Fakultät für Elektrotechnik und Informationstechnik
Technische Universität München

Towards Source-Level Timing Analysis of Embedded Software Using Functional Verification Methods

Martin Becker, M.Sc.

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:
Prof. Dr. sc. techn. Andreas Herkersdorf

Prüfende der Dissertation:
1. Prof. Dr. sc. Samarjit Chakraborty
2. Prof. Dr. Marco Caccamo
3. Prof. Dr. Daniel Müller-Gritschneder

Die Dissertation wurde am 13.06.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.04.2020 angenommen.

Abstract

***

Formal functional verification of source code has become more prevalent in recent years, thanks to the increasing number of mature and efficient analysis tools becoming available. Developers regularly make use of them for bug-hunting, and produce software with fewer defects in less time. On the other hand, the temporal behavior of software is equally important, yet rarely analyzed formally, but typically determined through profiling and dynamic testing. Although methods for formal timing analysis exist, they are separated from functional verification, and difficult to use. Since the timing of a program is a product of the source code and the hardware it is running on – e.g., influenced by processor speed, caches and branch predictors – established methods of timing analysis take place at instruction level, where enough details are available for the analysis. During this process, users often have to provide instruction-level hints about the program, which is a tedious and error-prone process, and perhaps the reason why timing analysis is not as widely performed as functional verification.

This thesis investigates whether and how timing analysis can be performed at source code level, by leveraging mature analysis methods from general software engineering, and thus departs from the traditional instruction-level approach. Specifically, we focus on computing the worst-case execution time (WCET) of a program, which is required to prove and certify the timeliness of software in real-time systems, and used to prove its adherence to deadlines under all conditions. Not only would a source-level timing analysis be more accessible for users, but it promises other advantages, as well. As opposed to traditional instruction-level analysis, where semantic information is obfuscated or removed by the compiler, the source code carries all that information, making it easier to automatically track control and data flows. As a result, a source-level timing analysis should be able to infer more properties on the program with less user inputs, and to provide more precise timing estimates.

The obvious challenge towards source-level WCET analysis is that the temporal behavior of a program is only defined by the machine code of the program and the microarchitecture of the target. The source code, however, is only specifying functional behavior, lacking all of this information. Many aspects, e.g., the specific instructions being used, are only decided during compilation from source to binary, and can have a dramatic impact on the timing. As a consequence, we have to annotate the source with a timing model computed from the binary and target properties, which requires establishing a mapping from instructions to source code. Another challenge is posed by performance-enhancing features in processors, such as instruction caches or branch predictors. Such features should be supported soundly and with reasonable precision; it is however not obvious whether and how they can be modeled in the source code, since its control flow structure can deviate significantly from the control flow of the machine instructions.

Our approach to source-level WCET analysis is structured as follows. We start by evaluating current methods and tools for formal functional verification of source code, to identify their shortcomings and select a viable analysis method. Based on the findings, we have chosen Bounded Model Checking as primary analysis method, which enables maximally precise timing estimates with a high degree of automatism, alleviating the need for user

inputs. To address the back-annotation problem, we introduce an automated, compiler- and target-independent method for instruction-to-source mapping and back-annotation, which can tolerate compiler optimization. It is based on a careful combination of hierarchical flow partitioning to subdivide the mapping problem into smaller problems, and a dominator homomorphism and control dependency analysis to construct the mapping from incomplete debugging information. To address the known complexity issues of Model Checking, we propose source code transformations that are based on program slicing, loop acceleration and program abstraction, which keep the complexity of the models low. We further introduce a novel process of "timing debugging", that is, a debugger-based reconstruction and replay of the WCET path, which sets critical variables identified during static analysis to steer into the execution into the worst-case path. Last but not least, we propose a precise source-level model for instruction caches. To guarantee its soundness, flow differences between binary and source are reconsolidated with an implicit path enumeration technique, exploiting the support for non-determinism in the analyzers. The increased program complexity caused by these microarchitectural models is controlled by computing microarchitectural invariants at either source or binary level, which effectively prevent a complexity explosion.

Our source-level timing analysis has been verified against both cycle-accurate simulations and existing binary-level analyzers, for two different microarchitectures. In both cases our source-level analysis produced comparable and often better results than traditional binary-level analysis, especially under compiler optimization and in the presence of caches, and with less user input. For a simple microcontroller, we obtained WCET estimates which were 17% tighter than binary-level analysis, and on a processor with caches this number improved to even 56% tighter estimates. As a downside, the scalability of the proposed source-level approach is inferior to traditional approaches. Typical analysis times range between seconds and minutes in our experiments, lagging behind the faster traditional methods, which consistently terminate within seconds. However, we found several ways how this could be improved. Furthermore, we have only considered mono-processors with single-level instruction caches. Modeling more advanced processors is feasible and has been sketched, but is also left for future work. Overall, the results demonstrate that source-level timing analysis is a promising research direction with certain benefits, but also that traditional binary-level analysis cannot be entirely replaced. Instead, future research should focus on hybrid approaches, leveraging the best of both source-level and binary-level worlds.

In summary, the main contributions of this thesis are as follows. (1) We introduce an overall workflow for WCET analysis at source code level using Model Checking, which addresses the known scalability issues with source code transformations. (2) We propose a generic, compiler- and target-independent method to establish a sound and precise mapping between machine instructions and source code, which is used to back-annotate the instruction timing to the source code, and serves as link between source-level analysis and microarchitectural analysis. (3) We propose a novel source-level model for set-associative instruction caches and address its complexity issues by computing invariants. We further sketch models for data caches, branch predictors and other performance-enhancing features, and discuss their impact on the analysis. (4) We present a novel process for "timing debugging" which allows the user to examine the program's timing behavior in a debugger, while we automatically steer it into the WCET case. This method can be applied to counterexample reconstruction in general functional verification without any changes. (5) Lastly, and opposing a commonly held view, the experiments in this thesis demonstrate that Model Checking can indeed be applied for the WCET problem and scale well, albeit only when used intelligently.

Zusammenfassung (German Abstract)

***

Formale funktionale Verifikation von Software kommt zunehmend häufiger zur Anwendung, dank der wachsenden Zahl von ausgereiften und effizienten Analysewerkzeugen. Entwickler nutzen diese regelmäßig zur Fehlersuche, und produzieren robustere Software in kürzerer Zeit. Andererseits ist das zeitliche Verhalten von Software ebenso wichtig, wird jedoch selten formal analysiert, sondern typischerweise durch Profiling und dynamische Tests bewertet. Obwohl Methoden zur formalen Zeitanalyse existieren, werden sie getrennt von funktionaler Verifikation angewandt, und sind schwierig zu bedienen. Da das Zeitverhalten eines Programms ein Resultat von Quelltext und der ausführenden Hardware ist – z.B. beeinflusst durch Prozessorgeschwindigkeit, Caches und Sprungvorhersagen – arbeiten etablierte Methoden der Zeitanalyse auf Instruktionsebene, wo genügend Details für die Analyse bereit stehen. Während dieses Prozesses muss der Benutzer oftmals Hinweise auf Instruktionsebene übergeben, was eine mühsame und fehleranfällige Arbeit darstellt, und vielleicht der Grund ist, warum eine formale Analyse des Zeitverhaltens wenig verbreitet ist.

Die vorliegende Arbeit untersucht ob und mit welchen Mitteln eine Ausführungszeitanalyse auf Quelltextebene durchgeführt werden kann, unter Zuhilfenahme von ausgereiften Analysemethoden aus der allgemeinen Software-Entwicklung, und weicht damit vom traditionellen Ansatz der Ausführungszeitanalyse ab. Im Speziellen betrachten wir die Berechnung der längsten Ausführungszeit (WCET) eines Programms, welche benötigt wird um die Rechtzeitigkeit von Berechnungen in Echtzeitanwendungen unter allen Umständen nachzuweisen. Eine Zeitanalyse auf Quelltextebene wäre nicht nur leichter für den Benutzer zugänglich, sondern verspricht einige weitere Vorteile. Im Gegensatz zur traditionellen Analyse auf Instruktionsebene, bei der semantische Informationen durch den Compiler verschleiert oder entfernt werden, enthält der Quelltext noch alle diese Informationen, was die automatische Analyse von Kontroll- und Datenflüssen erleichtert. Infolgedessen sollte eine Zeitanalyse auf Quelltextebene in der Lage sein eine größere Anzahl von Programmeigenschaften mit weniger Benutzerhilfe abzuleiten, und zudem genauere Zeitschätzungen zu liefern.

Die offensichtliche Herausforderung für eine Zeitanalyse auf Quelltextebene besteht darin, dass das zeitliche Verhalten eines Programms erst durch den Maschinencode und die Prozessoreigenschaften bestimmt wird, und beides nicht im Quelltext ersichtlich ist. Zahlreiche Aspekte, unter anderem die genauen Instruktionen, werden erst bei der Kompilierung vom Quelltext zum Maschinencode entschieden, und können einen drastischen Einfluss auf das Zeitverhalten haben. Infolgedessen muss der Quelltext mit einem Zeitmodell versehen werden, welches aus dem Maschinencode und Prozessoreigenschaften abgeleitet wird, was wiederum eine Zuordnung von Instruktionen zum Quelltext notwendig macht. Eine weitere Herausforderung sind hierbei Prozessorfunktionen die dessen Leistungsfähigkeit erhöhen, beispielsweise Caches oder Sprungvorhersagen. Derartige Funktionen müssen mit angemessener Präzision unterstützt werden. Es ist jedoch nicht offensichtlich ob und wie diese im Quelltext darstellbar sind, da die Struktur signifikant vom Maschinencode abweichen kann.

Unser Ansatz für die Zeitanalyse auf Quelltextebene ist wie folgt strukturiert. Zunächst evaluieren wir aktuelle Analysemethoden und -werkzeuge für die funktionale Verifikation von Quelltexten, um deren Schwächen zu identifizieren und geeignete Methoden auszuwählen. Hierbei hat sich ergeben, dass Bounded Model Checking die bevorzugte Verifikationsmethode zur Zeitanalyse ist, da sie maximal präzise und automatisierbar ist, wodurch die Notwendigkeit von Benutzereingaben entfällt. Um die Zuordnung von Maschinencode

auf Quelltext zu ermöglichen, führen wir ein automatisiertes, Compiler- und prozessorunabhängiges Verfahren ein, welches auch Compiler-Optimierungen toleriert. Es basiert auf einer Kombination von hierarchischer Unterteilung des Kontrollflusses um das Zuordnungsproblem in kleinere Teile aufzuspalten, sowie auf einen Dominatorhomomorphismus und einer Abhängigkeitsanalyse zur Berechnung der Zuordnung basierend auf unvollständigen Debugging-Informationen. Um die bekannten Komplexitätsprobleme von Model Checking zu umgehen, stellen wir Quelltexttransformationen basierend auf Slicing, Schleifenbeschleunigung und Schleifenabstraktion vor, welche die Komplexität gering halten. Weiterhin stellen wir ein neuartiges Verfahren des Timing Debugging vor, welches eine Rekonstruktion und Wiedergabe des WCET-Pfads ermöglicht, währenddessen automatisch kritische Variablen gesetzt werden, sodass das Programm gezielt in den längsten Ausführungspfad gesteuert wird. Weiterhin schlagen wir ein präzises Quelltextmodell für Instruktions-Caches vor. Strukturdifferenzen zwischen Quelltext und Maschinencode werden unter Ausnutzung von nicht-deterministischen Bausteinen in den Analysewerkzeugen kodiert, speziell in der Form der impliziten Pfadaufzählungstechnik (IPET). Die durch die Prozessormodelle erhöhte Programmkomplexität wird durch Berechnung von Invarianten – entweder auf Instruktions- oder Quelltextebene – reduziert, wodurch eine Zustandsexplosion effektiv verhindert wird.

Die Ergebnisse unserer Quelltext-basierten Zeitanalyse wurden mit sowohl taktgenauen Simulationen, als auch mit bestehenden traditionellen Zeitanalysen verglichen, für zwei verschiedene Mikroarchitekturen. In beiden Fällen erreichten wir vergleichbare und oft bessere Ergebnisse als traditionelle Ansätze, insbesondere unter Compiler-Optimierung und auf Prozessoren mit Caches, und mit weniger Benutzereingaben. Für einen einfachen Mikrocontroller erhielten wir WCET-Schätzungen welche durchschnittlich 17% genauer als jene bestehender Methoden sind, und auf einem Prozessor mit Caches verbesserte sich diese Zahl auf 56%. Als nachteilig ist die Skalierbarkeit des vorgeschlagenen Ansatzes zu sehen, welche unterhalb der traditionellen Methoden liegt. Typische Analysezeiten liegen zwischen Sekunden und Minuten in unseren Experimenten, womit wir hinter den schnelleren traditionellen Methoden zurückbleiben. Weiterhin unterstützen wir bisher nur Einkernprozessoren mit einstufigen Instruktions-Caches. Lösungswege zur Modellierung komplexerer Prozessoren werden aufgezeigt, aber für zukünftige Arbeiten zurückgestellt. Insgesamt zeigt die vorliegene Arbeit auf, dass eine quelltext-basierte Ausführungszeitanalyse ein vielversprechender Ansatz mit einigen Vorteilen ist, aber zeitgleich, dass traditionelle, instruktions-basierte Zeitanalysen nicht vollständig ersetzt werden können. Zukünftige Untersuchungen sollten sich daher auf hybride Ansätze konzentrieren, um die Vorteile von beiden Seiten zu kombinieren.

Zusammenfassend sind die wichtigsten Beiträge dieser Arbeit wie folgt. (1) Wir präsentieren einen Workflow für die Analyse der maximalen Ausführungszeit auf Quelltextebene, unter Nutzung von Model Checking, und adressieren das bekannte Komplexitätsproblem mit Quelltexttransformationen. (2) Wir schlagen eine generische, Compiler- und prozessorunabhängige Methode vor um eine korrekte Zuordnung von Maschinenanweisungen auf den Quelltext herzustellen, mit deren Hilfe das Zeitverhalten in den Quelltext eingefügt wird, und als Bindeglied zwischen Quelltext- und Prozessoranalyse wirkt. (3) Wir schlagen ein neuartiges Quelltextmodell für mengenassoziative Instruktions-Caches vor und adressieren die damit entstehenden Komplexitätsprobleme. Weiterhin bewerten wir die Möglichkeiten und Auswirkungen der Modellierung von Daten-Caches, Sprungvorhersagen und anderen Prozessormechanismen welche dessen Leistungsfähigkeit erhöhen. (4) Wir stellen ein neuartiges Verfahren zum "Timing Debugging" vor, welches es dem Nutzer ermöglicht das Zeitverhalten des Programms in einem Debugger zu untersuchen, während es automatisch in den längsten Ausführungspfad gesteuert wird. (5) Schließlich, und entgegen einer weit verbreiteten Ansicht, zeigen unsere Experimente dass Model Checking sehr gut zur Ausführungszeitanalyse geeignet ist, jedoch nur wenn es gezielt angewendet wird.

Contents

***

Chapter 1

Introduction

***

Throughout the history of software engineering, one of the most addressed issues has always been the detection and fixing of defects, or colloquially called "bugs". They can cause unexpected results, program crashes or freezes, or even hide for years until they manifest in some undesired behavior. While some of them can be circumvented by selecting the right programming language, it is mainly the rising software complexity in conjunction with shortcomings of human developers which leads to oversight, and gives rise to software defects. It is not surprising then, that one of the grand challenges of software engineering since the 1960s [Hoa03] is to build a *verifying compiler*, a computer program that checks the correctness of any program that it produces.

Although this ideal has arguably not been attained, recent advances in prover technology bring us closer than ever before. Today we find many tools in practice which can automatically identify large classes of defects at the press of a button. A workshop focused on tools for software verification in 2009 featured already no less than 65 different tools and languages [vst09, Fil11], and a 2016 study of more than 168,000 open source projects found that more than 50% of projects already made use of static analysis tools [BBMZ16].

On the other hand, even a defect-free program can still become problematic if its temporal behavior is unpredictable, or if it has too long response times. A recent study [ATF09] has shown that timing is the number one property of interest among the non-functional aspects. Especially in real-time and embedded systems, timing analysis is an important step during their verification, to gain confidence on, or even prove and certify their timeliness. For instance, airbag deployment in a car, collision avoidance systems in aircraft, and control systems in spacecraft have to meet deadlines for ensuring vehicle and passenger safety. The need to formally analyze the temporal behavior of such time-critical systems has been the main motivator [PK89] to compute an upper bound of the execution time of programs, called the *Worst-Case Execution Time (WCET)*. This metric is subsequently used to prove analytically that a software can always meet its deadlines, no matter what the operating conditions are. The WCET problem therefore has been widely studied over the past three decades [WEE+08], resulting in a number of academic and commercial tools.

However, static timing analysis is currently separated from functional verification, and works on a level that is less familiar to most developers. Unlike functional behavior, the timing of a program is a product of the processor architecture it is running on, for example, its

clock speed and cache size, as well as the program structure. On the hardware side, we need to determine which machine instructions have been chosen by the compiler to implement the software, how they access memories and thus modify cache states, and finally what the timing effects are. On the software side, control statements, such as loops and conditionals, need to be analyzed to find the longest path, which implies that much information about the control flow has to be available. Moreover, the analysis has to consider the interaction of these aspects, since not necessarily the longest path causes the worst processor timing, and vice versa. Consequently, timing analysis traditionally takes place at instruction level, where both the behavior of the compiled software and the processor can be deduced.

Unfortunately, the process of WCET estimation cannot be fully automated. Users are typically asked to provide hints to the analyzer [WG14, LES+13] during the analysis, caused by two fundamental limitations. First, a WCET analyzer is essentially a program analyzing another program, in an attempt to decide whether certain properties hold true. This is an instance of the famous *Decision Problem*, which was proven to be generally undecidable in a finite amount of time by Gödel, Church and Turing [Göd31, Tur38]. Therefore, no analysis method can decide all properties on all programs, and occasionally must rely on user inputs. Second, compilers translate a high-level description into a more low-level one, typically C source code to machine instructions. During this process, information is not only added (e.g., in which registers a variable is being stored), but also removed. Semantic properties, such as type and range information of variables, have little meaning for the processor, and are therefore obfuscated or "compiled away". Such information can however be beneficial or even necessary for WCET analysis, and thus needs to be reconstructed [BCC+16, GESL06, KPP10]. This again may require user inputs to specify the lost information, or to constrain program behaviors to be considered. Moreover, compilers can apply various optimization techniques, which can lead to vast differences between source code and machine instructions, such that the task of providing hints to the analyzer can quickly become tedious and overwhelming, and easily lead to human error [LES+13, AHQ+15]. Static timing analysis, although being a mature research domain, therefore fights an ongoing battle against precision, correctness and the degree of automation.

This thesis evaluates a radically different approach to static timing analysis, namely to shift the analysis from instruction level to *source code level*, where programmers have an intuitive interface to the analysis, and where state-of-the-art tools from functional verification can be leveraged to get precise timing estimates without the need for error-prone, manual user inputs. Whereas such a source-level approach was indeed pursued in the early years of WCET analysis [PK89, PS91], increasingly complex processors and optimizing compilers made it difficult to predict the timing behavior from the source code, and source-level verification tools have not been mature at that time to compensate for these difficulties. The source-level approach was thus quickly dropped in favor of an instruction-level analysis. As a consequence, the price which had been paid ever since, is that of a harder analysis, both for the tools that analyze a program's behavior, and for the users, which have to interact with the tools. There have been follow-ups on this idea over the years, but none of them addressed all challenges sufficiently. This thesis is the most comprehensive work describing an approach to source-level timing analysis which is *practical*, *scalable* and works for *modern processors*.

Specifically, we propose to shift the entire analysis to source code level and use *Model Checking* as primary analysis method, which is a formal method that can be automated and is precise, but has been deemed as too computationally expensive for WCET analysis [Wil04]. We show that it pairs well with the source-level approach, and, in fact, scales well if applied

intelligently. Towards this, we introduce new source-level models for microarchitectural components, and explain how these models can be computed automatically even in the presence of compiler optimization. This novel approach to WCET analysis results in often maximally precise timing estimates, with little to no manual user inputs, thereby reducing the impact of human error. We furthermore introduce a novel kind of *timing debugging*, which enables a user to examine the timing behavior of a program in a familiar, off-the-shelf debugger environment, where she can step through the program, set breakpoints and retrace the timing behavior. To enable this approach, we exploit modern tooling which did not exist many years back, when WCET analysis was established as we know it today, and we extend existing work in the domains of static timing analysis, virtual prototyping and functional verification with methods for timing debugging, automatic back-annotation, and source-level processor models. Our experiments show up to 260% improvements over traditional WCET analysis approaches, suggesting that the traditional approach is not necessarily the best one any more. With its precise results, higher automation and the user-friendly source code environment, this work does not only challenge the state-of-the-art approach to WCET analysis, but is also a step towards bringing static timing analysis to a broader application.

The rest of this chapter is organized as follows. The next section describes in detail the traditional, quasi-standard approach to WCET analysis, including its shortcomings. Section 1.2 introduces the class of functional verification tools that we considered in this work, since not all of them are suitable to compute safe estimates. Section 1.3 gives a preview of how source-level timing analysis looks like, and what kind of benefits it brings over traditional methods, as well as the challenges towards such an analysis. Finally, we conclude this chapter with a summary of technical contributions, and the list of publications contained herein.

1.1 Worst-Case Execution Time Analysis

The *Worst-Case Execution Time* (WCET) of a (sub)program P is the longest time it takes to terminate, considering all possible inputs and control flows that might occur, but excluding any waiting times caused by sleep states or interruption by other processes. In real-time systems, this estimate is subsequently used as an input for *schedulability analysis*, which then models the influence of other processes and computes an upper bound of the *reaction* or *response time* of P. The response time, finally, should be shorter than any deadline imposed on P. For example, the deadline for P could be given by the maximum time that is permissible to detect a car crash and activate the airbags. Consequently, the WCET estimate is a vital metric for real-time systems, and thus needs to be *safe*, i.e., never smaller than what can be observed when executing P, and *tight*, i.e., as close as possible to the observable value.

Figure 1.1 illustrates the basic terminology of WCET analysis. It depicts a hypothetical probability distribution of a program's execution time. The range of this distribution depends on the program itself, its inputs, processor states, and perhaps even on sources of entropy. The ideal is to compute the *actual WCET* of the program, which terminates the tail of the distribution. In principle this could be achieved by running the program and taking measurements, where eventually the largest observed value should approach the actual WCET. However, in analogy to finding defects by dynamic testing, the actual WCET can be unlikely to occur, which might render such a measurement-based strategy impractical. This is further aggravated by the fact that in practice neither the shape of the timing distribution is known, nor the inputs or processor states that cause the actual WCET.

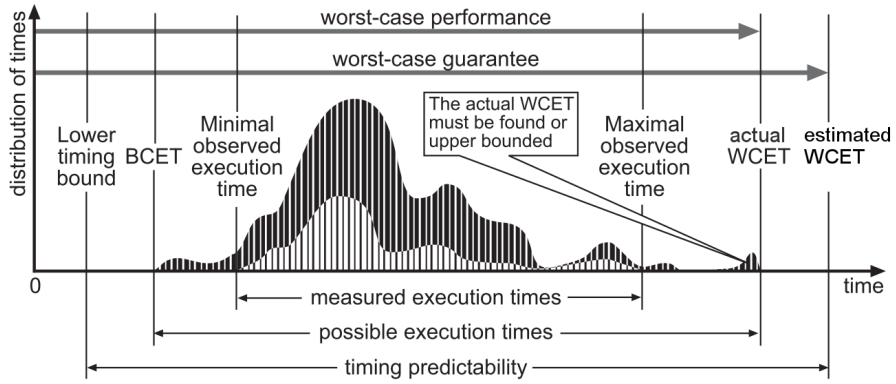


Figure 1.1: The worst-case execution time, adapted from [WEE+08].

In the most general case, the actual WCET is undecidable, if only for unknown loop bounds or variance in processor timing, and hence only an upper bound of the WCET can be determined through an automated analysis, which is the *WCET estimate*. Therefore, in a practical setting, the problem reduces to computing the *tightest* safe upper bound that can be deduced in face of the predictability of program, inputs and processor, and in particular using a technique that scales well with program size and complexity.

Control Flow Graph and Basic Blocks. Computing a WCET estimate entails finding a path through the program. Towards this, each function in the program – source or binary – is represented by a Control Flow Graph (CFG). This directed graph G is defined by the tuple

$$G := (V, E, I, F) \tag{1.1}$$

where V is the set of nodes, $E = V \times V$ the set of edges between them, I is the set of entry nodes and F the set of exit nodes. Without loss of generality, we assume that $||I|| = ||F|| = 1$.

We use the short notation $e_{i,j}$ to denote an edge in E that goes from node v_i to node v_j. We further write $v \succ u$ (or $u \prec v$) to denote that v is a successor of u, possibly via intermediate nodes. Furthermore, throughout this thesis, we use subscript "s" for elements in the source CFG, and "b" for those in the binary/instruction CFG.

The nodes V represent *Basic Blocks (BBs)*. These are maximal sequences of instructions or statements with at most one entry and one exit point [LMW95]. Consequently, basic blocks are terminated by branches or indirect jumps, which in turn are represented by the edges E. Therefore, we will often use the terms *node* and *basic blocks* interchangeably. Last but not least, we assume BBs are also terminated at function calls and returns, so that the callees can be analyzed separately.

1.1.1 Evolution of WCET Analysis

While there are several approaches to estimate the WCET, we only consider *static deterministic timing analysis* here, as opposed to probabilistic or measurement-based approaches, since it is the only one capable of providing a safe upper bound [AHQ+15]. Early methods were indeed based on the source code, specifically on the Abstract Syntax Tree (AST) of programs. Each source statement or expression was weighted according to a timing scheme, and the WCET was computed by a bottom-up summation of the AST elements, while choosing local maxima for alternative branches [PS91, PK89]. The timing schemata were obtained

using measurements, or by predicting which instructions would be used by the compiler. These approaches were therefore agnostic to data dependencies and data-dependent flow constraints, and also not guaranteed to be safe. Furthermore, the authors already recognized that variable instruction timing and optimizing compilers make it hard to anticipate the performance only based on the source code. Thereafter, source-level analysis was not actively pursued for several years to come.

The paper of Li, Malik and Wolfe from 1995 [LMW95] can be seen as the first breakthrough in WCET analysis. They proposed to compute the WCET at instruction level, since it precisely represents the program, using a method that considers all possible paths in the program, yet without requiring a costly explicit enumeration. Specifically, they proposed to cast the WCET problem to an Integer Linear Programming (ILP) problem as follows. Consider a subprogram given as a CFG, with the nodes $v_i \in V$ representing BBs. Furthermore, assume that the execution time for the BBs are known, and represented by c_i, in units of processor cycles. Then the execution time of the program is

$$\sum_{i=1}^{|V_b|} f(v_i)c_i, \tag{1.2}$$

with $f(v_i)$ denoting the execution count of BB v_i. The WCET then can be expressed as the attainable maximum by tuning the execution counts, i.e.,

$$WCET := \max_{f} \sum_{i=1}^{|V_b|} f(v_i)c_i. \tag{1.3}$$

However, further constraints need to be given, since otherwise we could choose arbitrary execution counts $f(v_i)$ which may not be realizable by the actual program. Li, Malik and Wolfe introduced the *Implicit Path Enumeration Technique (IPET)*, which generates constraints expressing a flow conservation in the CFG, that is

$$\forall v_i \in V.\ \sum \{f(e_{h,i}) \mid e_{h,i} \in E\} = f(v_i) = \sum \{f(e_{i,k}) \mid e_{i,k} \in E\}, \tag{1.4}$$

where $f(e)$ denotes the execution count of edge e in the CFG. In other words, the number of times each BB is entered must equal the number of times it is executed, which in turn must equal the number of times it is left. These so-called *structural constraints* given by Eq. (1.4) allow precisely those paths to be considered in Eq. (1.3) that are structurally feasible in the CFG, yet without requiring an explicit enumeration of all feasible paths. To avoid an unbounded result in the presence of loops, further *logical constraints* must be provided, which are primarily loop bounds expressed as constraints on the execution count of their headers, or back-edges. Equation (1.3) together with these constraints represents an ILP problem, for which efficient solvers exist. This IPET/ILP method therefore was the first one to precisely formulate the WCET problem, and was practically solvable.

The next concern had been caches. In the same paper [LMW95], Li, Malik and Wolfe proposed a way to integrate cache analysis into the same ILP formulation. In summary, they subdivided each BB v_i into n_i smaller chunks $v_{i,j}$ according to their cache line separation ("line blocks") and added another execution count variable to the equation which allowed

for each block to be either a miss or a hit as

$$WCET := \max_{f^{\text{hit}}, f^{\text{miss}}} \sum_{i=1}^{|V_b|} \sum_{j=1}^{n_i} \left(c_i^{\text{miss}} f^{\text{miss}}(v_{i,j}) + c_i^{\text{hit}} f^{\text{hit}}(v_{i,j}) \right), \tag{1.5}$$

$$\text{s.t. } f(v_{i,j}) = f^{\text{hit}}(v_{i,j}) + f^{\text{miss}}(v_{i,j}). \tag{1.6}$$

Finally, they statically computed more constraints on the hit/miss interdependence between the line blocks. The resulting ILP problem became however quickly too complex for the solvers, such that only small programs could be analyzed this way [Wil04].

An alternative method was proposed later by Ferdinand and Wilhelm [FW99]. They introduced a cache analysis based on the analysis framework *Abstract Interpretation*, which we introduce in detail in Chapter 3. This framework computes invariants at each program location through a fixed-point iteration, such that all paths reaching the respective locations are summarized in one abstract property. In their cache analysis, these invariants are the possible cache states at each program location, which therefore yield constant values for $f^{\text{miss}}(v_i)$ and $f^{\text{hit}}(v_i)$. Subsequently, the ILP formulation can be reduced back to Eq. (1.3), since the execution time for each BB is again given by single value. This cache analysis has a sufficiently low complexity to be used on large programs and in industrial settings [Wil04].

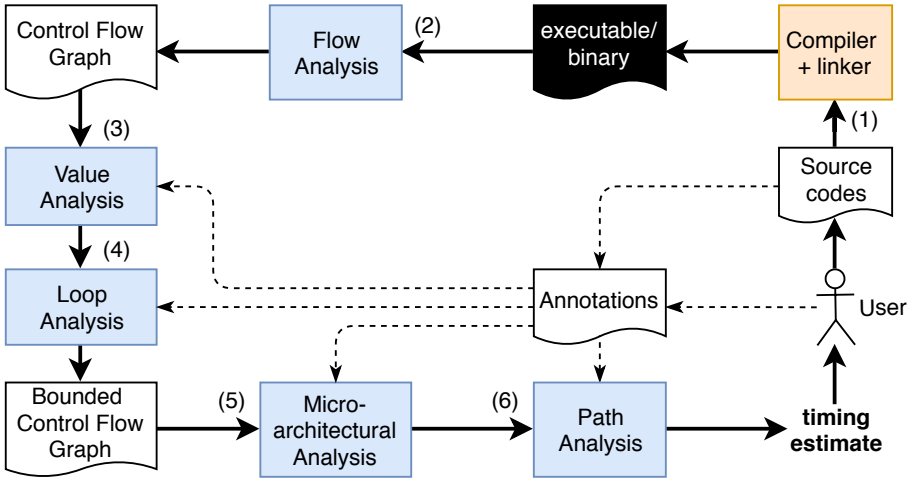


Figure 1.2: Traditional workflow for static timing analysis.

State of the Art. Many extensions have been developed over the years [WEE+08] and are discussed later, but the basic approach to WCET analysis has remained largely the same. Today, the traditional and quasi-standard workflow [WEE+08, WG14] to static deterministic timing analysis is depicted in Figure 1.2. It shows the workflow implemented by one of the leading commercial tools, *aiT*, currently referring to itself as "the industry standard for static timing analysis" [Gmb19]. The entire flow starts and ends with the user, who should be familiar with the source code of the program.

(**1**). **Compilation:** Cross-compile program P for the target processor. The source code of P is translated to machine instructions I, applying various optimizations.

(**2**). **Flow Analysis:** Analyze I to discover all possible control flows in the binary. This includes finding all potential branches in I and storing them in a CFG G, including their branch conditions and other meta-information.

(**3**). **Value Analysis:** Calculate possible ranges for operand values in I, to resolve indirect jumps and classify memory accesses into different memory regions (e.g., slow DRAM vs. fast core-coupled memory).
(**4**). **Loop Analysis:** Bound the control flow of G, that is, identify loops and compute their maximum execution counts based on branch conditions, and annotate the nodes and edges in G with those execution counts.
(**5**). **Microarchitectural Analysis:** Predict the timing effects of caches, pipelines and other architecture-dependent constructs, based on memory mapping and the paths in G. Annotate nodes and edges in G with instruction timing considering these features.
(**6**). **Path Analysis:** The discovered control flow graph G with its microarchitectural timing annotations and the computed loop bounds are analyzed together to find the longest path through the program, that is, the WCET.

Steps (2) through (5) are sometimes referred to as *low-level analysis*, and step (6) as *high-level analysis*. The employed methods for the shown steps typically involve a combination of Abstract Interpretation and Linear Programming [Wil04]: Value and loop analysis typically use Abstract Interpretation to deduce variable contents and iteration counts that may influence control flow or timing. Microarchitectural analysis, as well, typically builds on Abstract Interpretation to determine cache and pipeline effects. Finally, path analysis is often done by translating the annotated control flow graph into an ILP problem and solve for the WCET.

This traditional workflow has proven itself in many settings, among others, in asserting the real-time guarantees for Airbus' commercial passenger aircrafts [SPH+05] such as the Airbus A380, one of the most complex commercial aircrafts ever built.

Shortcomings. Even the best timing analysis tools require user annotations, as explained earlier. The main concern are loop structures, for which additional constraints must be computed, since Eq. (1.4) is otherwise unbounded. WCET analyzers are therefore trying to identify loop-controlling variables, and to automatically derive loop bounds [HS02, WEE+08]. Additionally, as evident from Eq. (1.4), path analysis is by default blind to data dependencies. The analysis might consider paths that are logically infeasible, for example, a path through the body of two sequential but logically mutual exclusive if-statements, and as a result produce an overestimation. WCET analyzers which aim to produce tight estimates must therefore run additional analyses to identify and exclude infeasible paths.

Due to these requirements, it is common that users are asked to provide manual annotations (sometimes also referred to as assertions or flow facts) to the analyzer, which can be used by all analysis steps to bound and tighten the estimate. As it can be seen in Fig. 1.2, these annotations usually refer to the program after compilation, hence users have to inspect machine instructions and understand compiler optimization to come up with such annotations. It is considered a challenge in itself to develop annotation languages that are expressive enough to capture the required knowledge [KKP+07, MRS+16].

The shortcomings of traditional WCET can therefore be summarized as follows:

- Existing approaches predominantly implement their analyses at machine code level, where the high-level information from the original program is hard to extract. Variables are distributed over multiple registers, type information is lost, loops and conditional statements can be implemented in different ways, and indirect addressing can make it close to impossible to track data flows and function calls. As a consequence, overapproximations have to be used, which usually result in pessimistic estimates [AHQ+15, WG14].

- Without further inputs or analyses, path analysis is agnostic to semantic flow constraints, which may thus consider infeasible paths, and consequently lead to an overestimation of the WCET. Another source for infeasible paths are overapproximations that bound the control flow, contributing further to less tight estimates [AHQ+15, MRP+17].

- User annotations, such as loop bounds, have to be provided [WEE+08, SPH+05], but are hard to obtain; they influence the tightness and may even refute the soundness of the WCET estimate. Providing too large bounds leads to a large overestimation, and too small bounds may yield an unsafe estimate. As a result, providing safe and tight bounds has become a research field on its own with a wide range of different approaches, e.g., using Abstract Execution [GESL06], refinement invariants [GJK09] and pattern matching [HSR+00]. Furthermore, it is a known and difficult problem to specify good constraints, requiring elaborate description languages [MRS+16].

- Existing tools are specific to a chosen target, a compiler, and its optimization settings. As little as an unexpected compilation pattern may disturb the analysis, and require further user interaction or even make the tool bail out. Some tools require optimization turned off to work [RMPV+19].

- Finally, practitioners are facing yet another challenge with today's analysis tools. Once the WCET of an application has been computed, the output offers little to no explanation of how the WCET has evolved, even less of how it can be influenced through changes in the program. The output of the ILP solver enables only the reconstruction of an abstract path, induced by the execution counts $f(v_i)$ and $f(e)$. However, neither does this path necessarily exist, nor does it contain any details to comprehend why this path was taken. Although there have been some attempts to visualize the results, e.g., [FHLS+08, FBvHS16], these results are on an abstract level and do not enable "debugging" of timing issues. In particular, this feedback should be at source level, since developers are more familiar with source than with machine instructions.

In conclusion, many of the difficulties in current WCET analysis approaches come from the disparity between the source code, which developers are familiar with and what contains some essential information for the analysis, and the machine instructions and processor details, which are required for low-level analysis but not easily understood.

1.2 Functional Verification of Source Code

Modern software development uses numerous tools to improve software quality [BBMZ16]. Many of these tools have been conceived for "bughunting", and made great progress in the past decades in being able to detect software problems reliably and efficiently, which is part of the motivation for this thesis.

Terminology around Bugs. Unexpected and unwanted behavior in software is colloquially referred to as "bug". This term is, however, too imprecise, since it cannot distinguish between cause, event chain, and finally the location where faulty behavior becomes visible. We therefore adopt the terms from Zeller [Zel09], who proposed the following definitions. A *defect* is a specific location in the program that implements functionality in a way that causes an *error*. An error, in turn, is any deviation of the program state w.r.t. its expected behavior or specification. Along with the control and data flow, errors may *propagate*. Finally, at some

point the error becomes visible, when the software exhibits some *failure* in its outputs, e.g., a crash, or wrong functionality. Normally only the failures are visible for the user. Furthermore, the propagation of errors may also cease, such that not every error (and thus defect) becomes visible, creating a dormant failure. Once failure and error have been identified, the event chain can be walked backwards, in an attempt to identify the original defect.

The methods considered in this thesis can only predict failure (program crashes) and errors (if the user provides a specification, or implicitly to avoid undefined behavior). In general they cannot identify the original defect unambiguously, since different parts of a program may collaborate in unwanted behavior. It can be an arbitrary decision which part is to blame and thus contains the defect. Therefore, whenever we refer to something as defect, we mean any possible cause, usually the one directly preceding the error or failure.

All verification tools can be broadly classified into *static* or *dynamic* verification tools.

1.2.1 Dynamic Verification

Dynamic verification requires executing the program under analysis, while observing its behavior. In the simplest case, this can be some form of monitoring during its operational use, but also a number of test runs specifically crafted to activate certain parts of the program. Examples of software tools for dynamic verification are all sorts of testing frameworks, but also tools that monitor program execution by intercepting and analyzing its activities, like the dynamic interpreter *Valgrind* [uC19a].

It is hard to give any guarantees with dynamic verification methods, unless the tools are used in such a way that all program points relevant for a defect have been reached. For example, consider the program shown in Listing 1. This program contains 2^{308} different paths, and therefore exhibits a computational complexity that is unsolvable with all available processing resources on Earth [Kli91], if we would try to cover all of its paths naively.

```
1 for (i=0; i<308; i++) {
2   if (b) {
3     bar();
4   } else {
5     baz();
6   }
7   b = foo();
8 }
```

Listing 1: A transcomputational program for naive dynamic verification.

Specifically for testing, software engineering has therefore developed more intelligent methods and coverage criteria, to enable a more intelligent exploration of program states, and to give at least some guarantees. Towards that, it is necessary to write a test *harness*, which starts the program, brings it into a defined state, runs it with specific inputs, compares the result with the expected one, and generates a report. In this sense, testing can only detect failures and errors, but not the underlying defects. Therefore, testing requires a substantial effort, and further requires the developer to specify not only the expected behavior, but also to specify *how* the behavior shall be tested. Methods like anti-random testing, combinatorial testing and model-based testing have evolved to address some challenges, but many questions are still unanswered [ABC+13]. This is perhaps most succinctly summarized by Edsger Dijkstra, in the words "Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence" during an ACM Turing Lecture in 1972.

Since this work focuses on static timing analysis and guarantees for the WCET estimate, dynamic verification is not considered.

1.2.2 Static Verification

Static verification tools reason about a program's properties without executing it, often taking into account multiple, if not all, execution paths. Unlike with dynamic verification, the developer does not have to specify how behavior shall be tested, but only provide a specification of the expected behavior, so that errors can be identified automatically. Furthermore, these tools can also reveal some underlying defects, such as incorrect implementations that lead to numeric overflows. Static methods do not require a test harness, a target to execute the program, or the operation in its natural environmental conditions, nor on-target mechanisms to log test results, which makes them especially amenable to use in embedded systems.

The general workflow in static verification is as follows. The source code of the program is fed to an automatic analyzer, which identifies potentially bad operations in the program, e.g., multiplications that may produce an overflow and divisions by zero, then generates verification conditions or proof obligations, and finally applies some reasoning to evaluate whether such operations can indeed be driven with inputs leading to the undesired behavior. If yes, then some tools provide a *counterexample* or *witness*, which demonstrates the faulty behavior to the developer. If not, then this usually implies that the program under analysis, for all possible inputs, will never exhibit erroneous behavior. Properties specified by the user, e.g., assertions on variable values, are often supported in a similar way.

We can further classify such tools into two categories:

1. **Heuristic checkers** are tools that can verify only some properties of a program, and often only in an *ad-hoc* fashion. That is, even if the tool checks for a property, it may only spend a limited amount of effort to evaluate this property under all execution traces, and thus miss defects. Examples for such programs are *lint* and *cppcheck*.

2. **Formal verification** tools build a model of the software, and then use a mathematical system to reason on the model and evaluate specific properties. Such methods can be sound or unsound (defined in Section 3.2), but they consider all possible execution traces and are therefore capable of providing certain guarantees. Such formal verification can be equivalent to exhaustive testing (yet scales better), and thereby can replace large parts of testing or manual inspection, while giving guarantees. Examples of such tools are *Astrée*, *Frama-C*, *cbmc* and *Polyspace*.

This latter class of tools is what this thesis is concerned with, since the right static formal verification method and tool can be leveraged to compute safe estimates of the WCET. In this thesis the considered tools are *Frama-C* [CKK+12], which implements a host of different analyses to prove, inter alia, the absence of overflows, out-of-bounds array access and compliance to first-order logic specifications via theorem proving, *cbmc* [CKL04], a tool for verifying a similar set of properties on C code using Model Checking, and *gnatprove* and its surrounding tools [HMWC15], a verifier for Ada programs that builds on contracts. All these tools have demonstrated their practicability in industrial settings, are continuously improved in their capabilities, scalability and user feedback, and helped developers in creating high-integrity software [KMMS16, BC16, BK11].

1.3 Combining WCET Analysis and Functional Verification

The goal of this work is to investigate whether WCET analysis can be fully shifted to source code level, where programmers are most comfortable, and where recent advances in functional verification tools can be leveraged. This shift would therefore be beneficial for tool developers (information is not obfuscated and precise analyses are possible) and users (annotations, if required, are provided in the well-known source code, and the result of the analysis is directly visualized in the source). In particular, an ideal workflow for source-level analysis would be as follows – see also Fig. 1.3:

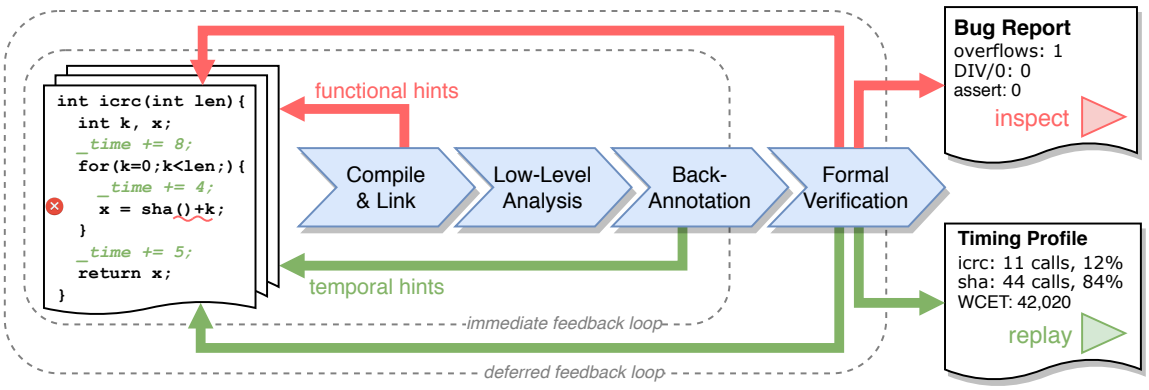


Figure 1.3: Workflow of combined timing analysis and functional verification.

(1). **Write Program.** A developer writes a program, which naturally happens incrementally. This following steps are therefore repeated numerous times during the development of an application.

(2). **Compile & Link.** As usual, the compiler is used frequently to build the program during the implementation of new parts, which detects simple syntactic and semantic errors, and highlights them in the source code. These *functional hints* are thus given immediately during development.

(3). **Low-Level Analysis.** Directly triggered by compilation, a low-level analysis on the binary is performed, which analyzes functional aspects (e.g., stack heights), but also microarchitectural events for timing properties. These analyses are supposed to be quick, such that the results can be visualized in the source code of the program, similarly to compiler errors.

(4). **Back-Annotation.** The results of the low-level analysis are visualized in the source code, such that they provide immediate feedback to the programmer, together with the compiler warnings and errors. Specifically, timing could be visualized in the source code as a dedicated variable `_time`, as read-only overlay in the editor. Note that no path analysis has taken place, yet. Therefore, this feedback is still immediate and local.

(5). **Formal Verification.** Using established source-level analysis tools, the developers occasionally run static analyses to ensure the absence of errors and defects. The preceding back-annotation of timing allows these tools to reason about temporal aspects as well, seamlessly integrating functional and temporal verification. Furthermore, in analogy to bug reports generated by these tools, the temporal aspects could be presented as a timing profile and a WCET value which can be replayed interactively in the source, such that developers can trace the decisions that lead to the worst timing behavior.

