 zero l2_                    |>
  sll      zero zero 0                    |>
  next

let ex1 ms = ms                           |>
  lw       v0 0(fp)                       |>
  addu     sp fp zero                     |>
  lw       fp 12(sp)                      |>
  addiu    sp sp 16                       |>
  jrds     ra                             |>
  sll      zero zero 0                    |>
  ret


let bblocks =
[|
{func=final;    name="final";   nextid=na_;    dist=0;
addr=0x00000000; caller=false;};
{func=fact;     name="fact";    nextid=l1_;    dist=2;
addr=0x00400018; caller=false;};
{func=l2;       name="l2";      nextid=l1_;    dist=2;
addr=0x00400038; caller=false;};
{func=l1;       name="l1";      nextid=ex1_;   dist=1;
addr=0x00400058; caller=false;};
{func=ex1;      name="ex1";     nextid=na_;    dist=0;
addr=0x00400068; caller=false;};
|]

(* -- Start of Analysis -- *)

let main =
        let _st_time = Unix.gettimeofday() in
        analyze fact_ bblocks gp_addr mem [] [] 0;
        printf "Time Elapsed: %fs\n" (Unix.gettimeofday() -. _st_time)
```

Appendix B

Mälardalen Benchmarks

Benchmark	Function	Input	Floating-Point	Known Issues	
				SWEET	KTA
adpcm.c	main				
	my_cos	a0=[0,6282]			
	my_sin	a0=[0,6282]			
bs.c	binary_search	a0=Any			
bsort100.c	main	-			
cnt.c	main	-			
compress.c	main	-		-O2, -O3: The compilation fails with an LLVM Error: emitConstant()	
	compress	-			
cover.c	main	-			Indirect Jumps: *jr*, not supported
crc.c	main	-			
	icrc	a0=[0,5] a1=[40,50] a2=[0,5] a3=[1,5]			
duff.c			✗		
edn.c	main	-		-O2, -O3: The compilation fails with and LLVM Error: unsupported: visitInsertElementInst()	
expint.c	main	-			
	expint	a0=[30,50] a1=[1,10]			
fac.c	fac	a0=[1,10]			
fdct.c	main	-			
fft1.c			✗		
fibcall.c	fib	a0=[0,10]			
fir.c	main	-			
insertsort.c	main	-			

janne_complex.c	main	-			
	complex	a0=[0,30] a1=[0,30]			
jfdctint.c	main	-			
lcdnum.c	main	-			Indirect Jumps: *jr*, not supported, but some optimization flags eliminate them.
lms.c			✗		
ludcmp.c			✗		
matmult.c	main	-			
minver.c			✗		
ndes.c	main	-			
ns.c	main	-			
nsichneu.c	main	-			
prime.c	prime	a0=[0,100]			
qsort-exam.c			✗		
qurt.c			✗		
recursion.c	fib	a0=[0,10]		The analysis terminates with failed assertion: assert(exit.nodes.size>0).	
	kalle	a0=[0,10]			
	anka	a0=[0,10]			
select.c			✗		
sqrt.c			✗		
statemate.c	main	-			
st.c			✗		
ud.c	ludcmp	a0=[0,30] a1=[0,30]			handles *long* as a 64-bit number, but the C language specification [26, 27] specifies *long* to be larger than or equal to 32 bits, and mcb32-gcc produces code for 32-bit *long* integers.
	main	-			
ud3.c	ludcmp	a0=[0,30] a1=[0,30]			Modified ud.c (above) that uses *long long* integers, which always correspond to at least 64 bits.
	main	-			
whet.c			✗		

Table B.1: The table shows the benchmarks of the Mälardalen Benchmark Suite. Expressiveness evaluation uses all these benchmarks that do not contain floating-point numbers. This table shows also the functions that the expressiveness evaluation uses. A number of benchmarks contain known issues for either SWEET or KTA.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*, volume 2. Addison-wesley Reading, 2007.

[2] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. *OTAWA: An Open Toolbox for Adaptive WCET Analysis*, pages 35–46. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[3] Bruno Blanchet. Introduction to abstract interpretation, 11 2002. URL http://www.cs.tau.ac.il/~msagiv/courses/asv/absint-1.pdf. Accessed: 2017-10-18.

[4] Claire Burguière and Christine Rochange. History-based schemes and implicit path enumeration. In *OASIcs-OpenAccess Series in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006.

[5] Stefan Bygde. Abstract interpretation and abstract domains. Master's thesis, Mälardalen University, June 2006.

[6] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multicore platforms. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):124, 2014.

[7] Patrick Cousot. Abstract interpretation. *ACM Computing Surveys (CSUR)*, 28(2):324–328, 1996.

[8] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of generalized type unions. In *ACM SIGPLAN Notices*, volume 12, pages 77–94. ACM, 1977.

[9] 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*, pages 238–252. ACM, 1977.

[10] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978.

[11] *cXT200 SoC Datasheet*. Creator. URL https://docs.creatordev.io/ci40/guides/hardwaredocs/cXT200_datasheet2.pdf. Accessed: 2017-06-29.

[12] Christoph Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, March 2013.

[13] David Broman. A Brief Overview of the KTA WCET Tool. *ArXiv e-prints*, December 2017.

[14] A. Ermedahl, J. Fredriksson, J. Gustafsson, and P. Altenbernd. Deriving the worst-case execution time input values. In *2009 21st Euromicro Conference on Real-Time Systems*, pages 45–54, July 2009.

[15] Christian Ferdinand and Reinhold Heckmann. *aiT: Worst-Case Execution Time Prediction by Static Program Analysis*, pages 377–383. Springer US, Boston, MA, 2004.

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

[17] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. Time for reactive system modeling: Interactive timing analysis with hotspot highlighting. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems*, pages 289–298. ACM, 2016.

[18] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. *An Abstract Domain of Uninterpreted Functions*, pages 85–103. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[19] Philippe Granger. *Static analysis of linear congruence equalities among variables of a program*, pages 169–192. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[20] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 57–66. IEEE, 2006.

[21] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In Niklas Holsti, editor, *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET09)*. OCG, June 2009.

[22] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[23] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[24] *InterAptiv™ Multiprocessing System Datasheet.* Imagination Technologies LTD, 5 2013. URL http://cdn2.imgtec.com/documentation/MD00903-2B-interAptiv-DTS-01.20.pdf. Accessed: 2017-06-29.

[25] *MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual.* Imagination Technologies LTD, 5 2016. URL https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MIPS_Architecture_MIPS32_InstructionSet_%20AFP_P_MD00086_06.05.pdf. Accessed: 2017-05-09.

[26] *N1548 - Committee Draft - C11 Specification.* ISO/IEC 9899, 9 2000. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf. Accessed: 2017-09-04.

[27] *N1256 - Committee Draft - C99 Specification.* ISO/IEC 9899, 9 2000. URL http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf. Accessed: 2017-09-04.

[28] Linus Källberg. Circular linear progressions in SWEET. Master's thesis, Mälardalen University, December 2014. Version 1.0.

[29] Daniel Kästner, Marc Schlickling, Markus Pister, Christoph Cullmann, Gernot Gebhard, Reinhold Heckmann, and Christian Ferdinand. Meeting real-time requirements with multi-core processors. In *International Conference on Computer Safety, Reliability, and Security*, pages 117–131. Springer, 2012.

[30] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 333–339, May 2008.

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

[32] Y-TS Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Real-Time Systems Symposium, 1995. Proceedings., 16th IEEE*, pages 298–307. IEEE, 1995.

[33] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *ACM SIGPLAN Notices*, volume 30, pages 88–98. ACM, 1995.

[34] Björn Lisper. *SWEET – A Tool for WCET Flow Analysis (Extended Abstract)*, pages 482–485. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[35] Jane W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, 1st edition, 2000.

[36] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2):183–207, 1999.

[37] Claire Maiza-Burguière and Christine Rochange. History-based schemes and implicit path enumeration. In *Proceedings of the 6th International Workshop On Worst-Case Execution Time (WCET) Analysis*, july 2006.

[38] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. WCET (m) estimation in multi-core systems using single core equivalence. In *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*, pages 174–183. IEEE, 2015.

[39] Amine Marref and Guillem Bernat. *Predicated Worst-Case Execution-Time Analysis*, pages 134–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[40] *PIC32MX3XX/4XX Family Data Sheet*. Microchip Technology Inc, 5 2011. URL http://ww1.microchip.com/downloads/en/DeviceDoc/61143H.pdf. Accessed: 2017-02-13.

[41] Antoine Miné. A few graph-based relational numerical abstract domains. *Static Analysis*, pages 527–531, 2002.

[42] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[43] *SWEET manual*. Mälardalen University, Sweden, 6 2016. URL http://www.mrtc.mdh.se/projects/wcet/sweet/manual/SWEET_manual.pdf. Accessed: 2017-08-23.

[44] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[45] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[46] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *ACM SIGARCH Computer Architecture News*, 12(3):348–354, 1984.

[47] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 741–746. European Design and Automation Association, 2010.

[48] Rathijit Sen and YN Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Formal Methods and Models for Codesign, 2007. MEMOCODE 2007. 5th IEEE/ACM International Conference on*, pages 39–48. IEEE, 2007.

[49] Gagandeep Singh, Markus Püschel, and Martin Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 46–59. ACM, 2017.

[50] Stephan Thesing. *Safe and precise WCET determination by abstract interpretation of pipeline models*. Pirrot, 2004.

[51] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7 (3):36, 2008.

[52] Wei Zhang and Yan Jun. Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering*, 6(4):267–278, 2012.

TRITA TRITA-ICT-EX-2017:207