This source-level approach to timing analysis fully integrates with the increasingly used functional verification tools, but also promotes timing to be no longer an afterthought of software development. The immediate feedback helps to steer the development such that developers become aware of time, and the deferred feedback – perhaps running on nightly builds as part of continuous integration workflows – provides detailed feedback in the familiar form of source-level analyzers. Similar ideas for compile-time feedback of temporal aspects have been proposed before [SEE01, HSK+12], yet they only visualize the timing in the source code. This work goes beyond that, proposing to leverage such feedback for a full source-level timing analysis.

Moreover, both functional and temporal verification inevitably may require some user annotations. Performing functional verification in parallel to software development has already been shown to be highly effective, as it captures the knowledge of the developer early, discourages committing faulty code, and leads to better software quality in less time [CS14]. By bringing both analyses together in the same source-level framework, developers simultaneously aid both analyses with their annotations. Timing analysis can therefore benefit from functional verification and vice versa.

Last but not least, a source-level timing analysis also enables time budgeting and continuous tracking thereof. Real-time constraints could therefore be kept under observation during the entire development cycle, allowing to react early if budgets need to be redistributed. This should be of special interest for the development of real-time programs, where the WCET is mostly an afterthought, and hard to correct once the software is implementation is complete.

1.3.1 Challenges Towards Source-Level Timing Analysis

To carry out WCET analysis at source level, the following major challenges need to be addressed.

1. **Mapping between Source and Binary.** The most prominent obstacle to source-level analysis is that the source code in itself does not fully specify the behavior of the implementation. For example, compilers may perform high-level and target-specific optimizations, which can reduce the number of instructions, type of instructions, and their ordering. As a consequence, the major challenge is to back-annotate timing-relevant properties from the binary to the source code, which requires us to establish a safe mapping between instructions and source. While some approaches exist in Virtual Prototyping (e.g., [LMS12]) and in early work of WCET analysis [PS91], none of them are guaranteed to be safe.

2. **Lacking Source-Level Processor Models.** To analyze the timing of programs on modern processors, the effects of performance-enhancing features must be modeled, for example, instruction caches and branch predictors. Overapproximating such features can lead to an overestimation of several orders of magnitude, which would be inacceptable [WEE+08]. It is however not obvious whether and how such features can be modeled in the source code, since its control flow can deviate significantly from the control flow of the machine instructions. At a higher level, the question is which kinds of processors can be modeled at all, and with what precision. Besides approximate approaches in Virtual Prototyping [BEG+15], we are not aware of any existing work on safe and precise models. Therefore, new source-level processor models must be developed.

3. **Keeping Precision.** The product of the mapping imprecision and processor models is likely to create overestimation in a source-level analysis, which is not directly comparable to traditional approaches. The challenge is to keep these two precise enough to gain an advantage over binary-level analysis.

4. **Controlling Computational Effort.** More precision usually implies more computational effort. Therefore, even if source level analysis turns out to be more precise, the scalability of the approach must be good enough to support the analysis of reasonably-sized programs. Approximations and abstractions could be applied, but are detrimental to analysis precision. Consequently, another challenge is how to balance these two aspects such that, overall, source-level analysis can be competitive to binary-level analysis.

5. **WCET Feedback.** To allow developers a replay of the WCET path, the analysis must be precise enough to contain decision variables that explain control decisions, and further also compute the inputs that lead to the WCET. However, both of these challenges have currently no practical solutions. An ILP-based path analysis is insufficient to reconstruct decision variables since it only generates an abstract path agnostic to variables, and computing the inputs of the WCET path has therefore never been done before, beyond computationally expensive test-based approaches without guarantees (e.g., [EFGA09]).

1.4 Contributions

This thesis makes the following technical contributions.

Evaluation of Current Methods and Tools for Functional Verification. We evaluated recent tools for static functional verification towards their applicability for source-level timing analysis, where we have identified their weaknesses, threats to soundness and usability. Specifically, we have evaluated Model Checking of C code using the tool *cbmc* to verify the functional correctness of an in-house parachute system for drones; we have evaluated Abstract Interpretation using the tool *Frama-C* on a series of small programs; finally, we have evaluated Deductive Verification on an in-house flight controller written in Ada, using the tool *gnatprove*. The three case studies on these three fundamentally different verification techniques have shown that they all share a common weakness with loops, which either cause long analysis times or imprecise results. Furthermore, all tools required a careful setup to model the behavior on the target processor correctly. We have proposed code design rules and code transformations, especially for Model Checking, which are beneficial for the analysis. Finally, we found that the most suitable method and tool for an automated and precise timing analysis is Bounded Model Checking.

Source-level WCET analysis based on Model Checking. We propose an overall approach to source-level WCET analysis using Bounded Model Checking, which, unlike argued in literature, is not prohibitive due to complexity issues. Towards this, we have developed further source transformations that significantly reduce the complexity issue encountered during the earlier case studies. These transformations are based on *Program Slicing*, *Loop Acceleration* and *Loop Abstraction*. We describe how to set up the analysis context properly, such that the behavior on the target processor is modeled correctly, to avoid the prediction errors initially encountered during the first case study. We have conducted experiments on an 8-bit microprocessor, in which we have consistently outperformed the formerly commercial WCET analyzer *Bound-T* in precision, and in one occasion also in analysis time, on a very

long program. This demonstrates that Model Checking is a viable, if not preferable approach to WCET analysis, which is enabled by our shift from binary- to source level, as well as by the proposed methods to reduce model complexity.

Timing Debugging. We introduce a novel kind of "timing debugging", which extends the ordinary debugging experience with explicit access to execution time. The developer can interactively set breakpoints, inspect any variable and so on, to comprehend how the WCET is produced. This technique is based on the counterexample from the model checker, which returns an incomplete trace of the WCET. We complete this trace by loading the program in a standard debugger attached to either the real target or a simulator, where we make use of hardware watchpoints and breakpoints to inject decision variables into the running program according to the counterexample. As an effect, the program is efficiently steered into the worst case, and can be stopped and inspected at any point in time. As a side product, we generate a timing profile similar to the output from general-purpose profilers, which however represents the worst-case profile, and can be used for time budgeting.

Generic Instruction-to-Source Mapping. We propose a fully automatic method to trace back instructions to source statements which works with mainstream compilers and in the presence of moderate compiler optimization. It adopts and combines methods from the Virtual Prototyping domain, but makes corrections to ensure a complete and safe mapping, as required for WCET analysis. Towards this end, source- and binary CFGs are first hierarchically decomposed according to their loop nesting structure, then matched pairwise, and subsequently pairwise given to a mapper. The mapper uses debugging information to compute control dependencies on both CFGs, computes a dominator-homomorphic discriminator mapping to disambiguate BBs, and then matches the nodes in both graphs by their control dependency and discriminator map. Finally, unmatched nodes caused by missing debugging information are lumped into their dominators, such that timing is lost. This mapping can subsequently be used for source-level WCET analysis to attribute instruction timing to source statements. We have compared this automatic mapping to a manually crafted one under different compiler optimization levels. The results show that the generic mapping is often close to the manually crafted mapping, and that the source-level WCET computed on this generic mapping is often tighter than the estimates of traditional WCET analysis, especially under compiler optimization.

Source-Level Processor Models. We introduce a novel source-level model for instruction caches, which enables to make the caching behavior visible for source-level analyzers, with a maximal precision w.r.t. a given mapping. The model soundly captures the timing effects incurred by instruction caches, by overcoming flow differences between source and binary with the help of nondeterminism. We encode all unmatched binary paths similarly to the IPET technique, such that they can be are considered in the source model despite flow differences, which further is an improvement over the lumping method described earlier. We conducted experiments on an ARM-like Reduced Instruction Set Computer (RISC) processor with two-way instruction caches, during which we have evaluated two methods to reduce the complexity of the resulting model back to the same order of complexity as for simple 8-bit microprocessors. First, we computed microarchitectural invariants at binary level, which resulted in source-level estimates slightly better than traditional WCET analysis. Second, we proposed a way to compute equivalent invariants from the full source model using Frama-C, together with a new way to compute scope-sensitive first-miss invariants. These source-level

invariants can better exploit the semantic information in the source code relevant for cache analysis, resulting in estimates that were up to 260% tighter than the traditional approach. Further, due to the high precision of our source-level analysis, we have discovered an error in the WCET analyzer OTAWA. Lastly, we assess the feasibility and complexity of modeling other architectural features in the source code, such as branch predictors, prefetchers, and bus accesses, concluding that all of them could be represented in the source code, as well.

Last but not least, during this work we have developed, integrated and tested a number of tools, some of which have been made publicly available for the benefit of other researchers. An overview on all the involved tools is shown in Figure 1.4 on page 16.

1.5 List of Publications

This monograph builds on the following peer-reviewed publications:

- Martin Becker, Markus Neumair, Alexander Söhn, and Samarjit Chakraborty. Approaches for software verification of an emergency recovery system for micro air vehicles. In Floor Koornneef and Coen van Gulijk, editors, *Proc. Computer Safety, Reliability, and Security (SAFECOMP)*, volume 9337 of *Lecture Notes in Computer Science*, pages 369–385. Springer, 2015

- Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chakraborty, and R. Venkatesh. TIC: a scalable model checking based approach to WCET estimation. In Tei-Wei Kuo and David B. Whalley, editors, *Proc. Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, pages 72–81. ACM, 2016

- Martin Becker, Emanuel Regnath, and Samarjit Chakraborty. Development and verification of a flight stack for a high-altitude glider in ada/spark 2014. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *Proc. Computer Safety, Reliability, and Security (SAFECOMP)*, volume 10488 of *Lecture Notes in Computer Science*, pages 105–116. Springer, 2017

- Martin Becker, Ravindra Metta, R. Venkatesh, and Samarjit Chakraborty. Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors. *Journal on Software Tools for Technology Transfer*, 21(5):515–543, 2019

- Martin Becker and Samarjit Chakraborty. WCET analysis meets virtual prototyping: Improving source-level timing annotations. In Sander Stuijk, editor, *Proc. International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2019

- Martin Becker, Ravindra Metta, R. Venkatesh, and Samarjit Chakraborty. WIP: Imprecision in WCET estimates due to library calls and how to reduce it. In N.N., editor, *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2019

The following publications of this author are generally related to the area of timing analysis and timing debugging, but not directly part of this thesis:

- Martin Becker and Samarjit Chakraborty. Optimizing worst-case execution times using mainstream compilers. In Sander Stuijk, editor, *Proc. Software and Compilers for Embedded Systems (SCOPES)*, pages 10–13. ACM, 2018

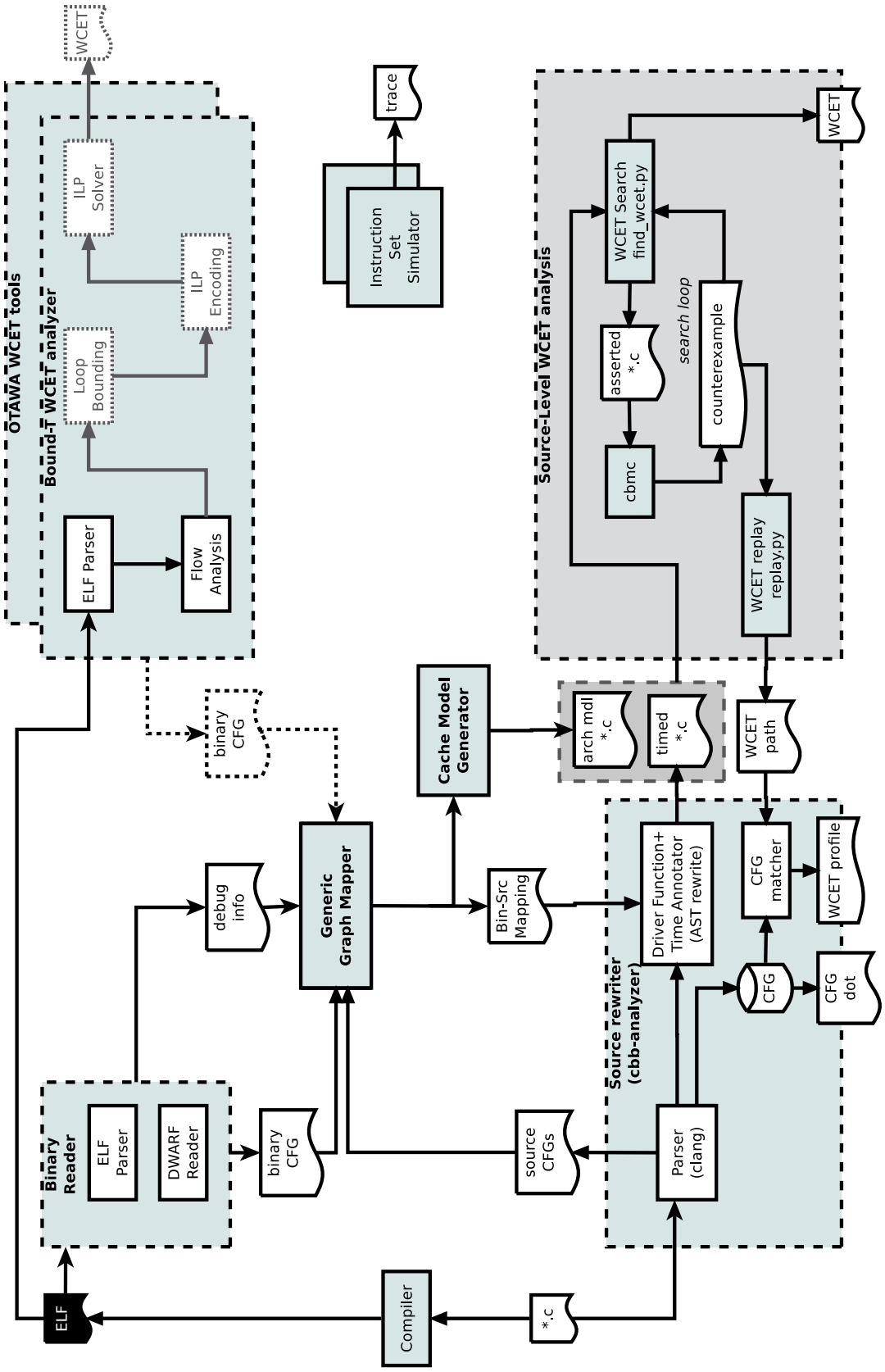


Figure 1.4: All the tools.

- Martin Becker, Sajid Mohamed, Karsten Albers, P. P. Chakrabarti, Samarjit Chakraborty, Pallab Dasgupta, Soumyajit Dey, and Ravindra Metta. Timing analysis of safety-critical automotive software: The AUTOSAFE tool flow. In Jing Sun, Y. Raghu Reddy, Arun Bahulkar, and Anjaneyulu Pasala, editors, *Proc. Asia-Pacific Software Engineering Conference (APSEC)*, pages 385–392. IEEE Computer Society, 2015
- Martin Becker, Alejandro Masrur, and Samarjit Chakraborty. Composing real-time applications from communicating black-box components. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 624–629. IEEE, 2015
- Martin Becker and Samarjit Chakraborty. Measuring software performance on Linux. *CoRR*, abs/1811.01412, 2018
- Martin Becker and Samarjit Chakraborty. A valgrind tool to compute the working set of a software process. *CoRR*, abs/1902.11028, 2019

1.6 Organization

The rest of this thesis is organized as follows.

- Chapter 2 summarizes related work to WCET analysis, source-level timing analysis and timing debugging, to put this work in a historical and technical context.
- Chapter 3 provides the formal background for the methods used in this work. It introduces basic terminology and describes the three fundamental methods of formal verification, namely *Model Checking*, *Abstract Interpretation*, and *Deductive Verification*.
- Chapter 4 evaluates recent tooling implementing these three methods by presenting and discussing three case studies that we have conducted. We identify the different strengths and weaknesses of the methods, and determine necessary boundary conditions to obtain sound analysis results. Moreover, we justify our selection of analysis methods in our source-level approach.
- Chapter 5 proposes a basic approach for a source-level WCET analysis based on bounded Model Checking, as well as scalability-enhancing methods and a process for timing debugging. Experiments have been carried out to compare against cycle-accurate simulation and a traditional binary-level analyzer, in the aspects of analysis precision and analysis time, as well as usability and safety.
- Chapter 6 introduces a novel generic, compiler- and target-independent algorithm for establishing an automatic and safe mapping from instructions to source code, which works with mainstream compilers and also supports compiler optimization.
- Chapter 7 introduces a novel source-level model for instruction caches, and presents an experimental evaluation. We further discuss the possibilities of modeling other architectural features, such as branch predictors, prefetchers, and bus transfers.
- Chapter 8 discusses the achieved results w.r.t. the initial hypothesis, i.e., that source-level analysis is feasible with state-of-the-art analysis tools, and that it can be more precise than binary-level analyzers, with less user inputs. We further identify promising future directions of research, which could improve both source- and binary-level timing analysis.

Chapter 2

Related Work

***

This chapter provides an overview about the current body of literature about static worst-case execution time analysis. The goal is to put our work into historical and technical context. More specific related work, for example about graph mapping algorithms or cache models, is discussed in the respective chapters focusing on such issues.

2.1 WCET Analysis

An survey about techniques and tools for WCET analysis was published by Wilhelm et al. [WEE+08], and more recently in [WG14].

In general, all approaches to WCET analysis can be classified into *deterministic*, and *probabilistic* estimation methods, whereas the first yields a unique WCET estimate that is provably safe, and the second a set of, or probability distribution of estimates, which can only be exceeded with a low probability [AHP+14]. Both approaches can further be subdivided into *static* and *dynamic* approaches. Static approaches derive the WCET from a model, by mathematical reasoning. In contrast, dynamic approaches execute the program and observe its behavior. Probabilistic analysis often uses dynamic methods, by taking measurements on the running program and deriving the tail of the execution time probability distribution. Similarly, deterministic analysis often uses static methods, to compute an upper bound that is guaranteed to be safe. However, there also exist static approaches that compute a probabilistic WCET estimate based on models [CQV+13], and dynamic approaches that compute a deterministic WCET estimate based on measurements [EY97].

In this thesis we only consider deterministic WCET analysis, with focus on static estimation techniques, as described in the previous chapter. This approach is the only one that yields a unique and safe upper bound on the WCET, which can be used in schedulability analysis to

guarantee adherence to deadlines. In the following, we summarize the body of research on such approaches.

2.1.1 Early Work on WCET Analysis

One of the first works dealing with WCET analysis was published by Puschner [PK89] in 1989, who proposed to compute the WCET at (indeed) source level. He introduced the notion of *timing schemata*, where each source-level construct is assigned a constant execution time. The WCET estimate was computed by a bottom-up traversal of the Abstract Syntax Tree (AST), propagating the maximum of alternative branches. The estimate was therefore rather untight. To enable this tree-based estimation, he further imposed constraints on the programming language, as well as annotation constructs to aid loop analysis. Some information was already generated automatically by the use of a special "precompiler" that computed loop bounds and the execution time of the instructions. The method was however neglecting the timing for control decisions, and thus was neither generally applicable nor safe.

A similar work was published shortly after that by Park and Shaw [PS91], mentioning issues with compilers and processor uncertainty. Their tool predicted how the compiler would translate source statements into groups of instructions, and estimated the WCET on the source, based on the timing of these groups. Control decisions were taken into account by assuming their worst-case behavior. All loops had to be statically bounded, and overestimation was a direct result of assigning a constant execution time to source level constructs (since the compiler may translate the same construct very differently, depending on the specific variable types and values). The results were finally not guaranteed to be safe, owing to the difficulty of predicting how the compiler implements certain constructs. The authors closed with the conclusion that more complex processors with caches and pipelines are notoriously non-deterministic, and can possibly not be predicted any longer.

This increasing complexity of compilers and processors had stirred a debate about whether timing analysis should be performed at source level or instruction level, which was effectively settled in 1995, with Li, Malik and Wolfe's seminal paper about an efficient modeling approach at instruction level [LMW95], as explained in Section 1.1. Static deterministic WCET analysis, since then, is predominantly based on their proposed ILP/IPET formulation, leading to the quasi-standard approach that we have described in the introduction.

For the remainder of this chapter, we refrain from any attempts to give a chronological overview, and instead organize the work in logical categories. We broadly classify all approaches into source- or binary-level, although some evade any clear classification. Arguably, our approach could also be considered to be one such case. In this work, we implicitly consider the venue of the path analysis as the deciding factor for categorization. That is, tools that compute the longest path on the source code, are considered source-level tools.

2.1.2 Analysis at Binary Level

The majority of WCET analyses takes place at instruction level, or also referred to as binary-level, since the binary program is analyzed. We categorize the various sub-analyses that are required (flow analysis, microarchitectural analysis, etc.) into *low-level* and *high-level analysis*.

Low-Level Analysis. Flow analysis the first step in all WCET approaches. It parses the binary, usually through disassembly, and discovers all possible control flows, yielding one or multiple CFGs. Since this merely interprets the machine instructions to find calls, returns,

jumps and branches, there is no need for a variety of methods. Many tools are available to perform this step, e.g. [BJAS11, S+89]. This work builds on such tools.

The next step, value analysis, computes possible register values to constrain the flows further [WEE+08]. For instance, addresses of indirect jumps are resolved, to complete the CFG. The usual methods are Abstract Interpretation (AI) [WEE+08], constant propagation and Presburger Arithmetics [HS02], although some different, integrated approaches exist as well, and are mentioned later. This part is usually coupled with loop analysis, which tries to identify loop repetition counts. Our approach currently does not make use of any value analysis, since the mapping from source to binary makes this currently unnecessary. However, we did not evaluate a data cache model, which will likely require to implement such methods. This challenge is discussed in Section 8.5.

Perhaps the most researched low-level task is microarchitectural analysis, that is, the modeling of the processor states. A first solution to model caches in the ILP/IPET framework has been proposed together with its introduction [LMW95], based on the concept of *cache conflict graphs*. Each BB was subdivided into smaller chunks according to distribution over multiple cache lines ("L-blocks"). Constraints on cache hits and misses were computed for each cache set individually on these graphs, and subsequently added to the ILP formulation, as explained in Section 1.1. The scalability was however low, calling for new methods to soundly model and estimate microarchitectural effects [Wil04].

A scalable solution to model caches has been proposed by Ferdinand, and has become known as classical cache and pipeline analysis [FW99, FHL+01]. The microarchitectural timing effects are first summarized by computing invariants at each program location using AI, and subsequently only these invariants are encoded in the ILP formulation. The details are explained later in Section 7.1.1.1. This effectively removes the microarchitectural model from path analysis, and thus increases the scalability of WCET estimation for large programs. Although this separation of path analysis and microarchitectural analysis renders the latter blind to flow facts (beyond structural constraints), it has been reported to cause only an insignificant loss of precision [Wil04] on industrial programs.

However, AI-based cache analysis was known to be imprecise on each JOIN operation (see Section 3.4). Every time paths are merging in the CFG, the AI merges the results of these paths into a single invariant, effectively loosing relational information between the blocks in the cache. A more precise abstract domain has been proposed recently [TMMR19], where possible sets of cache states are stored explicitly, but in a compressed representation using antichains. The results have been claimed to be equally precise as those from Model Checking (MC), yet 25 times faster. Pipeline analysis usually uses similar approaches as cache analysis [RS09, WEE+08]. Our work also makes use of the AI-based method for cache analysis to reduce the complexity of the WCET problem. We have introduced a fully precise cache model, which however did cause the expected state space explosion, thereby necessitating such abstractions.

Another method to improve AI-based microarchitectural analysis was proposed by Chattopadhyay, Banerjee and Roychoudhury in [CCR+14] and [BCR13]. The authors augmented the AI analysis with a Satisfiability (SAT) solver, which was used to exclude infeasible microarchitectural states from the computed invariants. Towards that, they track a bounded partial path along with the AI evaluation, and at each control decision, the path is sent to the SAT solver to ensure its feasibility. This reduces the imprecision due to the JOIN operation, and resulted in more precise WCET estimates. Incidentally, in their first paper they also used CBMC on the source code for infeasible path computation, yet stating that their binary-to-source mapping is not robust, and not able to handle compiler optimization. Fur-

thermore, the final timing analysis took still place at binary level. In principle their technique could be naturally integrated in our analysis framework, since we fully integrate path and microarchitectural analysis. However, we later show that such a partial path is implicitly available in the Frama-C tool (superposed states), which has a similar effect, but a higher scalability than their approach. The details are discussed in Sections 4.2 and 7.3. As a result, our approach attains a similar result, but with different and more efficient means, just by offering the complete model to state-of-the-art source level analyzers.

One weakness for microarchitectural analyses are loops. In particular, there has been much effort spent on classifying cache accesses in loops, to separate between a miss in the first iteration and subsequent ones. Initial approaches have been based on unrolling, introducing *persistence* analysis [FW99] which identified which cache blocks remain in the cache as long as the surrounding loop is running. This has later been generalized to compute the outermost persistence scope in nested loops [BC08], and is reviewed in more detail in Section 7.3. While most approaches are based on AI, some alternatives have been presented as well, e.g., in [CPHL01], which uses Presburger Arithmetics to compute the cache behavior in nested loops. Our microarchitectural analysis takes place at source code level and with standard tools from formal verification. Therefore, we propose a different method to calculate persistence information within loops in Section 7.3, which yields similar results as the multi-level persistence analysis.

High-Level Analysis. In traditional WCET analysis, the final longest-path analysis is usually done via solving an ILP formulation with implicit path enumeration (IPET), as described earlier. An alternative to the IPET formulation has been described by Puschner and Schedl [PS97]. While their path analysis ultimately still results in an ILP formulation, it was encoded as a maximum cost circulation in a graph. There is, however, no apparent advantage over the IPET approach, and hence this alternative formulation is rarely used today [WEE+08].

Another alternative path analysis was conceived by Stappert et al. [SEE01], based on Dijkstra's longest-path search algorithm. Flow facts were incorporated by checking the result against infeasible paths, blacklisting them, and restarting the search. This method explicitly searches the (potentially exponential number of) paths in a program and thus is expensive. However, the analysis time of this approach has not been given, but it also has never made it into any WCET tool implementation [WEE+08].

Although there are also structure-based approaches to path analysis (such as the early WCET estimation approaches on the source AST), we are not aware of any tool applying such a strategy at binary level.

Further search-based techniques for path analysis, but they require modeling the semantics of the program to enable a search. We therefore consider them part of the integrated approaches that combine all analysis steps, from value analysis over microarchitectural analysis to path analysis, in a single step. Such integrated approaches are summarized next.

Integrated Approaches. Lundqvist and Stenström [LS98] proposed an integrated path and low-level analysis, which is based on symbolic simulation. Loop bounds were calculated automatically, abandoning the need for manual user inputs. This however has only been shown for simple processors without caches, did not terminate in all cases, and was slow owing to the simulator being forced to handle unknown states. Furthermore, the approach was not sound in the presence of timing anomalies. Our approach does not share these weaknesses.

METAMOC [DOT+10] presents an integrated approach based on MC, combining low-level analysis for caches and pipelines with path analysis. They require an executable that is annotated with loop bounds, perform a value analysis, and then run the model checker UPPAAL to find the longest path. However, the path search is agnostic to program semantics, ignoring data flows and therefore not being able to identify infeasible paths. Unlike in our work, their estimates were not verified against a simulator, and the approach clearly shows scalability problems. A similar analysis based on a model checker has been proposed by Cassez [Cas11]. He models the processor as a network of timed automata in UPPAAL, at instruction level. Loop annotations are not required, since effectively the entire processor is modeled. To reduce the state space explosion problem he made use of manual abstractions, which results in a similar scalability as our source-level approach. As a downside, the processor has to be modeled as a timed automaton, which is not required in our approach, and also a source-level WCET feedback is not available. Comparing these two approaches confirms our findings that abstractions are the key for using MC in WCET analysis.

Another integrated approach was proposed by Metzner [Met04], solving the low-level and path analyses together with a model checker. This is also close to our approach, but was still done at binary level. The experiments did not include a comparison to WCET estimates using the traditional approach. Furthermore, Metzner also applied abstractions to improve the scalability of MC, but the scalability seems still rather low (his benchmarks had at most 2,500 instructions, yet it took several minutes to compute the WCET). In this work, we shift the analysis to source level to further increase scalability, and we also provide the lacking comparison to traditional WCET analysis. Closely related approaches at source-level are summarized next.

2.1.3 Analysis at Higher Levels of Abstraction

Although the quasi-standard workflow of WCET analysis had been successfully in industry, the original, source-level approach to WCET analysis has always been an ongoing research interest. Existing work can be categorized into true source-level approaches, and those that work on compiler intermediate representation.

Intermediate Representation (IR) Level. WCET analysis has also been proposed at an intermediate level in between source code and machine code, similarly because of easier analysis of data and control flow. Altenbernd [AGLS16] et al. developed an approximation method of the WCET which works on an ALF representation of the program, without analyzing the executable. They automatically identified a timing model of the intermediate instruction set through execution and measurement of training programs. As an effect, the analysis is efficient, but the estimate is not guaranteed to be safe.

Another IR-level WCET analyzer is the SWEET tool [LES+13]. It follows the traditional tool architecture, but all analysis takes place at the ALF representation, which can represent either source code or instructions. A converter is needed that also provides the timing information. Towards that, SWEET is integrated with a special compiler [GESL06]. In addition to the ILP-based path analysis, it also offers a cluster-based or path-based analysis. Low-level analysis uses a variant of abstract simulation. Program slicing is also applied to reduce the analysis time. This tool has an interesting architecture and was used several times in conjunction with other analyzers to automatically generate flow facts. Compared to our approach, it requires more user annotations [WEE+08], is limited to a specific compiler, and generally cannot provide the same precision.

Source-Level. Holsti [Hol08] proposed modeling execution time as a global variable, and to use a dependency and value analyses (Presburger Arithmetic) to determine the value of the variable at the end of a function and meanwhile exclude infeasible paths. Similar to our approach, slicing and loop acceleration were suggested. However, he showed by example that only some infeasible paths can be excluded by his method, whereas we can precisely detect all infeasible paths. Furthermore, his approach seems to have scalability issues, which unfortunately are not detailed further due to a lacking set of experiments. Caches were also not supported, since his work was rather an experiment revolving around analysis.

Kim et al. [KPE09] experimented with the model checker CBMC to compute the WCET at source-level of small programs. Their assumed processor was the PRET architecture [LLK+08], hence no caches have been considered. Further, they have not addressed the scalability issues, nor provided experimental data or user feedback.

Puschner [Pus98] computed the WCET on an AST-like representation of a program, where flow constraints were expected to be given by the user, and assuming that the compiler performs the back-mapping of actual execution times to the source code. He used ILP to compute the longest path. It should be noted that any ILP-based approach cannot handle variable execution times of basic blocks without large overapproximation, whereas Model Checking can encode such properties with non/determinism and range constraints. Furthermore, complete path reconstruction is not possible either with that approach. In contrast, our approach establishes an automatic mapping without requiring compiler modifications, computes flow constraints automatically, and furthermore supports caches.

A source-level timing analysis for Ada programs on simple, cache-less processors has been proposed by Chapman [CBW96]. Because of the assumed simple processor, the analysis is only concerned with the longest path problem. It integrates timing analysis and program proof by building regular expressions on the paths in the program, and then passing them to a prover to determine the longest path. Towards that, users have to annotate the program with loop information and other specifications for the prover. His work had mostly been a proof-of-concept, and unfortunately did not present any information about the analysis time.

In [KYAR10], WCET analysis using MC was evaluated for a model-based design of IEC-61499 function blocks. The processor was a Microblaze with disabled caches, and therefore the focus was also on the path problem. The WCET was estimated from a time-annotated source code with the model checker CBMC, and hence is close to our work. However, estimates were in average less precise than ours (reasons have not been given), and the compiler was modified to obtain a mapping. Finally, the state space explosion problem was not addressed in their work, either.

Last but not least, there are also some source-level approaches to timing estimation in the Virtual Prototyping [MG19] domain, but these are not sound for WCET analysis. We review and make use of these methods in Chapter 6.

2.1.4 Computing & Specifying Flow Facts

One of the main problems is specifying a sufficient number of flow constraints (also called "flow facts"), to reduce the number of infeasible paths, but also to aid value analysis. Since manual work is not desirable and bears a high risk of human error, various methods have been proposed which connect the traditional, binary-level WCET analysis with the information available in the source code.

One example is the SWEET tool mentioned earlier. Part of the *ALL-TIMES* project was the goal to leverage the source code information for a better WCET, and towards this SWEET was coupled with a source front-end to provide the IR-level input and compute function

pointers, and with the WCET analyzers RapiTime, aiT and SymTA/S, to produce tight estimates [LES+13]. SWEET provided loop bounds [GESL06, ESG+07] and other flow facts to these tools. A similar tool integration has also been done between SWEET and Bound-T later on. The underlying analysis, however, remains a binary-level approach, with a quite complex setup, and the interfacing of these tools requires elaborate annotation languages.

Another example is *orange* from the OTAWA WCET analyzer[BCRS10]. The tool evaluates the source code of the program to generate flow facts, and generates an annotation file that is read by the binary-level WCET analyzer. The tool however cannot bound loops in all cases, and more importantly, the annotations are only safe if the compiler does not alter the associated control flow.

An approach going even further that leverages the semantics of "predictable" programming languages, has been demonstrated by Raymond et al. in [RMPC13] for the synchronous language *Lustre*. The strict semantics of such languages simplifies the computation of infeasible paths at source level, which eventually can improve the resulting WCET estimate. In principle there is a synergy between their approach and the solutions presented herein. However, since our proposed approach to source-level analysis uses Model Checking, it does not suffer from precision problems. Consequently, it does not appear to be beneficial to exploit high-level semantics for better flow constraints in our approach, for it only duplicates the mapping problem, that is, once for the compilation from the high-level language (e.g., Lustre) to a general-purpose language (e.g., C), and subsequently from the general purpose language to the machine instructions.

Compiler-integrated approaches to compute flow facts at binary-level have been explored as well, e.g., in [HSR+00], [PSK08] and [BMP14]. Some of these approaches offer to back-map the WCET to the source code, for which the modified compiler maintains traceability between source and instructions.

All of these methods require annotation languages to transfer the computed information to the final path analysis. Defining a language that is expressive enough, is however a known problem [KKP+07], especially for flow facts beyond loop bounds, which is difficult in all known languages and remains an open challenge [KKP+11]. In contrast, our integrated approach alleviates the need for such annotations, since MC implicitly discovers such flow facts automatically.

A binary-only approach towards the computation of exact program flow constraints has been proposed by Marref [Mar11], to tighten an existing WCET estimate. She employs an elaborate workflow involving MC, dynamic testing, genetic algorithms and ILP path solving to compute flow facts automatically. However, such dynamic approaches are hardly scalable, as discussed in the introduction, which reflects in analysis times in the order of tens of hours.

2.1.5 New Hardware and Software Paradigms and Isolation Techniques

The complexity of modern processors has reached an extent that increasingly hampers static timing analysis. The necessary microarchitectural analysis in conjunction with the required path analysis is becoming more and more intractable. This view is currently shared by many researchers [Pus02, LLK+08, KP08, KP08, AEF+14, Sch09] and [MTT17, Chapter 5.20], and the need for better analyzability has generated the two research directions of more analyzable hardware, and more analyzable software, and a host of isolation techniques bridging the two.

We discuss the hardware trends in the context of our results in Section 8.6, where we argue that this development is beneficial for a source-level timing analysis. This comprises, inter alia, multi-core processors that attain temporal independence between their cores via time- and memory partitioning [SAA+15].

The software-only approach to counteract the complications of WCET analysis, however also requiring some hardware support, is focused on removing branches from the instruction stream, to produce *single-path* programs [Pus03]. This is an appealing idea, since it circumvents many problems of WCET analysis at once, namely, those of infeasible paths, exponential complexity in hardware states, and finally also removes the need to perform path analysis. The program could effectively be simulated to obtain a WCET estimate. Experiments have been published by Puschner [Pus05], where ordinary programs have been converted to single-path programs, and then compared in their performance. The results, although not statistically significant, show that single-path programs are competitive to their input programs in WCET, yet up to 40% slower in average. Since the execution of single-path programs always takes the WCET, this implies that the overall WCET of a program with many subprograms results in the sum of the individual WCETs, which would lead to an unacceptable loss of performance. The only alternative is an interprocedural conversion, perhaps with inlining of different call contexts, which would be complex and lead to an equivalent increase in program size (which was also not evaluated). Judging by recent publications, it seems that the single-path approach is no longer considered.

Finally, a growing body of research revolves around isolation techniques. Instead of assuming predictable hardware that is mostly academic at this point, isolation techniques aim to minimize temporal interference between tasks on modern COTS hardware. One notable work is the *WCET(m)* framework [MPC+15] proposed by Mancuso et al., which advocates the use of memory allocation schemes and budgeting on shared hardware resources. Through this it becomes possible to safely extrapolate the WCET of a single task running in isolation to its WCET in the presence of other tasks on an m-core processor. Another example is the *phased execution* model introduced by Pellizzoni et al. [PBCS08], which describes a co-scheduling algorithm for tasks and peripherals that reduces the interference caused by memory accesses.

Essentially, all those methods partially or fully turn shared resources into time- or space-partitioned resources, thereby attaining predictability and reducing model complexity at the cost of reduced execution efficiency.

2.1.6 Analysis Methods

There have been several publications in the domain of WCET analysis which discussed the suitability of various analysis methods. Owing to the complexity of modern processors, the main challenge is commonly seen as in the combination of microarchitectural analysis and path analysis, stemming from performance-enhancing features such as caches, pipelines, speculative execution and so on [Wil04, MTT17]. The complexity of their interaction is significant, prohibiting a precise and exhaustive model [WEE+08]. It is sometimes argued that a combination of AI and ILP methods appears to be without alternatives [Wil04].

Specifically, using MC as sole analysis method for all subtasks in WCET estimation is seen as intractable [Wil04]. One concern is that it can only evaluate Boolean predicates and not yield ranges of possible values at a given location, so it is not directly suited to estimating WCET. The several approaches to MC-based WCET analysis (including ours), however show that this is not an issue in practice. A binary search is quite effective, requiring typically between three and seven queries to the oracle (see Section 5.1.4.3).

More importantly, bounded MC itself has a (double) exponential complexity [CKOS04], since it explores the state space of the program to check global safety properties. Leaving microarchitectural analysis to MC as well, would dramatically increase the computational complexity, so that its scalability can be safely doubted. We indeed share this view, but we

address the issue in Chapter 7, showing that MC can still be used. Other authors have proven the same point, e.g., [Cas11], as discussed before.

Perhaps best quantifying this view are the experiments in [LGG+08], which confirmed that model checkers often face run-time- or memory-infeasibility for complex programs, whereas the ILP technique can compute the WCET almost instantly. However, because they used the same benchmarks that we are using (on a similar processor) *and* we come to the opposite conclusion, this confirms that our shift of the analysis to the source code indeed mitigates the scalability issue.

Additionally, it should be noted that also in the traditional approach to WCET estimation, an analysis with full precision is intractable. The ILP/IPET approach suffered this problem [LMW95] when the cache model was encoded directly [Wil04]. The size of the ILP problem grows exponentially with the number of cache blocks, so that this only works for small programs. The most common method for microarchitectural analysis is therefore AI [Wil04], to compute microarchitectural invariants, such that path analysis is relieved from analyzing the hardware. On the other hand, using AI for path analysis, too, is not a viable approach either. The computational complexity is not the issue, but instead that its overapproximative nature occasionally looses all information about the execution time along the program path, as we demonstrate in Chapter 5.

This work therefore shows that the argument that MC is unsuitable is flawed, for no method in isolation has proven to be precise and scalable enough for WCET analysis. We show in Section 5.2 how the particular weakness of MC can be mitigated through program transformations, and thereafter we also resort to AI for microarchitectural analysis in the presence of caches. However, our source-level representation of the temporal behavior actually leaves the choice of method with the developer. There are some tools that escape a rigid method classification, such as Frama-C's value analysis reviewed in Section 4.2, which allows trading analysis time for precision, and is also able to deduce a reasonable precise WCET estimate with some tuning. We therefore argue that no method should be categorically excluded, especially since more and more hybrid approaches are emerging in the domain of functional verification.

2.1.7 Parametric WCET Analysis

Another research direction is the computation of parametric or symbolic WCET estimates. Instead of computing one single bound for the program under all possible inputs, such approaches parameterize the WCET on the input values. Examples for such approaches are [VHMW01] (structural approach at binary-level, loop bounds must be known, no infeasible path detection), [HPP12] (structural approach at IR-level, algebraic simplification) and [Lis03] (generalizes the ILP/IPET approach with parametric integer programming) and finally also Chapman [CBW96], as discussed earlier.

All of these approaches are path-centric and do not consider microarchitectural models, mainly geared towards replacing the final path analysis. They therefore face the same challenges as traditional approaches, most importantly the flow fact and annotation challenge.

One notable exception is an early version of HEPTANE[CB02, HRP17], which did support caches, but also suffered from a weak identification of infeasible paths. We discuss the fundamental limitations of parametric estimates in Section 8.5, arriving at the conclusion that such estimates are hard to compute in the presence of caches, and furthermore they are not reusable. That is, they must be re-computed every time the program is changed, and therefore we consider such methods as an alternative to ILP/IPET, but not addressing any of the other challenges in traditional WCET analysis.

2.1.8 Tools

Table 2.1 shows a non-exhaustive overview of currently maintained static and deterministic WCET analyzers. In this work we used Bound-T and OTAWA as baselines.

Table 2.1: Overview of current static deterministic WCET analyzers.

		Analysis		
Tool	**Targets**	**High-Level**	**Low-Level**	**References**
aiT	I/D cache, in-order, out-of-order	ILP/IPET	AI	[Gmb19, LES+13]
Bound-T	no caches, in-order	ILP/IPET	Presburger Arithmetics	[HS02]
Chronos	I/D cache, in-order, out-of-order	ILP/IPET	AI, MC	[LLMR07, LRM06, CR13]
Heptane	I/D cache, in-order	ILP/IPET, tree	AI, sim.	[CP00, CB02, HRP17]
METAMOC	I/D cache, in-order	MC	MC	[DOT+10]
OTAWA	I/D cache, in-order	ILP/IPET	AI	[BCRS10]
SWEET	I/D cache, in-order	ILP/IPET	sim.	[GESL06, ESG+07, LES+13]
TuBound	no caches, in-order	ILP/IPET	Interval Analysis	[PSK08]
WUPPAAL	I/D cache, in-order	MC	MC	[Cas11, CdAJ17]

2.2 Timing Debugging

Understanding how the worst-case timing is produced is very important for practitioners, but there is only a small body of literature on this topic. Making the worst-case (and best-case) timing visible in the source code is a rather old idea, appearing first around 1995 [KHR+96]. The authors highlighted WCET/BCET paths in the source code, allowing the user to visually trace the WCET path and how time passes on this path. That work was later extended in [ZKW+04] to apply genetic algorithms for minimizing WCET, and still later to introduce compiler transformations to reduce the WCET. The WCET in these approaches estimates were however not safe, and the user feedback was still rather limited. A similar, but interactive tool was presented in [HK07] and extended in [HSK+12]. This was also a tree-based approach for it was focusing on analysis speed instead of precision. It moreover requires Java processors as [Sch03], and the approach itself only visualizes the timing in the source, but is not a source-level estimation. All these approaches do not provide tight results, as they were based on timing trees which did not consider data dependencies; they can only reconstruct the WCET path w.r.t. locations and time spent there, but lack a specific diagnostic trace.

The commercial WCET tool *RapiTime* [BDM+07] estimates the WCET presents timing bottlenecks in the source-code as color coding, similar to our source-level highlighting shown in Section 5.4.3. RapiTime further allows predicting what would happen if a specific function would be optimized. However, the tool is measurement-based and thus cannot give any guarantees. The commercial tool *aiT* [FHLS+08] provides time debugging capabilities at the assembly level, e.g., a call graph and a timing profile. It is also capable of mapping back this timing information to a high-level model, where the building blocks of the model are annotated with their contribution to the WCET. Further, the tool allows making changes to the model, and compare the results with the previous ones, essentially enabling a trial-and-error approach to improve the WCET. Finally, a similar toolset that realizes a generic formal interface between a modeling tool and a timing analysis tool has been presented in [FBvHS16], but relies on external tools for the actual analysis. All of these tools enable timing debugging to some extent, but they cannot provide a specific trace with values, or even allow the user to interactively step through the WCET path with a debugger, as we do.

A different view on bottlenecks has been presented by Brandner et al. [BHJ12]. They proposed to leverage dominator information to enable a static profiling of the WCET, which resulted in "criticality information" for each BB that is on, or close to, the WCET path. Although this does not quantify the amount of time spent in each BB, it ranks them by their importance for WCET. We have experimented with this method in [uC18b] in an attempt to leverage this information for automatic WCET reduction in mainstream compilers, and reached some moderate effects. However, this method provided no valuable insights for a user who seeks to reduce the WCET.

An method to derive the inputs that lead to the WCET has been presented in [EFGA09]. In principle it is a search-based method without any guarantees, unless the input space is exhaustively evaluated. In contrast, our approach also yields the WCET inputs as part of the WCET reconstruction, which only requires a single program run after the WCET analysis terminates, and furthermore our inputs are guaranteed to trigger the worst case.

2.3 Functional Verification

Although this work focuses on WCET estimation, we build on methods from general software engineering that were conceived for functional verification. We dedicate Chapter 3 to introduce the basic concepts, and Chapter 4 to evaluate current tools implementing them. Our method and tool selection was motivated as follows.

A large orchestrated survey from 2013 [ABC+13] summarizes the difficulties of test-based approaches for functional properties. Heuristics, approximations and models have to be used to pick some of the exponentially many paths in a program, yet most methods fail to provide any guarantees. This, of course, is aggravated when the temporal aspect is added, since the state space becomes only larger when the processor behavior needs to be considered. Especially search-based methods are increasingly popular, as found by a 2012 survey [HMZ12]. It lists more than 450 different methods in general software engineering, with some methods already addressing temporal properties. A similar survey limited to searched-based methods for non-functional properties [ATF09] shows that execution time is indeed the number one interest next to functionality. We believe that this trend is another indicator that source-level timing analysis could benefit from methods in general software engineering.

Looking specifically at *formal* methods for fully automatic functional verification, the 2014 status report on software verification [Bey14] features promising results that suggest that formal methods could be the answer to the problems encountered in test-based methods. It reports on the capabilities on 15 different formal verification tools in the context of a competition. It shows a variety of verification methods, making progress in many directions, with increasingly better support for complex programs. The tool *cbmc* [SK09], the bounded model checker which we use heavily in this work, has consistently been among the most top three in most categories, and was the overall winner of the competition. Yet another survey on formal verification tools [DKW08] concludes that Model Checking is currently the most applicable method for formal verification in practice, yet attesting that its Achilles heel is its poor scalability, constraining its usage to mostly shallow programs. We address this scalability issue specifically for timing analysis in Chapter 5.

2.4 Miscellaneous

WCET Benchmarks. While general software engineering has established benchmarks to test processor and compiler performance, such as the *SPEC2000* or *Dhrystone* suites, these are not made to stress program analysis tools, nor are they made for embedded systems, where WCET analysis is typically needed.

Therefore, the WCET community has established their on benchmark programs which specifically address these needs. One such WCET-specific benchmark is *debie1*, which was introduced in the WCET tool challenge 2008 as main benchmark [HGB+08]. Originating from an onboard software of the DEBIE-1 satellite, the program features several threads, some interrupt-driven. It has been stripped from the kernel and device-specific peripheral code, and the bodies of the threads have been singled out to provide entry functions for WCET analysis. However, the debie1 benchmark is only one single program, which does not explicitly address the known weaknesses of WCET analyzers. A similar benchmark is *PapaBench* [NCS+06], which we also use in this work. This benchmark is a similar adaption of a flight control software from a small UAV.

The most commonly used set of programs for WCET analysis are the *Mälardalen WCET benchmarks* [GBEL10]. It consists of 35 programs written in C, ranging from a small binary search of 15 elements, to code generated from statecharts, and systematically covers features that are challenging for WCET analyzers, e.g., nested loops, unstructured code, floating-point numbers, recursion, and so on. Over the years, more benchmarks have been added to reflect the needs of WCET research [GBEL10]. Judging by the number of references in WCET-related republications, this benchmark suite still is, by far, the most commonly used one today. We therefore based our experiments on these programs.

2.5 Summary

This work proposes an approach to WCET estimation that takes place at source level, to a larger and more complete extent than any existing work, with results that are competitive and often better than existing tools, and with an unprecedented user feedback of the WCET. While there have been source-level approaches in the early days of WCET analysis, they have been given up due to problems with compiler optimization and the difficulty of modeling performance enhancing features at source level. We solve both of these problems. Specifically, we integrate both microarchitectural analysis and path analysis together in the source code, which has not been done before. While several groups have proposed similar solutions for path analysis, they all build on microarchitectural analyses done at binary level, whereas we integrate the two.

Our source-level approach also conveniently circumvents the problem of computing and specifying flow facts, which is a frequent obstacle for all WCET tools. These facts can be mostly inferred automatically in our approach, thanks to our integration of low-level and high-level analysis, and the use of Model Checking. While other researchers have certainly made great progress in combining various source- and binary-level analyzers to reach a similar effect, they all face the problem of communicating flow facts between the tools, which in itself is a recognized research challenge. With our integrated approach, this is no longer necessary.

We leverage a number of previously explored methods from WCET analysis, most importantly Abstract Interpretation to analyze the cache behavior, and Model Checking for path analysis, yet not all of these methods were directly applicable. In particular, WCET analy-

sis based on Model Checking has been deemed as too computationally expensive. Several groups had tried it before, with varying results. We contribute new complexity-reducing methods to counteract this complexity issue. Comparing our results to those from existing WCET tools based on Model Checking, we can conclude that the conjunction of shifting the analysis to source level and complexity-reducing methods yields our scalability advantage, and demonstrates that Model Checking should not be dismissed for WCET analysis.

Last but not least, we introduce a unique and novel feedback to the user, which allows to interactively inspect the WCET path in a debugger. No similar methods have been proposed earlier, neither from a precision, nor from a usability standpoint. Instead, user feedback so far has been limited to visual highlighting of the WCET path, and displaying bottlenecks in callgraphs or flow graphs. On top of our novel timing debugging technique, we also offer such visualizations. All in all, this work therefore challenges the state of the art in WCET analysis, and fills a gap in existing approaches, namely an integrated source-level analysis with user-friendly feedback, supporting processors with caches, branch predictors and pipelines.

Chapter 3

Basics of Program Analysis

***

This chapter introduces the basic terminology and methods for formal verification of programs. The presented methods are commonly used to analyze a program for *functional* correctness, aiming to reveal defects or deviations from a high-level specification. We start with an introduction to basic concepts in program verification, before we describe the three prevailing formal analysis methods starting in Section 3.3.

3.1 Program Semantics

3.1.1 Concrete Semantics of a Program

The behavior of a program F when executed can be formalized as *concrete semantics* $(\Sigma, \twoheadrightarrow)$, where $s \in \Sigma$ is a tuple $s = (l, env)$ of the program location l and its current memory state env, and $\twoheadrightarrow$ is the set of possible transitions between these states, representing the effect of statements in a program. Specifically, env maps program variables to their current values in their respective domain $\mathcal{D}_{var}$ as

$$env : Var \to \mathcal{D}_{var}, \text{ therefore} \tag{3.1}$$

$$s : l \times (Var \to \mathcal{D}_{var}). \tag{3.2}$$

A *trace* of a program is a potentially infinite sequence of such locations and states as $\pi := s_i \to s_{i+1} \to \ldots$, usually starting at an initial state. Most programs have more than one path that can be executed (e.g., due to conditional statements), therefore the semantics of a program in general consists of *set* of possible traces.

Note that this can be represented as a finite transition system in practice, since the semantics of a program is finite if the number of locations and memory space are finite (although the trace can be infinite), which is the case in real computer systems.

3.1.2 Collecting Semantics

Collecting Semantics represent the strongest static properties that can be deduced on a program, by describing sets of reachable states. The specific contents of the sets depend on the property

of interest. For example, we could build a set of states that are reachable from the initial state, or a set of states which were executed before (and thus possibly influenced) a given state.

Here we consider *trace semantics*, which describe the *set* of possible traces T_c and the set of reachable states S_c in a program as

$$T_c := \{s_1 \to \cdots \to s_n \mid s_1 \text{ is an initial state } \wedge (s_i \to s_{i+1}) \in \twoheadrightarrow\}, \tag{3.3}$$

$$S_c := \{s \mid s_i \to s_i + 1 \wedge s_i \in T_c\}. \tag{3.4}$$

Collective semantics then records the precise sets of memory states at each program point l, and thus sets (and implicitly relations) of variable values that can occur at l as

$$Coll : l \to 2^{env}, \text{ i.e.,} \tag{3.5}$$

$$Coll : l \to 2^{Var \to \mathcal{D}_{Var}}. \tag{3.6}$$

We will use the notation $\mathcal{P}(...)$ to indicate such collections of elements, and therefore simplify this notation to

$$Coll : l \to \mathcal{P}(Var \to \mathcal{D}_{Var}). \tag{3.7}$$

Note that the use of collecting semantics for a static analysis is generally not tractable, since traces can be of infinite length.

3.2 Properties of Analysis Methods

Consider some analysis method $\mathcal{A}$ which takes a program F as input, computes some model M from it, and then performs some reasoning on M about a user-defined property R (e.g., "there are no integer overflows"). The method $\mathcal{A}$ can either raise an alarm, indicating that it deduced that R does not always hold true in F, or it can stay silent, indicating that it deduced that R always holds true in F.

The Decision Problem. A *decision problem* is a computational problem where the answer is Boolean. It is called *decidable* iff there exists a method or algorithm to find the correct solution in a finite amount of time, *undecidable* otherwise. Kurt Gödel has proven in his incompleteness theorems [Göd31] that any sufficiently expressive proof system must either be incomplete or inconsistent. His results were later used to find an answer to the *Halting Problem*: Given any program F and its inputs, is it possible to determine whether the program terminates (halts), or continues to run forever? Church and Turing leveraged Gödel's results to prove this problem to be undecidable in general, which leads to the consequence that no analysis method is able to decide all properties on all programs [Tur38].

3.2.1 Completeness and Soundness

Whether a property R actually holds true in a program F, may deviate from what $\mathcal{A}$ is claiming, by virtue of the underlying model M and reasoning. There are two basic properties of formal analysis methods:

- **Soundness:** $\mathcal{A}$ is sound if any property R it claims to hold true, is never violated by any feasible execution of F. That is, $\mathcal{A}$ emits an alarm for every violating program. In other words, any conclusion that is being made by the analysis must be correct for all possible executions.

- **Completeness:** $\mathcal{A}$ is complete if it indeed can prove any true R. That is, it stays silent for every compliant program. Or in other words, any property that is satisfied by the concrete semantics, must also be proven by the analysis method.

Thus, a method that raises an alarm for every R on any program is trivially sound, but most will be *false* alarms. In practice, soundness often requires overapproximation (i.e., to model more behaviors than possible in F) to keep the problem tractable, which causes these false alarms. Vice versa, a method that raises no alarms at all is trivially complete, but no use for detecting defects in a program, since it always claims all properties are satisfied.

An ideal analysis method would be both sound and complete, but it cannot exist in general due to the fundamental undecidability problem mentioned earlier. Therefore, on realistic programs, a method either cannot prove some true properties (incomplete), or it proves some wrong properties to be true (unsound). There are three ways to consolidate these two opposing goals. (1) We can prevent the analysis from terminating, (2) we can allow the analysis to be unsound for the sake of completeness, or (3) we can allow the analysis to be incomplete for the sake of soundness.

In practice, verification tools for static analysis are often designed to terminate and be sound, but to emit a few false alarms, i.e., to be incomplete in some aspects. Further, when computing the model M of the program F, they often apply abstractions, and thus usually give up more completeness in various ways. Therefore, different methods and tools in static analysis mainly differ in their extent of completeness, and directly linked to that, also in their computational complexity. In summary, since all tools should be sound but completeness can generally not be achieved, we need a more quantitative measure to compare different analyses in practice, which is the *precision* of an analysis.

Remark to soundness definitions. There exist also some other definitions using the term *soundiness*, arguing that virtually no sound tools exist [LSS+15], because all of them deliberately take unsound decisions to retain precision. For example, by making assumptions that certain constructs do not have side effects. However, in this thesis we still consider such tools as sound, yet only sound *w.r.t. certain assumptions*. As long as the tool warns about these potentially unsound decisions, such constructs can be avoided, or otherwise the safety of the analysis can be established by checking the assumptions manually.

3.2.2 Precision and Abstraction

Abstraction. Informally, the term *abstraction* denotes that a model M omits some details of F, by approximating some behaviors, usually with the goal of reducing the state space. A *sound* abstraction does that in a way such that the model M subsumes the original behavior of F, and includes some behaviors and states Σ_M that are infeasible in the concrete semantics Σ, i.e. $\Sigma_M \supseteq \Sigma$. We will refer the result of sound abstraction as *overapproximation*, as opposed to just approximation, which just aims for $\Sigma_M \cap \Sigma \approx \Sigma$, but is not necessarily a superset and thus possibly unsound.

Precision for a Program. Informally, the *precision* of an analysis $\mathcal{A}$ corresponds to how close it gets to completeness. Precision is, however, a function of the program F and the property R, and not an inherent property of an analysis $\mathcal{A}$. Given F and a property R, it quantifies the amount of false alarms that $\mathcal{A}$ raises, in relation to all alarms being raised. The less false alarms, the more precise it is for F w.r.t R. Note that precision does not equal the "degree of completeness", since an analysis method may not be able to decide some property.

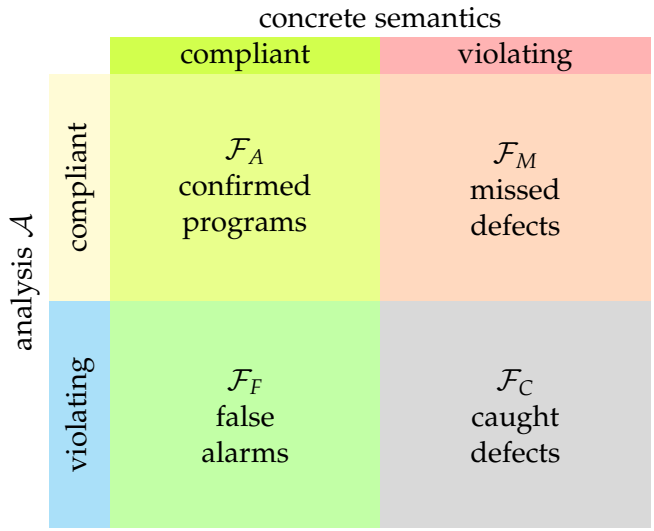


Figure 3.1: Classification of analysis results.

For a formal definition, first consider the illustration in Fig. 3.1. It summarizes the different combinations of what the analysis $\mathcal{A}$ claims for a property R, and what the concrete semantics allow. Consider n programs $\mathcal{F} = \{F_1, \ldots, F_n\}$, each of them falls into exactly one of the four cases shown in the Figure, i.e., the four cases are disjoint.

Since we are only concerned with sound analysis methods as to not miss any behaviors that are feasible in the concrete semantics, it holds that $|\mathcal{F}_M| = 0$, i.e., no programs violating R are missed. More interestingly, we can now define the *precision* of an analysis for set of programs $\mathcal{F}$ and a property R can be defined as

$$\rho := 1 - \frac{|\mathcal{F}_F|}{|\mathcal{F}_C| + |\mathcal{F}_F|}, \tag{3.8}$$

whereas a value of one denotes full precision, and zero none. Thus, an analysis $\mathcal{A}$ is maximally precise if it raises no false alarms, that is, every claimed violation of R is indeed a violation in the concrete semantics. In analogy, one could define how far an analysis $\mathcal{A}$ is away from soundness:

$$\zeta := \frac{|\mathcal{F}_M|}{|\mathcal{F}_M| + |\mathcal{F}_C|}. \tag{3.9}$$

However, in this thesis we only consider analyses for which $\zeta = 0$, i.e., which are perfectly sound, possibly to specific assumptions.

Since abstraction removes information from the M by subsuming more behaviors than possible in the concrete semantics, an analysis naturally suffers a loss of precision for some programs, since a superset of behaviors may include some that violate R.

3.2.3 Sensitivity

In program analysis, another characterization of methods can be done by evaluating their sensitivity to certain aspects of the semantics, namely [DKW08]:

- **Flow sensitive**: The analysis considers the order of execution of statements (or even expression evaluation).

- **Path sensitive**: The analysis distinguishes between different paths through the program.
- **Context sensitive**: Functions are analyzed in their call context/per call site, i.e., with their specific arguments.
- **Inter-procedural**: The body of callees is analyzed, as opposed to making assumptions or omitting them at all.

An ideal analysis would fulfill all of these aspects, but naturally has to spend more computational effort compared to a simpler analysis which omits some of these aspects by abstraction to reduce analysis time, but possibly looses precision.

3.2.4 Computational Complexity

A last important property of an analysis method is its *computational complexity*. For each of the analysis methods, there are two different measures. (1) The cardinality of the state space of M, and (2) the computational complexity of the decision procedure relative to aspects of M. One important difference therefore lies in how M is constructed from F, and in the specific decision procedure that is applied thereafter. Therefore, when referring to complexity of an analysis method, we have to distinguish between these two aspects.

In any case, it is intuitive that analyses with a higher precision require capturing a more detailed version of F in M, which therefore requires more memory and more processing time, up to the point where it can become *intractable* due to resource or time constraints.

Next, we briefly introduce three sound methods for formal static verification. These methods are all capable of identifying defects in the source code of a program, while considering all possible execution traces. Therefore, as far as the boundary conditions and limitations of the method and tools allow, we can obtain a guarantee that the program under analysis is free of certain defects. For example, if such a tool has been used correctly and concludes that there is no possibility that a numeric overflow occurs in the program, then we have a guarantee of that behavior.

3.3 Model Checking (MC)

Model Checking [CGP99] is a formal analysis technique used to verify a property on a model with finite state abstractions. Given a model M and a property R, a *model checker* – a tool implementing this technique – determines whether M satisfies R in every possible execution, by performing a reachability analysis over a finite-state transition system that represents M. If the property does not hold, the model checker produces evidence for the violation, called a *counterexample*. The latter is a practically important advantage of MC over other verification methods, because it aids the debugging process. Another benefit of MC is that it can be fully automated, unlike Deductive Verification discussed later on.

We now briefly introduce the principles of MC based on a summary of one of its creators, Edmund Clarke [CKNZ11], only presenting the details required for this thesis. The model M is defined as a finite state transition system $M := (S, I, \twoheadrightarrow, L, AP)$, where

- S is the set of states in the transition system,
- I is the set of initial states $I \subseteq S$,
- $\twoheadrightarrow \subseteq S \times S$ is the set of state transitions,
- AP is a set of *atomic propositions* (Boolean functions over S), and
- $L : S \to 2^{AP}$ is a *labelling* function that assigns the subset of APs that hold in a state $s \in S$.

The model is naturally close to the program semantics introduced earlier, such that each $s \in S$ can represent the tuple (l, env) of program location l and environment env. The state space of M is therefore a product of program locations l and the variables in env, and grows exponentially with the number of variables, known as the *state (space) explosion problem*.

An atomic proposition $p \in AP$ is a state formula, describing whether a certain Boolean predicate is true or not, and the labelling function $L(s)$ defines which combination of them holds in a given state s. The APs must be chosen to support the specific property to be evaluated. For example, consider a program that writes data to a file, and we are interested in whether the file has been opened before writing to it. Then the APs could be chosen as $AP = \{\text{file opened}, \text{data written}\}$, and L would indicate for each state s which combination of these two predicates holds. However, the APs do not involve any notion of time or progression in M. Instead, it is the formal language of property R that captures such relations. One such formal language is *linear temporal logic* (LTL), which is suitable to formally express the property "whenever data is written, the file must have been opened before".

The process of MC then consists of the following steps:

1. The user specifies a to-be-verified property R in a formal language, e.g., in LTL.
2. A model checker constructs a model M with all information relevant for R, possibly with abstractions suitable for R. For example, if the property is only concerned with one specific particular program variable, then M can omit all variables which cannot influence the one under analysis (see also Section 5.2.1 about *slicing*).
3. Given M and R, a verification algorithm performs an intelligent exhaustive search over the state space to determine whether M satisfies R in all feasible traces. Towards this, it finds all reachable states $s \in S$ that satisfy R. Conversely, it can abort the search if one reachable s_v is found that does not satisfy R.
4. If one s_v violating R is found, then the model checker constructs the *counterexample*, which is a program trace $s_0 \to \cdots \to s_v$ leading to the unwanted state.

The specification of R is traditionally based on *Computational Tree Logic* or *Linear Temporal Logic* [CKNZ11], but details are not of importance for this work, since software model checkers generate the according specifications automatically from source-level specifications, such as assertions. The traditionally used verification algorithms were based on graph theory, *inter alia* based on reachability tests. Since each state $s \in S$ was visited, it is known as *explicit-state* MC [CKNZ11]. However, due to the exponential complexity of the verification procedure itself, this original approach to MC is limited to smaller models.

3.3.1 Bounded Model Checking

The newer approach of *bounded* MC avoids exhaustive verification by only examining paths in M up to a given length. It has mostly replaced the earlier verification algorithms [KOS+11, CKNZ11], and therefore, for the rest of this thesis, when referring to MC, we mean bounded MC. Given a bound k, it first generates a propositional formula of R that is violated iff there exists a counterexample of length k. This formula is encoded such that it can be evaluated as a SAT/Satsfiability Modulo Theorem (SMT) problem. Towards that, the states of the model are represented as vectors of Booleans, such that the MC problem is reduced to evaluating the satisfiability of a bit vector equation.

Example 3-1

The SAT/SMT formula in bounded MC is generated by first generating a Static Single Assignment (SSA) form of the program, followed by generating bit vector equations and bringing them into Conjunctive Normal Form (CNF) for the SAT/SMT solver. Consider the following example taken from [CKL04], where the `assert` specifies the property R to be verified:

```
1 x = x + y;
2 if (x != 1) {
3   x = 2;
4 } else {
5   x++;
6 }
7
8 assert(x <= 3);
```

$\xrightarrow{SSA}$

```
1 x1 = x0 + y0;
2 if (x1 != 1) {
3   x2 = 2;
4 } else {
5   x3 = x1 + 1;
6 }
7 x4 = (x1!=1)?x2:x3
8 assert(x4 <= 3);
```

$\xrightarrow{bitvector}$

$$
\begin{aligned}
C :=& x_1 = x_0 + y_0 \wedge \\
& x_2 = 2 \wedge \\
& x_3 = x_1 + 1 \wedge \\
& x_4 = \text{if } (x_1 \neq 1) \text{then } x_2 \text{ else } x_3 \\
R :=& x_4 \leq 3
\end{aligned}
$$

The resulting two equations for C – representing the program constraints and R – representing the property to be checked – are then turned into a SAT problem as $\varphi = C \wedge \neg R$. Next, φ is shaped into CNF using standard Boolean algebra, and fed to a SAT/SMT solver, which tries to find assignments for the variables to satisfy the formula. If such an assignment exists, then the program can violate R, and it can be used to construct a counterexample. If no assignment is found, then the program satisfies R up to the bound k. ■

If loops are encountered, they must be *unwinded*, which is the equivalent to loop unrolling in a compiler. The model checker copies the loop body n times, each instance surrounded by the loop condition that may or may not be satisfiable. Since we operate under a bound k, loop unwinding stops when n reaches k. Similarly, function calls are expanded (the equivalent to inlining in the compiler), and recursion is handled like loops [CKL04]. Specifically, loop unwinding itself does *not* lead to an exponentially large number of paths, but it is only linear in the depth and size of the program [DKW08].

Another advantage of bounded MC is, that it can build on mature SAT/SMT solver technology, which has made great progress in the past years, and developed a number of different solvers (e.g, *minisat*, *yices*, *boolector*). Therefore, tools for bounded MC usually support a variety of solver backends.

3.3.2 Soundness, Completeness and Precision

It is sometimes argued that (bounded) MC cannot be used for formal verification, since the bounding renders MC unsound. For example, a property R could be violated on a path with length $k+1$, and therefore we could miss a violation of R (unsound), or vice versa fail to prove compliance with R (incomplete). However, one solution is to compute a *completeness threshold*. This is a bound k_c, such that if no counterexample of length k_c or less to a property is found, then the property in fact holds over all infinite paths in the model [KOS+11]. As a side note, completeness might still not be guaranteed, subject to the abstractions being performed when the model has been constructed. Computing the completeness threshold is hard in general, but a practical solution exists in the case of software analysis. We can start by guessing a bound k and run MC to explore all paths up to depth k. When reaching the bound, we add a new property R', which is violated by all paths that exceed depth k, and let MC evaluate R'. If R' can be proven (i.e., is unreachable), then k is indeed the completeness threshold of the program under analysis. In practice, k is defined on the number of loop iterations,

and MC tools are trying to guess k by a syntactic analysis of loops, often alleviating the user from having to guess the completeness threshold [BHvM09]. Completeness can however only be expected for shallow or abstracted programs, since otherwise the completeness threshold might be unattainable. Therefore, we consider (bounded) MC as a sound analysis technique in practice, which is – as any other analysis method in practice – subject to certain assumptions, and can even help to prove these assumptions.

Regarding precision, it should be noted that bounded MC is maximally precise if the initial abstractions during the buildup of M are not removing information relevant to the verification outcome. It is fully flow-, path- and context-sensitive, since it models the semantics along the CFG, unwinds loops, and expands all function calls.

3.3.3 Complexity

As discussed, bounded Model Checking builds up the state space up to a certain depth, and this state space can be infinite. Its cardinality is given in $\mathcal{O}(|l| \times |env|)$, which is known as the state space explosion problem. Furthermore, the decision procedure is (double) exponential in complexity [CKOS04], although fast SAT/SMT solvers can solve large problem instances. MC therefore must be expected be intractable on programs which have a large state space, viz., a large number of variables and statements. As a consequence, we may be forced to either select a bound below the completeness threshold and thus loose guarantees, or to apply transformations or abstractions to the program under analysis, to reduce its state space.

Predicate Abstraction. One way to enhance scalability of MC, is to apply *predicate abstractions*. These are abstractions specific to the program under analysis, in contrast to the abstractions used in AI. The goal is to retain just as much information as necessary to decide a property, and thus to loose as much precision as possible without changing the verdict. One such successful abstraction is *counterexample-guided abstraction refinement* (CEGAR) [DKW08]. It starts with a (most) abstract model, and refines (adds more information to) the model whenever a counterexample is generated that does not exist in the concrete execution of the program [DKW08], which is determined by simulation. However, since this refinement loop might not terminate, we do not consider predicate abstraction here.

3.4 Abstract Interpretation (AI)

AI was introduced in 1977 by Cousot and Cousot [CC77], as a generic, unifying framework for sound static and approximative analysis and verification of computer programs. The goal is to analyze a program's behavior under all possible inputs and execution paths, and to deduce invariants at the program locations. It is often used for range analysis of variable values (e.g., in compilers), but not limited to that. Its purpose is to address two problems in static program analysis: (1) Because of the *Halting Problem*, the termination of programs in undecidable in general, and so are some properties in static analysis. AI is a method that is designed to always terminate by making use of overapproximations. (2) Any static analysis that tracks/simulates all possible execution paths individually, would have exponential complexity. AI is designed to avoid having to analyze all possible paths. Towards that, AI employs abstractions of the concrete program states, by computing invariants from all paths leading to one location, and summarizing them as one abstract state that subsumes all feasible states. These abstractions are sound overapproximations of the collecting semantics, such that the output of AI can be used for any sound analysis following thereafter. However,

because of the overapproximation, AI is not complete. We can always find a program on which the abstraction removes critical information for the property R to be proven, such that the proof fails in the analysis, although the concrete semantics do not violate R. Furthermore, AI cannot generate a counterexample, due to loss of information in the abstraction.

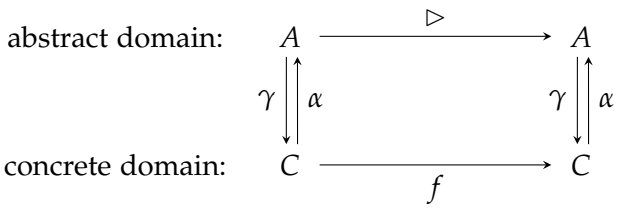


Figure 3.2: Basics of Abstract Interpretation.

AI computes the invariants on the abstract domain A, as illustrated in Fig. 3.2. It represents the program's concrete semantics $f : C \to C$ by an abstraction $\triangleright : A \to A$. Specifically, c is usually an element in the collecting semantics as described in Section 3.1, representing a *set* of possible program states, and $a \in A$ is a single abstract value representing all the states in c. Towards this, AI takes sets of concrete states, and uses an *abstraction function* $\alpha : C \to A$ to map them to a value a in the abstract domain, which models a superset of the semantics in the concrete domain. The opposite mapping from abstract to concrete domain is called a *concretization function* $\gamma : A \to C$. A necessary condition for AI to be sound, is that f and $\triangleright$ must agree, that is

$$\forall c \in C, \forall a \in A. \quad c \subseteq \gamma(a) \Leftrightarrow \alpha(c) \sqsubseteq a, \tag{3.10}$$

where $\sqsubseteq$ is abstract equivalent of the concrete subset operator $\subseteq$. This property is known as *Galois Connection*, which ensures that $\alpha(c)$ is a correct and sound abstraction of c, and vice versa for the concretization. It further requires the functions γ and α to be monotonous. The abstract semantics of Δ can be systematically computed from the concrete semantics f and this Galois connection.

The details of how to construct these functions are not further discussed here, since we do not require defining new abstract semantics in this thesis. Crucially, however, the default choice during the construction of abstract domains is to discard relations between variables. Let $\mathcal{P}_l(Var \to \mathcal{D}_{Var})$ denote the collection of variable states at some program location l as determined by the collecting semantics, then

$$\mathcal{P}_l(Var \to D_{var}) \underset{\gamma}{\overset{\alpha}{\rightleftharpoons}} Var \to \mathcal{P}_l(\mathcal{D}_{var}). \tag{3.11}$$

In other words, while the collecting semantics captures entire memory states and therefore relations between each pair of variables at each location l, whereas abstraction usually represents these sets of states as *one* set of variables, where each variable has a collection of possible values at location l. AI therefore looses the relation between variables, unless special *relational* domains and abstractions are used.

In AI, both the set of concrete states $c \in C$ and abstract states $a \in A$ are elements on a *lattice*, that is, a partially ordered set in which each two elements have one Least Upper Bound (LUB) and one Greatest Lower Bound (GLB). Each program step $\in\!\!\to$ is then represented by abstract operations $\triangleright$ on that lattice. Last but not least, the lattice is extended by two elements $\top$ and $\bot$, which ensure that each two subsets of the lattice have a defined LUB ($\bot$) and GLB ($\top$),

if no other element in the lattice qualifies. Since a lattice is finite, this means that for each operation in the abstract domain the result can only stay the same abstract element, or grow wider, until the top-most LUB of the lattice is reached.

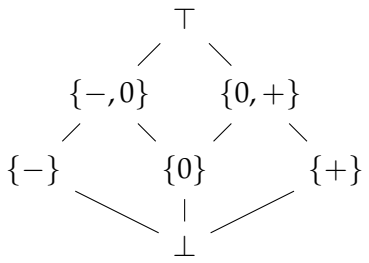


Figure 3.3: The complete *sign lattice* for AI.

Example 3-2

The lattice is chosen according to the needs of the required static analysis. Let us assume that we are interested in whether variables hold negative, positive or zero values at each program location. We could use the *sign lattice* as shown in Fig. 3.3. Here, the abstract domain is induced by three states (1) the value is negative ($\{-\}$), (2) the value is positive($\{+\}$), or (3) the value is zero ($\{0\}$). All values that a variable might carry can only be represented by those three cases, that is, AI on this domain does not distinguish between value -127 and -3,809, since these would both be represented by $\{-\}$. The remainder of the lattice defines combinations over these three abstract elements, given by the LUB. For example, the element $\{-, 0\}$ is the LUB of elements $\{0\}$ and $\{-\}$, and represents all concrete states where the variables are either negative or zero. Finally, $\top$ is called the *top* element and subsumes all states, and $\bot$ is called the *bottom* element, usually representing an unknown state. ■

Another classical abstract domain is that of intervals [CCF+05], which is also used in the tool Frama-C [CKK+12] that we evaluate later on.

Once the abstract lattice and the Galois connection have been chosen, running AI on a CFG works as follows. AI is based on a fixed-point iteration over the statements in the CFG. Given the abstract state on entry of one BB in the CFG, AI applies $\triangleright$ to compute the effect of the contained statements until the end of the BB is reached. The resulting abstract state is then "forwarded" to the successors of that BB, for which the same procedure is applied. If a BB has multiple predecessors, then the multiple abstract states are consolidated into a single abstract state that represents their LUB, before the $\triangleright$ is applied again. This is propagation is repeated over all BBs until the abstract states at each locations no longer change, i.e., reach a least fixed point.

It follows that the second source of imprecision in AI comes from joining paths in the CFG, since the LUB (sometimes called JOIN) can also loose precision. For example, assume we are running AI analysis using the lattice of integer intervals, and that the two abstract states $s_1 = (l_1, env_1)$ and $s_2 = (l_2, env_2)$ contain only $var_a \in env_1 = [-10, -5]$ and $var_a \in env_2 = [2, 4]$. Their merged state $s_{1,2} = LUB(s_1, s_2) = (l_{1,2}, \{var_a \mapsto [-10, 4]\})$ now contains the whole interval from -10 to 4, although the values $[-4, 1]$ are not part of the collecting semantics at $l_{1,2}$, i.e., infeasible. Consequently, AI generally overapproximates whenever multiple paths in the CFG are joining.

3.4.1 Widening

If all value domains are countable and finite, then the fixed-point algorithm would in principle always lead to converging values at all locations l, and thus terminate. However, this may take a long time. The concept of *widening* accelerates the convergence and ensures that there is a least fixed point in all cases. Specifically, the abstract value sets are overapproximated to fixed points after a certain number of analysis steps, often defined by the number of loop iterations.

While such widening ensures termination of the analysis and further also reduces analysis time, it can cause loss of precision. For example, if a loop iterates $n > m$ times, and widening is applied after m loop iterations, then all variables modified within the loop can be overapproximated.

3.4.2 Soundness, Completeness and Precision

Abstract Interpretation is sound by construction, but, as any other static program analysis, incomplete. AI is therefore suitable for functional verification, and only impeded by the loss of precision that it entails. AI-based analyses are usually flow sensitive but not path-sensitive. That is, the order of statements is followed, but merging paths in the CFG via the LUB discards path-sensitive information. Further, depending on how a tool implements AI, context-sensitivity may or may not be provided. Therefore, AI usually can only verify simple properties, due its loss of precision.

However, the three identified sources of imprecision, namely missing relations between variables, joining paths in the CFG and widening, do not necessarily manifest in every program. First, besides the use of relational domains to prevent the first problem, variable relations may not always play a role in verification of all properties. Second, the loss of precision at joining paths only happens when (a) the program under analysis is a multi-path program, and (b) the abstract domain cannot precisely represent the LUB of both merging states. Third, widening is only expected in programs with loops that iterate a large or unknown number of times, and can usually be controlled by the user. It follows that AI is likely to yield precise results on single-path programs, when the abstract domain is suitably chosen and widening carefully controlled. On the other hand, AI can be expected to include spurious program states when the program has multiple paths or many variable interdependencies.

3.4.3 Complexity

The model M that is being used has a reduced state space compared to the original state space of F, since AI is not path sensitive. It performs a path summary at each program location, and therefore collapses the product $l \times env$ to a state space in the order of $O(|l|)$. The decision procedure of AI has a linear to quadratic complexity [KW92] in the best case. In practice the scalability depends much on the chosen domain and its implementation. The use of relational domains can increase decision complexity to an exponential rate, but is not considered in this work. AI can therefore be used for large programs, and specifically to deduce value invariants at program locations.

3.5 Deductive Verification

This approach to formal verification of software, sometimes also called *program proving*, formulates desired properties of a program as mathematical statements, and then attempts to prove these statements through mathematical reasoning. While this idea can be traced back to program proofs made by Alan Turing (1949), via Hoare's axiomatic basis [Hoa69] and Dijsktra's seminal paper from 1975 on a formal calculus for program analysis [Dij75], mechanized deductive verification only became possible in the last few decades, with languages, methods and tools appearing that allow performing such mechanized proofs efficiently [Fil11].

Deductive verification requires a specification language in the background. This is a mathematical language to model the program behavior, which is also integrated into the programming language in which the program is written. The latter enables to specify *contracts* for subprograms, such as preconditions, postconditions, assertions and so on. This language is usually a first-order logic [Fil11], and therefore Deducive Verification (DV) allows specifying complex properties beyond just overflows, and can prove the correctness of a whole program w.r.t. its specification.

The basic idea is model each subprogram F as a Hoare triplet [Hoa69] $\{P\}F\{Q\}$, where P and Q are Boolean predicates of the program environment before and after the subprogram. Specifically, P is the *precondition*, expressing allowed inputs to F (assumptions), and Q the *postcondition*, expressing the output that F provides (guarantees), i.e. [Fil11]

$$\forall env.\ P(env) \Rightarrow Q(env, F(env)). \tag{3.12}$$

However, P and Q are just specifications, and the goal is to verify whether F can operate accordingly. That is, given P, does F always terminate, and if so, does it always satisfy Q? In a larger context, F can be any stretch of a program, even a single statement. Therefore, Q can represent a property R that the user wants to verify, e.g., whether the result of a calculation is never an overflow, or whether a variable carries a certain value at a program state.

Weakest Precondition (WP). Let F be a code fragment (e.g., a subprogram), and Q the program state thereafter. Then the *Weakest Precondition* $wp(F, Q)$ is an assertion that is true for precisely those initial states for which F terminates (total correctness) and Q is satisfied. In other words, $wp(F, Q)$ is the necessary and sufficient Boolean predicate just before F executes, such that F produces the expected result Q [Dij75].

The workflow in DV is then as follows:

1. Identify property Q. Some properties can often be inferred automatically, e.g., no numeric overflows, or no division by zero.
2. Specify P by subprogram contracts or other means. If P is left unspecified, it is usually implicitly defined as all semantically possible environments at the current location. E.g., if F is a function and P is unspecified, then P is implicitly given by its parameter types.
3. Compute $wp(F, Q)$ by reasoning backwards from Q to the beginning of F, to deduce the necessary and sufficient conditions for Q to hold true.
4. Generate a mathematical statement, a *verification condition* (VC), which is a proposition that, when provable, guarantees that the WP is always satisfied, and by transitive property, that Q is always satisfied.
5. Attempt to prove the VC mathematically. If the VC can be proven ("discharged"), then indeed F can operate as intended.

In contrast to AI, no iterations over the CFG take place, but instead a theorem prover is used, applying proof strategies. Similarly to (bounded) MC, DV eventually uses SAT/SMT solvers as backends. The details of this proving process are not of relevance for this thesis, and therefore we refer the reader to [Fil11] and [FP13].

Example 3-3

The WP is computed backwards from a required postcondition to the beginning of the subprogram, using *Predicate Transformer Semantics* (also called *WP Calculus*) [Dij75]. Consider the following program:

```
1 // -3 ≤ a ≤ 15 = wp(..., Q)
2 v1 = a + 5;
3 //  2 ≤ v1 ≤ 20
4 v2 = v1 / 2;
5 assert(1 ≤ v2 ≤ 10); // = Q
```

We start in line 5, which specifies the postcondition Q of this code fragment, and doubles as the property R that we want to verify. Next, we reason backwards. In order to satisfy this assertion, we evaluate the necessary inputs for the preceding statement in line 4, which are $2 \leq v_1 \leq 20$. Similarly, we now propagate this condition backwards to the necessary inputs of the statements in line 2. The result is the weakest precondition, for the inputs of F, such that Q is satisfied and F terminates. The latter can make difficulties in loops.

As a next step, a VC is generated as $P \Rightarrow wp(F, Q)$, i.e., the proposition that the known precondition implies the weakest precondition. In our program above, assume that the known precondition is $P := 1 \leq a \leq 10$. Then the verification condition would become $1 \leq a \leq 10 \Rightarrow -3 \leq a \leq 15$.

Finally, similar to bounded MC, this proposition can eventually be encoded as a SAT or SMT problem $\varphi = P \wedge \neg wp(F, Q)$. If the solver finds a satisfying instance, then this demonstrates how the specified precondition can violate the weakest precondition, and therefore that the property R is possibly violated. In that case the program must be corrected, either by reworking F, or by changing P. Towards this, it is possible, although more complicated than in MC, to compute a counterexample from the satisfying assignment [KMMS16]. ■

3.5.1 Complexity

Similarly to MC, DV without any abstractions is a fully precise verification approach, and therefore has a state space with a size in the order of $(|l| \times |env|)$. Furthermore, is has been shown that a short proof (i.e., one of polynomial length in the size of the program) is not guaranteed to exist even for simple specification languages [CR79]. Despite these complexity challenges, DV has been successfully used in practice in a number of projects [CS14, Fil11].

3.5.2 Soundness, Completeness and Precision

As the other methods, DV is sound by construction, but incomplete. Theorem proving is not fully automatable because of the Halting problem and decidability. It therefore needs human intervention for some proofs, or must otherwise raise False alarms, e.g., when it cannot prove that the user-provided invariants are inductive. One infamous obstacle are loops, which often require the user to specify *loop (in)variants*, which are hard to come up with. Therefore, some tools [HMWC15] apply strategies similar to unwinding, to handle at least some loops automatically.

Due to the comparatively low scalability of theorem proving, verification often requires *assume-guarantee* reasoning. That is, subprograms are often analyzed individually and in a context-insensitive fashion, by proving their postcondition only under the assumption that the precondition holds, while disregarding actual call parameters. Vice versa, postconditions are then used in lieu of the callee's body when analyzing the caller, therefore building on the assumption that the callee's postcondition holds true. As a consequence, if the precondition of a callee is violated, then its postcondition may no longer hold, and vice versa, if a callee violates its postcondition, then the proofs in the caller might be refuted. This modularization, while allowing for incremental development and verification, and also appearing quite natural and useful for assigning responsibilites during software development, therefore leads to a loss of precision and may lead to unsound results, unless all VCs can be discharged.

3.6 Chapter Summary

This chapter has introduced the formal background for this thesis, in particular the used terminology and the three analysis methods *Model Checking*, *Abstract Interpretation* and *Deductive Verification*. We have seen that all of them are *sound*, that is, any property that they claim to hold true is never violated by the program under any inputs, but that they differ in their computational complexity, degree of automatism and precision.

Chapter 4

Evaluation & Selection of Formal Verification Methods

***

In this chapter we review current tooling for the verification methods *Model Checking (MC)*, *Abstract Interpretation (AI)* and *Deducive Verification (DV)*. The goal is to identify boundary conditions for the tools, to learn which class of properties can and cannot be proven, and to rank them with respect to their suitability for source-level timing analysis.

Since the main concern of formal verification methods is *functional* correctness, we first and foremost conduct our evaluation under functional aspects, and later discuss how these methods would carry over to analyze *temporal* correctness. Towards that, this chapter presents three case studies, one for each method, whereas two of them took place in the context of real-world, autonomous and safety-critical embedded systems that we have developed.

4.1 Case Study 1: Model Checking of C Code

This section summarizes our case study published in [uNSC15], which uses MC for formal verification. We verified an automatic parachute rescue system for small drones, as depicted in Fig. 4.1, to ensure the system operates correctly under all conditions. It is a mixed hardware-software solution, which can be "plugged-on" and can bring down the drone safely in case of a malfunction. The only interface to the drone is the power line, for which the parachute system acts as a proxy. If it detects an emergency through its own sensor readings (barometer, acceleration sensors), it cuts off the power to the drone (to stop the spinners) and ejects the parachute. Therefore, the system must operate correctly and with as little as possible false alarms, to not impair the reliability of the drone.

Static formal verification is an attractive approach to ensure the correctness of this system, since it can be exposed to a variety of operational conditions. Testing therefore is hard (setting up that environment) and furthermore not desirable (crashing drones). In an ideal case, all testing can be replaced by static analysis, such that we are guaranteed the absence of errors.

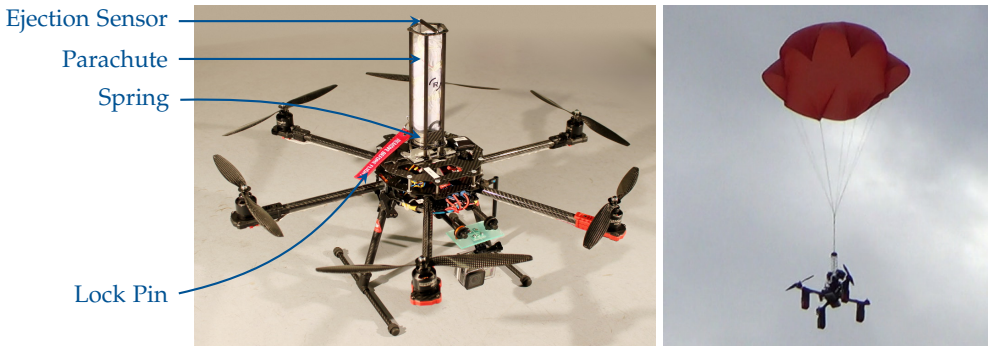


Figure 4.1: Prototypes of our *Emergency Recovery System* mounted on a hexacopter (left) and deployed on a quadrocopter (right).

4.1.1 Tools and Methods

We have chosen the model checker *cbmc* [CKL04], due to its robustness and consistent performance in recent software verification competitions [Bey14]. It is a *bounded* model checker which accepts models and properties in the form of ANSI-C programs. More model checkers for C code were compared in [SK09] and [Bey14]. Some important features of cbmc that we use here, are:

- **Assertions**: cbmc allows expressing properties with assertions. In our case, these assertions are of the form `assert(_time<X)`, where the constant `X` is the proposed WCET, and `_time` denotes the counter variable reflecting the time passing by in the program.
- **Non-determinism**: cbmc allows any of the program variables to be assigned a non-/deterministic value. This is done using assignments of the form `y=nondet()`, where `nondet()` is an undefined function having the same return type as `y`. This results in `y` being assigned a non/deterministic value from the range specified by its type.
- **Assumptions**: cbmc allows the use of `assume` statements to block the analysis of undesirable paths in a program. A statement of the form `assume(c)` marks all those paths infeasible for which `c` evaluates to `false` at the execution of this statement. This feature can be used to constrain the value domain of non/deterministic assignments.
- **Checking multiple assertions**: cbmc allows multiple assertions in the input program, which can be checked at once using the `--all-properties` option. This option uses an optimal number of solver calls to verify programs with multiple assertions.

Before explaining the verification process, we briefly introduce the system under analysis.

4.1.2 Emergency Recovery System for MAVs

The parachute system, called *Emergency Recovery System* (ERS) was developed for Micro Air Vehicles (MAVs), a class of light-weight, predominantly civil drones. No modifications to the existing MAV are required, e.g., neither altering the flight controller nor the propulsion system. Our system effectively acts as a power proxy between MAV battery and MAV. The only (necessary) interface for our ERS is the power connector, which is why we call it a "plug and play" solution. A second optional interface is for one RC channel, allowing the pilot to trigger the parachute manually.

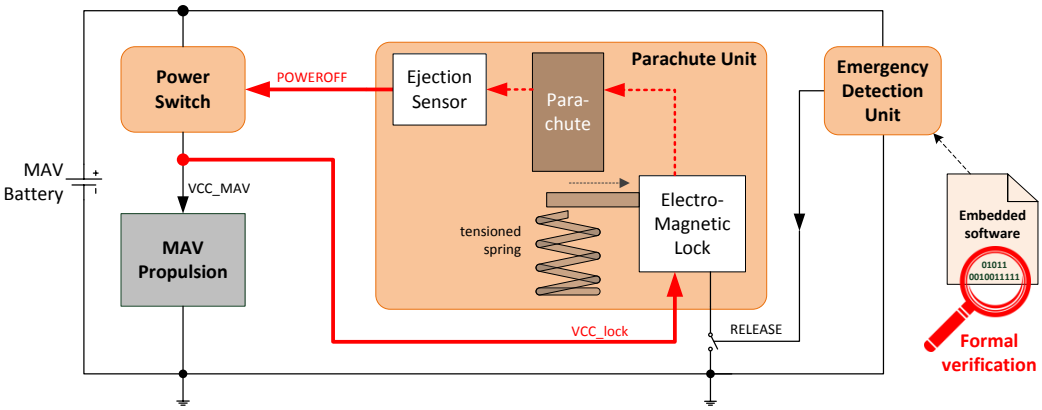


Figure 4.2: Internal Structure of our *Emergency Recovery System*: The *Emergency Detection Unit* on the top right is running the to-be-verified software.

4.1.2.1 Internal Structure

The ERS consists of the following three components, as illustrated in Fig. 4.2:

- **Emergency Detection Unit (EDU):** A Printed-Circuit Board (PCB) with sensors and a microprocessor running software to detect emergencies. In case it detects an emergency, it can trigger the ejection of the parachute.
- **Power Switch (PS):** A PCB with power electronics, acting as a proxy between the MAV's battery and the propulsion. In case of emergency, it cuts off the power.
- **Parachute Unit (PU):** This is a housing holding the parachute. It is also comprising an ejection sensor and an electro-magnetic (EM) lock, which, when opened or powerless, releases a compressed spring, which in turn ejects the parachute.

Mode of Operation: The EDU features an Atmel ATmega 328p microprocessor (Harvard, 8 MHz, 32 kB Flash, 2 kB RAM, no caches), a barometer sensor and an accelerometer sensor. The embedded software evaluates those sensors periodically, and estimates the MAV's air state. When it detects emergency conditions, it triggers the parachute ejection by emitting a `RELEASE` signal, which opens the EM lock. This releases a compressed spring, which can now eject the parachute from its housing. Simultaneously, when the parachute is pushed out, an ejection sensor detects this and sends a `POWEROFF` signal to the *Power Switch*. This ensures, that the MAV's propulsion is deactivated as soon as the parachute is ejected.
Emergency Conditions: The root causes for failure in MAVs are wide-spread. Due to tight integration of functionality and the imperative minimalism in redundancy, even errors in non-critical components can evolve quickly into fatal failures. Therefore, it seems more efficient to apply a holistic monitoring, instead of monitoring single components. Accordingly, an emergency is considered as the MAV being *uncontrolled*, that is, when the pitch or roll angles exceed user-defined thresholds, or when the descent rate gets too high. These conditions cover the most important malfunctions, such as Flight Control System (FCS) failure (e.g., badly tuned controllers or error in software logic), electrical or mechanical failure of propulsion (propeller, Electronic Speed Controller (ESC)), loss of power and partially even human error (in the form of initiating an uncontrolled state).
Influence of the Software: The fault tree of the MAV with parachute system is depicted in Fig. 4.3. It can be seen, that the three *uncontrolled* system states which lead to a crash, can only

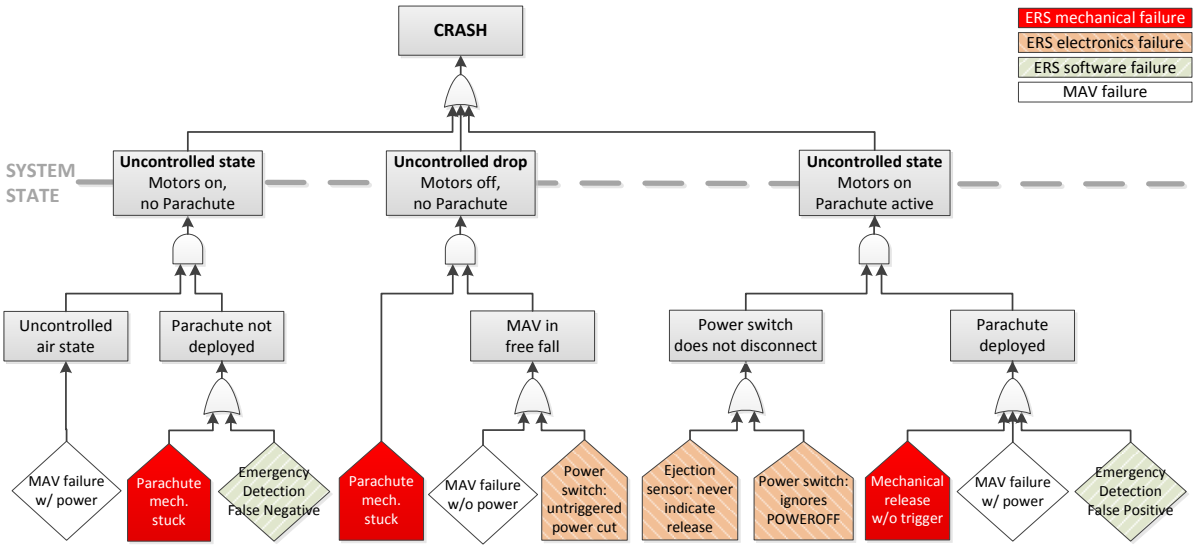


Figure 4.3: Fault Tree for the top event "crash", valid for any electric Micro Air Vehicle equipped with our Emergency Recovery System.

be reached if at least two failures occur at the same time. As indicated with the color coding, there are four categories of failures: (a) mechanical failure in ERS (red), (b) electronics failure in ERS (orange), (c) software failure in ERS (green) and (d) MAV failure (white). Although there are many kinds of errors possible in software, from a system point of view we are only interested in the two consequences depicted in the fault tree:

- **Emergency Detection False Negative**: The embedded software does not trigger the emergency sequence despite emergency conditions.
- **Emergency Detection False Positive**: The embedded software does trigger the emergency sequence without emergency conditions.

While both software failure events can have the same impact at system level (both can lead to crash if a second failure occurs), the case of a *False Negative* is practically more critical, since MAV failures with power are more likely than a second independent failure occurring in the ERS. Furthermore, the ERS runs self-checks during initialization, reducing the probability of being used in the presence of internal failure. For these reasons, our verification efforts that we explain in the next section, focused on (but were not limited to) finding defects that lead to False Negatives.

4.1.2.2 Software Structure

The software running on the EDU is therefore our program under analysis. It can be partitioned into four sequential parts:

1. **Initialization:** Initializes all sensors, and captures environmental conditions (e.g., pressure at ground level). When completed, the ERS switches to *self-check mode*.
2. **Self-Check:** To ensure that there is not already a failure in the ERS during start-up, we added built-in self tests covering the major subsystems of the ERS. When completed, the ERS switches to *detection mode*.

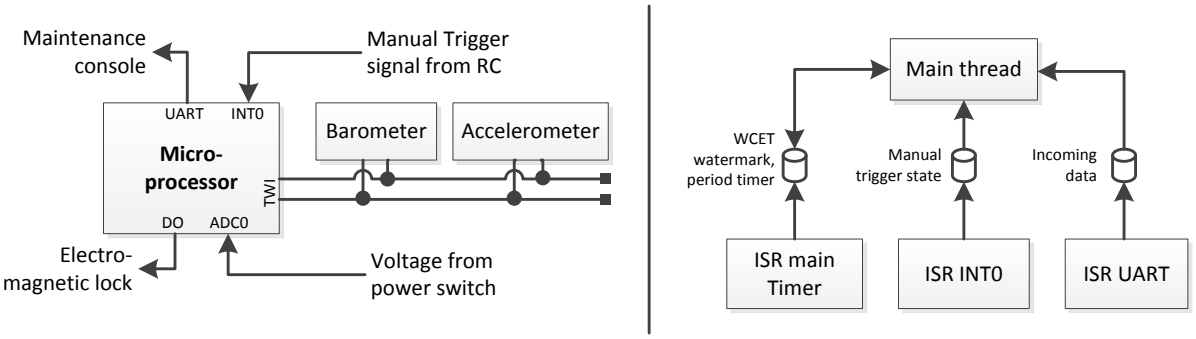


Figure 4.4: Microprocessor with interfaces to its environment (left) and the resulting concurrency in the software (right).

3. **Detection:** The software periodically reads all sensors and estimates the MAV's air state. If the emergency conditions apply, the EM lock is released and the software switches to *emergency handling mode*.

4. **Emergency Handling:** Current sensor data and decision conditions are written to EEPROM, to enable a post-flight analysis.

The sensors and actuators are connected to the microcontroller as depicted in Fig. 4.4 on the left. The interfaces impose some concurrency in the software, which is shown on the right. For example, the maintenance console and manual trigger signal both require interrupts (polling would be too slow), thus each introduces one thread concurrent to the main program. Additionally, a timer interrupt is used to support a time-triggered execution of the detection loop, contributing one further thread.

4.1.3 Verification Process

We continue with the verification goals, the process and encountered difficulties, followed by the results.

4.1.3.1 Verification Goals

Proper Timing: The mentioned concurrency poses the first verification task. To ensure that the detection loop always runs at the desired rate – which is important for correctness of computed data, e.g., the descent rate – we need to show that the required computations can be completed in all cases, before the next period begins.

Towards that, the WCET of the main loop must be determined. Here we took a dual approach: On one hand, we performed a static WCET analysis with a freely available analyzer tool [HS02], but we also monitor the execution time on the microprocessor with a *high watermark*.

For the static analysis we made the assumption that the sensors are healthy, and follow their datasheets' timing specification. The resulting WCET was 2.7 ms for the detection loop, which is well below the 5 ms-period in the EDU. However, interrupts also need to be considered. The *worst-case response time* (WCRT) is (in this context) the maximum amount of time that the detection loop needs to finish processing, under the preemption of interrupts. Only if the WCRT is less than the period, then it can be concluded that the timing is correct.

However, without further provisions the minimum inter-arrival time (MINT) for the event-based interrupts (manual trigger from RC, UART) have no lower bound, i.e., it would be

possible that a broken RC receiver or UART peer could induce so many interrupts, that the detection could never execute, resulting in an unbounded WCRT. To avoid this situation, the inter-arrival times of all event-driven interrupts are also measured in the microcontroller. If an interrupt occurs more often than planned, the attached signal source is considered failing, and the interrupt turned off.

With these bounded MINTs and the WCET values from the static analysis, a standard response time analysis yielded a WCRT of 2.89 ms for the detection loop. Again, this is for the case of healthy sensors.

The purpose of the high watermark is to detect those cases when sensors are failing, but also to gain confidence in the above analysis. The response time of the detection loop is continuously measured using a hardware timer, and maximum values are written to EEPROM. With rising number of flying hours, the watermark should approach the WCRT. If it exceeds the statically computed WCRT, then a sensor failure is likely, which triggers the emergency sequence.

In practice, the watermark measurements were observed approaching the statically computed WCRT up to a few hundred microseconds with healthy sensors, thus giving confidence in the analysis. By construction of the software, it can be concluded that the timing of the detection loop is correct, unless the parachute is deployed. However, there are more timing-related issues to be considered, namely, the time-sensitive effects of interrupts upon the control flow in the main program. This was addressed later during the verification process.

Proper Logic: The ultimate goal of the software verification is to ensure that the emergency detection algorithm works as intended. As explained before, the main concern was to avoid False Negatives, i.e., the error that the embedded software does not trigger the emergency sequence, despite emergency conditions.

An obvious reason for such failure is, that the software is not running because it crashed or got stuck. This can be a consequence of divisions by zero, heap or stack overflow[1], invalid memory writes, etc. Note that a reboot during flight is not possible, since the initialization and self-checks need user interaction (open and re-close the ejection sensor to ensure it works correctly), and making them bypassable is not desirable for practical safety reasons. Therefore, crashes and stuck software have to be avoided.

The second reason for not recognizing an emergency is an incorrectly implemented detection algorithm. This entails both an error in decision taking (i.e., which sensor has to tell what in order to classify it as emergency), and also numerical problems (e.g., overflows) in data processing. Identifying these kinds of problems also decreases the number of False Positives.

The majority of those defects is checked automatically by *cbmc*, if requested during instrumentation. The correctness of the decision taking part, however, must be encoded with user assertions. Since our detection loop runs time-triggered, properties such as “latest 100 ms after free fall conditions are recognized, the parachute shall be deployed“ can be encoded with some temporary variables. With that, verification of arbitrary properties of the decision algorithm follows the same workflow as the automatically instrumented properties, which is why we do not elaborate on the specific properties that were eventually verified, but rather show how we set up the workflow correctly.

[1] Heap was not used, and stack size was checked with *Bound-T*.

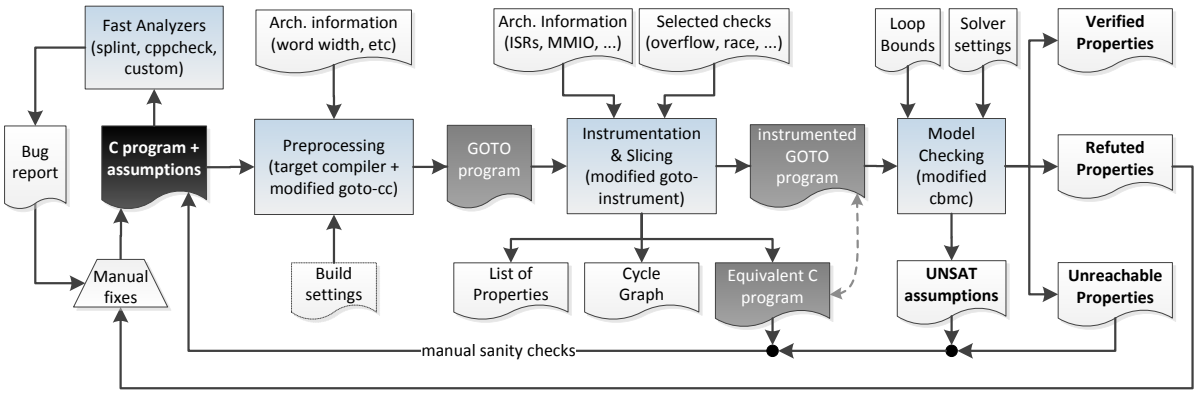


Figure 4.5: Workflow for formal verification of the embedded software written in C.

4.1.3.2 Verification Workflow

The toolchain that we set up around *cbmc* is shown in Fig. 4.5. We start with a C program, written for the target. First, we run fast static checkers such as *splint* on the program, to identify and remove problems like uninitialized variables, problematic type casts etc. Not only does this help to avoid defects early during development and thus to reduce the number of required verification runs later on, but also it complements the verification. For example, the semantics of an uninitialized variable depends on the compiler and the used operating system (if any); *cbmc*, however, regards these variables as nondeterministic and therefore approximates the program without a warning.

After passing the fast checks, the C code is given to *goto-cc*, which translates it into a *GOTO-program*, basically a control flow graph. During this process, all the macros in the C code are resolved by running the host compiler up to the preprocessing stage.

The *GOTO-program* is subsequently fed into *goto-instrument*, which adds *assert* statements according to user wishes. For example, each arithmetic multiplication can be checked for overflow, array bounds can be ensured, divisions by zero can be detected, and so on. Note that the original code may contain user-defined assert statements, which are preserved.

The resulting *instrumented GOTO-program* is finally handed over to *cbmc*, which performs loop unwinding, picks up all *assert* statements, generates VCCs for them and – after optional simplifications such as slicing – passes the problem to a solver back-end (here we use *minisat2*; SMT solvers like *z3* and *yices*, were experimental at the time of this case study).

After the back-end returns the proofs, *cbmc* post-processes them (e.g., building the counterexample, if any) and provides a list of verified properties, and for each refuted one a counterexample. These lists can be used to fix defects in the original code, clearing the way for the next iteration.

4.1.3.3 Missing Architectural Information

A problem in static verification is implicit semantics that depends on the target, for example that certain functions are set up as Interrupt Service Routines (ISRs) and thus their effect needs to be considered, although they never seem to be invoked. Another example is memory-mapped I/O, which may seem like ordinary reads from memory, but in fact could inject nondeterministic inputs from the environment.

Neglecting such context can easily lead to a collapsing verification problem and result in wrong outcomes. In our program, there were initially 351 properties, from which 349

were unreachable due to missing contextual information. Annotating all the necessary places manually is an error-prone labour, which bears the risk of having wrong or missing annotations and more importantly it is practically infeasible for our small program already. In the following we discuss how we addressed this problem.

Accounting for Interrupts: The preprocessed C code contains the ISR definitions, but naturally no functions call to them. The ISR is only called because its identifier is known to the cross compiler, and because particular bits are being written to registers at the start of the program; something that the model-checker lacks knowledge of. Consequently, it concludes that the ISR is never executed, and – through data dependencies – our detection algorithm seems to be never executed. This makes all properties within that algorithm unreachable and thus incorrectly evaluates them as "verified".

To overcome this, a nondeterministic invocation of the ISR must be considered at all places where shared variables are being evaluated, as described in [BK11]. This can be done with *goto-instrument* as a semantic transformation (flag `--isr`). Fig. 4.4 shows the respective data that depends on interrupts in our case. Unfortunately, this technique not only grows the to-be-explored state space, but it even overapproximates the interrupts: The ISR could be considered too often in the case when the minimum inter-arrival time is longer than the "distance" of the nondeterministic calls (e.g., ISR for periodic timer overflow) that have been inserted. However, even if we would include execution time and scheduling information from parts of the main thread (to be computed by WCET and WCRT tools), the points in time where the ISR is called could be drifting w.r.t. to the main thread. This is true even for perfectly periodically triggered programs, solely due to different execution paths in the main thread.

Nondeterminism from Frequency-Dependent Side Effects: There exists another problem with interrupts that has not been addressed in [BK11] nor in `goto-instrument`. It stems from the frequency-dependent side effects of ISR invocation: In general, interrupts could also execute *more often* than the places where nondeterministic calls have been considered before. If there exist side effects other than changes to shared variables (i.e., if the ISR is non-reentrant in general), this can break the correct outcome of the verification. For example, ISRs that on each invocation increment some counter variable which is *not* shared with any other thread, could then in reality have a higher counter value than seen by the model checker[2]. In other words, all persistent variables that are manipulated by the ISR have to be modeled as nondeterministic, not only shared variables. In our case there were only three such variables (one was for the time-triggered release of the detection loop), which have been identified and annotated manually.

Memory-Mapped I/O: All I/O variables (the sensor inputs) must be annotated to be nondeterministic. One option for that would be using the flag `--nondet-volatile` for *goto-instrument* to regard all volatiles as nondeterministic, however, this results in overapproximation for *all* shared variables (which are volatile as well), allowing for valuations which are actually infeasible due to the nature of the algorithms operating on the shared variable. Furthermore, this can override user-defined assumptions on the value domain of sensors, considering actually impossible executions and thus produce false alarms on the verified properties.

[2] A *lower* value is not possible, because all considered invocations are nondeterministic possibilities, and not enforced invocations.

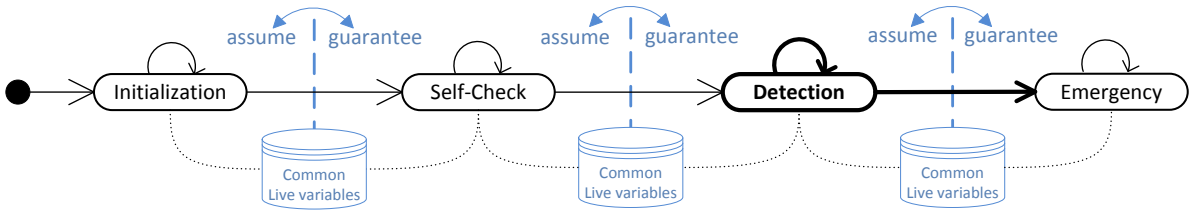


Figure 4.6: Partitioning of software into strictly sequential modes, each verified individually and cascaded using *assume-guarantee* reasoning.

In our case the microcontroller runs bare-metal code and uses memory-mapped I/O to read sensors, i.e., accesses show up in the preprocessed C code as dereferencing an address literal. In principle, it is therefore possible to identify such reads after the C preprocessing stage. However, in general it is a non-trivial problem to identify all these places, since indirect addressing is possible, which would require a full value analysis of the program to figure out whether the effective address is in the I/O range. At the moment we do not have a practical solution to this problem, which is why we instrumented all inputs manually. To support this process, we developed a *clang*-based [Lat08] tool which generates a list of all dereferencing operations, suggesting the places that should be considered for annotating nondeterminism in the C code. Since we minimized the use of pointers to keep verification effort lower, the majority of the entries in this list is indeed reading input registers.

4.1.3.4 Preprocessing against State-Space Explosion

After all architectural information has been added, the next big challenge is to verify the instrumented properties. A problem here is, that the state space grows rapidly from the architectural features, especially from the ISRs. In our case, the program has around 2,500 lines of C code, and running *cbmc* already fails for two reasons: (1) the program contains unbounded loops and (2) even if the loops were somehow bounded, there would be too many SAT variables to be considered (millions in our case).

Building Sequential Modes: The original structure of our program could not be verified, because the initialization and self-checks, were implemented as part of one hierarchic state machine, executed in main loop. The necessary loop unwinding then expanded the entire state machine as a whole. This resulted in too many SAT variables and could not be processed on our machine (we run out of memory after hours, having done only a fraction of the necessary unwinding).

To overcome this state space problem, we first partitioned our program into sequential modes, see Fig. 4.6. Each the initialization, the self-tests and the detection were refactored into their own loops, which take place one after another. Additionally, interrupts were enabled as late as possible, to reduce the number of states that have to be explored.

Assume-Guarantee Reasoning: However, at this point it turned out, that the initialization and self-checks still contributed too many variables for the program to be analyzed as a whole. As a countermeasure, the modes should now be analyzed independently and reasoning on the overall correctness should be done using *assume-guarantee* reasoning. Towards that, it was necessary to identify all possible program states between the modes, e.g., the detection mode can only be properly analyzed, if all possible program states after initialization and self-check are considered. One concrete example is, that the ERS determines the air pressure at ground level during the initialization, which is used later during detection. Verifying the detection

mode thus involves considering all possible pressure levels, by assuming nondeterministic values for them.

To reduce the complexity of assume-guarantee reasoning, we first turned each mode into a potentially infinite loop which can only exit, if everything works as expected. These "guards" reduce the number of program states to be considered for the successor modes. For example, when analyzing the detection mode, we only need to consider program states corresponding to *successful* initialization and self-checks.

To construct the program states between modes, we identified all *live variables* between each two successive modes, i.e., all variables which are written in one mode and possibly being read in its successor modes. As this is another error-prone work that should not be done manually, we extended our *clang*-based tool to take this step automatically.

After having identified the live variables at the end of each mode, we instrumented them as illustrated in Listing 4.1: First, we added a nondeterministic assignment to each variable just before the new mode starts (line 6). This allows for *all* possible values, once the analysis on the new mode starts. Then, if due to some logical reason the value range could be limited, we used an `assume` statement to restrict analysis to this value range (line 7). However, to *guarantee* that the value domain is indeed complete, i.e., ensuring that no possible execution has been neglected, we added a matching `assert` statement at the exit of the predecessor mode (line 3).

Listing 4.1: Illustration of assume-guarantee reasoning using *cbmc* at the program point between two sequential modes `X` $\rightarrow$ `Y`, sharing one live variable `sharedvar`.

```
1 // end of mode X
2 #ifdef ANALYZE_MODE_X
3   assert(sharedvar > -10.f && sharedvar < 50.f);
4 #endif
5 #ifdef ANALYZE_MODE_Y
6   sharedvar = nondet_float(); // introducing nondeterminism
7   assume(sharedvar > -10.f && sharedvar < 50.f);
8 #endif
9 // beginning of mode Y
```

A successful verification of the predecessor mode (here: `X`) means the `asserts` hold true, therefore *guarantees* that live variables indeed satisfy the assumptions we make at the beginning of the new mode (here: `Y`). Assume-guarantee reasoning therefore is sound. Finding the value ranges is currently done manually; in doubt one can omit the ranges, which leads to a safe over-approximation. However, tool support would be favorable, since having tight ranges means less false alarms during verification.

In summary, this mode-building reduced the number of properties from 458 to below 250 in each mode, with 31 shared variables between them that were subject to assume-guarantee process (see Table 4.1).

Removing Dead Code: When going through the verification process shown in Fig. 4.5, it is desirable to entirely remove dead code (especially after mode-building and analyzing the modes separately), otherwise a lot of unreachable properties will be there, slowing down the analysis and cluttering the results. Although *goto-instrument* offers two slicing options, none of them removes dead code. This task is not trivial, since in our case the modes share code, e.g, both self-check and detection use a function that reads out the accelerometer. Again, we used our clang-based tool for this task, which operates on the C code that is equivalent to the GOTO-program and removes dead functions and variables, see also Fig. 4.5.

Bounding Non-local Loops: A complexity-increasing problem for verification are nested, stateful function calls, as they occur in hierarchical state machines. Our program uses such hierarchical state machines to interact with the barometer and accelerometer peripherals. If

Table 4.1: Complexity of the verification before and after preprocessing. Unlike the full program, which cannot be analyzed, assume-guarantee reasoning between sequential modes *Initialization*, *Self-Check* and *Detection* was computationally tractable.

Mode →	Initialization	Self-Check	Detection	All
lines of code	1,097	976	1,044	2,513
#functions	36	29	43	94
#persistent variables	36	38	59	72
#live variables at exit	31	31	n.a.	n.a.
#properties	249	221	175	458
#VCCs	11,895	35,001	15,166	330,394
#SAT variables	5,025,141	8,616,178	6,114,116	n.a.
SAT solver run-time[a]	16 min	14 min	28 min	intractable[b]

[a] On an Intel Core-i7 vPro at 2.8 Ghz and 4 GB RAM.
[b] Out of memory after 3 hours; #VCCs and SAT variables were still growing.

one of the inner states has transition guards, then the *entire* hierarchy needs unrolling until these guards evaluate to true. In our case, we have guards like *waiting for Analog-Digital Converter (ADC) to finish*. Unfortunately, hierarchic state machines are a popular design pattern in model-based design (e.g., Statemate, Stateflow, SCADE), which therefore needs to be addressed rather than avoided.

We found that some guards in the inner state machines can be removed safely, reducing costly unrolling. Assume that the guard will eventually evaluate to true (even if there is no upper bound on the number of steps it takes): If all *live* data that is written *after* this point is invariant to the number of iterations, then the guard can be removed. Consequently, such irrelevant guards can be identified by first performing an impact analysis (find all variables that are influenced by the guard), followed by a loop invariance test (identify those which are modified on re-iteration) followed by a live variable analysis on the result (from the influenced ones, identify those which are being read later during execution). If the resulting set of variables is empty, then the guard can be removed safely. This technique is of great help for interacting with peripherals, where timing may not influence the valuations, but otherwise contribute to state space explosion. The technique is easily extended, if there are multiple guards.

On the other hand, if a guard potentially never evaluates to true, e.g., due to a broken sensor, then there are two ways to treat this: If this is valid behavior, then this guard can be ignored for the analysis (no execution exists after it). If it is invalid behavior, then the guard should be extended by an upper re-try bound and this new bounded guard can then be treated as explained above. After these transformations all state machines could be successfully unrolled.

4.1.3.5 Keeping Assumptions Sound

We made use of assumptions for limiting value domains where possible, and to perform assume-guarantee reasoning. Assumptions are a powerful tool in *cbmc*, however, it is easy to add assumptions which are not satisfiable (UNSAT). Those rule out *all* executions after the `assume` statement and thus might lead to wrong verification results.

Therefore, we have to ensure that the *composite* of all annotations is sound, otherwise the verification outcome may be wrong despite the individual annotations being correct. To check whether assumptions can be satisfied, we added a new check to *cbmc*, which does

the following: It inserts an `assert(false)` after each assumption and subsequently runs the SAT solver on it. If the solver yields UNSAT for the assertion, it means it is reachable and thus the assumption is valid. If it yields SAT, then all executions were ruled out and thus the assumption is UNSAT and thus unsound. Finally, we warn the user for each UNSAT assumption.

4.1.3.6 Verification Results

With our extensions of existing tools we were able to set up a correct verification workflow for the software of the ERS. The complexity of the analysis (for each mode: run-time, number of variables etc.) is summarized in Table 4.1. During the process we identified several trivial and non-trivial defects, some of them caused a deadlock in a state machine, multiple overflows in sensor data processing and even one timing-related error (barometer update took more steps than anticipated, which lead to wrong descent rate). Interestingly, during flight tests we sporadically experienced some of these errors, which by then could not be explained. One reason being, that there was little information about these errors due to limited logging and debugging facilities on the microcontroller, and that we could not reproduce the environmental conditions in the lab.

4.1.4 Related Work

Model-checking the entire C code running on microprocessors has been reported only a couple of times, e.g., with *cbmc* on an ATmega16 processor in [SK09] and on an MSP430 in [BK11], but either it failed because of state space explosion and missing support for concurrency, or succeeded only for smaller programs.

However, recent developments that turn concurrency into data nondeterminism [LR09], spot race conditions [WWC+13] and support for interrupts in *cbmc* [BK11] can solve the concurrency issues and make bounded model checking an interesting approach. In this case study we took together all these ideas, pointed our their problems, and proposed abstractions which mitigate the state space explosion problem, enabling a workflow which allows verifying an entire real-world program running on a microcontroller.

4.1.5 Conclusion

This case study has shown that formal verification of the entire, original software running on a microcontroller using MC is possible, if appropriate preprocessing techniques are applied. The state space can be reduced to a size that can be covered by existing tools in an acceptable time. This suggests that further efforts should be spent in developing more such techniques, especially on loops, to increase scalability of MC further.

Several threats towards soundness have been identified. First, information beyond the that in the source code must be provided. *Inter alia*, architectural information of the target (e.g., the word width), memory-mapped I/O and negligence of interrupts. A further threat to soundness may come from user-provided assumptions, which, when inconsistent, can cut out all execution traces, leading to unreachable code. Furthermore, in the case of cbmc, every unreachable property is marked as verified, since no violating trace can be found. This then might hide errors.

The precision of MC was confirmed. We were able to identify several defects that cause erroneous behavior observed during operation that could not reproduced in a lab environ-

ment. Towards that, the counterexample turned out to be especially useful, since the system does not have a debugging interface, nor sufficient resources to log critical events.

4.2 Case Study 2: Abstract Interpretation of C Code

In this second, shorter case study, we explore Abstract Interpretation with the tool Frama-C [CKK+12]. It implements a number of different software analyses for C programs, one of which is an AI-based value analysis. Since AI is the only one of the considered analysis method which enforces abstractions and is known to be suitable for large programs, the main purpose of this case study is to understand threats to validity, precision loss, and how it can be controlled. Therefore, and in contrast to the other two case studies, we explore a number of small, mostly synthetic examples instead of one larger application.

From our previous case study, we have identified that non-determinism and the ability to provide external knowledge (e.g., by assumptions), are essential for program analysis. The goals of this case study therefore are:

- Identify mechanisms to provide analysis context,
- identify mechanisms to indicate non-determinism,
- investigate the impact of missing relational domains and mitigation strategies, and
- investigate the impact of widening.

Frama-C was chosen first and foremost because it is an open source tool implementing AI, able to handle complex C programs, and being actively developed. It further offers features that allow parametrization and investigating of precision loss, and comes with a graphical user interface to visualize the results. One of its other analyses used in later chapters is program slicing.

4.2.1 Frama-C's Value Analysis

Frama-C was conceived for static formal verification, and therefore the value analysis aims to be sound. That is, Frama-C statically analyzes a C program for possible environments at each program location l, abstracting the collecting semantics. It is further parametrizable in many aspects, inter alia in its context-sensitivity and in the granularity of the abstract domain.

Abstract Domain. The default abstract domain can take different representations, depending on the program semantics and user settings [CYL+16]. The most important representations of the sets of abstract values $\mathcal{P}(\mathcal{D}_{var})$ of a variable *var* are:

- **Integer enumeration:** If $|\mathcal{P}(\mathcal{D}_{var})| \leq X$, then the precise set at each location l is stored.
- **Integer intervals:** When the enumeration size exceeds the threshold X, the domain only keeps the lower and upper bound of the value set. It can optionally carry periodicity information (shown later).
- **Float literal or interval:** For floating-point types, the domain either keeps a single literal or an interval, since rounding must be accommodated.
- **Set of addresses:** For pointers, the abstract domain tracks the precise sets of targets.

Some further representations are possible, e.g., when the analysis looses track, but this is not further discussed. It should be noted that X is a user setting, therefore this is a first option to control the precision.

Relational Domains. Frama-C can use external libraries [JM09] for relational domains. This aspect was however not explored here, since Frama-C offers another workaround presented later, and since in general the selection of the right relational domain depends on the program.

Alarms. The value analysis may generate proof obligations which are to be evaluated by other plugins of the Frama-C tool. For example, it generates assertions that array accesses must be within bounds. This not only raises warnings, but also the value analysis assumes them to be true, i.e., it does not consider execution traces with such offending behavior after the respective program location. The most important alarms are [CYL+16]

- logical shifts of signed numbers,
- division by zero,
- invalid memory access,
- signed overflows,
- floating-point ideals, and
- reading uninitialized variables.

Note that some of these events may lead to program termination if encountered during execution (e.g., division by zero), whereas others (e.g., signed overflow) may continue execution. As an effect, **the analysis might be unsound unless all proof obligations are verified**. This is in stark contrast to the previous case study with *cbmc*, where such undefined behavior was overapproximated. Some warnings can be disabled, as the signed overflow check, others cannot.

4.2.2 Experiments

We conducted a set of small experiments with Frama-C version *Aluminium* and its value plug-in "EVA" [CYL+16].

4.2.2.1 Providing Analysis Context

The first experiment seeks to understand Frama-C's handling of initial states, such as global variables, their initialization, and target-specific properties beyond the source code.

Before running any source analysis, Frama-C first requires specifying an entry point. By default, the tool considers that the program under analysis is a full program, as opposed to a library. Therefore, initialization of static and global variables is taken into account. Consider the following example:

```
1  static int mem = 0;
2
3  int filter(int x) {
4     int ret;
5     if (mem == 0) {
6       mem = x;
7       ret = x;
8     } else {
9       ret = lowpass(mem, x);
10    }
11    return ret;
12 }
```

If we analyze function `filter` in isolation (e.g., later for its WCET), then Frama-C considers the initial value of `mem`, and deduces that the else-branch is *unreachable*. Clearly, this is not the intended behavior in most cases, and it may lead to unsound results.

The command line option `-lib-entry` has the desired effect of setting `mem` to unknown, however, it has another drawback. If our source-level analysis itself introduces global variables register certain properties, then this option would havoc their initial state, as well. As a consequence, our source-level timing analysis requires adding a driver function which sets up the analysis context as desired.

Regarding the platform-specific properties, as endianness, word size and others, we have to use specific types to represent architectural sizes, and provide all target-specific header files and definitions. Otherwise, similar to the previous case study, the results may again be unsound.

4.2.2.2 Assertions and Assumptions

Assertions can be used to generate proof obligations, some of which can be resolved by a value analysis, if precise enough. They are therefore semantically equal to proof obligations generated by the value analysis (the alarms), and have the same consequences. That is, they cut out execution traces which violate the asserted expression. Consider the following test program:

```
1  #include "__fc_builtin.h"
2  #define assert(X) Frama_C_assert(X)
3
4  int main(int argc, char**argv) {
5    assert(argc < 15); // fails
6    // argc \in [0, 14]
7    int k = argc + 1;
8    assert(k <= 15); // succeeds
9    return k;
10 }
```

The assertion in line 5 fails, since the there is no guarantee that the number of arguments is always less than 15. However, the value analysis subsequently assumes (line 6 onwards) that this assertion holds true, such that the next assertion in line 8 always can be proven. This demonstrates that assertions imply assumptions, and conveniently links run-time with analysis, but it also means we cannot simultaneously verify multiple properties without a potential mutual influence on the outcome.

An individual `assume` statement is not available in Frama-C, but it can trivially be replaced by an assertion. The possibilities for constraining non-determinism as desired are available nevertheless, as discussed next.

4.2.2.3 Non-Determinism

Frama-C supports non-determinism, however, differently than cbmc. According to the C standard, uninitialized variables can lead undefined behavior. As already mentioned earlier, Frama-C assumes that no undefined behavior takes place, and therefore stops propagating such execution traces. As a consequence, uninitialized variables, perhaps used for the sake of setting up the analysis context, can lead to unreachable code and therefore again lead to unsound verification results.

Instead, non-determinism is only supported explicitly by either invoking a function for which no body has been specified (similar to cbmc), or by invoking built-in functions assigning non-deterministic values in a specific range, e.g.:

```
1 #include "__fc_builtin.h"
2 #include <limits.h>
3 #define nondet_h() Frama_C_interval(INT_MIN, INT_MAX-1)
4 #define nondet_int() Frama_C_interval(INT_MIN, INT_MAX)
```

```
int nondet();

int main() {
  int h = nondet_h(); // \in [-2_147_483_647 - 1, 2_147_483_647]
  int l; // \in UNINITIALIZED
  if (h > 0) {
    l = nondet(); // \in [--, --]
  }
  int k = nondet_int(); // \in [--, --]
  if (k > 10) { // result \in {0; 1}
    l = 1; // \in {1}
  }
  return l; // \in [--, --]
  // => means entire range of type
  // if init is skipped, then frama-c carries a distinct value
}
```

Note that since we lack an `assume` statement and that because `assert` has other side effects, complex constraints (e.g., "assume only odd numbers") on non-deterministic values are not directly possible. One workaround is to introduce an if-circuitry just for the analysis, which filters the execution traces accordingly.

4.2.2.4 Widening

The widening can be controlled to accelerate the convergence to a fixed point. This experiment seeks to understand which variables are approximated when widening happens within loops. We use the following test program, which contains a preview on what our future source-level timing analysis could look like. Let the variable `timeElapsed` represent how execution time passes by, along with the source code. Towards that, assume that we increment this variable according to how many processor cycles each source block consumes.

```
int main(int argc, char**argv) {
  int iter = 2;
  unsigned long timeElapsed = 0;
  while (1) {
    if (iter <= 0) break;
    iter--;
    timeElapsed += 12;
    Frama_C_show_each_loop(iter, timeElapsed);
  }
  timeElapsed += 1;
  return (int)timeElapsed;
}
```

This while-loop is in principle limited to two iterations, but this is only controlled implicitly by variable `iter`, which breaks the loop when it reaches zero. To yield a timing estimate, the value analysis therefore has to determine how often the loop repeats.

The fixed-point iterations were made visible with a tool-specific function call which writes the intermediate abstract states to the command line:

```
main.c:4:[value] entering loop for the first time
[value] Called Frama_C_show_each_loop({1}, {12})
[value] Called Frama_C_show_each_loop({0; 1}, {12; 24})
[value] Called Frama_C_show_each_loop({0; 1}, {12; 24; 36})
[value] Called Frama_C_show_each_loop({0; 1}, [12..2147483652],0%12)
[value] Called Frama_C_show_each_loop({0; 1}, [12..2147483664],0%12)
[value] Called Frama_C_show_each_loop({0; 1}, [0..4294967292],0%4)
[value] Recording results for main
[value] done for function main
```

The output shows that `iter` stabilizes quickly, but the iterations continue since the relation between `iter` and `timeElapsed` is not captured, and thus the latter one is still incrementing. After three iterations (the default setting), the widening operator is applied, setting the upper bound of this variable to the maximum integer multiple of 12 processor cycles that can be represented by its type[3]. Two more iterations follow, which accommodate the effect of a possible integer overflow in line 7, leaving the final value of `timeElapsed` almost maximally imprecise. The precise upper bound for the variable at program exit would have been 25, i.e., significantly lower, exhibiting a significant loss of precision in this case.

Consequently, although the loop bound has been established, the missing relation between controlling variable and other effects can rendering the analysis result practically useless. Next, we investigate ways to mitigate this issue, beyond just increasing the setting that controls when widening is applied.

4.2.2.5 Parametrization of Precision

We investigate three of Frama-C's options to control the precision of the analysis, namely *semantic unrolling*, *loop invariants* and *function specifications*.

Superposed states. Frama-C supports the propagation of unjoined states ("s-level") up to a certain path length, that is, it can be made path sensitive up to a bounded length. This option has three fundamental effects:

1. Relations between variables are implicitly represented by the disjunction of abstract states. That is, by not joining different abstract states into their LUB for a certain path length, the individual states are approximating the collecting semantics, and thus alleviate the need for relational domains.

2. For loops, superposed states have the same effect as loop unwinding, therefore they counteract widening.

3. The state space of the model is increased at an exponential rate in an extreme case, slowing down the analysis.

To illustrate the first effect, consider the following program:

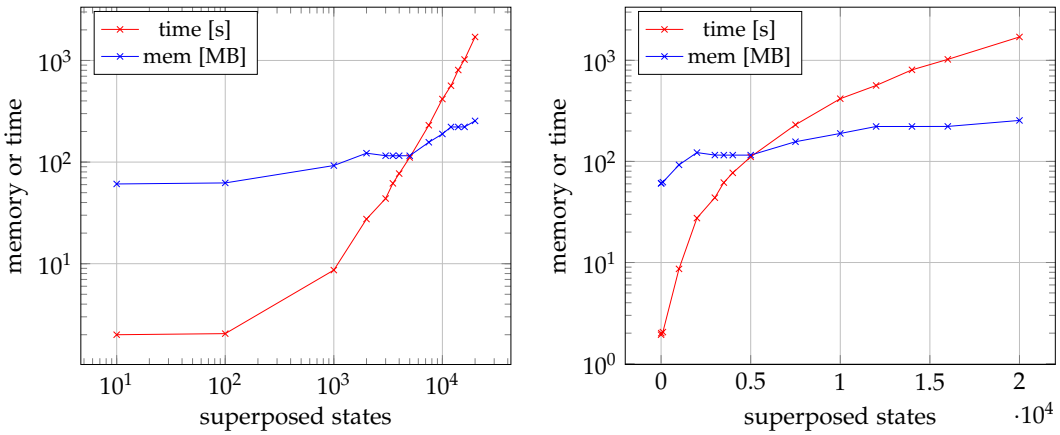
```
1 int main() {
2   int timeElapsed = 0;
3   for(int i=0; i<100; i++) {
4     if (i<5) timeElapsed+= 20;
5   }
6   return 0;
7 }
```

Like before, the relation between variables `timeElapsed` and the loop counter is unknown. However, when setting `s-level` to 5, i.e., we propagate unjoined states up to 5 steps ahead, then Frama-C keeps the precise environments for the first five loop iterations unjoined, just long enough to deduce that thereafter the variable is no longer incremented. As a result, this loop can be analyzed precisely thanks to superposed states.

The second effect follows from this observation.

[3]The trailing notation $r\%m$ conveys the periodicity information, denoting that the value range only consists of values whose remainder in the Euclidean division by m is equal to r [CYL+16].

The third effect – exponential increase of the state space – has to be investigated separately. Towards this, we analyzed an implementation of the bubblesort algorithm for 1,000 non-deterministically initialized elements, and successively increased the path length for superposed states. The result is depicted in Fig. 4.7. The results indicate a polynomial increase of analysis time, and no clear relationship for memory usage, suggesting that Frama-C internally re-uses or shares sets of abstract states with identical values. In any case, it can be seen that this "extended AI" is computationally complex, and therefore not a general answer to AI's precision loss.



(a) both axes logarithmic **(b)** logarithmic y-axis

Figure 4.7: Computational effort of superposed states on *bubblesort*.

Loop invariants. Another approach to gain precision in loops are loop invariants. Frama-C intersects the widened state with user-provided loop invariants [CYL+16]. However, it is generally non-trivial to obtain loop invariants. Especially if we model the elapsed time as a variable and want to avoid its widening in loops, then we would have to find an upper bound to formulate a useful (in-)variant, which in turn does not exist before the analysis is terminated.

Function Calls and Specifications. Last but not least, Frama-C offers the possibility to specify the behavior of callee analysis. By default the context-sensitivity is limited to a certain depth, but can be controlled. Furthermore, instead of analyzing an entire function on every invocation, it is possible replace the function with a formal specification. Such functionality could be useful for library routines or compositional analysis, but in general only reduces the precision.

4.2.3 Conclusion

The AI-based value analysis of Frama-C has again confirmed that a proper setup of the analysis context is essential. It creates a model assuming the absence of undefined behavior. For example, direct memory access via pointers and reading uninitialized variables both trigger alarms, and execution traces are subsequently considered ceased, which is not necessarily the case in an implementation. As a consequence, improper setup may result in unsound results. Some of its assumptions can be configured (e.g., for overflows and memory accesses),

whereas others (interpretation of uninitialized variables) cannot. In general, any undefined behavior should therefore be avoided.

Furthermore, properties build on each other by the same mechanism. Even if a property is violated, Frama-C assumes that it holds true for all downstream code. As an effect, multiple properties cannot be verified independently, and each of them is only guaranteed to be sound if all of them are verified.

The flexibility of the abstract domain allows the user to control the precision of AI. It collects precise sets of values up to certain size, before reverting to sets of ranges. Similarly, widening can be controlled, which can mitigate the weakness of AI in loops. With the standard domain AI does not keep track of variable relations, but Frama-C offers an extension called *superposed* states, which propagates abstract sets forward on the CFG to a given length, which effectively tracks relations between variables. The price is, however, at least a in increase in complexity polynomial to this path length.

The possibility to use function specifications in lieu of their bodies does not have any obvious benefit for timing analysis, since it needs manual annotations. It could however be a means to model library functions, for which no source code is available.

In summary, AI seems capable of proving only simple properties. If widening is kept under control, it might work for source-level timing analysis, with the drawback that it cannot precisely identify infeasible paths. An additional use case might be the computation of invariants, since they naturally result from this type of analysis.

4.3 Case Study 3: Deductive Verification of Ada/SPARK Code

This section summarizes our case study published in [uRC17], which uses DV for formal verification. The system under consideration is a novel kind of weather balloon which is actively controlled, and thus requires verification to ensure it is working properly in public airspace. Specifically, this case study is concerned with the verification of a flight control software for the autonomous glider shown in Fig 4.8, which is attached to this balloon.

The mode of operation is as follows. As any normal weather balloon, the system climbs up to the stratosphere (beyond an altitude of 10 km), while logging weather data such as temperature, pressure, NO_2-levels and so on. Eventually the balloon bursts, and the sensors would be falling back to the ground with a parachute, drifting away with prevailing wind conditions. However, in our system the glider carries these sensors, and it separates itself from the balloon at a pre-defined altitude. It then stabilizes its attitude and performs a controlled descent back to the take-off location, thus, bringing the sensors back home.

The entirety of this mission is running autonomously, and is controlled by the software running on a microcontroller inside the glider. This software, therefore, has to implement all functionality necessary to realize this mission, such as attitude estimation and control, navigation (via GPS and barometer), launch monitoring, unhitching, and last but not least, also logging of the weather data.

Since operating such a system requires permission and is subject to certain operational constraints (such as pre-defined time windows), the software has to perform correctly as to not endanger other airspace users, and furthermore testing is not an option beyond basic ground-level tests. Analog to the first case study, the environmental conditions can also not easily be reproduced. Temperatures may range from 30 °C down to −50 °C, winds may exceed 100 kph, and GPS devices may yield vastly different output in those altitudes due to decreasing precision and the wind conditions. The combination of those extreme values is likely to trigger corner cases in the software, and thus hard to reach by testing. Static formal

Figure 4.8: The glider/drone whose software was subject to verification.

verification is one way to ensure that this software behaves as intended, and is free of defects. In this case study, verification therefore took place *in parallel* to software development.

4.3.1 Tools and Methods

We have chosen the Ada/SPARK 2014 programming language to implement the flight stack from scratch, only building on a Run-Time System (RTS) for Ada on closely related ARM processors. It comes with new and state-of-the art verification tools of based on DV [HMWC15], and leverages the strict semantics of SPARK. Its predecessor has proven itself on large-scale systems [DEL+14, CS14], and was rated by the National Institute of Standards and Technology as approach with "dramatic future impact", reducing the defects of a program by two orders of magnitude [BBGF16]. This combination of pedigree, endorsement and latest technology made it a natural choice to investigate the strengths and weaknesses of current DV technology.

The SPARK language – for the rest of this chapter we refer to SPARK 2014 simply as SPARK – was conceived with verification in mind. It is mostly a subset of the strongly typed *Ada* programming language, but additionally integrates a specification language for functional contracts and data flow contracts. Subprograms (procedures and functions) can be annotated with pre- and postconditions, as well as with data dependencies. GNATprove, the (only) static analyzer for SPARK 2014, aims to prove subprograms in a modular way, by analyzing each of them individually using DV. The effects of callees are summarized by their post-condition when the calling subprogram is analyzed, and the precondition of the callee is imposing a proof obligation on the caller, i.e., the need to verify that the caller respects the callee's precondition. Additional proof obligations arise from each language-defined check that is executed on the target, such as overflow checks, index checks, and so on. If all proofs are successful, then the program is working according to its contracts and no exceptions will be raised during execution, for short, *Absence of Run-Time Errors (AoRTE)*.

Internally, GNATprove [HMWC15] builds on the Why3 platform [FP13], which performs WP calculus on the proof obligations to generate verification conditions (VCs), and then passes them to a theorem solver of the user's choice, e.g., *cvc4*, *alt-ergo* or *z3*.

4.3.2 System Details

Target Hardware. We have chosen the *Pixhawk* autopilot [MTFP11]. It comprises two ARM processors; one Cortex-M4F (STM32F427) acting as flight control computer, and one Cortex-

M3 co-processor handling the servo outputs. We implemented our flight stack on the Cortex-M4 from the ground up, thus replacing the original PX4/NuttX firmware that is normally used with this hardware.

Board Support, Hardware Abstraction Layer & Run-Time System. We are hiding the specific target from the application layer by means of a board support package (not to be confused with an Ada package). This package contains a hardware abstraction layer (HAL) and a run-time system (RTS). The RTS is implementing basic functionality such as tasking and memory management. The HAL is our extension of AdaCore's Drivers Library [Ada15], and the RTS is our port of the Ada RTS for the STM32F409 target. Specifically, we have ported the Ravenscar Small Footprint variant [Bur99], which restricts Ada's and SPARK's tasking facilities to a deterministic and analyzable subset, but meanwhile forbids exception handling, which anyway is not permitted in SPARK.

Separating Tasks by Criticality. SPARK in conjunction with the chosen RTS supports multitasking, which enabled us to separate the functionality into multiple tasks, such that 1. termination of low-critical tasks shall not cause termination of high-critical tasks, 2. higher-criticality tasks shall not be blocked by lower-critical tasks and, 3. adverse effects such as deadlocks, priority inversion and race conditions must not occur.

We partitioned our glider software into two tasks (further concurrency arises from interrupt service routines):

1. The *Flight-Critical Task* includes all execution flows required to keep the glider in a controlled and navigating flight, thus including sensor reads and actuator writes. It is time-critical for control reasons. High-criticality.

2. The *Mission-Critical Task* includes all execution flows that are of relevance for recording and logging of weather data to an SD card. Low-priority task, only allowed to run when the flight-critical task is idle. Low-criticality.

The latter task requires localization data from the former one, to annotate the recorded weather data before writing it to the SD card. Additionally, it takes over the role of a flight logger, saving data from the flight-critical task that might be of interest for a post-flight analysis. The interface between these two tasks would therefore be a protected object with a message queue that must be able to hold different types of messages.

4.3.3 Verification Process

Functional verification was performed in parallel to the development of the software, with focus on the *flight-critical* task. In the following, we describe the verification process with its goals, difficulties and results. We use the following nomenclature:

- **False Positive.** Denotes a failing check (failed VC) in static analysis which would not fail in any execution on the target, i.e., a false alarm.

- **False Negative.** Denotes a successful check (discharged VC) in static analysis which would fail in at least one execution on the target, i.e., a missed defect.

4.3.3.1 Verification Goals

First and foremost, AoRTE shall be established for all SPARK parts, since exceptions would result in task termination. Additionally, the application shall make use of as many contracts and checks as possible, and perform all of its computations using dimension-checked types.

Last but not least, a few functional high-level requirements related to the homing functionality have been encoded in contracts. Overall, the focus of verification was the application, not the BSP. The BSP has been written in SPARK only as far as necessary to support proofs in the application. The rationale was that the RTS was assumed to be well tested, and the HAL was expected to be hardly verifiable due to direct hardware access involving pointers and restricted types.

4.3.3.2 Assertions and Assumptions

In SPARK and GNATprove, assertions build on each other, similarly to Frama-C. This brings the verification semantics close to run-time semantics and makes it therefore intuitive. Nevertheless, it has the same downside as Frama-C, in that we cannot verify multiple properties simultaneously without a potential mutual influence. Additionally, even a verified program can fail, if its preconditions are violated, or if its callees fail to fulfill their postcondition. Therefore, unless a software is testified AoRTE, there may be missed errors and defects due to the DV approach.

Assumptions can be provided manually like in cbmc, but this is again a dangerous way to convey information to the verifier. If incorrect or inconsistent, the soundness of the results can be refuted again. Since DV does not have, nor requires, an equivalent to explicit non-determinism (this is captured implicitly in the contracts), the `assume` statement is not inevitable, and should better be avoided.

4.3.3.3 Threats to Soundness

There are a few situations in which static analysis can miss run-time exceptions, which in a SPARK program inevitably ends in abnormal program termination. Before we show these unwanted situations, we have to point out one important property of a deductive verification approach: Proofs build on each other. Consider the following example (results of static analysis given in comments):

```
1   a := X / Z; -- medium: division check might fail
2   b := Y / Z; -- info: division check proved
```

The analyzer reports that the check in line 2 cannot fail, although it suffers from the same defect as line 1. However, when the run-time check at line 1 fails, then line 2 cannot be reached with the offending value of z, therefore line 2 is not a False Negative, unless exceptions have been wrongfully disabled.

Mistake 1: Suppressing Alarms. When a developer comes to the conclusion that the analyzer has generated a False Positive (e.g., due to insufficient knowledge on something that is relevant for a proof), then it might be justified to suppress the failing property. However, we experienced cases where this has generated False Negatives which where hiding (critical) failures. Consider the following code related to the GPS:

```
1  function toInt32 (b : Byte_Array) return Int_32 with Pre => b'Length = 4;
2  procedure Read_From_Device (d : out Byte_Array) is begin
3    d := (others => 0); -- False Positive
4    pragma Annotate (GNATprove, False_Positive, "length check might fail", ...);
5  end Read_From_Device;
6
7  procedure Poll_GPS is
8    buf    : Byte_Array(0..91) := (others => 0);
9    alt_mm : Int_32;
10 begin
11   Read_From_Device (buf);
12   alt_mm := toInt32(buf(60..64)); -- False Negative, guaranteed exception
```

```
13 end Poll_GPS;
```

Static analysis found that the initialization of the array `d` in line 3 could fail, but this is not possible in this context, and thus a False Positive[4]. The developer was therefore suppressing this warning with an annotation pragma. However, because proofs build on each other, a severe defect in line 12 was missed. The array slice has an off-by-one error which *guaranteed* failing the precondition check of `toInt32`. The reason for this False Negative is that everything after the initialization of `d` became virtually *unreachable* and that all following VCs consequently have been discharged. In general, a False Positive may exclude some or all execution paths for its following statements, and thus hide (critical) failure. We therefore recommend avoiding to suppress False Positives, and either leave them visible for the developer as warning signs, or even better, rewrite the code in a prover-friendly manner following the tips in Section 4.3.3.6.

Mistake 2: Inconsistent Contracts. Function contracts act as barriers for propagating proof results (besides inlined functions), that is, the result of a VC in one subprogram cannot affect the result of another in a different subprogram. However, these barriers can be broken when function contracts are inconsistent, producing False Negatives by our definition. One way to obtain inconsistent contracts, is writing a postcondition which itself contains a failing VC (line 2):

```
1  function f1 (X : Integer) return Integer
2    with Post => f1'Result = X + 1 is -- overflow check might fail
3  begin
4    return X;
5  end f1;
6
7  procedure Caller is
8     X : Integer := Integer'Last;
9  begin
10    X := X + 1; -- overflow check proved.
11    X := f1(X);
12 end Caller;
```

Clearly, an overflow must happen at line 10, resulting in an exception. The analyzer, however, proves absence of overflows in `Caller`. The reason is that in the Why3 backend, the postcondition of `f1` is used as an axiom in the analysis of `Caller`. The resulting theory for `Caller` is an inconsistent axiom set, from which (*principle of explosion*) anything can be proven, including that false VCs are true. In such circumstances, the solver may also produce a *spurious* counterexample.

In the example above, the developer gets a warning for the inconsistent postcondition and can correct for it, thus keep barriers intact and ensure that the proofs in the caller are not influenced. However, if we change line 4 to `return X+1`, then the failing VC is now indicated in the body of `f1`, and – since the proofs build on each other – the postcondition is verified and a defect easily missed. Therefore, failing VCs within callees may also refute proofs in the caller (in contrast to execution semantics) and have to be taken into account. Indeed, the textual report of GNATprove (with flag `--assumptions`) indicates that AoRTE in `Caller` depends on both the body and the postcondition of `f1`, and therefore the reports have to be studied with great care to judge the verification output. Finally, note that the same principle applies for assertions and loop invariants.

Mistake 3: Forgetting the RTS. Despite proven AoRTE, one procedure which rotates the frame of reference of the gyroscope measurements was sporadically triggering an exception

[4]This particular case has been fixed in recent versions of GNATprove.

after a floating-point multiplication. The situation was eventually captured in the debugger as follows:

```
1 -- angle = 0.00429, vector (Z) = -2.023e-38
2 result(Y) := Sin (angle) * vector(Z);
3 -- result(Y) = -8.68468736e-41 => Exception
```

Variable `result` was holding a *subnormal* floating-point number, roughly speaking, an "underflow". GNATprove models floating-point computations according to IEEE-754, which requires support for subnormals on the target processor. Our processor's Floating Point Unit (FPU) indeed implements subnormals, but the RTS, part of which describes floating-point capabilities of the target processor, was incorrectly indicating the opposite[5]. As a result, the language-defined float validity check occasionally failed (in our case when the glider was resting level and motionless at the ground for a longer period of time). Therefore, the RTS must be carefully configured and checked manually for discrepancies, otherwise proofs can be refuted since static analysis works with an incorrect premise.

Undefined Behavior and Target Properties. Undefined behavior is not an issue in SPARK, since the language does not leave it completely open, but instead specifies a *set* of behaviors which compilers can choose from, such that the analyzer can take these into account. Closely related, Ada/SPARK allows *implementation-defined* behavior, which additionally requires the compiler to document which of the possible behaviors was chosen.

Similarly to cbmc, we need to consider architecture-specific properties that may impact the behavior of the program (again, endianness, word widths, float size and others). This is ensured by providing a target configuration file. Another aspect, as mentioned earlier, is the setup of the RTS, which may claim a certain behavior of the hardware (e.g., floating-point rounding modes) that is picked up by the analyzer. Additionally, the analyzer is currently assuming IEEE-754 semantics, illustrating that hardware, RTS and analyzer have to agree on certain properties for correct results.

4.3.3.4 Design Limitations

We now describe some cases where the current version of the SPARK 2014 *language* – not the static analysis tool – imposes limitations.

Access Types. The missing support for pointers in SPARK becomes a problem in low-level drivers, where they are used frequently. One workaround is to hide those in a package body that is not in SPARK mode, and only provide a SPARK specification. Naturally, the body cannot be verified, but at least its subprograms can be called from SPARK subprograms. Sometimes it is not possible to hide access types, in particular when packages use them as interface between each other. This is the case for our SD card driver, which is interfaced by an implementation of the FAT filesystem through access types. Both are separate packages, but the former one exports restricted types and access types which are used by the FAT package, thus requiring that wide parts of the FAT package are written in Ada instead of SPARK. As a consequence, access types are sometimes demanding to form larger monolithic packages, here to combine SD card driver and FAT filesystem into one (possibly nested) package.

Polymorphism. While being available in SPARK, applications of polymorphism are limited as a result of the access type restriction. Our message queue between flight-critical and mission-critical task was planned to hold messages of a polymorphic type. However, without

[5]This also has been fixed in recent versions of the embedded ARM RTS.

access types the only option to handover messages would be to take a deep copy and store it in the queue. However, the queue itself is realized with an array and can hold only objects of the same type. This means a copy would also be an upcast to the base type. This, in turn, would loose the components specific to the derived type, and therefore render polymorphism useless. As a workaround, we used mutable variant records.

Interfaces. Closely related to polymorphism, we intended to implement sensors as polymorphic types. That is, specify an abstract sensor interface that must be overridden by each sensor implementation. Towards that, we declared an abstract tagged type with abstract primitive methods denoting the interface that a specific sensor must implement. However, when we override the method for a sensor implementation, such as the Inertial Measurement Unit (IMU), SPARK requires specifying the global dependencies of the overriding IMU implementation as class-wide global dependencies of the abstract method (SPARK RM 6.1.6). This happens even without an explicit `Global` aspect.

As workaround, we decided to avoid polymorphism and used simple inheritance without overriding methods.

Dimensioned Types. Using the GNAT dimensionality checking system in SPARK, had revealed two missing features. Firstly, in the current stable version of the GNAT compiler, it is not possible to specify general operations on dimensioned types that are resolved to specific dimensions during compilation. For example, we could not write a generic time integrator function for the PID controller that multiplies any dimensioned type with a time value and returns the corresponding unit type. Therefore, we reverted to dimensionless and unconstrained floats within the generic PID controller implementation. Secondly, it is not possible to declare vectors and matrices with mixed subtypes, which would be necessary to retain the dimensionality information throughout vector calculations (e.g., in the Kalman Filter). As a consequence, we either have split vectors into their components, or reverted to dimensionless and unconstrained floats. As a result of these workarounds, numerous overflow checks related to PID control and Kalman Filter could not be proven (which explains more than 70 % of our failed floating-point VCs).

4.3.3.5 Weaknesses of the Analyzer

We now summarize some frequent problems introduced by the current state of the tooling.

Missing Models. The `'Position` attribute of a record allows evaluating the position of a component in that record. However, GNATprove has no precise information about this position, and therefore proofs building on that might fail. Another feature that is used in driver code, are *unions*, which provide different views on the same data. GNATprove does not know about the overlay and may generate False Positives for initialization, as well as for proofs which build on the relation between views.

Uninitialized Variables. We had several False Positives related to possibly uninitialized variables. SPARK follows a strict data initialization policy. Every (strict) output of a subprogram must be initialized. In the current version, GNATprove only considers initialization of arrays as complete when done in a single statement. This generates warnings when an array is initialized in multiple steps, e.g., through loops, which we have suppressed.

Loops. In general DV requires providing loop invariants to compute VCs. This requires manual inputs from the user, and is not an easy task in practice. However, GNATprove

attempts to generate *frame conditions* for loops automatically, which have been sufficient for this program. No manual annotations were necessary.

4.3.3.6 Results

In general, verification of SPARK 2014 programs is accessible and mostly automatic, yet a small number of properties required most attention from the developers. Figure 4.9 shows the results of our launch release. As it can be seen, we could not prove all properties during the time of this project (three months). The non-proven checks have largely been identified as "fixable", following our design recommendations given below.

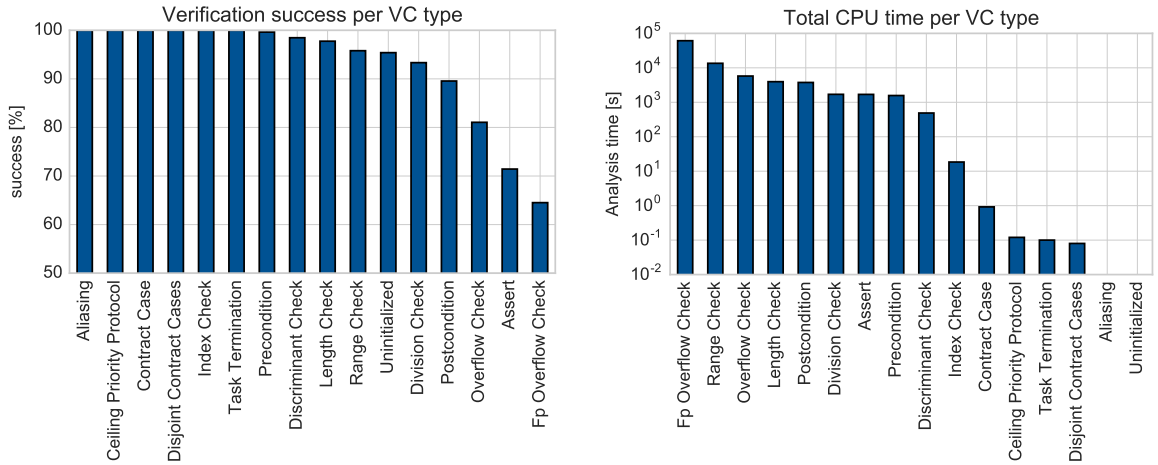


Figure 4.9: Statistics on Verification Conditions (VCs) by type.

The complexity of our flight stack and verification progress are summarized in Table 4.2. It can be seen that our focus on the application part is reflected in the SPARK coverage that we have achieved (82 % of all bodies in SPARK, and even 99 % of all specifications), but also that considerably more work has to be done for the BSP (currently only verified by testing). In particular, the HAL (off-chip device drivers, bus configuration, etc.) is the largest part and thus needs a higher SPARK coverage. However, we should add that 43 % of the HAL is consisting of specifications generated from CMSIS-SVD files, which do not contain any subprograms, but only definitions of peripheral addresses and record definitions to access them, and therefore mostly cannot be covered in SPARK. Last but not least, a completely verified RTS would be desirable, as well.

Floats are expensive. Statistically, we have spent most of the analysis time (65 %) for proving absence of floating-point overflows, although these amount to only 21 % of all VCs. This is because discharging such VCs is in average one magnitude slower than discharging most other VC types. In particular, one has to allow a high step limit (roughly the number of decisions a solver may take, e.g., deciding on a literal) and a high timeout. Note that at some point an increase of either of them does not improve the result anymore.

Multi-Threading. By using the Ravenscar RTS, our goals related to deadlock, priority inversion and blocking, hold true by design. Several race conditions and non-thread-safe subprograms have been identified by GNATprove, which otherwise would have refuted task separation. To ensure that termination of low-criticality tasks cannot terminate the flight-critical task, we provided a custom implementation for GNAT's last chance handler (outside of the SPARK language and therefore not being analyzed) which reads the priority of the

Table 4.2: Metrics and verification statistics of our Flight Stack.

		Board Support Package		
Metric	Application	HAL	RTS	All
lines of code (GNATmetric)	6,750	32,903	15,769	55,422
number of packages	49	100	121	270
cyclomatic complexity	2.03	2.67	2.64	2.53
SPARK body/spec	81.9/99.4 %	15.5/23.5 %	8.6/11.8 %	30.0/38.5 %
number of VCs	3,214	765	2	3,981
VCs proven	88.1 %	92.5 %	100 %	88.8 %
analysis time[a]	–	–	–	19 min

[a] Intel Xeon E5-2680 Octa-Core with 16 GB RAM, timeout=120 s, steps=inf.

failing task and acts accordingly: If the priority is lower than that of the flight-critical task (i.e., the mission-critical task had an exception), then we prevent a system reset by sending the low-priority task into an infinite null loop (thus keeping it busy executing `nops`, and keeping the flight-critical task alive). If the flight-critical task is failing, then our handler allows a system reset. Multi-threading is therefore easy to implement, poses no verification problems, and can effectively separate tasks by their criticality.

High-Level Behavioral Contracts. Related to the homing functionality we proved high-level properties with the help of ghost functions, although this is beyond the main purpose of SPARK contracts. For example, we could prove the overall behavior in case of loosing the GPS fix, or missing home coordinates.

Design Recommendations. It has become evident that the software structure can have a large impact on verifiability. There are usually different ways to implement a specified functionality, and some turned out to more amenable to verification, namely:

1. Split long expressions into multiple statements → discharges more VCs.
2. Limit ranges of data types, especially floats → better analysis of overflows.
3. Avoid interfaces → annotations for data flows break concept of abstraction.
4. Emulate polymorphic objects that must be copied with mutable variant records.
5. Separation of tasks by criticality using a custom last chance handler → abnormal termination of a low-criticality task does not cause termination of high-criticality tasks.

4.3.4 Related Work

Only a small number of experience reports about SPARK 2014 have been published before. A look back at (old) SPARK's history and its success, as well as an initial picture of SPARK 2014 is given by Chapman and Schanda in [CS14]. We can report that the mentioned difficulties with floating-point numbers are solved in SPARK 2014, and that the goal to make verification more accessible, has been reached. A small case study with SPARK 2014 is presented in [TE14], but at that point multi-threading (Ravenscar) was not yet supported, and floating point numbers have been skipped in the proof. We can add to the conclusion given there, that both are easily verified in "real-world" code, although float proofs require more (computational and mental) effort. Larger case studies are summarized by Dross et al. in [DEL+14], with whom we share the opinion of minor usability issues, and that some small amount of developer training is required. Finally, SPARK 2014 with Ravenscar has recently been

announced to be used in the Lunar IceCube [BC16] satellite, a successor of the successful CubeSat project that was implemented in SPARK 2005. It will be a message-centric software, conceptually similar to NASA's cFE/CFS, but fully verified and striving to become an open source platform for spacecraft software.

4.3.5 Conclusion

The flight stack software was developed under continuous verification, therefore we were able to identify program constructs which are amenable and detrimental to the tool and method. The modular, *assume-guarantee* style of verification had a positive impact on two levels. First, because any failing property could be blamed to a subprogram and thus suggests where the defect could be located. Second, the program rarely had to be verified as a whole. Instead, the tools only re-compute proofs if relevant parts of the program have changed, which did therefore not result in any scalability problems. Further, this modularization also allows analyzing translation units and subprograms in isolation, and thus non-determinism is implicitly captured in this verification model in the contracts.

Regarding analysis precision and gap to completeness, it is hard to come to a conclusion. Not every property got proven, and for a developer it was hard to tell whether a proof fails because of lacking theorems or timeout, unless a counterexample was generated. The usual strategy for failing proofs was to "test the knowledge" of the provers by strategically placing assertions and observing whether they could be discharged. The strong typing system has been beneficial for many proofs, confirming that source-level information is valuable for an analysis. In summary, the degree of automation is high, but failing properties require tedious investigations.

For those properties that raised alarms, there does not necessarily exist a violating call context, owing to the modular approach. DV is therefore well-suited to develop robust and reusable software (specify contracts, ensure postcondition under all inputs) and satisfy the precision requirements from a developer's point of view, but its otherwise imprecise inter-procedural analysis would make it likely too overapproximative for a timing analysis. Since the contracts subsume all call contexts, the precision of a timing estimate would depend on how much details are conveyed in them, and how much timing variability each subprogram has. In an extreme case, the WCET could be estimated as the sum of the individual worst cases induced by the contracts, which in general would lead to overestimation.

As the other approaches, DV in this tooling is sound, but subject to certain assumptions and boundary conditions. Whereas the logical aspects of the analysis context are no concern (e.g., states of globals), architectural properties have to be communicated to the verifier correctly, and contracts need to be free from defects themselves. Additionally, the run-time system's behavior must consistent with the prover assumptions (i.e., configured correctly), since otherwise the results can become unsound. Last but not least, undefined behavior, thanks to the SPARK language, was not an issue.

Finally, the tasking abilities and semantics of Ada have proven useful to analyze concurrency correctly, and to separate high-criticality from low-criticality tasks. This shows that program structure can be especially helpful for analyzing aspects of concurrency.

4.4 Discussion of Analysis Methods and Tools

We now compare the different analysis approaches and tools, to identify which one is the most suitable for a source-level timing analysis.

4.4.1 Link between Execution Semantics and Analysis

For the code of the program under analysis, naturally all of the reviewed tools and methods must adhere to execution semantics, if we leave out the different interpretations of undefined behavior, if any. A difference exists in how the tools model the effect of analysis constructs (e.g., an `assert` to denote a property to be checked) with the run-time of the program. This is a tool choice, and not one of the underlying analysis.

We have seen that in SPARK/gnatprove the run-time and verification are tightly linked, as long as the contracts themselves are free of defects. The contracts are (by default) even executed during run-time, such that this tool offers the closest link between run-time and verification among those evaluated. However, since the proofs build on each other, an independent verification of multiple properties is not directly possible. On contrast, cbmc chooses to have a weak link, because multiple assertions do not influence each other. In Frama-C we again have a similar behavior like with gnatprove, although not as strongly, possibly owing to the undefined behavior of the C language in contrast to SPARK.

In this aspect, all tools seem equally suitable for a timing analysis. Independent of which analysis method and tool is chosen, such links, if not mutual influences, between analysis and run-time of a program, must be carefully checked.

4.4.2 Threats to Soundness

The case studies have shown further causes of unsound results. Before we summarize those, we should emphasize that the unsound results do not stem from the underlying analysis methods, but derive from improper use of the tools. The most important causes of unsound results were:

- **Incorrect analysis context**. To model the behavior of the program under analysis correctly, the user needs to provide information about target properties (e.g., word width, endianness), run-time systems and hardware modes (e.g., libraries and floating-point behavior). Furthermore, the initial program environment must be set up, most importantly global and static variables must be modeled according to possible run-time states. Last but not least, the source-level analysis can only be correct if the compiler translates the program correctly, which is not always the case [FFL+11, WZKS13].

- **Undefined behavior**. Closely related to considerations regarding the compiler is avoiding behavior that is left undefined by the language (e.g., the C standard). If such behavior is present, all analysis methods must necessarily produce wrong results for at least some programs. By definition, undefined behavior can have any effect. Static analyzers usually choose to model it such that it agrees with the behavior most often observed in practice (e.g., uninitialized variables often carry random values, thus non-determinism is a good choice). Whereas cbmc approximates such behavior, Frama-C assumes the absence thereof. Although the former one seems more intuitive, Frama-C's treatment is no less justified. Especially under compiler optimization it is known that unexpected consequences can side-step the semantics of the source code completely [WZKS13], and thus all tools may yield unsound results. Since all of them can identify undefined behavior, the best way forward is to repair the program based on the tool's warnings, before proceeding with any other analysis.

- **Defects in the property specification**. If the verification tool comes with its own specification language (like most DV-based tools), then it is essential that these specifications

themselves are free from defects. Otherwise, as experienced in the last case study, incorrect contracts can result in *ex falso quodlibet* proofs. It further may be the case that the specification language uses unbounded domains, such that specifications have to be formulated carefully, as to still meet the implementation semantics.

- **Incorrect annotations**. If the analysis tool allows the user to constrain the traces to be analyzed (e.g., through `assume`), then such constraints can cut out too many execution traces. The result can be unreachable properties which thus cannot be violated, and as an effect pass as verified. Assumptions should only be used to constrain inputs to the analysis, e.g., values coming from sensors, and even there this practice is questionable (the software could crash if the sensor malfunctions). In any case, the user is responsible to verify the correctness of any annotation externally.

Since all threats are tool-specific, they cannot be used to rank the analysis methods. These threats must be kept in mind during source-level timing analysis.

4.4.3 Software Structure vs. Analysis

The structure of the analyzed program has a large impact on analzability. We have seen that especially in the case studies for SPARK and cbmc, task separation and modes helped to chunk the program into smaller modules. By structure we not only mean modularization, encapsulation, etc., but also to introduce types specifying ranges, as well as contracts. The latter two aspects, of course, are only of relevance for programming languages that support these, like Ada/SPARK.

Ideally, functional and temporal verification should happen in parallel to the development of the software, to capture the programmer's implicit knowledge on the program behavior, and to attain a program structure that is amenable to formal verification. A source-level timing analysis could then be seamlessly integrated into software development, along with the use of static analysis tools.

4.4.4 Support for Parallelism

Support for parallelism has only been investigated coarsely. The tool cbmc has mechanisms to model ISRs, yet they can be unsound. In the last case study, we have demonstrated that language constructs can be used to separate tasking and sequential code effectively. However, if the program under analysis uses interrupts (as many embedded systems do), then the shared variables between ISR and main program must still be modeled. For a WCET analysis, which by definition only computes uninterrupted time, this is enough. On the other extreme, if a software does run multiple threads concurrently and does not take any provisions to logically separate them, then any of the shown analyses can break.

4.4.5 Scalability

In practice, there is no clear winner in the aspect of scalability. The two methods which theoretically have the highest complexity – DV and MC – could both successfully verify a full embedded system within minutes. Although this is slower than traditional timing analysis methods, and DV cannot directly be compared because of its modular approach, this seems an acceptable duration, as long as it can be automated.

The scalability of MC is especially at risk in the presence of many loops. As we have seen there are some possibilities to alleviate this problem through loop transformations. However,

more effective transformations would be beneficial, since the programs will only get more complex once the source code also contains the timing.

The best scalability can be attested to AI, as expected from theory. However, due to its fundamental imprecision (see next section), extensions like superposed states or relational domains might be necessary in practice to obtain usable results. Our experiments have shown that the complexity then reaches at least polynomial levels, almost diminishing the scalability advantages of AI.

Independent of the analysis method, further scalability-enhancing methods should be considered, as done with the dead code removal.

4.4.6 Precision

The best analysis precision can be expected from MC, followed by DV. These approaches are maximally precise, but MC analyses the program as a whole, therefore only considering feasible flows. In contrast, DV uses assume-guarantee reasoning to analyze subprograms individually, thereby loosing precision. While this is actually preferable for general software engineering (see usability), this has an adverse impact for timing analysis

AI, by its very nature, sacrifices precision for decidability and scalability (see above). Although relational domains can improve precision, it there exists no systematic method to choose the best domain for a given program, other than trial-and-error. Furthermore, each JOIN in the CFG still causes overapproximation. Using superposed states as a workaround brings much better precision, but in practice requires more analysis time than a MC run on the same program.

4.4.7 Usability

All in all, MC is the method which we expect to be easiest to use for a developer during later source-level timing analysis. Whereas DV and AI both require user interaction on loops (invariants and parametrization of widening), MC has no such drawbacks and is fully automated. Additionally, it provides a counterexample, which has proven very useful in practice to identify and fix defects. In our first case study, MC has delivered a counterexample that precisely explained an observed behavior that was not understood before, and could not be reproduced or logged otherwise.

DV-based verification methods, one on hand, bring a couple of advantages for developing the *functionality* of the program. On the other hand, they loose ground when they cannot decide properties due to lacking theorems, reminding us on the fundamental difficulty of automating them. The overall workflow based on contracts and modular analysis might indeed be what programmers want for developing correct programs. It allows assigning responsibilities for correct behavior, and to tracing back errors to their underlying defects, thereby supporting the development of robust, re-usable software. The degree of automatism was surprisingly high, yet a number of properties also timed out, and then it was difficult for a user to tell whether the decision procedure is overwhelmed by the state space, or genuinely stuck. In any case, a rewrite of the program is a common reaction, which would not be desirable for a source-level timing analysis.

We have also evaluated another plug-in of Frama-C's ("WP"), which is also based on DV. The degree of automatism was inferior to the SPARK/gnatprove solution, and the tool was therefore not further considered. While this can most likely be attributed to the strong typing of the Ada/SPARK language, which provides more information (e.g., type ranges) for the provers, a source-level timing analysis should not have to rely on this.

Last but not least, the usability of AI was acceptable, as well, when disregarding Frama-C's specific choices about undefined behavior. But eventually, as soon as the precision of the default AI is insufficient to conclude on a property, more user interaction is needed to understand where exactly precision is lost, and to parametrize the algorithms to get better results. This would be detrimental for an automatic source-level timing analysis.

4.4.8 Choosing a Suitable Analysis Method

Based on the reviewed tools and considered properties, MC is the most suitable analysis method for source-level timing analysis. It is fully precise by default, can be automated and does not require user annotations. Furthermore, the production of counterexamples as an explanation for property violations turned out to be very useful. In an ideal case, we could leverage this capability to produce a *timing* trace, which explains how the WCET came to live. Its scalability with program size has been acceptable in our case study, and the completeness threshold could also be reached. Yet, scalability will become worse when the source code is annotated with timing, and therefore methods to enhance scalability will be necessary.

As a secondary analysis, AI might fill this gap. It has a lower complexity and thus scales better, but suffers from precision loss. For the second reason, we find it unsuitable as a primary analysis method. It is generally easier to abstract a program and thereby relinquish some precision in an otherwise precise analysis method, than increasing a precision of an analysis. The secondary nature that we consider is to leverage AI's ability to compute invariants, something that MC is unsuitable for. Invariants could replace some complex stretches of code which for MC does not scale. Further, AI may yield precise results on some programs, and therefore could be used as an initial analysis, and then refined by MC.

On the other hand, DV, although appealing in its workflow for developers, must be deemed unsuitable for timing analysis. It has a built-in nature of assume-guarantee reasoning, which is in favor of scalability, modularization and assignment of responsibilities, yet the very same aspect would lead to overapproximation in a timing analysis, and require specifying contracts. While in principle DV could operate on a whole program (e.g., by inlining all callees), none of the tools currently support this[6], since for general software engineering this would only lead to a monolithic analysis and no further benefits. We have also briefly investigated Frama-C's DV capabilities, which in addition has a lower degree of automation when the decision procedure is performed, therefore not suitable for an automatic timing analysis, either.

4.5 Chapter Summary

In this chapter we have reviewed the three most commonly used methods of formal functional verification and current tooling, namely *Model Checking*, *Abstract Interpretation* and *Deductive Verification*. We have presented three case studies which highlighted their strengths, weaknesses and boundary conditions. All of them have proven capable of performing a sound analysis of whole embedded programs, and in principle all of them could be used as a basis for a source-level timing analysis. However, Bounded Model Checking is the most suitable candidate and will therefore be used as a primary analysis method in upcoming chapters. It is automated, maximally precise and produces a counterexample, which would

[6]The *gnatprove* verifier does perform inlining for some callees, but that is an exception.

be a useful artifact to describe the timing of a program. However, its scalability remains a major concern.

We have also seen that independent of the tool and method, two major challenges are reoccurring. First, it is essential to set up the analysis context properly. That is, properties not directly visible in the code, e.g., endianness, sensor readings or initial states of programs, need to be properly communicated and handled. Otherwise the analysis results may become unsound. Second, loops are problematic in all analyses. They impede scalability and may require manual annotations (bounds, invariants). Since we have chosen Model Checking as analysis method for the next chapters, addressing the loop issue means that its weakness of scalability is addressed simultaneously.

Last but not least, *undefined behavior*, if existing in the source programming language, can refute any analysis. We therefore assume for the rest of this thesis that the program, if not completely functionally verified, is at least free from undefined behavior.

Chapter 5

Source-Level Timing Analysis

***

In this chapter we propose an approach towards a precise and scalable *source-level* timing analysis of embedded software, building on our findings about functional analysis in the previous chapter. Specifically, our goal is to introduce the basics of WCET estimation at source level, and to demonstrate the potential of such an approach. Therefore, we make two simplifications in this chapter. First, we assume that we have a way to determine the precise timing of each source statement, for which we show a general approach in Chapter 6. This chapter only sketches this part. Second, and closely related to this, the data shown here is based on experiments on an easy-to-analyze microprocessor, which is extended towards more complex processors in Chapter 7.

The chosen analysis method is Bounded Model Checking (MC), because it can precisely determine the longest feasible path, as it explores all paths in the program explicitly. Therefore, we expect tighter estimates, and ideally we no longer need flow constraints to be specified by the user, which makes WCET analysis safer and easier to conduct. It has been demonstrated that model checkers are useful at computing the WCET of simple programs [KPE09, KYAR10].

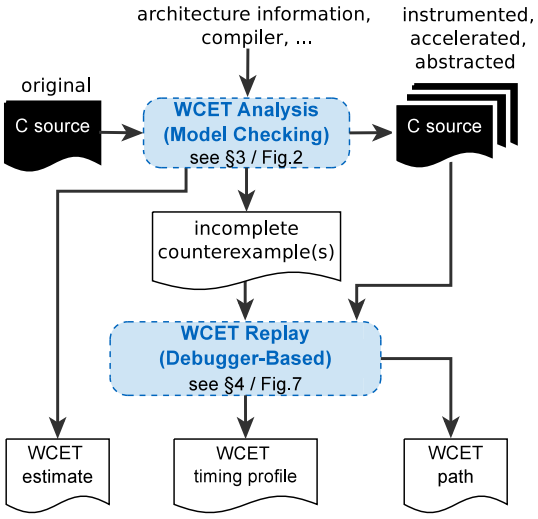


Figure 5.1: General workflow for source-level WCET analysis.

The idea is to take a program that is annotated with timing information, translate it into a model and verify the property "at program exit, elapsed time is always less than X", where X is a proposed WCET value, or in short, a WCET candidate. The model checker acts as an oracle, determining if there exists any execution that takes longer than this candidate. This process is repeated with changing candidates, until we arrive at the lowest upper bound where no counterexample is generated.

However, MC was found to scale poorly with program complexity [LGG+08, Wil04], and thus has never been seen as a competitive technique for WCET estimation. We show that this scalability problem can be mitigated by the shift of the analysis from machine code to source code level, and that source transformation techniques, such as *program slicing* and *loop summarization*, can be used to reduce the complexity and allow Model Checking to scale better. Meanwhile, we also reduce the number of oracle queries by applying a generalized binary search that is augmented by counterexample values.

Last but not least, this chapter introduces the process of *Timing Debugging*. We leverage the counterexample that is generated by the model checker, to reconstruct the precise path that is taken in the WCET case. This enables an interactive replay in an off-the-shelf debugger, showing not only the path being taken and how time is accumulating, but also the contents of all program variables.

Summary of our approach: The overall workflow consists of two steps, as depicted in Fig. 5.1. First, we estimate the WCET using a model checker, then we reconstruct the WCET path from the counterexample. The WCET estimation – shown separately in Fig. 5.2 – starts with evaluating the instruction timing of the program and meanwhile establishing a mapping between the instructions and the source code. Given this information, we annotate the source code with increments to a counter variable which is updated at the end of each source block according to the instruction timing. The resulting code is then sliced with respect to time annotations, to remove all computations not affecting the control flow. In the next step, we accelerate all loops that can be accelerated. Next, we over-approximate remaining loops with a large or unknown number of iterations. Then, we perform an iterative search procedure with a model checker to find the WCET, which is equivalent to the largest possible value of the counter variable that can be reached. We terminate the search procedure as soon as we find a WCET estimate within a precision specified by the user. Finally, we reconstruct the

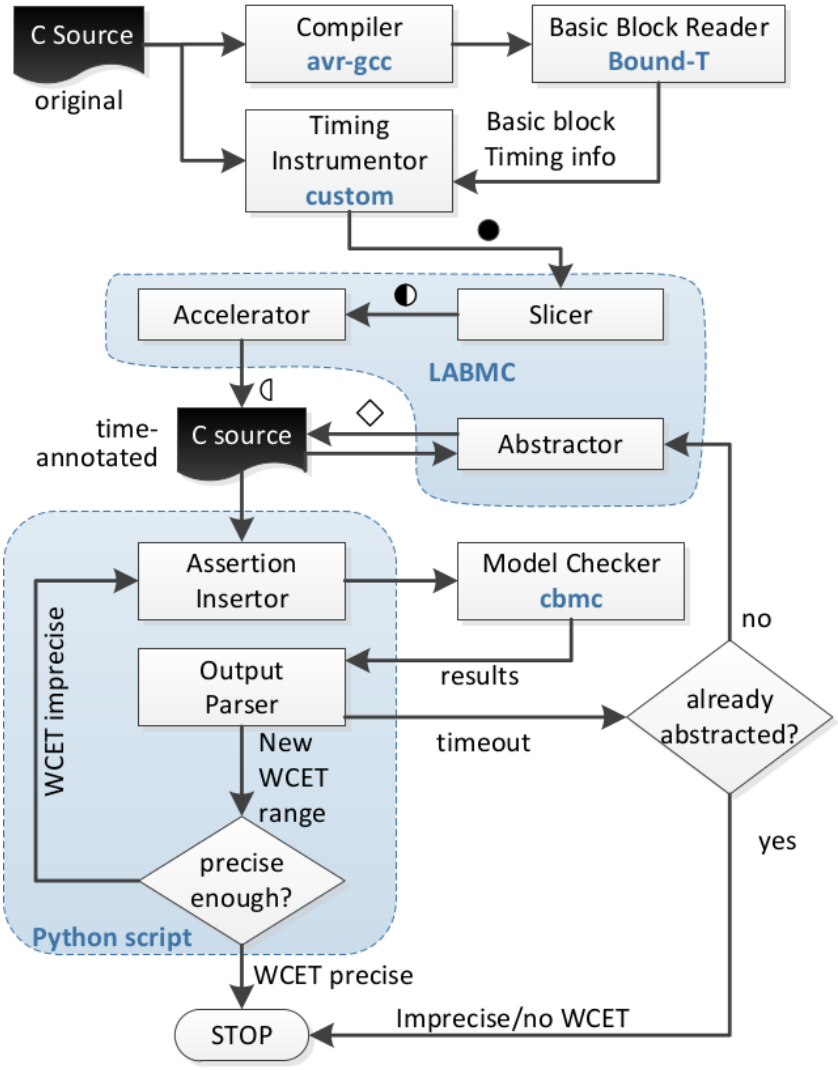


Figure 5.2: Source-Level WCET analysis using Model Checking.

WCET path from the counterexample, which contains the witness that the counter variable can reach the WCET value. Towards this, we execute the program in a debugger, while forcing decision variables to the values given in the counterexample. This reconstructs the precise path leading to WCET, and simultaneously collects a timing profile similar to *gprof*.

This chapter is organized as follows. In Section 5.1, we describe the workflow for source-level WCET analysis. In Section 5.3, we propose a debugger-based technique to reconstruct and replay the precise control flow, all variable contents, and a timing profile of the path leading to the WCET. Section 5.4 shows experiments with the commonly used Mälardalen WCET Benchmark Suite [GBEL10], to assess the impact of the source code transformations on scalability and tightness of the WCET estimates, in comparison to an ILP-based analyzer and a cycle-accurate simulator. Finally, we discuss the effectiveness, strength and weaknesses of source-level WCET analysis in Section 5.5.

5.1 A Method for WCET Analysis at Source-Code Level

The overall workflow is illustrated in Fig. 5.2 and shall now be explained in detail. Given a C program, a corresponding executable that results from cross-compilation for the target, and information about the target architecture, we take the following steps:

1. Estimate the time taken for each basic block in the machine code of the program.
2. Establish a mapping from these basic blocks to a set of source lines and instruments the C program with the timing obtained in the previous steps.
3. Detect and prepare the handling of persistent variables, i.e., the internal states of the program, since those may have an impact on the control flow and thus on the WCET.
4. Apply source code transformations to eliminate all irrelevant data computations and to summarize loops.
5. Add a *driver function*, which represents the entry point for the model checker, sets up the analysis context, and encodes the search for WCET as a functional property.
6. Use a model checker and an appropriate search strategy to estimate the WCET with desired precision.
7. If the model checker runs into timeout/memout, then we abstract the program and repeat the search.

The details of each of the above steps are given in the following sections. As a running example, we use a simplified version of the *fir* benchmark from the Mälardalen WCET benchmark suite.

5.1.1 Low-Level Analysis

We first analyze the executable file produced by the target-specific compiler and construct the CFG. Towards this, we identify basic blocks. These become the nodes in our control flow graph, and the edges represent branches or function calls, labelled with their branching condition. Finally, we determine the execution time for each basic BB by adding up the time taken by the contained instructions, and annotate we allow for branch instructions to have a variable timing (e.g., the instruction may take longer if a conditional branch is taken), we annotate the precise timing of the jump instructions to the edges of the graph. Through this, there is no overestimation due to such timing variations.

Our prototype implementation re-uses parts of the Bound-T WCET analyzer [HS02] to build the control flow graph and estimate basic block times. We therefore have exactly the same inputs for both our approach and the ILP-based analysis implemented in Bound-T.

5.1.2 Back-Mapping and Annotation of Timing

The next task is to match the control flow of the machine instructions with the control flow of the source code, and back-annotate the basic block timing in the source code, in the form of increments to a counter variable. The result shall be a source code that is instrumented with timing, in the following called *instrumented program*.

Specifically, here we annotate the basic blocks of the source with the instruction timing as close as possible to the corresponding source construct. That is, each maximal sequence of statements that is free of branches (just like basic blocks at machine code level) is immediately followed (or preceded, in the case of loop headers) by a timing annotation that reflects the execution time of the instructions corresponding to that block.

First of all, an approximate and incomplete mapping of machine instructions to source line numbers is given by debug information. However, it is incomplete because some instructions do not have a direct match in the source code, and it is only approximate because it does not allow distinguishing multiple basic blocks on the same source line. Complex expressions falling into multiple basic blocks, such as loop headers or calls in compound expressions, cannot be resolved with this information (more details in Section 6.2.2). We therefore use this information as an initial mapping, and apply safe heuristics to complete it.

Towards a mapping, we have to handle two cases:

1. *One-to-Many mapping.* One basic block in the executable corresponds to one or more continuous statements in the source code. This case is trivial to handle, all that needs to be done is to instrument the basic block before the last statement with the corresponding timing information.

2. *Many-to-one mapping.* Several basic blocks in the executable map to a single source statement. Typically, this case arises when a statement splits into different control paths in the machine instructions, for example in case of a division or a bit shift operation being converted to function calls or loops. In such cases, it is not directly possible to instrument the source code with this information, since it would require translating back the loop into the source first. To tackle this issue, we summarize the timing information from such multi-path instruction blocks, and instrument the source code with its worst-case timing value.

As an example, the timing of the source code in Figure 5.3a can be mapped as shown in Table 5.1. The table contains timing information of each basic block along with the span

Table 5.1: Mapping of instruction basic blocks (BB) to source code.

Start Line	End Line	BB (#)	Time	Comment
1	3	1	44	including for init.
3	3	2	21	for conditional
3	6	3	36	including for iterator
7	7	4	24	before division
7	7	mult.	737	division block
7	18	6	5	after division

of the block in the source code. This information is used to instrument the source code as shown in Figure 5.3b: We introduce a global counter variable `_time`, which is incremented through macro `TIC` by the corresponding amount of time at the respective source locations.

Basic block 2 is an instance of a one-to-many mapping. This block maps to the conditional part of the for-statement on line 3 of Figure 5.3a, and therefore the annotation is inserted just before the statement on line 8 in Figure 5.3b. Similarly, basic blocks 1 and 3 are one-to-many mappings. Basic block 1 implements the start of the function line 1, as well as the declarations and initialization on line 2 and also the for-initialization block on line 3. All these source level lines fall into a single source basic block. Similarly, the for-loop increment on line 3 along with statements on lines 4 and 5 map to basic block 3. The instrumentation in this case is placed in lines 6 and 12 respectively.

The division assignment in line 7 is an instance of a many-to-one mapping, i.e., many basic blocks, and therefore also some conditional jumps, map to this single statement. Here, these basic blocks include a compiler-inserted function call (for the long division). In particular, the compiler-inserted function contains a loop, for which we do not compute the timing precisely. Instead, we over-approximate this loop with its worst-case bound, and subsequently use a single timing annotation for the entire function. Thereafter, the statement at line 7 can be represented by three parts; one before the division (function call), one for division (function WCET) and the final one after the call. The resulting three timing annotations are shown on lines 14, 16 and 17 in Figure 5.3b.

This concludes the shift of WCET analysis from machine code level to source level. At this point, we have an *instrumented program*, i.e., a source code carrying the execution timing. From now on, we continue the analysis with the instrumented source code only.

5.1.3 Handling Initial Program States

A source-level analysis technique, such as the C model checker *cbmc*, could now be used to analyze the timing behavior of the program. However, with only the timing annotations, the analysis would be unsound.

A WCET estimation of a function f (including all its direct and indirect callees) must consider all possible inputs to f, and from those derive the longest feasible paths. Such inputs can be function parameters, but also referenced variables that are persistent between successive calls of f (they can also be seen as the hidden program state). In the C language, such persistent variables are *static* and *global* variables that are referenced by f or its callees.

We over-approximate the content of such persistent variables by initializing them with a non-deterministic value, as explained in Section 3.3. This guarantees that all the feasible and infeasible values of the persistent variables, as allowed by their data type size, are considered as inputs to f, and thus ensures soundness. Thus, the WCET estimate for f is always a safe over-approximation of the maximum observable execution time of f. It is possible to remove (some of) the infeasible values either with manual inputs by users or by analyzing the callees of f. This may lead to a tighter WCET, but we did not explore these approaches as they are orthogonal to our main work, and they run the risk of creating UNSAT assumptions (see Chapter 4), any thereby could refute the safety of the WCET estimate.

5.1.4 Timing and Path Analysis

Now that timing is visible in the source, existing (sound) code analyzers can be used to compute a safe upper bound of the WCET at source code level.

5.1.4.1 The Need for Target-Specific Analysis

Since the analysis takes place at source level, it is essential to include target-specific information, such as word width, endianness, interrupts, I/O facilities etc. If neglected, the behavior between the model in the analysis and the real target may differ, leading to unsafe results, as seen in Chapter 4. Most importantly, we provide target-specific pre-processor definitions and the word width with the corresponding flags that cbmc offers. For more details on how to include target-specific information for the model checker, we refer the reader back to said Chapter, where the specific pitfalls for cbmc have been explained. We will further employ checks during the WCET path reconstruction, which can identify missing or incorrect target information.

5.1.4.2 Setup of Analysis Context

To run a model checker on the instrumented (sliced, accelerated, abstracted) source code, we add a *driver* function to the source that implements the following operations in sequence:

1. Initialize counter variable `_time` to zero.
2. Initialize all input variables of the program to nondeterministic values according to their data type. This includes handling of persistent variables as described in Section 5.1.3
3. Call function f for which WCET shall be estimated.
4. Encode assertion properties to query for WCET, as described in the next section.

The driver function is subsequently used as entry point for the analysis by the model checker.

5.1.4.3 Iterative Search for the WCET

At this point, a model checker such as *cbmc* can verify whether the counter variable `_time` always carries a value less than X after the program terminates. In this section we explain how to choose candidates for X, such that we eventually approach the WCET.

A WCET candidate X is encoded as `assert(_time <= X)`, and subsequently passed to the model checker. Unless the model checker runs into a timeout or out of memory, only two outcomes are possible:

1. Successfully verified, i.e., `_time` can never exceed X. Therefore, X is a valid upper bound for the WCET.
2. Verification failed, i.e., `_time` may exceed X in some executions. Therefore, X is a lower bound for WCET. If a counterexample was generated, then it may contain a value $Y > X$, which then is a tighter lower bound for the WCET.

Our strategy is to use both outcomes to narrow down on the WCET value from both sides: Initially, we start with lower bound e_{lower} as zero, and upper bound e_{upper} as a very large value (*intmax*). We now place a number of assertions[1] in the program, where each is querying one WCET candidate X. In particular, the candidates are equidistantly spaced between e_{lower}

[1] We have empirically chosen N_{assert}=10; placing either more or less assertions usually takes longer, because either the computational effort is growing, or more iterations are required.

and e_{upper}; except for the first step, where we use a logarithmic spacing to initially find the correct order of magnitude. Subsequently, we invoke the model checker to verify the assertions – in the case of cbmc, all at once. For each assertion we obtain a result. We set e_{lower} as the largest X where the assertion was failing (or, when a counterexample with $Y > X$ was generated, to Y), and e_{upper} as the smallest X where the assertion was successfully verified. The search is now repeated with the new bounds, and stopped when these upper and lower bounds are close enough to each other, which can be interpreted as a *precision goal* for the WCET estimate. The full algorithm is given in Algorithm 1.

Algorithm 1: Iterative search for WCET bound

Input: instrumented C source code C, required precision P
Output: WCET estimate e_{upper}, s.t. $e_{\text{upper}} - e_{\text{lower}} < P$.
begin
  $N_{\text{assert}} \leftarrow 10$ `// number of assert per call`
  $e_{\text{lower}} \leftarrow 0$
  $e_{\text{upper}} \leftarrow intmax$
  $p = e_{\text{upper}} - e_{\text{lower}}$
1 **while** $p > P$ ***and not*** *timeout* **do**
    if $e_{upper} = intmax$ **then**
      candidates $\leftarrow$ logspace($e_{\text{lower}}..e_{\text{upper}}, N_{\text{assert}}$)
    else
      candidates $\leftarrow$ linspace($e_{\text{lower}}..e_{\text{upper}}, N_{\text{assert}}$)
2   $C' \leftarrow$ insert `asserts` for candidates into C
3   results $\leftarrow$ model checker (C')
    for $i = 1$ **to** N_{assert} **do**
      if verified(results[i]) **then**
        $e_{\text{upper}} \leftarrow \min(e_{\text{upper}}, \text{candidates[i]})$
      else
4       $B \leftarrow$ getCounterexample(results[i])
        $e_{\text{lower}} \leftarrow \max(e_{\text{lower}}, \text{candidates[i]}, B)$
5   $p = e_{\text{upper}} - e_{\text{lower}}$

At any point in time the model checker could terminate due to either a user-defined timeout or when it runs out of memory. In such cases the algorithm returns the WCET estimate as at-this-point tightest bound e_{upper} that could be verified. In combination with the precision goal P, this gives the user a fine control over how much effort shall be spent on computing the WCET. For example, an imprecise and fast estimate may be sufficient during early development, whereas a precise analysis may be only of interest when the WCET estimate approaches a certain time budget.

The maximum number of search iterations can be determined in advance; in the worst case the number of search iterations n is

$$n = \left\lceil \log_{N_{\text{assert}}} \left(\frac{e_{\text{upper}} - e_{\text{lower}}}{P} \right) \right\rceil , \tag{5.1}$$

where $e_{\text{upper}} = intmax$ and $e_{\text{lower}} = 0$, if no a-priori knowledge about the bounds of WCET is available. Usually the number of iterations is lower, since the values found in the counterexamples speed up the convergence (point 4 in Alg. 1).

Leveraging A-Priori Knowledge. In this chapter we assume that no information about the timing behavior of the program is available. If, however, the user has some knowledge on the WCET bounds already, then these bounds can be tightened by the algorithm, reducing the number of iterations. If an upper bound is known, then e_{upper} can be initialized with that bound, and the algorithm tightens it up to the required precision. Similarly, if a lower bound is known from measuring the worst-case execution time on the target (e.g., from a high watermark), then e_{lower} can be set to that measured value.

Implementation. The WCET search procedure was implemented as a Python script. It further implements monitoring of memory and CPU time as given in the experiments section. As model checker we used *cbmc* with various solver backends, since it is currently ranks among the top MC tools [Bey14]. However, other model checkers, such as *cpachecker* could be used as alternatives with only little changes.

5.2 Enhancing Scalability

The source code is now instrumented with its timing behavior, and ready to be analyzed for WCET. However, a direct application of Model Checking to compute the WCET of large and complex programs would not scale due to the size of state space, as seen in the previous chapter. The analysis time could quickly reach several hours for seemingly small programs, and memory requirements may also quickly exceed the available resources. We have seen this problem in our case studies, where loop structures have proven to be a bottleneck. Our next step, therefore, are source code transformations which retain the timing behavior, but reduce the program complexity, especially addressing the loop problem.

The transformations that we propose are a mix of existing techniques from general software engineering, with new aspects specific for timing analysis. They exploit the additional information available in the source code, such as data types and clearly visible control flows, to enable effective semantic transformations. The proposed transformations are executed sequentially in three *stages*, which we explain in the following. All three stages require only little computational effort themselves, and therefore speed up the overall process of WCET estimation.

5.2.1 Stage 1: Slicing (◐)

In Chapter 4, we have identified that dead code can adversely impact the scalability. However, there we only considered whole functions that are being unused. There exists the more fine-granular method of *program slicing*, which can safely remove all parts of the code, even reachable ones, which do not affect the property being analyzed.

Program slicing was first introduced by Mark Weiser in 1981 [Wei81]. Given an imperative program and a slicing criterion, a program slicer uses data flow and control flow analysis to eliminate those parts of the program that do not impact the slicing criterion. That is, it removes statements which do not influence the control flow related to the criterion, and do not change its value. The resulting program is called a *program slice*, and behaves identically w.r.t. the slicing criterion, but has a reduced complexity. For the specific case of WCET

```
int task(int fir, int scl) {
  int i, out, acc=0;
  for(i = 1; i < 35; i++) {
    acc += fir;
    fir <<= 1;
  }
  out = acc/scl;









  return out;
}
```

(a) Original code

```
#define TIC(t) (_time += (t))
unsigned long _time = 0;

int task(int fir, int scl) {
  int i, out, acc=0;
  TIC(44);    // BB1
  for(i = 1;
    TIC(21),    // BB2
    i < 35; i++) {
    acc += fir;
    fir <<= 1;
    TIC(36);    // BB3
  }
  TIC(24);    // BB4
  out = acc/scl;
  TIC(737);  // mult.  _divmodsi4
  TIC(5);     // BB6
  return out;
}
```

(b) Instrumented

```
#define TIC(t) (_time += (t))
unsigned long _time = 0;

int task(int fir, int scl) {
  int i, out, acc=0;
  TIC(44);
  for(i = 1 ;
    TIC(21),
    i < 35; i++) {
    acc += fir;
    fir <<= 1;
    TIC(36);
  }
  TIC(24);
  out = acc/scl;
  TIC(737);
  TIC(5);
  return out;
}
```

(c) Sliced

```
#define TIC(t) (_time += (t))
unsigned long _time = 0;

void task(void) {
  int i, _k0, _k1;
  TIC(44);
  {
    _k0 = 1;
    i = 35;
    _k1 = i - _k0;
    TIC(21*_k1 + 36*_k1);
    TIC(21);
  }
  TIC(24);

  TIC(737);
  TIC(5);

}
```

(d) Accelerated

Figure 5.3: Example for source code instrumentation and transformations to compute the WCET of a function `task`. Steps c) and d) enhance scalability while preserving precision.

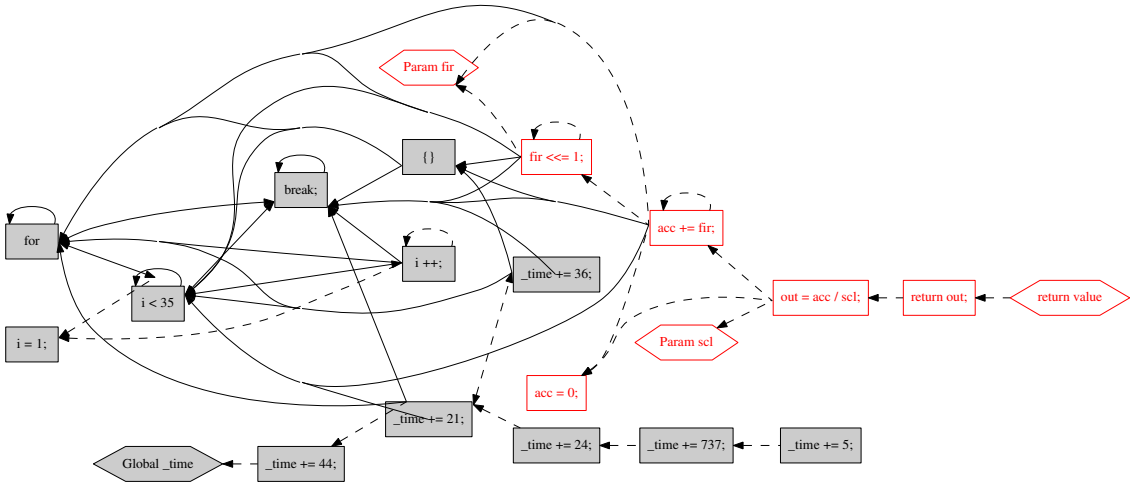


Figure 5.4: Program Dependence Graph of the program from Fig. 5.3b. Sliced statements are unshaded/outlined in red.

analysis using a counter variable, our slicing criterion is the value of this variable upon program termination. For example, Figure 5.3c shows the slice of the instrumented program from Fig. 5.3b, when the slicing criterion is the value of the variable `_time` at program exit. As it can be seen, several parts of the original program can be omitted (e.g., the actual computation of the return value), since they do not influence the value of this variable.

Slicing works by constructing and evaluating a Program Dependency Graph (PDG), which captures the control and data flow dependencies of a program. Figure 5.4 shows such a PDG corresponding to the example code. This graph has two kinds of edges to denote dependencies: dashed edges denote a data dependency, and solid edges denote a control flow dependency. For example, the loop increment `i++` in the center of the graph is control-dependent on the loop condition `i<35`, and data-dependent on itself as well as on the loop initialization.

Given the PDG from Fig. 5.4 and a slicing criterion, we start at the node that corresponds to the location of the criterion, and traverse the PDG from this point until all the root nodes of the graph are reached. Subsequently, we remove from the program all statements and expressions that correspond to nodes that have not been visited during this traversal. For our example program in Fig. 5.3b, the criterion is the latest possible assignment to variable `_time` in line 17, and the corresponding PDG node is `_time += 5` (in the lower right corner of the graph). When traversing the graph from there, the outlined/red nodes (e.g., `out = acc/scl`) are not reachable. Therefore, these parts of the program do not impact the value of variable `_time` at our location, and can be safely removed from the program.

We used a program slicer that builds an inter-procedural program dependency graph [PB95], capturing both intra/procedural and inter/procedural data and control dependencies. It then runs one pass over this graph to identify statements that impact the property to be verified and outputs the sliced code. The slicer is a conservative one, which means that it discards statements only when sure that the statements do not affect the timing analysis. In all other cases, the statements are preserved.

5.2.2 Stage 2: Loop Acceleration (◮)

Another scalability issue in source-level verification that we have identified earlier, are loops. However, whereas we had then applied a specific technique to reduce loops, there exists a more general one. *Loop acceleration*, also known as *loop summarization*, describes the action of replacing a loop with a precise closed-form formula capturing the effects of the loop upon its termination. The resulting program has a reduced number of loops, and therefore is easier to analyze. This has been shown to be effective for in particular for MC of source code [DCV+15, Bey14].

For loop acceleration to be applicable, the loop should have the following characteristics:

- The loop should iterate a fixed number of times, say n, which could either be a constant or a symbolic expression. For example, the loop may execute 10 times or n times, or $x \cdot y$ times and so on.
- The statements in the loop constitute only of assignments and linear arithmetic operators, that is, first order polynomial computations (and no higher order).
- The loop body consists of straight line code, that is, there are no conditional statements such as if-else blocks inside the loop body.

When a loop satisfies the above constraints, it is possible to replace the loop with simple linear recurrence relations that precisely compute the summarized effect of all the iterations of the loop, on each of the assignments in the loop body.

As an example, consider the loop in Figure 5.3c: Here, `_time` is incremented in line 12 within a loop body through macro `TIC`. The effect of this repeated increment of `_time`, can be summarized by the expression `_time` = `_time` + $n \cdot 36$, where n is the number of loop iterations. Line 11 in Figure 5.3d shows the accelerated assignment. Note that two new variables `_k0` and `_k1` have been introduced, representing the initial value of the loop counter (`_k0`) and the number of loop iterations (`_k1`). After accelerating all variables, the loop in Figure 5.3c can be removed and replaced by the statements given in Figure 5.3d lines 7 through 13. Here, lines 8 to 11 represent the effect of all the 34 iterations of the loop. Line 12 captures the time taken for evaluating the loop condition after the final iteration.

As the accelerated program of Fig. 5.3d is free of loops, it is less complex than the instrumented program. For this example, the program size (as determined by cbmc) reduces from 325 steps for the instrumented program to only 60 steps for the accelerated program. Last but not least, note that a prior slicing is important for loop acceleration. If we had not sliced the instrumented program w.r.t. `_time`, then the above loop could not have been replaced, as the assignment to `acc` on line 4 of cannot be accelerated due to the assignment to `fir` on line 5.

We implement acceleration as proposed in [DCV+15], mainly because it has been proven effective on both the SVCOMP C benchmark programs and industrial programs [Bey14].

5.2.3 Stage 3: Loop Abstraction (◇)

Finally, further reduction of complexity be accomplished by abstraction (see Section 3.2), at the cost of reduced precision. In contrast to abstractions in AI, where the analysis itself models a superset of all program traces, here we abstract the program before giving it to the analyzer, which therefore is effective for all source analysis methods.

Specifically, we propose to abstract loops that did not fulfill the criteria for acceleration, and therefore still impede scalability. For instance, suppose a loop contains assignments in

```
1  for (i=1;
2       TIC(20), i<=len;
3       TIC(8), i++) {
4     TIC(35);
5     if(ans & 0x8000) {
6        TIC(31);
7        ans ^= 4129;
8     } else {
9        TIC(23);
10       ans <<= 1;
11    }
12 }
```

(a) Loop with conditional

```
1  int _k0, _k1;
2  _k0 = 1;
3  i = len + 1;
4  _k1 = i - _k0;
5  ans = nondet();
6  TIC(_k1*(20+8+35)+_k1*31);
7  TIC(20);
8
9
10
11
12
```

(b) Abstracted loop

Figure 5.5: Abstraction of loops with conditional statements in body.

the form of if-else statements as shown in Fig. 5.5a, so that it cannot be accelerated. It is still possible to replace the loop with an over-approximate formula as follows. The easiest way to achieve this is to replace the problematic assignment with a non-deterministic one (which must be supported by the analysis method) that allows for more values than the original program, i.e., overapproximates.

Figure 5.5b shows the abstracted version of this loop, which makes such a non-deterministic assignment to the variable `ans`. By calling the function `nondet()` we specify that `ans` can take a value in the range of its type, which then is a superset of all the feasible values among all possible executions. As a result, the if-else construct can be removed.

After this abstraction, the loop can be accelerated as explained before. Line 6 contains the accelerated assignment to `_time`. In this, `_k1*(20+8+35)` accounts for the time increments in Figure 5.5a corresponding to the loop condition evaluation (line 2), loop counter increment (line 3) and the first part of the loop body on line 4. Finally, line 7 captures the time taken for evaluating the loop condition after the final iteration.

If a loop contains unstructured code, such as `break` and `return` statements, these are easily handled through a non-deterministic Boolean variable that allows these statements to be executed or not executed.

5.2.3.1 Overapproximation of Timing

The trailing `_k1*31` in Fig. 5.5b on line 6 summarizes the time taken by the if-statement forming the remainder of the loop body, but it is somewhat special. It contains a WCET-specific modification of the abstraction. Since now the values of `ans` have been abstracted, we can no longer reason about which branch of the if-statement is taken in the WCET case. Therefore, when abstracting such undecidable conditional statements, we over-approximate the effects on our time variable by only considering only the longest branch (here: the then-branch in line 6 with 31 clock cycles). Note that this does not change the WCET estimate, because the model checker would have picked the longer branch anyway, since the possible values of `ans` include the case allowing the then-branch to be taken. It is thus a safe modification to the abstraction in the context of WCET analysis, which however reduces the complexity of the program further.

To implement abstraction, we again used the LABMC tool [DCV+15]. However, the standard abstraction has been tailored for the counter variable `_time`, as explained above, to pick the maximum time increment from the branches.

5.2.4 Further Transformations

A number of other source transformations could be applied to further reduce the program complexity and yet retain the worst-case timing information. For example, after the back-annotation of timing, the counter variable could be summarized by merging increments belonging to the same source block. However, this would make it harder to understand how source-level statements contribute to timing, and thus has not been investigated in the context of this work.

Furthermore, cbmc itself (more specifically, goto/instrument, a front-end of cbmc), features transformations that can be applied to the source program, such as k-induction for loops, constant propagation and inlining. While our toolchain supports using those cbmc features, none of them have proven effective in reducing the complexity or analysis time in our WCET framework. In particular, the loop acceleration included there took longer than our entire WCET analysis and did not improve the analysis time thereafter, and "full slicing" even produced unsound results. Therefore, our source transformations introduced above are justified, because they have a proven impact on WCET analysis.

5.3 A Method for Reconstructing the WCET Trace

From a software developer's point of view, the WCET value in itself (and its inputs) are of limited use. Without understanding the exact path and the decision variables leading to the WCET, counteractive measures are limited to a time-consuming trial-and-error approach. Instead, the following information would be of high value for comprehension and active mitigation of the WCET case:

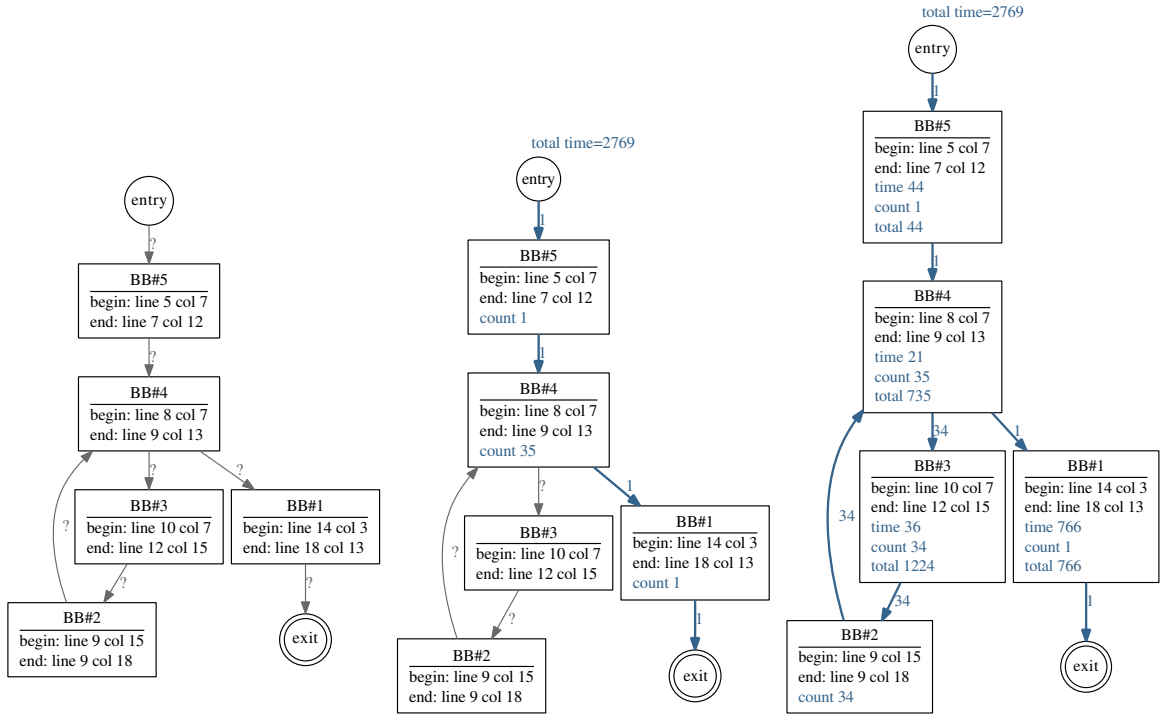
1. The exact inputs leading to the WCET path,
2. a concrete walk-through of the worst/case path where all variables can be inspected, to identify the timing drivers, and
3. a timing profile of the worst/case path (how much time was spent where?).

Especially a detailed walk-through of the worst-case path is very important to understand why a certain path incurs high timing cost. From our experience, it is not sufficient to know only the WCET path. Oftentimes the specific values of decision variables are important to understand why this path was taken, but such information is not provided by any tool that we know of. The user is therefore left with the mental puzzle of finding an explanation how the presented path came to be, and at times there can be very subtle and unintuitive reasons. For example, in our *prime* benchmark, we found that an integer overflow in a loop condition caused the loop to terminate only after a multiple of its expected bound. In turn, this overflow depends on the word width of the processor being used (more details about this case are given in Section 5.4.3). In such a case a developer would likely struggle to explain why the loop did not obey to its bounds, and thus not understand the WCET estimate.

In summary, the WCET trace that we want to reconstruct shall be detailed enough to inspect not only the control flow and its timing, but also all data that is relevant for the specific path being taken. Towards that, we want to leverage the final counterexample from Alg. 1 to provide a maximum-detail explanation of how the WCET can be reached.

5.3.1 Trace Reconstruction Method

The challenge in reconstructing the WCET path from a counterexample is illustrated in Figure 5.6, using the program introduced earlier in Figure 5.3a. The goal is to annotate the



(a) Control flow graph **(b)** Typical output of analysis **(c)** Fully reconstructed path

Figure 5.6: Reconstruction problem of WCET path for example code from Figure 5.3a.

corresponding control flow graph in Fig. 5.6a with execution counts at both its nodes (basic blocks in source code) and its edges (branches being taken). However, the counterexample produced by the model checker only contains sparse information as shown in Fig. 5.6b. Typically, it only provides a subset of the visited code locations, variable assignments that are relevant for branches being taken, and assignments to `_time` oftentimes occur only at the end of the counterexample, depending on which solver backend was chosen. In fact, the counterexample is only guaranteed to provide exactly one value for the variable `_time`, which is at the location where the WCET assertion is failing.

It is clear that the counterexample provided by the model checker is insufficient as an explanation for the WCET path, much less for values of arbitrary variables. It could carry even less information than what is provided by an ILP-based approach, where execution counts for all basic blocks on the WCET path are available. Without such data, neither can we compute a timing profile in the presence of function calls, nor is it possible for the developer to understand or even walk through the WCET path.

Therefore, towards reconstruction of the WCET path, we have to interpolate the control flow in between the locations given in the counterexample, and deduce variable valuations that are not available (most importantly, variable `_time`). There are two fundamental approaches for reconstructing the path:

1. **By Analysis:** Use SAT solvers, AI, or a similar technique to fill the "location gaps" of the counterexample, and to conclude about assignments of all (possibly sliced) variables. This is expected to be computationally complex and not necessarily precise.

2. **By Execution:** Execute the code with the worst/case inputs. An "injection" of critical values beyond inputs is required, for example to all persistent variables. This

could be compiled into the application through look-up tables that contain the known assignments from the counterexample, or done dynamically during execution.

It should be clear that an execution is preferable in terms of precision and speed, however, there are some challenges in such an approach:

1. **Using an Interpreter:** Whereas the most logical choice, only a few good interpreters are available for the C language. Most of them have limited "stepping" support (e.g., *cling* uses just-in-time compilation; which means that stepping through statements would inline/flatten functions), and in general they are slow. There would be no support for target-specific code, such as direct pointer dereferencing. Additionally, often only a subset of the language is implemented.

2. **Instrumented Execution:** Compile and run the application, preferably on the target for maximum fidelity, while capturing an execution trace which subsequently could be replayed. Capturing a complete trace including variable valuations could produce a huge amount of data and incur memory and timing overhead. If the program under analysis is supposed to run on an embedded target, then it might not be feasible to capture the trace for reasons of memory limitations or missing interfaces.

We decided for an execution-based approach, since this is computationally less complex than analysis, and thus expected to be faster. The problem of insufficiently granular interpretation and trace capturing has been addressed as follows: Our path reconstruction is accomplished by means of executing the application in a debugger, whilst injecting variable valuations from the counterexample when necessary. By choosing a debugger as replay framework, replaying the WCET has the potential to become intuitive to most developers, and thus could be seamlessly integrated into the software development process. Furthermore, debuggers are readily available for most targets, and some processor simulators even can be debugged like a real target (in our case, *simulavr* allows this). And, finally, target-specific execution context, such as word widths, endianness etc, are bound to be correct, which otherwise would be a common pitfall for incorrect results (see Chapter 4). However, to use these advantages to full capacity, the replay process must be of low overhead, and able to be fully automated to not require any additional inputs compared to traditional debugging.

5.3.1.1 Preparing Replay

Our proposed replay process is illustrated in Figure 5.7. We start by compiling the instrumented (or sliced, accelerated, abstracted) program with additional stubs (empty functions) for each "nondet" function (see Section 3.3). Then, we load the program into a debugger. Depending on which compiler was chosen, this could either be the host's debugger, or the one for the target (connecting to either an actual target or to a simulator). Since in all cases the replay process is similar, for the remainder of this chapter we do not distinguish anymore between a host-based replay, a simulator or a real target.

Finally, it should be mentioned that the replay path is based on the time instrumentation in the sources, which is why the path always exists unless the model checker is handed wrong target-specific settings (e.g., word widths), or unless the analysis is unsound.

5.3.1.2 Injection of Nondeterministic Choices

To inject the critical variables as found by the model checker, we set breakpoints on the inserted nondet stubs and start the execution. As soon as the debugger reaches one of our

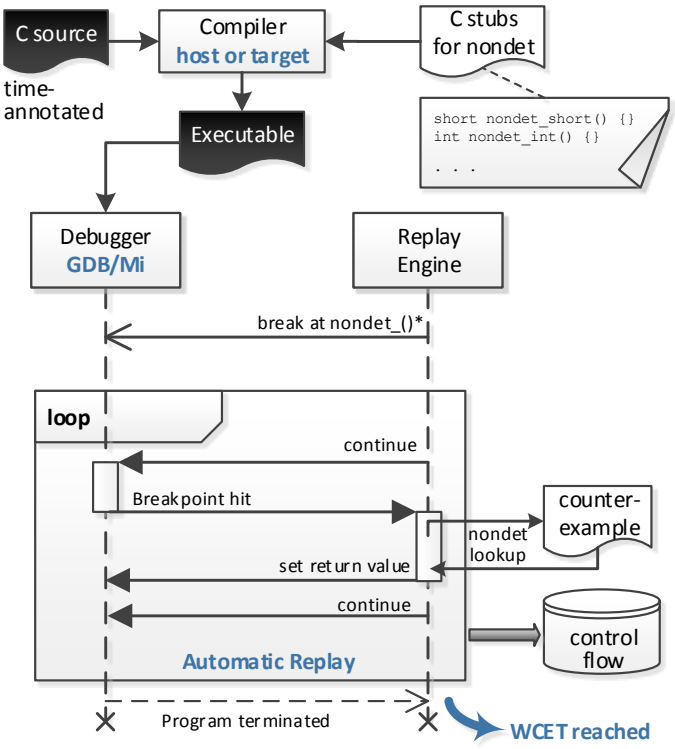


Figure 5.7: Automatic WCET replay using a debugger.

breakpoints (additional user breakpoints for stepping may exist and are not interfering), we automatically inject the correct value as follows: First, we query the current call stack to identify the call site, i.e., the source location of the nondet call. Knowing the location of the designated nondet assignment, the value that leads to WCET is extracted from the counterexample, and subsequently forced as a return value in the debugger. As an effect, the call to the empty nondet stub returns the critical value suggested by the model checker. After that, the execution is resumed automatically.

However, there exists a second source of non/determinism, besides the explicit "nondet" function calls. In the C language, every uninitialized variable that is neither static nor at file scope, initially carries an unspecified value. Therefore, every such uninitialized variable is considered by the model checker as non/deterministic input, as well. Since no explicit "nondet" function calls exist, the breakpoints inserted earlier do not allow us to inject values into those variables. As a solution, we first identify all uninitialized local variables from the parse tree of the C source (declarations without right-hand sides, mallocs, etc.), and then insert additional breakpoints for every such variable that is mentioned in the counterexample. Through this, injecting values into uninitialized variables is handled in the same way as the nondet function calls (not shown in the Fig. 5.7 for clarity).

With this technique, the injection of the critical variable values from the counterexample is accomplished without any user interaction, and without embedding the assignments in the program itself (zero memory overhead). Furthermore, this live injection allows for additional checks during the execution, such as matching the assignment with the actual call site, and ensuring that the execution path does not deviate from the analysis.

5.3.2 Identification of WCET-Irrelevant Variables

The valuations of some variables do not have an effect on the control flow, and thus do not influence the timing[2]. As explained before, such variables are identified and sliced away during the analysis phase. In particular, both our pre-processing (slicing, acceleration, abstraction), as well as the model checker itself remove such variables. Consequently, the counterexample does not include valuations for variables that have been found irrelevant for the WCET.

As an effect, any location having a non/deterministic assignment that is visited during replay *and* and does not have an equivalent assignment in the counterexample, indicates that the respective assignment is irrelevant for WCET. We highlight such irrelevant statements to the developer, to help focus on the drivers for the worst/case timing, and not get distracted by surrounding code.

5.3.3 Collecting a Timing Profile

When larger programs are being analyzed, it may quickly become impractical to step through the whole path to find the main drivers of the worst/case execution time. To help the developer identify interesting segments that need to be stepped through in the debugger, we also generate a timing profile of the WCET path, showing which location was visited how often, and how much time was spent there.

Towards this, we capture the complete control flow on the fly during the replay. Since the timing profile is especially useful for larger programs, capturing the control flow during the debugger execution must scale well with growing path length and thus cannot be realized by a slow step-by-step execution in the debugger. Instead, we set a hardware *watchpoint* on our counter variable. That is, every time this variable is modified, the debugger pauses execution and notifies the replay engine of the new variable content. Since hardware watchpoints are realized through exceptions, the debugger can run without polling and interruptions between the watchpoints, and therefore the control flow is captured with little additional delay. Considering that the counter variable is embedded at least once per source block, the sequence of all reached watchpoints (their valuation and location), represents the actual control flow in the source code. As a result, a timing profile similar to the outputs of the well-known tools *gprof* or *callgrind* can be reconstructed and intuitively used by the developer. Table 5.2 shows the resulting flat WCET timing profile for the *adpcm decode* benchmark. Note that additionally to the shown per-function metrics, the execution times are also available at the even finer granularity of source blocks, which helps pinpointing the timing bottlenecks to specific source blocks within the functions.

More Safety Checks. Last but not least, it is essential to detect any problems during the complex processes of slicing, abstraction and replay that could lead to wrong WCET estimates. We have discussed the issue of unreachable properties, due to inconsistent assumptions in Chapter 4. These issues can be ensured to be absent, since the WCET replay offers the chance to insert many checks. Inter alia, we can ensure that the sequences and stacks of functions calls are matching between model and target and that all assignments in the counterexample are actually part of the replay, in particular the valuations of the counter variable. Additionally, during replay we replace every `assume` statement with an `assert`

[2]Note that this only holds true for cache-less processors.

Table 5.2: Timing profile obtained from WCET path of *adpcm decode* benchmark.

%total	cycles	%self	cycles	calls	self/call	total/call	name
100.0	69,673	35.3	24,593	1	24,593	69,673	decode
13.7	9,522	13.7	9,522	2	4,761	4,761	upzero
13.2	9,168	13.2	9,168	2	4,584	4,584	uppol2
12.9	8,984	12.9	8,984	2	4,492	4,492	filtez
9.3	6,472	9.3	6,472	2	3,236	3,236	uppol1
5.5	3,854	5.5	3,854	2	1,927	1,927	filtep
5.1	3,576	5.1	3,576	2	1,788	1,788	scalel
2.5	1,758	2.5	1,758	1	1,758	1,758	logscl
2.5	1,746	2.5	1,746	1	1,746	1,746	logsch

statement to ensure that the assumptions encoded in the model are satisfied on the WCET path. These measures have been found effective in detecting user errors, such as debugging the program with the trace of an unrelated program (or a different version), and even more subtle problems, such as providing wrong target architecture information to the model checker (e.g., wrong word width or endianness).

5.4 Experimental Evaluation

We applied our source-level timing analysis to both the Mälardalen WCET Benchmark Suite [GBEL10] and the more realistic PapaBench autopilot benchmark [NCS+06], to evaluate the performance and the tightness of WCET estimates. The benchmarks are described in Table 5.3, and also used in subsequent chapters. As a target, we used the Atmel ATmega 128 [Kuh98], for which WCET analyzers (Bound-T [HS02]) and simulators (simulavr) are freely available and can be used as a baseline for evaluating our approach. Bound-T is a traditional, binary-level WCET analyzer based on the ILP/IPET approach [WG14], but additionally tries to exclude infeasible paths by a combination of call context separation, constant propagation, and arithmetic analysis.

5.4.1 Setup and Reference Data

Selected Benchmarks. From PapaBench, we selected the periodic task of the autopilot, which has features such as peripheral accesses, floating-point operations, unstructured code and bitwise operations, but rather few loops. The benchmarks shown in Table 5.3 were chosen to form a representative subset of the Mälardalen WCET benchmark suite, such that interesting properties are covered, such as multi-path flows, nested loops and multidimensional arrays, except for recursion (this is currently not supported by the timing instrumentor, but in principle can be done).

Host Platform. We conducted our experiments on a 64bit machine with a 2.7GHz Intel Xeon E5-2680 processor and 16GB RAM, using cbmc 5.6 as model checker. As cbmc's backend we have used a portfolio of solvers, consisting of minisat (v2.2.1), mathsat (v5.3.10/GMP), cvc4 (v.1.5pre/GLPK-cut), z3 (v.4.5.1), glucose (v4.1-simple) and yices (v.2.5.1/GMP). We stopped the analysis as soon as the first solver provided a WCET estimate. Solvers finishing within the same second were considered equally fast. All programs have been analyzed sequentially,

Table 5.3: Benchmark selection.

Benchmark	Description	S	F	N	A	B	C	U	SLOC
adpcm	adaptive PCM algorithm								879
bs	binary search				✓				114
bsort100	bubblesort			✓	✓				128
cnt	counts positive numbers in matrix			✓	✓				267
cover	large switch case in loop	✓							240
crc	CRC computation on data				✓	✓	✓		128
fdct	fast DCT transform	✓			✓	✓			239
fibcall	Fibonacci number calculation								72
fir	FIR signal filter on 700 elements sample			✓	✓				276
insertsort	insertion sort			✓	✓				92
jfdctint	DCT on 8x8 pixel block	✓			✓				375
matmult	multiplication of 20x20 matrices	✓		✓	✓				163
ndes	complex embedded code				✓	✓	✓		231
ns	highly nested array search			✓	✓				535
nsichneu	Petri net simulation, huge function						✓		4,253
prime	prime number test								47
ud	LU decomposition on 50x50 matrix			✓	✓				163
PapaBench	Paparazzi UAV autopilot	✓	✓		✓	✓	✓	✓	4,732

A=arrays, B=bit ops, C=multi-clause conditionals, F=floating-point
N=nested loops, S=control flow independent of data, U=unstructured

to minimize interference between the analyses and with third-party background processes. The computational effort (CPU time, peak memory usage) are derived from the Linux system call `wait3`, and thus expected to be accurate.

Bounding the Control Flow. In most cases our approach could bound the control flow automatically, with no manual inputs required. However, in two benchmarks, namely the instrumented versions of *bs* and *insertsort*, cbmc could not find the bounds automatically and we were not able to accelerate or abstract either, and thus bounds had to provided. This occasional need for manual bounds is discussed in detail in Section 5.5.4, and traces back to the well-known Halting Problem [Tur38]. However, it should be noted that a guessed bound is sufficient, since a non-tight specification has no impact on the precision, and a too low specification can be detected by the Model Checker and subsequently corrected.

In contrast, we frequently had to provide loop bounds for the ILP-based WCET analyzer. Furthermore, in an ILP approach the bounds cannot be verified, and thus have to be specified correctly and tightly in the first place. Towards this, whenever the ILP-based estimation required manual loop annotations, we have taken the deduced precise bounds from our approach and handed them to the ILP-based analyzer. Consequently, both techniques had similar preconditions for WCET estimation.

Simulation Baseline. All benchmarks were also simulated with the cycle-accurate instruction set simulator *simulavr*, with the goal of having sanity checks for the WCET estimates, and also to get an impression on their tightness. Whenever possible, the simulation was performed with those inputs triggering the WCET. In other cases, we used random simulations in an attempt to trigger the WCET, but naturally we cannot quantify how close to the actual

WCET we have come (e.g., in nsichneu or PapaBench[3]). Hence, the simulation results can only serve as a lower bound for the actual WCET (i.e., no estimate must be less) and as an upper bound for the tightness (i.e., the overestimation of the actual WCET is at most the difference to the simulation value).

Abstract Interpretation. To provide a comparison to an AI-based approach to source-level WCET analysis, we also used Frama-C [CKK+12] to obtain estimates. It can be used on our time-annotated (or sliced, accelerated, abstracted) source codes, in lieu of the Model Checker, with only some minor syntactic modifications. Although Frama-C offers additional means to increase precision (see Section 4.2), we have not made use of them, to enable a comparison between MC and *classical* AI. Table A.3 in the appendix gives the results with these features enabled, which however needs manual tuning and increases computational complexity, as explained in the previous chapter.

5.4.2 Results

Figure 5.8 summarizes our experiments. We evaluated our technique of WCET estimation for each source processing stage (Section 5.2) of the selected benchmarks, that is:

1. instrumented with execution times (●),
2. sliced w.r.t. timing (◐),
3. loops accelerated (◗) and
4. abstracted (◇).

We compare the tightness of our WCET estimate with that of Bound-T, an ILP-based binary-level WCET analyzer. We computed the tightest possible WCET, i.e., setting precision $P = 1$, while allowing for a (relatively long) timeout of one hour, to show how our source transformations influence the tightness. Cases denoted with *timeout* are those where no solution was found within that time budget. The two vertical dashed lines denote the average overestimation relative to the simulation for both source- and binary-level analysis. In the figure, we omit sliced and accelerated versions, since they have identical results as the instrumented version. The raw data is given in Table A.1 in the appendix.

Table 5.4 summarizes the computational effort of the estimation process for a practical time budget of ten minutes and a precision of 10,000 clock cycles. The table also quantifies the speedup we have achieved with our source transformation techniques. Again we denote *timeout* in cases where the WCET could either not be bounded within the time budget, or not up to the required precision. The column *prog.size* shows the number of program steps found by cbmc. Cases where the program size is not given (viz., in *fir* and *prime*) indicate a state-space explosion. That is, the model checker never finished constructing the state space before the timeout. The column *iter* denotes the number of iterations of the search procedure (Algorithm 1) and the column *time* denotes the total time taken by the search procedure in seconds. Cases with a valid program size *and* a timeout (e.g., *bsort100*●) indicate that the solver backend could not verify the given properties within 10 minutes.

Finally, some benchmarks are lacking a sliced, accelerated or abstracted version. This occurs when the respective source transformation technique did not result in any changes to

[3]For PapaBench, we simulated the PPM and GPS inputs to enable the software transitioning into automatic flight mode. However, not all subprograms were triggered, i.a., `approaching` and error handling stayed inactive. A worst-case simulation is practically impossible here.

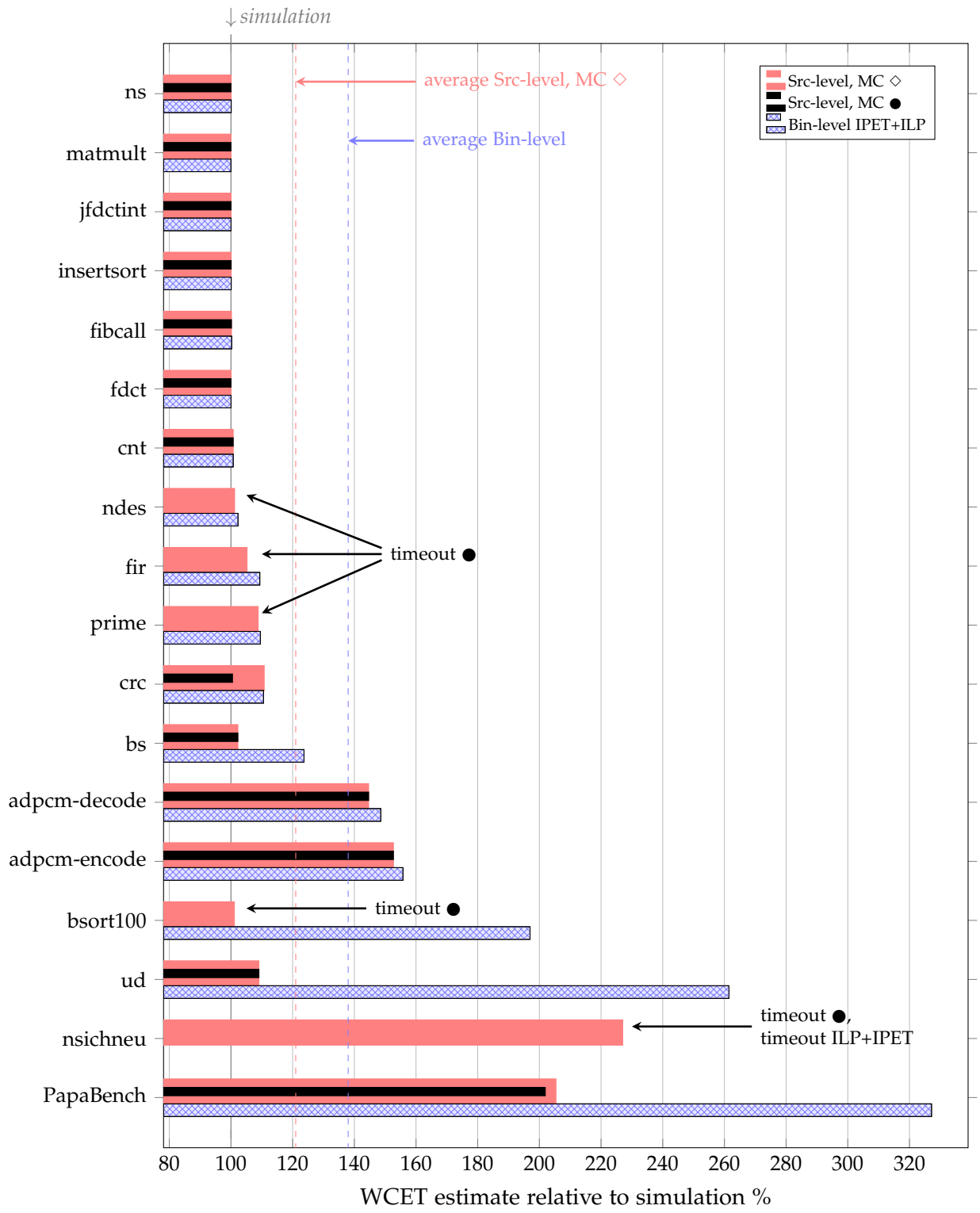


Figure 5.8: Tightest WCET estimates for different analysis methods.

Table 5.4: Computational effort for WCET estimation using different analysis methods.

	based on Instructions		based on Source Code					
	ILP/IPET				Abs. Interpretation		Model Checking	
benchmark	time [s]	mem [MB]	stage	prog.size	time [s]	mem [MB]	time [s]	mem [MB]
adpcm-decode	3.7	46.0	●	3537	1.4	55.7	8.0	356.0
			◐	2412	1.0	50.3	5.0	31.7
			◗	2133	1.0	50.1	5.0	32.1
			◇	1665	1.3	54.1	3.0	123.5
adpcm-encode	2.4	51.6	●	4911	1.6	60.5	291.0	688.0
			◐	4121	1.4	58.0	13.0	43.8
			◗	3978	1.4	57.4	14.0	45.1
			◇	1832	1.2	54.9	3.0	136.6
bs	0.7	4.3	●	372	0.5	46.9	1.0	117.3
bsort100	0.2	4.5	●	157376	0.4	46.1	601.0	11,572.4
			◇	5833	0.8	51.7	3.0	114.1
cnt	0.2	4.7	●	3239	0.6	49.5	10.0	240.9
			◐	2212	0.7	48.3	24.0	100.6
			◇	373	0.5	46.1	1.0	104.9
crc	0.3	5.3	●	41269	3.1	92.3	5.0	407.1
			◐	40519	2.9	92.3	5.0	367.8
			◗	39684	2.8	92.3	5.0	268.9
			◇	11013	0.9	55.7	2.0	103.2
fdct	0.9	12.8	●	989	0.7	48.3	1.0	101.9
			◐	182	0.5	45.7	1.0	96.8
			◗	98	0.5	45.7	1.0	97.3
fibcall	0.2	4.2	●	591	0.5	46.3	1.0	96.3
			◐	495	0.5	46.0	1.0	96.2
			◗	78	0.5	45.6	1.0	96.5
fir	0.9	40.8	●	–	0.6	48.4	timeout	6,718.2
			◐	–	0.6	47.0	timeout	16,322.3
			◗	50487	2.4	92.3	19.0	1,574.8
insertsort	0.2	4.7	●	1662	0.6	46.7	3.0	127.2
jfdctint	0.5	10.4	●	786	0.7	48.8	1.0	101.6
			◐	168	0.5	45.7	1.0	96.2
			◗	100	0.5	45.7	1.0	98.6
matmult	0.2	4.7	●	70768	0.5	46.1	23.0	773.4
			◐	62366	0.5	45.8	6.0	661.8
			◗	9637	0.9	57.5	2.0	101.4
ndes	2.2	54.5	●	75932	1.7	57.3	timeout	40,867.3
			◐	75797	1.6	57.0	timeout	40,877.7
			◇	5707	1.3	60.6	1.0	167.5
ns	0.2	4.7	●	22398	0.6	46.9	29.0	827.3
			◇	8271	0.6	47.1	7.0	200.3
nsichneu	timeout	2,734.6	●	23244	3.3	92.4	timeout	25,491.2
			◇	84	0.8	48.8	1.0	120.8
prime	1.0	40.8	●	–	0.5	45.9	timeout	22,052.9
			◐	–	0.5	45.9	timeout	23,532.8
			◇	157	0.5	45.6	1.0	99.7
ud	1.1	40.8	●	2381	1.0	49.6	2.0	101.7
			◐	1880	0.8	47.7	1.0	98.7
			◗	1375	0.7	47.6	1.0	99.5
PapaBench	3.6	79.7	●	10506	1.3	92.4	561.0	364.0
			◐	8963	1.3	65.4	523.0	352.0
			◇	11585	1.7	91.0	194.0	351.0

● instrumented, ◐ sliced, ◗ accelerated, ◇ abstracted, Δ upper bound for tightness

the source code. For example, *bsort100* remained unmodified post slicing and acceleration and thus does not have a dedicated sliced or accelerated version.

5.4.3 WCET Path Reconstruction

We were able to reconstruct and replay the WCET path for all benchmarks. The time taken for the debugger-based replay is in the range of a few seconds in all cases. For a better usability, we enabled our replay engine to output the trace as a cbmc-like counterexample, but additionally augmented with all assignments to `_time`. Recall that the complete information about variable `_time` implicitly carries the control flow, because each block in the source code increments this variable. As an effect, the graphical user interface can load this augmented counterexample, and map back the counterexample to the control flow graph, as well as compute a timing profile at a granularity of block- or function-level.

Identifying a Timing Hotspot. As an example, we show the resulting annotated control flow graph for the *ud* benchmark in Figure 5.9. There, we have applied a heatmap over timing, where red marks the timing hotspots. At first glance, it is apparent that there exists one source block forming a timing bottleneck, which consumes about one third of the overall execution time.

The *ud* program performs a simple form of LU decomposition of a matrix A, and subsequently solves an equation system $Ax = b$. The timing bottleneck in this program occurs in the LU decomposition, but interestingly not at the innermost or most frequently executed loop, but at a location where a division occurs. With this, the reconstruction of the WCET path made it easy to spot that for the chosen target, a long division incurs a high execution cost (see also the inline annotations `TIC`), and that this is the single driver for the WCET of this program.

Discovering a Bug. The replay of the WCET path can also reveal defects in the program. Such a defect has been found in the *prime* benchmark on a (simulated) 32-bit target, where initially the WCET estimate just seemed suspiciously high. After reconstructing the WCET path, an unexpected implausibility showed up. The path was indicating that the following loop had been executed 349,865 times:

```
1 for (uint i=3; i*i <= n; i+=2) {
2   if (divides (i, n)) return 0;
3   ...
4 }
```

where `n` is a formal parameter of the surrounding function that shall be checked for being a prime number. At first glance it seems like the loop can only execute 65,536 times: the maximum 32-bit unsigned integer `n` that the user can provide is 4,294,967,295, therefore the loop must execute at most $\lceil \sqrt{4,294,967,295} \rceil = 65,536$ times in order to evaluate whether that number is prime. Thus, we replayed the WCET path in our debugger, setting a watchpoint on `i`. It turned out that the expression `i*i` will overflow for some specific values of $\texttt{n} > 4,294,836,225$, since this would require at least $\texttt{i} \geq 65,536$, which, when squared, lies beyond the maximum representable unsigned integer and thus overflows. Specifically, this happens for only those numbers which are also not divisible by anything less than 65,536, hard to replicate if only the path and no values would be available. As an effect, the loop keeps running until finally the 32-bit modulo of `i*i` is larger than `n` before the loop terminates (which luckily is always the case). Clearly, this is a defect in the context of this

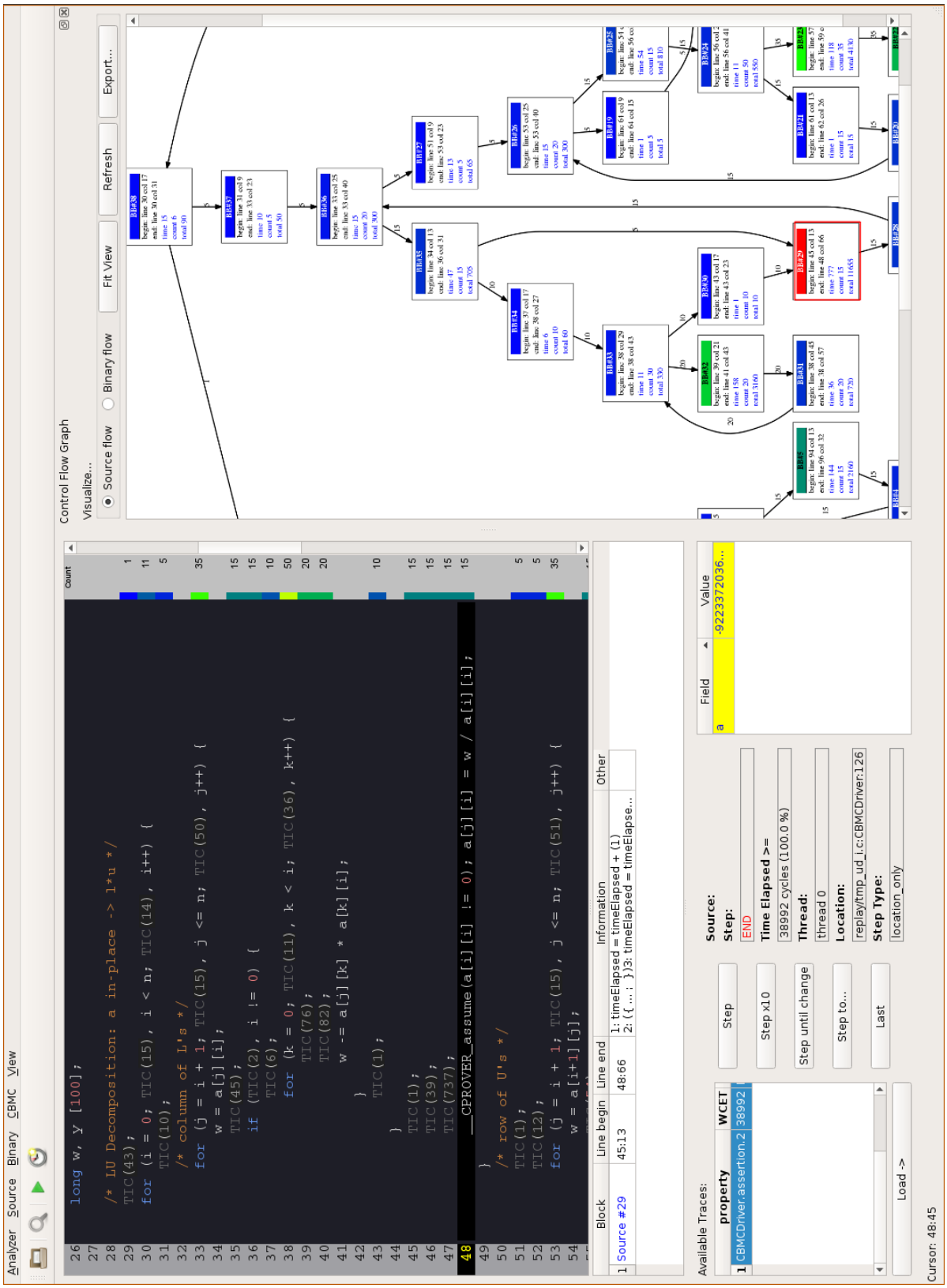


Figure 5.9: WCET path reconstruction for benchmark *ud* with identified timing hotspot.

program, resulting in a WCET which was orders of magnitude higher than it should have been[4]. Thanks to the ability of interactively replaying the WCET path and inspecting variable values, such defects can be easily spotted.

Top Ten Longest Paths. In principle our approach could also provide the "top 10" longest paths in the program. This could be done by repeating the overall WCET estimation, while excluding already known paths from the solution, and decreasing the WCET proposal to the Model Checker if no counterexample can be found. However, then this would become an explicit enumeration of paths, typically exponentially in the size of the program [LM97], and would still leave us with the question of how the program's overall probability distribution looks like (unless we repeat our estimation until every possible path length has been found). We did not investigate this further, as this clearly would not scale well with program complexity.

5.4.4 Abstract Interpretation

The result for an AI-based source-level timing analysis confirm our conclusions from earlier chapters. AI scales better, yet it is less precise than MC. AI fails to determine the WCET for most programs, returning the top element. The numbers are given in Tab. A.1 on page 202.

These problems arise due to missing relational domains and widening. This is especially the case when the number of loop iterations depends on data being manipulated within the loop. Even if the loop bound can be established, the *relation* to WCET stays unclear. On the other hand, if we make use of Frama-C's additional capabilities, like superposed states (to implicitly capture relations), then more benchmarks could be solved, but some (viz., *prime* and *nsichneu*) remained unsolvable within reasonable time. If the user would be willing to provide manual annotations for such loops (namely, loop invariants), then the WCET could also be bounded for them. The results for such a setup are shown in Tab A.3 in Appendix A. However, this is neither fully automated nor safe, as opposed to the Model Checking approach, and also yields less tight WCET estimates.

Interestingly, our proposed source code transformations can make a visible difference for AI. Both acceleration and abstraction remove loops, therefore removing with them the problems arising from data-dependent loops. This enables AI to yield a WCET even for formerly problematic programs, without the need for user annotations.

5.5 Discussion

Using source-level analysis based on MC, the WCET could be estimated for all benchmarks, usually within a few seconds, including a timing profile and path replay of the WCET case. Through that, not only did we provide an estimate with mostly no user inputs, but also we generate useful insights into the WCET path, enabling a true "time debugging". As we elaborate on the following pages, source transformation techniques had a noticeable impact on scalability of Model Checking, and thus paved the way to use Model Checking for WCET analysis and thus for source-level timing analysis. As a result, the estimates are comparable to those of an ILP-based technique, but with less effort and more feedback for the user.

[4]Note that this code works correctly for 16-bit targets, as $\forall n \in$ `uint16`, $\exists i \leq 255$, s.t. `divides(i,n)`.

5.5.1 Fairness of Comparison

One could argue that it is expected that our tool can outperform an ILP-based tool, due to the constraints that we impose on the target processor. An existing WCET tool should be more complex for the architectures that it must support, and therefore naturally slower and more approximating. However, the tool that we compare against here, Bound-T, is implicitly making similar assumptions as we do. It is also solving exactly the same subproblems (flow analysis, bounding of control flow, modeling of instruction timing, path analysis) as our tool set. Furthermore, we provide the exact same inputs to both tools, since we re-used Bound-T's parser front-end to obtain the time-annotated flow graph of the machine instructions. Therefore, we have ensured equal starting conditions for both tools.

In fact, one could argue the opposite: Since Model Checking is exploring paths explicitly, our tool effectively performs the *additional* task of identifying infeasible flows. From that perspective, we conclude that the experiments presented here are not biased towards our approach.

5.5.2 Tightness of Estimate

The tightness of the WCET estimate can be expected almost exact when Model Checking is allowed to explore all paths (i.e., no abstraction applied). When abstractions are applied, results become less tight and comparable to the result of an ILP-based solver which only gets loop bounds with no further flow analysis. In fact, Figure 5.8 shows that our source-level WCET estimates often are even tighter than the ILP-based approach. Overall, the upper bounds for overestimation (Δ) averaged over all benchmarks were as follows:

- ILP-based analyzer: +37.8%
- source-level analysis, MC: +20.3%
- source-level analysis, AI: +51.9%

In words, our source-level approach to WCET estimation based on MC resulted on average in 17.5% less overestimation than the traditional binary-level approach, and clearly outperforms a source-level approach based on AI.

For the *adpcm* benchmarks, there might be room for improvement. The computed WCET, even on the original version, is likely a large overestimation, considering the simulation value and the fact that this program has few paths only. The reasons for this are discussed in the following.

5.5.2.1 Overestimation from Non-Sources

Source level analysis suffers a fundamental limitation when it comes to complex control flows that have no source code equivalent, as happening for library calls. This is the reason why the estimates for the *adpcm* benchmarks are not tight. Although still better than the binary-level analyzer, the estimates are far off the simulation (+45% for *decode* respectively +53% for *encode*). These programs perform numeric operations that cannot directly be done in hardware, such as multiplication of numbers larger than the architectural word size, arithmetic shifts, and so on. Such functionality is provided by the target's C and math libraries, and implemented in assembly language for better performance. A pure source-level analysis thus cannot track the control flow and data dependencies for such functions, and must assume the worst case timing for such library calls. In fact, the same effect occurs in the binary-level analyzer, but for reasons of tractability. Other benchmarks subject to the same

limitation are *cnt*, *jfdctint*, *matmult* and *ud*, although those estimates are acceptable in their WCET tightness and on average close to or better than the binary-level analysis.

Table 5.5: Upper bound on WCET overestimation Δ due to functions without source code in *adpcm*.

	Sim.max.	WCET Binary-Level		WCET Source-Level	
function	time (calls)	time (calls)	Δ	time (calls)	Δ
__ashrdi3	1,887 (17)	27,013 (17)	**25,126**	27,013 (17)	**25,126**
__ashldi3	880 (10)	15,230 (10)	**14,350**	15,230 (10)	**14,350**
__muldi3[a]	18,420 (60)	19,467 (63)	1,047	15,141 (49)	-3,279
__adddi3	432 (36)	456 (38)	24	432 (36)	0
__adddi3_s8	15 (1)	30 (2)	15	0 (0)	-15
__negdi2	19 (1)	38 (2)	19	38 (2)	19
__cmpdi2_s8	255 (17)	270 (18)	15	90 (6)	-165
__muluhisi3[a]	492 (12)	492 (12)	0	574 (14)	82
__subdi3	72 (6)	72 (6)	0	36 (3)	-36
sum	22,472 (160)	63,068 (168)	40,596	58,554 (137)	36,082

[a] callees included

Table 5.5 quantifies the overestimation in *adpcm* caused by all implicit library calls. Here we show their *observed* WCETs during simulation runs (not necessarily occurring together in the same run) with their respective WCET estimates. It first should be noted that simulation and worst-case paths are close together, as indicated by the call counts. However, the number of processor cycles between simulation and WCET estimate substantially differs for the functions `__ashrdi3` and `__ashldi3`. Those functions implement arithmetic shifts using loops, which makes their execution time proportional to the shift distance. Further, we know from a manual inspection of the program, that these functions are never called with the largest possible shift distance. Consequently, large overestimation errors occur when we assume the worst case. Clearly, this explains why the simulated value is well below the estimates.

Other benchmarks subject to the same limitation are *cnt*, *jfdctint*, *matmult* and *ud*, although those estimates are acceptable in their WCET tightness and still mostly close or better than the binary-level analysis.

On the other hand, Tab. 5.5 also shows that source-level analysis is able to bound the number of some calls more precisely than the binary-level analyzer. This suggests that source-level analysis can identify more infeasible paths, and thereby does not need to assume that the global WCET is the sum of the WCETs of the individual functions.

In general, this context-agnostic approach to library calls can also fail if a callee is truly unbounded when call parameters are unknown (e.g., `memcpy`). Consequently, not all programs can be handled this way, making the approach incomplete. We discuss possible solutions later in Section 8.5, since they depend on the cache model that we introduce in Chapter 7..

5.5.2.2 Overestimation from Intrinsic Overapproximation

During the back-translation of timing information from machine instructions to source code, we used overapproximations used as described earlier. We argue that these over-approximations are common even when using the existing ILP-based techniques. During this back-translation, wherever there is a difference between the source blocks and basic blocks in the machine instructions, we over-approximate that part machine instructions into one block. However, these over-approximations are usually small, since these differences

are often formed by few and small basic blocks, amounting to only a few clock cycles per iteration.

Without any abstractions, our method exhaustively explores all feasible paths in the (instrumented) source code, and therefore does not make any overapproximations. Thus, when identifying the longest path, the result is never worse than an ILP-based path search. In fact, the ILP solver could only perform better, if the control flow could be bounded *exactly* and then encoded in the ILP formulation.

A typical case is that of *crc*. When no abstractions were applied, our estimate is around 10% tighter than that of Bound-T (130, 114 vs. 143, 137), which we tracked down to infeasible paths being considered by Bound-T: In *crc*, there exists a loop containing an if-else statement, whereas the if-part takes more time than the else-part. Bound-T, not analyzing the data flow and dependency between these two parts, always assumes the longer if-branch is taken. However, this is only feasible 70% of the time, whereas the other 30% the shorter else-branch takes place. Without additional user annotations or a refined flow analysis, ILP cannot discover this dependency and thus must overestimate the WCET.

Consequently, our source-level analysis performs better than an ILP-based approach when abstractions are left aside. For the remaining cases, where abstraction was necessary to make Model Checking scale, we now must discuss the overestimation caused thereby.

5.5.2.3 Overestimation due to Abstraction

Abstraction overapproximates the control and data flows, trading analysis time for tightness. When applying loop abstraction, the abstracted code forces the model checker to pick times along the branch with the longest times, cutting out shorter branches. Thus, we compute the WCET assuming that every branch inside the loop will always take the worst local choice, which may be infeasible. Naturally, this leads to an overestimation of WCET. However, ILP-based analyzers usually over-approximate with similar strategies when bounding the control flow. The result is a WCET estimate comparable to that of an ILP-based analyzer.

Again, consider *crc* as a typical case. When no abstractions were applied, our estimate was around 10% tighter than that of Bound-T. When applying abstractions, the estimate became very close to the estimate of Bound-T, whereas the complexity was cut down to approximately 25%.

Surprisingly, in some benchmarks, namely *adpcm-decode* and *encode*, *cnt*, *fdct*, *jfdctint* and *ud*, the loop abstraction did not lead to a higher WCET estimate. This is because in all the loops with branches in these programs (a) either there is an input to the program that forces the branch with the highest time to be taken in all the iterations of the loop, or (b) there is a break or return statement that gets taken in the last iteration of the loop. These cases match the exact pessimistic loop abstraction. In short, these loops do exhibit the pessimistic worst case timing behavior.

An extreme overestimation due to abstraction seems likely (reminder: the simulation is not guaranteed to contain the worst-case) for *nsichneu*, where the estimate is 127% higher than the observed WCET. This benchmark has only one loop with two iterations, but its simulated WCET is far away from the observed WCET. Upon inspection, we found that this loop has 256 if-statements in a single sequence, many of which do not get executed in every iteration. However, our loop abstraction pessimistically assumes that all the 256 if-statements do get executed in each iteration, which explains the overestimation in this case. Note that this benchmark could not be solved with Bound-T, at all.

Consequently, our method performs close to and often slightly better than an ILP-based approach when abstractions are used. However, this result depends on the way control flows

are bounded before the ILP solver is called, and thus it might not hold true when AI and ILP are combined.

5.5.3 Computational Effort

Our claim was, that the scalability issues of Model Checking could be mitigated with appropriate source preprocessing techniques, which we expected to be particularly effective at source code level. The experimental data clearly confirms that claim. In all cases, the analysis time – usually in the range of minutes up to unsolvable size for the original version – could be reduced to an acceptable time of a few seconds in average, which makes source-level timing analysis an interesting alternative to existing WCET estimation techniques.

However, taking the analysis time (computational effort) as measure of the computational complexity can be misleading. The time to solve the WCET problem in our approach consists of two fundamental parts: 1. building the SAT/SMT model and 2. solving the model. Whereas the time to build the model is often proportional to the size of the program (number of steps as found by cbmc), the time for solving the model cannot be predicted trivially by looking at program metrics. In particular, we have found no correlation between any of program size, cyclomatic complexity, number of statements, number of loops and the analysis time. The reason for this is that the analysis time of a SAT or SMT problem can vary significantly between two problems of the same size or between two solvers, due to the nature of modern solver algorithms [DPVZ15]. For instance, compare *adpcm-encode*● (program size 4,911 steps, analysis time around 5 minutes) with *ndes*◇ (program size 5,727 steps, solved in less than one second).

Therefore, we used a portfolio of solvers to reduce the effect of solver-specific strengths and weaknesses, and we consider the program size (last column in Table 5.4) as a prime indicator for the complexity of the program. With this, we show in the following that the complexity of a program can be significantly reduced with our techniques. Nevertheless, note that the solving time also points towards our interpretation. In all benchmarks we were able to greatly reduce the computational effort with each source processing stage. In particular, several benchmarks could not be solved in a reasonable amount of time without our proposed processing, viz., *adpcm-encode*, *bsort100*, *fir*, *ndes*, *nsichneu* and *prime*. The memory usage suggests, that a state-space explosion prevents building the model. Their program size could be reduced to a tractable size with acceleration and abstraction. Moreover, the benchmark *nsichneu* could not even be processed with Bound-T, since bounding of the control flow had failed because of too many variables and flows (out of memory after running for several hours). After applying our abstraction, the WCET of this benchmark could be successfully estimated with Model Checking, within one second of analysis time.

The overall impact of source transformations is quantified in Figure 5.10, where the program size after the respective processing stage is compared to the instrumented program and over all benchmarks. It can be seen, that on average each additional processing stage reduces the program complexity; in average we reach 84% of the original complexity after slicing, 75% after acceleration, and 27% after abstraction. Furthermore, it can be seen that in the worst case the source code transformations have no effect on the complexity. In the case of PapaBench◇ the complexity even seems increased, however, the analysis time was still reduced.

Note that the data used for Figure 5.10 exclude those benchmarks where we could not determine the program size due to state-space explosion. For each of those benchmarks, we have additionally spent 24 hours of processing to determine the program size, but without success.

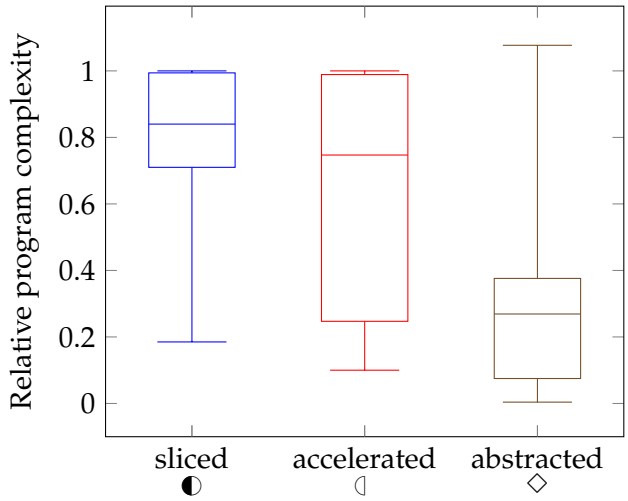


Figure 5.10: Box plot showing reduction of program complexity of after different source transformation stages, relative to original program. Whiskers are denoting the min/max values, diamonds the average value.

As explained in Section 5.1.4, a further reduction of the analysis time is possible if a lower bound for the WCET is already known. This is often the case for safety-critical systems, where run-time measurements (like high watermarks) are common practice. The user would initialize the search algorithm with an observed WCET, thereby reducing the analysis time. However, this does not change the structure of the model under analysis, and therefore is not considered a complexity reduction.

5.5.3.1 Further Potential to Reduce Analysis Time

There are a number of opportunities to further reduce the analysis time, which were not considered in this chapter. For the sake of completeness, we want to mention two enhancements that we have evaluated:

1. Parallelization of the solver backend may provide a linear speedup without loosing any precision. This is possible, for example, by using the parallel version of the solver backends (e.g., *glucose-parallel*). No changes to the workflow are required.

2. An enhanced compositional approach could be realized via assume-guarantee reasoning. Dividing the program into smaller units and composing the result would reduce the computational complexity at the cost of precision, since infeasible paths would be created. For example, one could analyze each subprogram individually, and accumulate the individual estimates along the call graph. This would discard call contexts, but produce a number of smaller programs to be analyzed. Such a trade-off is usually a user option, offered by many WCET analyzers. In essence this is assume-guarantee reasoning as introduced earlier, and works best if the source language supports contracts.

5.5.3.2 Alternative Analysis Methods

The results included AI-based estimates for reference. However, in principle we could also use an ILP-based WCET analysis at source level, similar to traditional WCET analysis. This was not evaluated, since it implies that we either discard all semantic information, or have to use

an additional method to extract and represent flow constraints. There exists some research in that direction, which could be combined with our approach [RMPV+19, MRS+16]. However, this would bring us back to the problem of generating sufficiently precise constraints, and in general cannot reach the precision achieved in our experiments.

5.5.4 Safety and Usability

Most programs can be analyzed automatically, without any user input. However, two programs, namely *bs* and *insertsort*, could not be bounded automatically. The loop bounds could not be deduced by cbmc, which shows up as an infinite loop unwinding phase. In such cases, the user has to specify an *unwinding depth*, at which the loop unwinding stops. The tool then places an assertion at this point, to check that the given depth is sufficient. In case of an insufficient bound, the model checker identifies this as a failed property. In case of a too generous bound, the unwinding may take longer, but WCET is not overestimated. Therefore, unsound or too imprecise flow constraints do not refute (or even worsen) the WCET estimate, which makes our approach safer than others.

Another contribution to the safety of our toolchain are the run-time checks introduced during WCET replay. As discussed earlier, a wide range of problems can be discovered by this, including confounding of input data and failure to specify the details of the target architecture.

Regarding the usability of our toolchain, we argue that the results of a source-level WCET analysis, especially the possibility to replay the worst-case path, should increase the level of acceptance for WCET analysis in a practical setting. As opposed to machine code or assembly, developers can be expected to be proficient in understanding a source code, for its higher level of abstraction and higher readability. By offering the possibility to replay the WCET path through an ordinarily looking debugging session – which can be controlled in the usual ways of stepping, inspecting and resuming – it becomes intuitive to walk through the path an identify the drivers of WCET. This is also supported by generating a profiler-like timing profile of the worst-case path, which is another, well-known view at a program, and a quick entry point for WCET inspection. A developer can identify those parts of the WCET path that need an in-depth inspection, and subsequently dive into debugging.

The technique we present here does not depend on the chosen tools. For example, the model checker cbmc could be replaced by any other model checker, possibly through an additional front-end that translates C code into a model to work with.

In summary, our approach is inherently safer than existing approaches for its protection against unsound user input, and because it enables software developers to understand and analyze the WCET path of a program without additional tools or training.

5.5.5 Correctness and Threats to Validity

The WCET estimate computed by our method is always an upper bound of the observable WCET. This is easy to see as in each step, we either over-approximate or preserve the timing information of machine instructions. In step 1, while the execution time computed for each basic block (Section 5.1.1) is precise, the back-annotation done in Section 5.1.2 over-approximates the time in the case where multiple basic blocks map to one source line. Therefore, at the end of this step, the WCET of the machine instructions is either preserved or over-approximated in the source.

The slicing removes only those statements that do not impact the values of the counter variable, thus preserving the WCET as per the instrumented program.

In the next stage, acceleration preserves the computation of the counter variable and abstraction potentially over-approximates it. So, at the end of this step, too, WCET is either preserved or over-approximated. Finally, the iterative search procedure of Algorithm 1 terminates only upon finding an upper bound of the WCET as per the accelerated or abstracted program. Thus, if our method scales for the input program, it provides a safe upper bound for the WCET.

Microarchitectural Features. We presented our results for a simple microcontroller. Modern processors, such as ARM SoCs, have architectural features like caches and pipelines, and memory-mapped I/O. Chapter 7 discusses how to model these features soundly at source code level.

Back-Annotation of Timing. The mapping of temporal information from machine instructions to source-level is a rather technical problem. A compiler could be build which enables complete traceability in at least one direction. Plans for such a compiler have been made in [PPH+13], unfortunately, we are not aware of any implementation. As mentioned earlier, although the gcc compiler recently improved the debug information (added discriminators), this is not sufficient to solve the task of matching instructions to source code.

Our mapping used for these experiments requires that the program is compiled using gcc 4.8.1 for the 8bit AVR microprocessor family [Kuh98], with standard compilation options. If we change the target-compiler pair or the compilation options, then our backmapping strategy may not work. Since it was established on a case-by-case basis, our strategy might not be complete, but it was extensive enough to cover all cases in the Mälardalen benchmarks. In general, compiler transformations are a common problem for all WCET techniques. And, to the best of our knowledge, there has been no generic solution to this problem, except to provide support for each architecture, compiler and transformation individually, often in the form of pattern matching [WEE+08].

In Chapter 6, we propose a generic and compiler-independent mapping strategy, which is able to handle some compiler optimization.

Compiler Builtins. The compiler may insert low-level procedures, such as loops for arithmetic shifts or make implicit library calls (e.g., for soft-floating point operations). These are not covered in our current tool, but taken from a pre-computed WCET library for the specific version of libc being used. Naturally, the correctness of the overall estimate depends on the library entries to be safe. In our experiments, we used the same values for both Bound-T and our tool.

5.5.6 Amenable Processors

In principle, the analysis presented so far is best suited for microcontrollers, or simple processors with a scalar in-order pipeline and no caches, and a fixed instruction timing. An extension for more advanced architectures is presented in Chapter 7.

Slight overapproximations, such as using only the maximum execution time for time-variable instruction, might be applied to use our approach even for processors that are not strictly compliant. The processor family that we target in this chapter, the 8-bit Atmel AVR family [Kuh98], is a good example for this. While there is a dependency of the instruction timing on the operands in some cases – viz., slight variations in instruction

timing for flash memory access – this is negligible and can be overapproximated. Our results shown later in this chapter justify this approach. Other processors that are a good fit are the SPARC V7 (specifically the ERC32 model [Sun87]), processors of the ARM7TDMI family (vanilla implementations without co-processors [Ltd08]), and the Analog Devices ADSP-21020. Academic examples include the Java-Optimized Processor [Sch03], the Precision Timed Architecture [LLK+08] with minor modifications (namely, port-based I/O to avoid time variances in load/store instructions, and absence of structural hazards), and the CoMPSoC multi-processor architecture [GAC+13].

5.6 Comparison to Related Work

Source-Level WCET Analysis. Excluding the early and unsafe work in WCET analysis that we reviewed in Section 2.1.1, there are only a handful of similar approaches.

In [KPE09], the authors experimented with *cbmc* to analyze a time-annotated source code. The assumed target processor implements the PRET architecture [LLK+08], which has similar properties as our AVR device. The scalability problem of MC is however not addressed in that report, nor are any performance figures given. Furthermore, their search strategy is solely based on counterexamples and requires around ten times as many iterations as ours, although this might not be representative, since only one program was analyzed. Similar experiments have been conducted by Kuo et al. [KYAR10] on source code generated for IEC-61499 function blocks. However, their estimates were in average less precise than ours (reasons have not been given), and the state space explosion problem was again not addressed. This confirms that our source code transformations (slicing, acceleration, abstraction) are essential.

Another source-level approach to WCET analysis has been proposed by Chapman [CBW96], for Ada programs. He integrated timing and program proof by building regular expressions on paths in the program, converting them to verification conditions, and analyzes them with a symbolic execution system. The result is a WCET estimate and a path description, however, with reportedly 22% overestimation in the best case. Furthermore, it was required to annotate the program with loop information and other specifications for the prover, whereas our analysis requires no such annotations. His work has been a proof-of-concept only, and no analysis times have been shown.

In summary, our approach is more scalable and precise than any earlier attempts to source-level WCET analysis thanks to our source code transformations. We do not require user annotations, and leverage the counterexamples for WCET path reconstruction. In addition, we have evaluated our method on a large set of programs and provided detailed results.

Model Checking for WCET Analysis. The academic tool METAMOC [DOT+10], uses Model Checking at instruction level to estimate the WCET for ARM7, ARM9 and AVR processors, for the same benchmarks. Modeling of caches and pipelines is included, but loop bounds have to be specified by the user. The framework is based on the UPPAAL model checker. The WCET search is eventually agnostic to data flows, and therefore, call contexts are not considered, resulting in higher overestimation. This is further worsened by their abstractions, which are mandatory in contrast to our tool.

A similar work has been published by Cassez [BC11], but with a different modeling and search strategy. They also use slicing and confirm that it is crucial for scalability. Unlike the computationally complex METAMOC framework, the analysis time in [BC11] is comparable to our tool set, however, does not scale as well for complex programs. This can be attributed to our additional processing steps to reduce program complexity.

Program Slicing, Acceleration and Abstraction. Hatcliff [HDZ00] was the first to suggest the use of program slicing to help scale up Model Checking, and we applied this technique to constrain the analysis to timing-relevant program constructs. In this work, we have also built on the acceleration and abstraction capabilities of LABMC [DCV+15]. Different abstractions for improving the precision of WCET computation or determining loop bounds have been explored by other researchers. Ermedahl et al. [ESE05] show precision improvements in WCET computations by clustering basic blocks in a program. Knoop et al. [KKZ12] use recurrence relations to determine loop bounds in programs. Blazy et al. [BMP14] use program slicing and loop bound calculation techniques to formally verify computed loop bounds. Černý et al. [CHK+15] apply a segment abstraction technique to improve the precision of WCET computations. While in these abstractions could be used in some situations, in general they are either too restrictive because they do not work for a large class of programs, or they fail to address the scalability issue arising in the context of using a model checker to compute the WCET. Al-Bataineh et al. [ABRF15] use a precise acceleration of timed-automata models (with cyclic behavior) for WCET computation, to scale up their IPET technique. However, these ideas are not readily applicable to loop acceleration in C programs in the absence of suitable abstractions. Cassez [Cas11], although working at binary level, also made used of program slicing to accelerated the WCET estimation process with the model checker UPPAAL. These works show that slicing, acceleration and abstraction are beneficial for source code analysis and MC, yet they have never been used together in a source-level WCET analysis, which results in a particularly scalability boost for MC.

WCET Path Debugging. Similar work related to timing debugging has been summarized in Section 2.2. None of them, however, can reconstruct a precise execution trace for the WCET path. Reconstructing the inputs leading to WCET has been done before by Ermedahl [EFGA09], and is perhaps the closest work in respect to the degree of detail that is provided on the WCET path. Our approach, however, uses entirely different methods. While Ermedahl applies a mixture of static analysis and measurements to perform a systematic search over the value space with WCET estimation, and eventually only provides the inputs to the WCET path, our approach is leveraging the output of the model checker that witnesses the WCET estimate, performs only a single run of the application, and reconstructs the WCET inputs, as well as the precise path being taken together with a timing profile. It is thus less expensive and better integrated with the actual WCET estimation. Furthermore, by allowing the developer to interactively explore the trace in a well-known debugger environment, ours is also a more intuitive workflow that integrates timing debugging with functional debugging.

Possible Enhancements. The approach of Henry et al. [HAMM14] could be combined with our source-level timing analysis. They also employ an SMT solver to compute the WCET (just like our back-end), but generate additional constraints to have the solver ignore infeasible paths. This helps to further increase the scalability, but under the assumption that the programs are loop-free, or that loops have been unrolled. This therefore fits well with our loop acceleration and abstractions. Brandner [BHJ12] computes the time criticality on all parts of a program, to help focus on optimizing paths that are close to WCET, and not only those on it. His approach requires an external WCET analyzer that annotates basic blocks with their WCETs, and therefore can be naturally combined with our work.

5.7 Chapter Summary

We have presented the basic workflow for a source-level WCET analysis on a simple microcontroller, with MC as the primary analysis method. The experiments show this approach to be competitive to binary-level analysis, with in average 17% less overestimation. In addition to computing a more precise estimate, we also reconstruct and replay the WCET path in an ordinary debugger, while providing profiling data. This allows the user to inspect arbitrary details on the WCET path at both source code and machine code level. Although our approach does not entirely remove the need for manual inputs from the user, WCET analysis is no longer prone to human error coming from there, because the model checker also verifies whether such inputs are sound. In particular, users only have to provide loop bounds for few, hard-to-analyze loops. If too small bounds are given, an error is flagged. Too large bounds, on the other hand, only influence the analysis time, but not the outcome. In summary, we therefore arrive at a safer WCET analysis and a more intuitive understanding of the estimate.

An essential part of our approach is the shift of the analysis from machine instructions to the source code. Through this, data and control flows can be tracked more precisely, and source code transformation techniques can be applied to summarize loops, to remove statements not related to timing, and to over-approximate larger programs. As a result, the analysis time can be reduced significantly, making MC a viable approach to the Worst-Case Execution Time problem, despite its still exponential computational complexity. As an alternative, we have also evaluated source-level AI to estimate the WCET, which sometimes yields similar results, but in general cannot bound the WCET for its lack of precision.

What we have shown in this chapter is merely the concept and potential benefits of a source-level timing analysis. However, there is more work to be done to catch up with the traditional binary-level approaches. One question answered in the next chapter, is that of how the mapping can be established in a generic way. Another question answered in the subsequent chapters, is how to model more complex processors. While certainly processors for real-time applications should be simplified to address the self-made and acknowledged problem of unpredictable processors, some essential features, such as caches, need to be covered. This will result in support for a much wider range of processors. Pursuing this route should be worth the efforts that are on the way, by reason of the practical benefits demonstrated in this chapter, namely higher precision, higher automation and better usability.

Chapter 6

Generic Mapping from Instructions to Source Code

***

This chapter proposes an automatic instruction-to-source mapping strategy which yields the time-annotated source code that we need for a source-level timing analysis. Although such a mapping is a rather technical problem that can always be established by reversing transformation patterns for specific pairs of compiler and target, a *generic* mapping approach is required for source-level analysis to be of use in a practical setting. This implies not only a unified handling for different targets, but also for different compiler versions and flag settings. Such a generic and automatic approach prevents human error that may otherwise occur when a mapping strategy is developed for specific compilation patterns.

The approach presented here also allows enabling compiler optimization, unlike most other approaches from the WCET literature. This is another important capability in practice, since optimization can often significantly reduce execution times, and thus lower requirements to

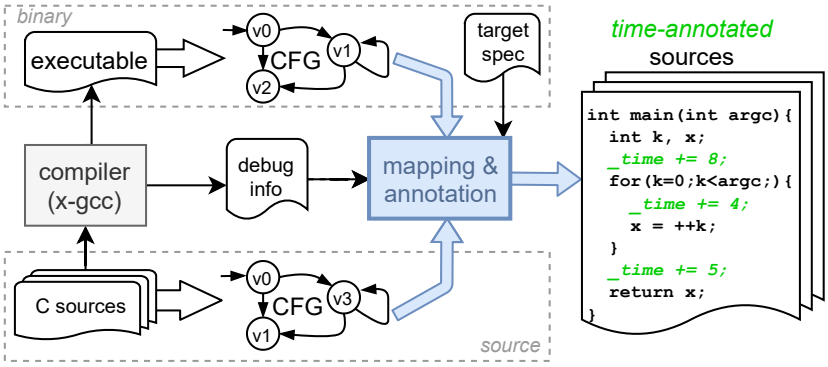


Figure 6.1: A common problem in Virtual Prototyping and WCET Analysis: matching flow graphs and back-annotation of timing from machine instructions to the source code.

the target platform for a given computational task. Nevertheless, we only consider moderate optimization, since static timing analysis is primarily a concern for time-critical systems or those that must undergo certification, for which higher optimization levels are often not permitted [FFL+11]. In such systems it is meanwhile more important to reduce potential impact of human error than to obtain highly efficient implementations.

6.1 The Mapping Problem

Relating machine instructions to source code is an often-needed capability in software engineering, that has been repeatedly addressed in different problem contexts, most famously to enable source-level debugging of software [C+10], so that programmers can follow a program's behavior at the easier-to-understand source code level, instead of instruction level.

WCET analysis was initially also concerned with this problem, as described in Chapter 1. There have been publications which implied to have solved the mapping problem, e.g. [KPE09, JHRC12, CR13, RMPC13], with some mentioning compiler modifications [Pus98, PPH+13]. Yet, a complete solution was never presented, and also compiler optimization was rarely considered. Meanwhile, WCET analysis predominantly takes place at binary level today, such that the mapping problem is merely in the way of generating flow facts from the source, or transferring user annotations (mostly loop bounds) from source to the binary [LES+13]. A complete mapping was therefore not required so far.

Meanwhile, another research domain has addressed the mapping problem for its own needs – *Virtual Prototyping* [MGG17]. The central goal there is to simulate a program's timing behavior when executed on a specific target, as quickly and accurately as possible, and without the need to set up the target hardware or running painfully slow cycle-accurate instruction set simulators [BEG+15, CZG13]. Instead, the time-annotated source code reflects the target's timing, but is executed on a much faster simulation host. Towards this, automated methods for source-level timing annotations have been proposed in recent years [MLS11, SBR11a, BEG+15]. These can be carried over to WCET analysis, and vice versa, VP can learn from methods that meanwhile have evolved in the WCET domain. The caveat, however, is that in VP the mapping is allowed to contain slight errors, both towards under- and over-estimation, whereas for WCET purposes only the latter one is allowed.

This chapter contributes the following: 1) We compare the requirements and goals of timing annotations in VP and WCET, 2) we evaluate the applicability of VP methods in WCET applications, and propose ways to fix them and increase their precision, 3) we propose a

generic instruction-to-source mapping algorithm for WCET analysis of simple processors, competitive to classic approaches, and 4) we discuss further synergy in both research communities.

6.1.1 Problem Setting, Challenges and Goals

The ultimate goal is to annotate the source code of a program with statements keeping track of its execution time, as experienced on a specific target. Fig. 6.1 illustrates the overall workflow: The program is first cross-compiled for the target processor, which results in a binary/executable with the machine code. We analyze this binary to obtain 1) the control flow of the machine instructions, and 2) the timing behavior of the elements in the control flow. Analogously, we obtain the control flow from the source code.

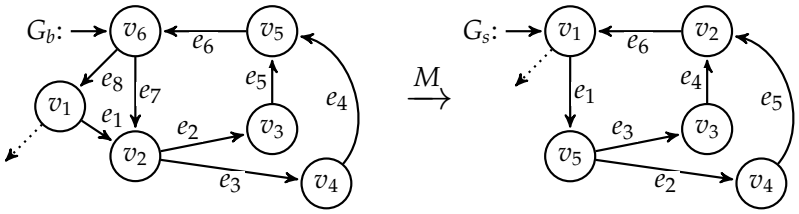


Figure 6.2: Example of the mapping problem.

The core of the timing annotation problem then is the following: Given two control flow graphs, G_b for the binary and G_s for the source code, we want to establish a mapping

$$M : V_b \to V_s \tag{6.1}$$

between the nodes V_b and V_s of the respective control flow graphs, and then annotate the nodes V_s with the timing according to the mapped V_b, to make the timing behavior visible in the source code. In other words, we want to associate blocks of machine instructions to blocks of source statements. Even without any compiler optimization, G_s and G_b can exhibit differences due to calls to library functions for which no source code exists, but also stemming from architectural constraints. For example, consider the two flow graphs in Fig. 6.2. The additional node v_1 in G_b must be produced on a 16-bit processor whenever a 32-bit comparison is required. Consequently, the graphs cannot even be considered isomorph in the absence of compiler optimization, and must be expected to differ significantly when optimization is enabled. It follows that the searched-for mapping is in general not bijective, nor even a mathematical function. Additionally, some nodes might be indistinguishable (e.g., v_3 and v_4 in Fig. 6.2) when only the graph structure is considered. Therefore, debugging information is often consulted to obtain more insights.

Since the two flow graphs are rarely isomorph, the mapping has to consolidate flow differences between them. In general, such differences could be addressed by decompiling the machine instructions back to source code [BLSW13]. However, we do not consider this option here, because it not always safe, and undercuts the goals of source-level timing annotations, as discussed next.

The goals for such a mapping are as follows:

- Timing annotations shall be as precise as possible.
- The structure of the source code shall be preserved to retain readability.
- Abstract as much detail of the target as possible, to keep complexity low and speed up the analysis.

- The compiler shall not be modified, and different compilers should be supported.
- Compiler optimization shall not be reverse-engineered.
- Compiler optimization should be tolerated.

In addition to these goals, which are shared with the VP domain, WCET analysis also comes with the following goals:

- Underestimation of the execution time is not allowed, but overestimation is acceptable.
- It is acceptable to forbid certain code constructs to enable an automatic analysis [FFL+11].
- There is no need to support maximum optimization levels, as systems subject to WCET analysis are often verified by additional means, where optimization can be considered a threat to soundness, and is sometimes forbidden by certification rules [FFL+11].

These WCET-specific goals prevent us from applying VP methods directly. For example, in VP unmapped nodes are often resolved by heuristics [SBR11a, MLS11], but this could result in an unsafe WCET estimate. Furthermore, substituting such blocks by their local WCET is not always possible either, since the WCET of unmapped flow parts may become unbounded without execution context.

6.2 Background

6.2.1 Dominators

In directed graphs G, a node u *dominates* another node v – in short, $u \operatorname{dom} v$ – if all paths from I that reach v must also go through u. In other words, the execution of v implies the (prior) execution of u, but not vice versa. Further, a node u *strictly* dominates another node v if $u \operatorname{dom} v$ and $u \neq v$. Analogously, y *postdominates* x – in short, $y \operatorname{pdom} x$ – if all paths from x towards F must also go through y. Pre- and postdominator trees are data structures that can be computed with standard methods from graph theory, and capture such domination relationships for all nodes in G at once [Hav97].

6.2.2 Debugging Information

Debugging information, such as the DWARF [C+10] format, has been introduced as means to relate instructions to source statements during source-level debugging. In principle, the compiler performs a translation from source code to machine instructions while maintaining a logbook, showing which source locations are causing each instruction, and eventually includes this information in the binary.

Unfortunately, maintaining precise and complete debug information is nontrivial, especially under optimization. Not all optimizers maintain full traceability between source and machine code, which may result in incomplete (some instructions have no source equivalent and vice versa), ambiguous (multiple relations) and imprecise (slight mismatches) information [SAA+15, LMS12, WH12].

Debug info vs. Basic blocks: Although debug information relates instructions to source locations, it does not explicitly specify equivalent source ranges for the binary BBs. Instead, potentially each instruction may have its own location info, or may share it with instructions that immediately precede or follow itself in the address space. Therefore, the source range of basic blocks must be reconstructed from the potentially multiple locations in the debug info that belong to its instructions. Let $locs(v)$ be the list of debug locations associated with a BB v, sorted by execution order (which is not necessarily instruction address). Then we can obtain

```
1 int main () {
2   [[int x,y,n,m[2];
3   if ((x >= 3]]) &&
4       ([[y + 3 <= 6]]) &&
5       ([[m[1] == m[2]]])) {
6             [[n = 2]];
```

(a) frontend/source

```
1 int main () [[{
2   int x,y,n,m[2];
3   if ((x >= 3)]][[&&
4       (y + 3 <= 6)]][[&&
5       (m[1] == m[2])) {
6             n]][[= 2;
```

(b) debug info

Figure 6.3: Discrepancies of basic block ranges (min='[[', max=']]') between compiler frontend and debug info.

at least two types of ranges: 1. begin $l_{\text{beg}}(v)$ to end $l_{\text{end}}(v)$: denotes the location info of the instructions that are the first and last in $locs(v)$, and 2. min $l_{\text{min}}(v)$ to max $l_{\text{max}}(v)$: captures the extremes of the of source locations in $locs(v)$. These may or may not be identical. For example, the compiler may choose to schedule instructions first that are not at the beginning of the source block, and thus *begin* can be greater than *min*. Furthermore, the last instruction of the block may share location info with prior instructions, and thereby not precisely capture the precise end of the basic block (and in fact, might point to the beginning of a source token, and thus not be column-precise either). Consider the example in Fig. 6.3 lifted from the *nsichneu* benchmark. There are malign differences in lines 3 and 4, where the source has only one BB in each line, but the binary has two. Another reason for more binary BBs on the same line as in the source may come from inlined compiler intrinsics or implicit library calls. We address all of these issues in our mapping algorithm.

Discriminators have recently been introduced into the debug format, to distinguish between multiple binary basic blocks that fall into the same source code line. For every control transfer that is detected during compilation, the according target instructions are labeled with a new discriminator value if they fall into the same source line as their predecessors. Unfortunately, the DWARF specification allows arbitrary enumeration of the binary basic blocks [C+10], which means that neither enumeration order nor maximum value have to agree with source locations and block counts (and indeed do differ in practice). As a consequence, it is not safe to pair up the nth source BB on a given line with the binary BB that is labeled with discriminator n. In principle the column information could be used to match binary discriminators to source blocks, but column data is imprecise, as we have seen just before. In fact, we have witnessed cases where column numbers even contradict the semantics of the assembly. Further, having more source BBs than binary discriminators carries no useful information either, since source BBs could have been optimized out.

Last but not least, source code layout has an influence on the debug info, but only to some extent. For example, writing a loop header such that its three parts occupy three different source lines, still results in only one line number for all parts. Consequently, the correctness of debug information may also depend on the source layout, and multiple BBs on one line cannot be avoided in general.

6.3 Review of Existing Work

In this section, we review the mapping algorithms from the VP community, to evaluate their applicability for source-level WCET analysis. Towards this, we define the following two

properties: (1) lower-bounding of execution count $f(v)$ under all traces, i.e.,

$$\forall v \in V_b.\ f(v) \leq f(M(v)) \tag{6.2}$$

and (2) preservation of execution order in the annotations, i.e.,

$$\forall u, v \in V_b.\ v \succ u \Leftrightarrow M(v) \succ M(u) \wedge v \prec u \Leftrightarrow M(v) \prec M(u). \tag{6.3}$$

Equation (6.2) ensures that there is no underestimation caused by the mapping, and Eq. (6.3) is relevant for processors with caches, since there the access order influences temporal behavior.

All proposed mapping algorithms, directly or indirectly, are based on the source locations contained in the debug info. Binary BBs which have similar locations as source BBs are considered to be related to each other. Initial approaches only relied on this property (such as [RFMdS11] in the WCET domain and [MSSL08] in the VP domain, but this information is insufficient to establish a mapping, since debug locations can be incomplete and ambiguous. Since then, the consideration of further BB properties has been proposed to solve address this problem.

6.3.1 Basic Block Properties

In [MLS11, SBR11b, BEG+15], BBs are eventually matched pair-wise to maximize similarities between their source location, loop membership, structural properties (control dependency and dominators) and the last branching decision. We ignore the last branching decision here, since it only facilitates a "more synchronous" annotation, but has no impact for WCET analysis. Loop membership can be implicitly captured as a side effect of a hierarchical decomposition, which we discuss in the following. Finally, structural properties have proven effective in VP, and are reviewed in detail after the decomposition.

6.3.2 Hierarchical Flow Partitioning

Several groups have proposed to partition the control flows hierarchically into smaller subgraphs [WLH11, LMS12]. This limits the impact of annotation errors and ambiguities in the debug information, and reduces the problem size for the node matching algorithm. Specifically, loops and branches were used as partition boundaries. Unmatched regions are eventually lumped into single nodes, and substituted by their local WCET. As a side effect, such a hierarchical matching implicitly captures loop memberships, which has been shown to improve the final node matching [MLS11].

Safety: It is unclear how in general it can be ensured that binary and source partitioning do not diverge during a hierarchical decomposition. In principle it could happen that a difference in the control flow leads two different splits in source and binary, which prevents the optimal mapping from taking place. However, under one of the following assumptions, such an approach can be justified: 1) the boundaries which are used for decomposition are guaranteed to be preserved, or 2) external information about structural changes, e.g., extended debug information, can be taken into account.

In the following, we look at two in VP commonly used properties beyond debug info that are used for matching the individual nodes.

6.3.3 Dominator Homomorphism Mapping

A *dominator homomorphism* [SBR11a] is a partial mapping M between two digraphs that preserves dominator relationships between pairs of nodes. The idea is that such a mapping preserves execution order. Specifically, if the mapping is defined for two binary nodes u and v, then it must hold that

$$\forall v_1, v_2 \in V_b.\ v_1 \operatorname{dom}_b v_2 \leftrightarrow M(v_1) \operatorname{dom}_s M(v_2). \tag{6.4}$$

Several details should be noted about the dominator homomorphism, which may refute Equations (6.2) and (6.3) if not considered.

First, the mapping is not unique. Thus, the algorithm constructing the homomorphic map has a large influence on the annotation precision. The algorithm in [SBR11a] is meant to preferably match dominated binary nodes with dominating source nodes, but we found that it cannot guarantee that. Since multiple map entries are added simultaneously during the iterative map construction, false conflicts may be detected: The homomorphism is checked for all pairs in the current map, including the (not yet verified) entries that have just been added. Checking is done one by one in an unspecified order. Whichever inconsistency w.r.t. Eq. (6.4) is found first, is noted as a conflict. If one of the newly added entries is a bad pick and breaks the dominance homomorphism for another newly added, but correct entry, then we reject both the correct and incorrect entries, although one of would have been the optimal choice. Together with the fact that picking *dominated* nodes first represents a topological order (which is not uniquely defined), conflicts are won by whichever dominated node is picked first.

To fix this algorithm, we must extend the map by one entry at a time. This impacts the run-time negatively, but it yields a better mapping and makes the result independent of the (also unspecified) node iteration order. With these changes, the algorithm indeed preferably ends up in pairing nodes in the chosen preference.

Second, the mapping differs depending on whether strict domination is used or not. Using *strict* domination, two nodes with $u \operatorname{dom} v$ cannot be mapped to the same source node s, since $M(u) \operatorname{dom} M(v) \neq s \operatorname{dom} s$, because s does not strictly dominate itself. Two unrelated binary nodes can however still be mapped to the same source node. This means execution order *is* preserved, as far as it is captured by the dominator relationship. On the other hand, when using a *non-strict* dominator relationship, preservation of execution order depends on whether dom is seen as a property or as an operator.

Mathematically, equation (6.4) requires us to also evaluate the reverse relationship, i.e., if $v_d \operatorname{dom}_b v_f$, then both of the following conditions must hold

$$v_d \operatorname{dom}_b v_f = true \leftrightarrow M(v_d) \operatorname{dom}_s M(v_f) = true \tag{6.5}$$

$$v_f \operatorname{dom}_b v_d = false \leftrightarrow M(v_f) \operatorname{dom}_s M(v_d) = false. \tag{6.6}$$

The algorithm in [SBR11a] only tests for the first condition. As a consequence, the *execution order between v_d and v_f can be lost*. This can lead to body nodes in loops to be mapped to loop headers, which *produces overestimation*. On the other hand, if Eq. (6.6) *is* checked, then such binary nodes are competing for the source node in question. That is, only one of multiple nodes can then be mapped to a single source node, leaving the others unmapped. This again can lead to overestimation in cases where $v_1 \operatorname{dom} v_2$, and v_1 being the only predecessor of v_2, and v_2 the only successor of v_1. Here v_2 steals the source node, leaving v_1 unmapped. This

happens in the binary CFG due to *jump threading*. To mitigate this issue, such nodes should be fused in both CFGs to prevent overestimation.

Execution order: Dominator relationships cannot capture all execution orders, thus this method cannot guarantee the preservation of order in the mapping. For example, consider the code fragment `if(u){v;} z;`, where we have $u \operatorname{dom} v \wedge u \operatorname{dom} z$, and but not $v \operatorname{dom} z$. If v executes, then it must still happen before z in the associated flow, yet, the dominator tree does not carry this information. In practice, this becomes a problem only when debug info between such nodes is ambiguous, which is the case for multiple source blocks on the same line (see Section 6.2.2).

Execution count: In general, every time multiple BBs are not distinguishable by their debug info nor their dominance relationships, BBs can be mapped arbitrarily, which may or may not underestimate the execution counts. Consider the one-liner `if (a){b} else {c} z` . Further, let a and z have the same dominator relationship to the surrounding flow (in both source and binary), such that all of b, c, z are siblings in the dominator tree. Thus, one arbitrarily selected node among b, c, z will map to the source of a, and the others have to be overapproximated later on. This does not violate our execution count property. Now assume z is absent. The mapping of b and c is now arbitrary, possibly violating Eq. (6.2).

Apart from that, counts are only weakly constrained. If $u \operatorname{dom} v$, then the only guarantee is that $f(v) > 0 \Rightarrow f(u) > 0$, and otherwise their count is unrelated (u can be a loop header which v is not a member of, thus $f(u) > f(v)$ is possible, and vice versa, v may be member of a loop that u is not, such that $f(v) > f(u)$ is possible). Consequently, a dominator homomorphism does not maintain much of the execution count relations.

Under optimization: As the experiments in [SBR11b] have shown, the dominance relationship can be changed when the compiler splits complex conditional statements in the source into multiple binary branches. Confusion of BBs cannot be provably avoided.

6.3.4 Control Flow Dependency Mapping

Müller-Gritschneder et. al [MLS11] proposed an alternative property to match BBs. The idea is to match up those BBs in source and binary that execute under the same condition. Towards this, the first step is to identify *control edges* and *controlled nodes* in both G_b and G_s. An edge $e = (u, v)$ is a control edge if not $v \operatorname{pdom} u$, i.e., if this edge enables the execution of v. Vice versa, the set of nodes $C(e)$ immediately controlled by an edge e is computed as

$$C(e) = \{w \mid w \in path(v, \mathrm{lca}(u, v))\} \setminus A, \tag{6.7}$$

$$A = \begin{cases} \emptyset & \text{if } u = \mathrm{lca}(u, v) \\ \{u\} & \text{otherwise} \end{cases}, \tag{6.8}$$

where $\mathrm{lca}(u, v)$ is the least common ancestor of u and v in the postdominator tree, and $path(u, v)$ is the sequence of nodes on the path between (and including) u and v in said tree.

In a second step, they assign labels to all control edges in source and binary, such that edges representing the same decision and outcome obtain the same label. Let $c_X(q)$ be a label representing "decision X, outcome q", and $\mathcal{MC}_{c_X(q)}$ be the set of edges that fall into this label. Given any node v, the new *control dependency property* is formally defined as

$$p_{\mathrm{ctrl}}(v) = \left\{ c_X(q) \;\middle|\; v \in C(e) \wedge e \in \mathcal{MC}_{c_X(q)} \right\}. \tag{6.9}$$

In other words, the property $p_{\text{ctrl}}(v)$ holds all labels representing the necessary conditions for the immediate execution of the basic block v, whereas multiple labels describe a logical disjunction. Thus, BBs in binary and source which have been assigned the same labels are executing under the same conditions. Note, however, that the mapping is not fully defined, since we may have multiple source nodes matching each binary node.

The key in this approach lies in how the labels X and q are assigned to the edges. Since we want to assign common labels for edges representing the same decision and outcome in binary and source, some form of edge matching is required. The authors propose to use the debug locations of the blocks that are at the outgoing and incoming end of branching edges. That is, both X and q are essentially defined by their source line numbers. Note that this allows multiple binary edges to obtain the same label, as long as they all jump to/from the same lines.

Execution count: Under the assumption that the edge labels represent the decisions and outcomes correctly, each binary node maps to a source node that executes under the same immediate condition. If each of the decision nodes itself maps to their correct conditions, we fulfill Eq. (6.2) by transitive property. However, using source lines to label the decisions and outcomes as in [MLS11], is *unsafe*. Edges can be confused when multiple BBs share the same line info. Furthermore, there are multiple ways to select the line number, since there is more than one way to define source ranges (see Sec. 6.2.2). By definition of what constitutes a basic block, we could be tempted to use the location of the entry/exit of the binary BB, which however does not necessarily coincide with the begin/end of the source BB. Apart from that, execution count is well-defined in relation to the controlling edges and co-controlled nodes. Except for loop header nodes, for a control-dependent node v, we know that $f(v) = \sum_{v \in C(e)} f(e)$. In other words, control-dependent nodes execute as often as their controlling edges, except loop headers, which execute more often. In summary, there is no execution count guarantee with the proposed edge labeling.

Execution order is only ensured towards the node that precedes the immediate control edge, and that node in turn is either mapped, thereby obeying its own execution order, or unmapped, and must be overapproximated later. Execution order towards other nodes controlled by the same edge and enclosing conditionals is undefined.

Under optimization: This method can tolerate more compiler optimization than the homomorphism, as shown in [MLS11]. Since the mapping is semantically tied to execution conditions, this method should not produce false mappings under optimization, if labels can be assigned correctly. However, it may still fail to map some blocks. Further, it is in principle possible to detect invariant conditions that have been optimized out.

6.3.5 Handling Optimization

Since the mapping methods may either make errors or produce fewer map entries under optimization, handling for several specific optimizations have been proposed in the VP community. For example, in [LMS12, WH09a], structural loop changes are addressed by mimicking the transformation rules of a certain compiler. This way they handle loop splitting, do-while transformations, unswitching, and blocking. In [SBR11b], full loop unrolling is handled in a similar way.

However, a myriad of effects on the CFG is possible, since the optimizers can interact with each other in unexpected ways. It is therefore tedious and likely unsafe to reconstruct them on a case-by-case basis. While some unsupported optimizations could be forbidden to alleviate this problem, it is often not possible to prevent all of them, even when optimization is turned off. Therefore, a generic mapping method must still be able to handle some effects.

In general, all optimization can be seen as either 1. change of execution order (e.g., instruction, trace and superblock scheduling; note that this is different from the *annotation* order required in Eq. (6.3)), 2. change of execution conditions (e.g., hoisting), 3. duplication (e.g., tail duplication) or 4. complete omission ("optimized out"). These need to be supported in a generic way, to avoid reverse-engineering compiler-specific patterns.

6.4 A Generic Mapping Algorithm for WCET Analysis

This section describes our compiler-independent instruction-to-source mapping algorithm for WCET analysis. Our algorithm builds on all the mapping strategies described in the previous section, but carefully adapts and extends them to prevent underestimation and ensure tight results. The source code is made publicly available at `https://github.com/tum-ei-rcs/vigilant-insn2src-mapper`.

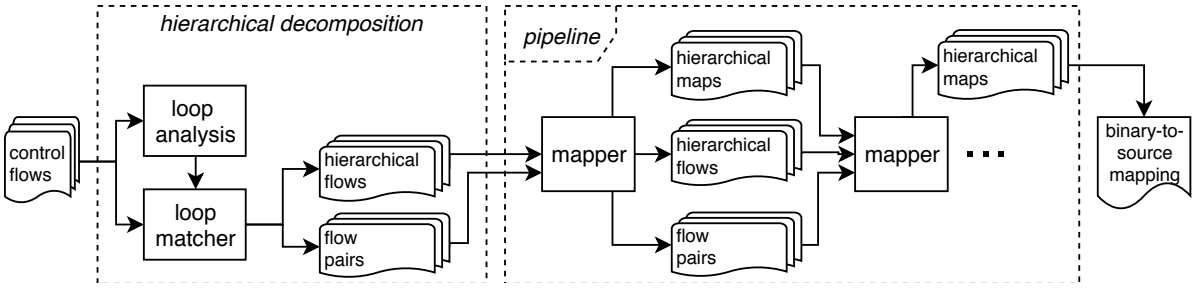


Figure 6.4: Anatomy of the mapper.

Our mapping workflow is comprising the following major steps:

1. Parse binary and source, and compute CFGs.
2. Annotate binary CFG with debug information.
3. Process inlining and loop transformations.
4. Hierarchical decomposition and matching.
5. Computation of partial mapping.
6. Overapproximative completion of mapping.
7. Back-annotation of instruction timing to source.

Detailed elaborations follow now.

6.4.1 Building annotated control flow graphs

6.4.1.1 Source code

The control flow graph of an imperative language can be obtained by parsing the source code and processing the abstract syntax tree. This processing is language-specific, since the semantics of the statements define what constitutes a source block. While the CFG is constructed, the nodes must be annotated with the counterpart of the debugging information. In this thesis we focus on the C language, and the control flows are obtained using the LLVM/clang front-end [Lat08]. The nodes of the resulting CFG are annotated with the following information: 1) range (begin, end) of source code location and 2) list of function calls (referenced CFGs). The resulting annotated sources are ready to be mapped to the binary CFG, which we obtain next.

6.4.1.2 Machine code

The binary CFG is reconstructed by first cross-compiling the program for the intended target, and then analyzing the semantics of the contained machine instructions. Towards this, we build on existing tools that decode the binary into the individual assembly instructions. We then compute the CFG on an abstraction of the machine instructions, to allow a uniform processing for different targets. In particular, it is only necessary to identify instructions that impact the shape of the CFG, whereas others can be ignored. Specifically, we only discover 1) function calls, 2) jumps and branches, and 3) return instructions. Special care is required for indirect jumps and anonymous function calls (where one instruction may be part of more than one function). Finally, we annotate nodes in the CFG with the debug information, specifically source code locations, and inlining stacks [C+10]. The resulting annotated binary CFGs are now agnostic to target- and compiler-details, and ready to be mapped to the annotated source CFGs obtained earlier.

6.4.2 Preprocessing

To prepare both the source and binary CFGs for the upcoming mapping, we pair them by identifier, take note of potential user-specified loop transformations, and process inlining stacks by copying nodes of inlined functions into their caller, as in [MLS11, SBR11b]. We further ensure that the code is not self-modifying, since this would refute a static analysis. In the next step, we solve the problem of multiple BBs per source line.

6.4.2.1 Discriminator Matching

To prevent the mapping algorithm from confusing BBs located at the same source line (neither discriminators nor columns are sufficient, see Sec. 6.2.2), we establish a mapping between binary and source discriminators using their structural information in the CFG. Towards this, we first compute discriminators for the source code by enumerating sets of BBs at each source line individually. Next, we establish a mapping from source to binary discriminators using the corrected dominator homomorphism. Let $D_b(l) = \{d_b^1, d_b^2, \dots\}$ be the set of binary discriminators at line l, and $D_s(l)$ the source counterpart. Note that binary discriminators d_b may be shared by multiple binary BBs, unlike source discriminators. Therefore, we arbitrarily pick one binary BB from each $d_b \in D_b(l)$, denoted as $pick(d_b)$. Further, let $G_b'(l) = \left(V_b', E_b', I, F\right)$ be a reduced graph as follows:

$$V_b' = \{v \mid v = pick(d_b(l)) \wedge d_b \in D_b(l)\} \cup \{I_b, F_b\}, \tag{6.10}$$

$$E_b' = \{(u, v) \mid path(u, v) \in G_b\}. \tag{6.11}$$

This graph contains the original entry I and exit nodes F of the whole binary flow, all BBs from the current line l to be mapped, and edges between them iff there exists a path between them in G_b. For the source code, $G_s'(l)$ is obtained similarly. We finally apply the dominator homomorphism to compute a map $M'(l) : V_b' \to V_s'$ between the reduced graphs (and thus between the discriminators). The mapping algorithm was corrected as proposed in Sec. 6.3.4, and additionally we removed ambiguous map entries (siblings in the dominator tree and have an indistinguishable dominator relationship to the surrounding code). The matching preference we use is *dominated* binary blocks to *dominated* source blocks.

We only accept complete maps for discriminators. Otherwise, we explicitly mark all binary and source BBs of the given line l as incompatible, forcing overapproximations later on. Although a proof for the correctness of discriminator matching is still pending, we have not

seen any BB confusions in our benchmarks. Alternatively, one could use other parts of the debug info to match the discriminators (i.e., accessed variables), or compare the semantics of source and binary blocks.

Finally, we group matching discriminators under a common label $\mathcal{D}_i(l)$ that is unique per line l, based on the mapping $M'(l)$, i.e.

$$\mathcal{D}_i(l) = \left\{d_s^i\right\} \cup \left\{d_b \mid M'(l)(d_b) = d_s\right\} \text{ for } i = 1 \ldots ||D_s(l)||. \tag{6.12}$$

6.4.3 Hierarchical decomposition & matching

Before the mapping algorithm starts, we apply a hierarchical decomposition of the CFGs to reduce the overall mapping problem into a number of smaller ones, as described in Sec. 6.3.2. As a side effect, this enables a safe WCET analysis, as shown later. We recursively partition the CFG into *subflows* along its loops. That is, if a graph G contains a loop, then we replace the loop by a surrogate node, and create a new subflow for the loop, which becomes a child of G. Towards that, we compute a loop nesting tree [Hav97], in which each node represents a loop, and child nodes are contained loops.

The decomposition process is repeated recursively on all subflows, until no further loops are found. The result is illustrated in Fig. 6.5. This decomposition results in two per-subgraph properties that we can leverage: 1) $u \operatorname{dom} v$ implies $f(u) \geq f(v)$, i.e., dominating nodes execute at least as often as dominated ones, 2) entries nodes are either loop headers or initial nodes in the top-level CFG, both of which can be taken as *fixed-points* in the upcoming node mapping.

Loop optimization: In this algorithm, we only require that existing loops are not vanishing, or that otherwise the user can provide that information during the preprocessing. This requirement is usually satisfied at all except the highest optimization levels. Since those usually clash with WCET analysis [WZKS13], this is not considered a limitation. If support for such aggressive loop optimization is still needed, this could be realized using the optimization reports generated by modern compilers, which indicate which loops have been peeled, blocked and unrolled [Fre19, Cla19]. Nevertheless, some loops might still vanish if the compiler detects their infeasibility, or even get introduced [uMVC19a]. To be safe, we check for loop preservation and consult the user for deviations between source and binary.

6.4.4 Precise & partial mapping

The actual mapping between nodes in binary and source graphs is now applied independently on pairs of subflows in the hierarchy. We establish a precise but partial mapping, containing only those nodes which have an unambiguous and safe match, as discussed in detail before. Additionally, the map is pre-populated with the fixed-points obtained during the previous step, ensuring that each subflow has at least one mapped node (its entry).

We use the control dependency mapper as described in Sec. 6.3.4, since it can be modified to preserve execution counts and order (Eq. (6.2) and (6.3)), as described in the following. First, a correct edge labeling is ensured by making some corrections to the original control dependency mapping algorithm. Towards this, we require that the lines (excluding columns and discriminators) in the debug info are correct, but missing information is still allowed. Edges (u, v) in binary and source are grouped under labels $\mathcal{MC}_{c_X(q)}$ as

$$\mathcal{MC}_{c_X(q)} = \{(u, v) \mid a(u) = X \wedge b(v) = q\} \tag{6.13}$$

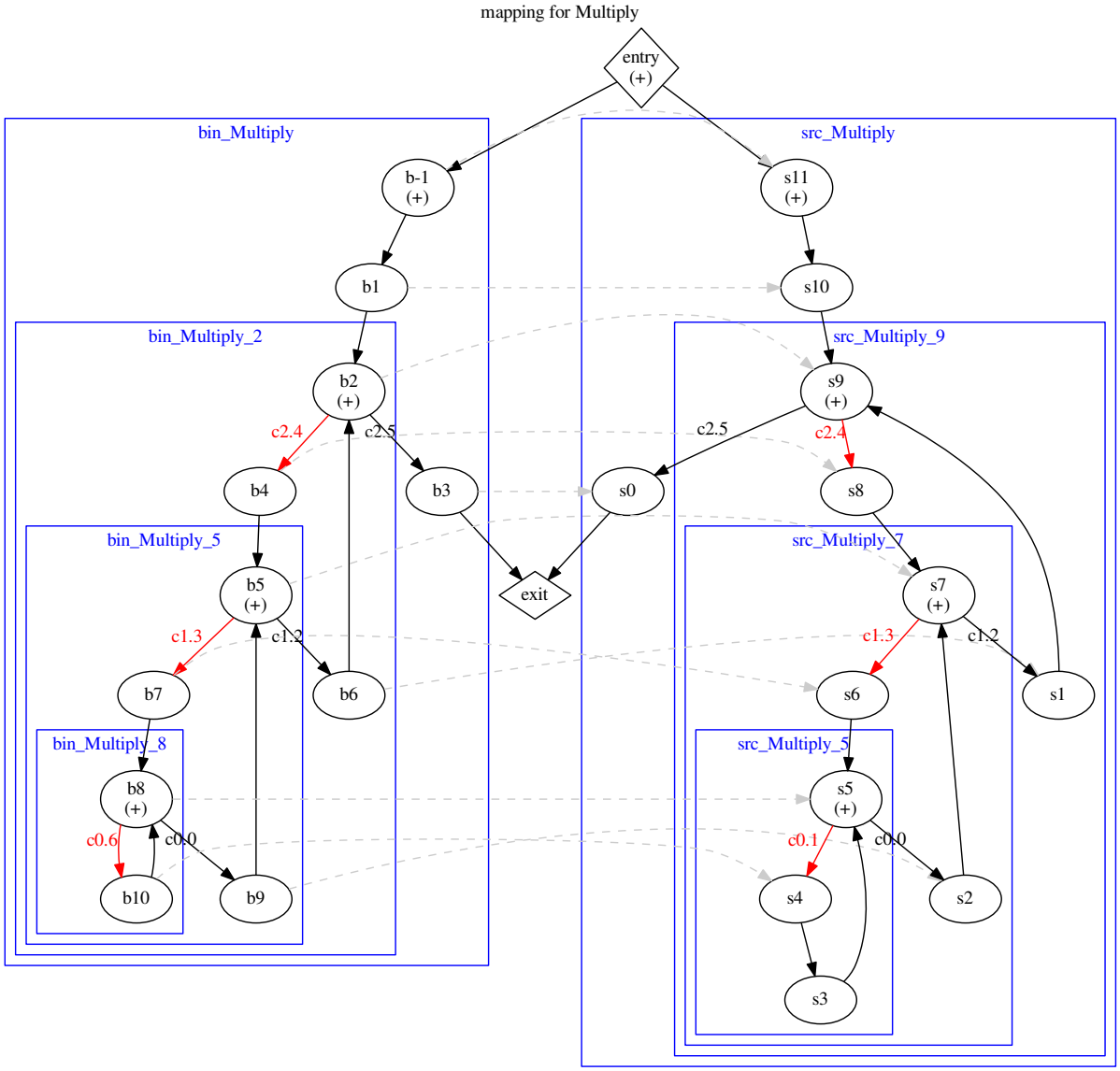


Figure 6.5: Example of generic graph mapping and hierarchical decomposition.

where functions $a()$ and $b()$ enumerate code locations of both source and binary nodes v according to

$$a(u) = (line(l_{\max}(u)), i), \quad \text{s.t. } disc(l_{\max}(u)) \in \mathcal{D}_i(line(l_{\max}(u)))$$
$$b(v) = (line(l_{\min}(v)), i), \quad \text{s.t. } disc(l_{\min}(v)) \in \mathcal{D}_i(line(l_{\min}(v))),$$

with *line* and *disc* yielding only the line number respectively discriminator of a debug location, $l_{\min}(v)$, $l_{\max}(v)$ being these locations as defined in Sec. 6.2.2, and $\mathcal{D}_i(l)$ is the discriminator label from Eq. (6.12). Using discriminator labels ensures that only edges between basic blocks that are equivalent in source and binary get the same label, removing the BB confusion that appears in [MLS11]. The BB properties are then computed as defined by Eq. (6.9).

Furthermore, when computing the controlled nodes of an edge, we exclude self-dependencies of loop headers by always applying the second condition in Eq. (6.8). This is justified by the hierarchical decomposition, since entries of subgraphs are always loop headers, and the information is therefore not lost. This results in precisely constrained execution counts, and allows us to map a binary BB to any of the matching source BBs whilst guaranteeing preservation of execution count. If execution order shall also be maintained (depends on the microarchitecture, see Sec. 6.3), this can be established using our modified dominator homomorphism.

6.4.5 Overapproximative Completion of Mapping

At this point, some binary BBs may remain unmapped, e.g., due to missing debug information, as illustrated in Figure 6.6; only the green shadowed BBs on the left side been precisely mapped, whereas the remaining ones are still missing, thus their execution time would be missing in the WCET estimate. To prevent underestimation, we lump the timing if unmapped binary BBs into the closest mapped binary BB, where it gets annotated in the corresponding source block. Note that this is a simple overapproximation, which we later refine in Chapter 7.

To implement the lumping, we first check for "simple paths". Let P be a path in G_b, with u being one unmapped node in P, and such that P can only be entered at the first node, and only left at the last. Assume further, that P is loop-free, which is guaranteed by our decomposition. Consequently, all nodes on P have the same execution frequency. Given the unmapped node u, we walk along P in both directions. If we encounter a mapped node v, then the timing of u is lumped into v.

If no binary BB can be found on a simple path, we lump the timing into the closest ancestor of u in the dominator tree. Note that this is safe only in our mapping, since in the worst case we reach the entry of each subflow, which was a fixed-point in the mapping and therefore guaranteed to be represented in the source code. Furthermore, the graph hierarchy also ensures that dominators are always guaranteed to execute at least as often as their dominated nodes, and thus underestimation of the execution count is also avoided.

Finally, execution order can be maintained by lumping iteratively in reverse topological order on G_b.

6.4.6 Back-annotation of timing to source code

The final step to enable source analysis, is to annotate each source BB with the timing of the mapped binary BBs. For simple processors as in the previous chapter, this boils down to summing up the instruction times of all binary BBs that map ta source BB, and annotate

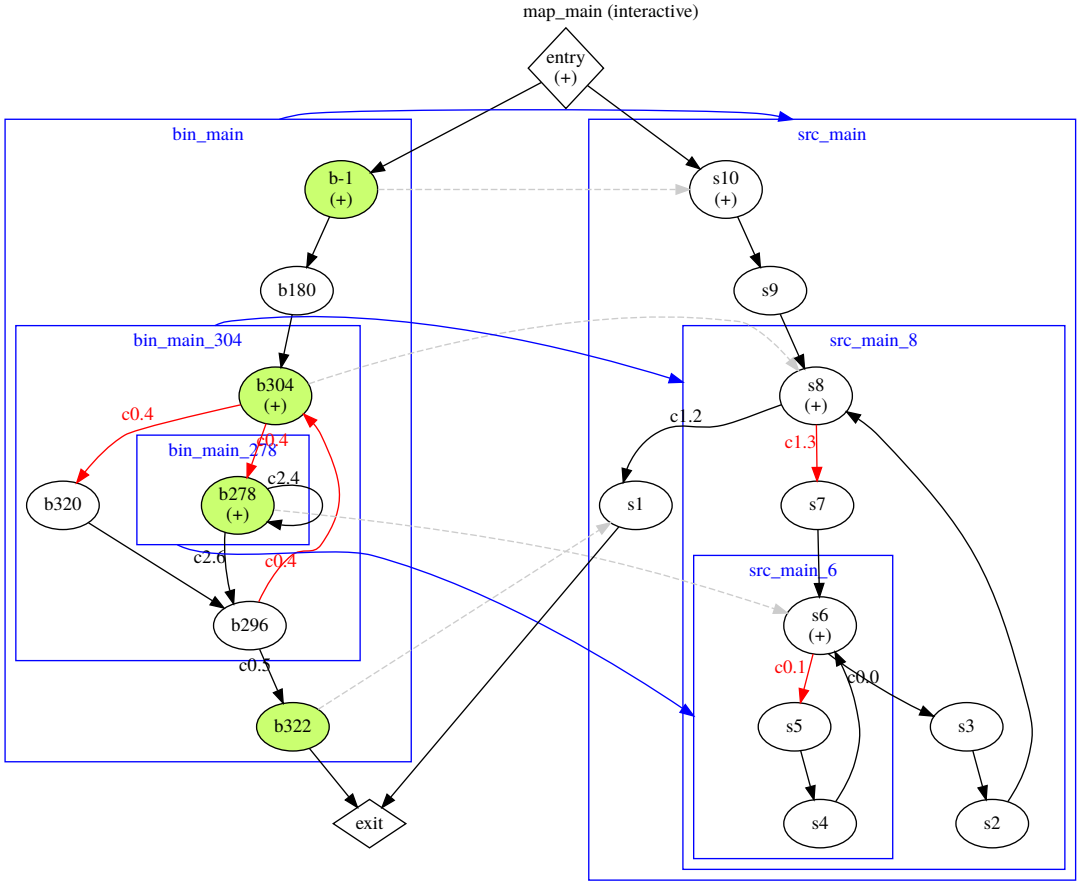


Figure 6.6: Example of partial mapping.

them at the beginning of the source BB. For targets with more complex timing behavior with caches, we refer the reader to the next chapter.

6.5 Experiments

We evaluated our mapping algorithm again the Mälardalen WCET benchmarks [GBEL10], using the WCET overall method described in the previous Chapter. As before, our target processor is an Atmel Atmega128, which implements a cache-less, in-order, pipelined microarchitecture. As a compiler, we used gcc 7.3 without any changes.

As baselines for our WCET estimates we have again performed both random simulations with a cycle-accurate simulator, and WCET estimations with the traditional, binary-level WCET analyzer *Bound-T* [HS02]. The simulation is performed with known worst-case inputs and thus close to the WCET path, which highlights overestimation caused by the mapping, as well as unsafe results.

Moreover, we compare the results of the here-proposed general, compiler-independent mapping with our existing mapper from the previous chapter, which was crafted specifically for one compiler version and flag setting, and is thus referred to as *precise mapping*. Since that mapper cannot handle other compiler versions or flags than the ones being reverse-engineered, it can only serve as a reference for optimization level O0. As we will shortly show, the results indicate that our generic mapping is in fact close to that precise mapper, but naturally a generic mapping does not need to be re-developed for every triple of target, compiler (version) and its flags.

The results are summarized in Table 6.1. For each of the three WCET estimation methods we also give an upper bound on the tightness of the estimate, which is obtained comparing the estimate to the simulated value. Note that this can only be an *upper* bound, since there is no guarantee that the simulation indeed has reached the WCET case at instruction level. In other words, column Δ gives a possibly pessimistic overestimation error for each method.

Notes on Benchmarks. The programs have been selected to stress-test various aspects of the mapping. 1. The benchmarks *crc, fdct, fibcall, jfdctint, matmult* and *ud* are single-path programs in the source code. That is, there are no flow dependencies on external variables. Therefore, the binary-level analyzer should produce tight results, and mapping imprecisions become obvious. 2. The benchmark *fibcall* shows what happens if a function is completely optimized out, due to constant propagation and inlining. 3. The benchmark *nsichneu* is a smoke test for BB confusion, missing hierarchy and further also scalability. It consists of 378 nested, often multi-clause if-statements, wrapped in a single loop. The compiler can optimize aggressively here, and debug information is often ambiguous. 4. The benchmark *cover-50* consists of a loop containing a switch-case statement with 50 consequents. We have turned off jump tables, to force the compiler to implement a binary search which differs heavily from the source in its CFG structure. 5. The benchmarks *adpcm, jfdcting, matmult* and *ud* contain implicit library calls caused by arithmetic operations, which source-level analysis can only over-approximate.

Overall impressions. As Tab. 6.1 shows, the generic mapping is unsurprisingly less precise than a custom-built mapping. Nevertheless, it is only little worse in most cases. Some benchmarks seem to be mapped too coarsely, e.g., *cover-50*, with more than 1,000% overestimation, but also *adpcm*, where the precise mapper had at most 44% overestimation, but the generic mapping more than twice as much. This is analyzed in detail in the next section.

Table 6.1: Tightest WCET estimates per method and benchmark. Entries marked in bold where our approach is within 1% of binary-level WCET estimate or better.

		Binary Level			Source Level			
		Sim.	ILP/IPET		Generic Map		Precise Map	
benchmark	opt.	WCET$\geq$	WCET$\leq$	Δ%	WCET$\leq$	Δ%	WCET$\leq$	Δ%
adpcm	O0	44,045	88,872	+101.8	84,744	+99.2	63,710	**+44.6**
	O1	31,699	75,370	+137.8	73,512	**+131.9**	–	–
cnt	O0	8,318	8,376	**+0.7**	8,915	+6.4	8,376	**+0.7**
	O1	1,621	1,663	**+2.6**	1,983	+22.3	–	–
cover-50	O0	3,524	4,100	**+16.3**	58,029	+1,546.6	3,524	$\approx$ **0**
	O1	1,369	2,205	**+61.0**	10,985	+702.4	–	–
crc	O0	130,325	143,646	+10.2	137,163	+5.2	131,652	**+1.0**
	O1	40,612	43,953	**+8.2**	48,052	+18.3	–	–
fdct	O0	22,097	22,097	$\approx$ **0**	25,013	+13.1	22,097	$\approx$ **0**
	O1	7,691	8,628	**+12.2**	8,648	**+12.4**	–	–
fibcall	O0	1,820	1,830	+0.5	1,904	+4.6	1,830	**+0.5**
	O1	6	6	$\approx$ **0**	6	$\approx$ **0**	–	–
insertsort	O0	5,476	5,476	$\approx$ **0**	6,236	+13.8	5,476	$\approx$ **0**
	O1	943	1,519	+61.1	1,142	**+21.1**	–	–
jfdctint	O0	14,143	14,143	$\approx$ **0**	15,906	+12.4	14,143	$\approx$ **0**
	O1	7,427	7,427	$\approx$ **0**	8,361	+12.5	–	–
matmult	O0	984,816	984,816	$\approx$ **0**	1,040,705	+5.6	984,816	$\approx$ **0**
	O1	294,413	294,413	$\approx$ **0**	309,571	+5.1	–	–
ns	O0	56,434	56,450	$\approx$ **0**	58,473	+3.6	56,438	$\approx$ **0**
	O1	9,757	11,410	**+16.9**	14,133	+44.8	–	–
nsichneu	O0	33,203	75,383[b]	+127.0	53,130	+60.0	35,195	**+5.9**
	O1	21,625	44,341	+105.0	35,024	**+40.7**	–	–
ud	O0	34,153	87,560	+156.4	49,907	+46.1	37,304	**+9.2**
	O1	24,885	58,452	+134.9	38,429	**+54.4**	–	–

Δ: upper bound WCET tightness; [b]const. & arith. analysis disabled to avoid timeout

More importantly, the generic mapper also works with optimization, unlike the precise/-custom one. It is interesting that half of the time, despite its apparent weaknesses, the generic mapping still outperforms the binary-level WCET analyzer. This suggests that mapping imprecision can be compensated by virtue of detecting more infeasible paths, and makes the *generic and compiler-independent* strategy, as presented here, attractive for WCET analysis.

6.6 Discussion

Although the results show room for improvement, they support the feasibility and usefulness of implementing a generic, compiler-independent mapping and timing annotation method. As we argue in the following, methods from VP can already be beneficial in source-level WCET analysis, but there are opportunities for improving them for both VP and WCET analysis.

6.6.1 Imprecision of Estimates

Since the results of our WCET analysis are maximally precise (we did not apply abstractions here), any imprecision of the generic mapping can be attributed to either a bad mapping quality, context beyond the analysis domain, or a combination thereof.

6.6.1.1 Mapping Imprecision

Not all imprecision can be attributed to the mapping. As in the previous chapter, implicit library calls are causing some overestimation in the benchmarks *adpcm*, *cnt*, *jfdctint*, *matmult* and *ud*, but have equal effects on both source- and binary-level estimates. We will return to this problem in Section 8.5. In the following we only review the imprecision caused by the mapping algorithm.

When source and binary CFGs start to diverge, the handling of flow differences becomes the key factor for precise results, often enforcing approximations. Unlike applications in virtual prototyping, WCET analysis is required to always *over*approximate, therefore approximation errors do not cancel out, and lead to greater deviations from cycle-accurate simulations. Whereas we propose a refined strategy for flow differences in the next chapter (necessitated by a cache model), it is worth examining a few cases that are especially challenging for the proposed mapping.

Consider the two benchmarks *nsichneu* and *cover-50*, which we had chosen because they were likely candidates for such overapproximation. Indeed, the results show large overestimation compared to the binary-level analyzer and the simulation. However, *nsichneu* is still better than the binary-level analyzer, possibly caused by the overestimation of the latter. A clear case presents itself with *cover-50*, shown in Figure 6.7.

The switch-case was implemented as binary search, and the control-dependency mapping failed to map any of the cases. Consequently, all timing has been lumped in the switch header, accumulating a large error due to the surrounding loop. As a side note, the pure dominator homomorphism (see appendix) was able to map the final case nodes, but also failed for all the binary search decision nodes. In contrast, our *precise mapper* implements a special switch-case handling and ends up with zero overestimation. This is explained in the following.

Switch-Case Improvements. The flow differences of switch-case constructs can be mapped precisely by attributing the timing of all decision BBs that lead to a case, to that case BB

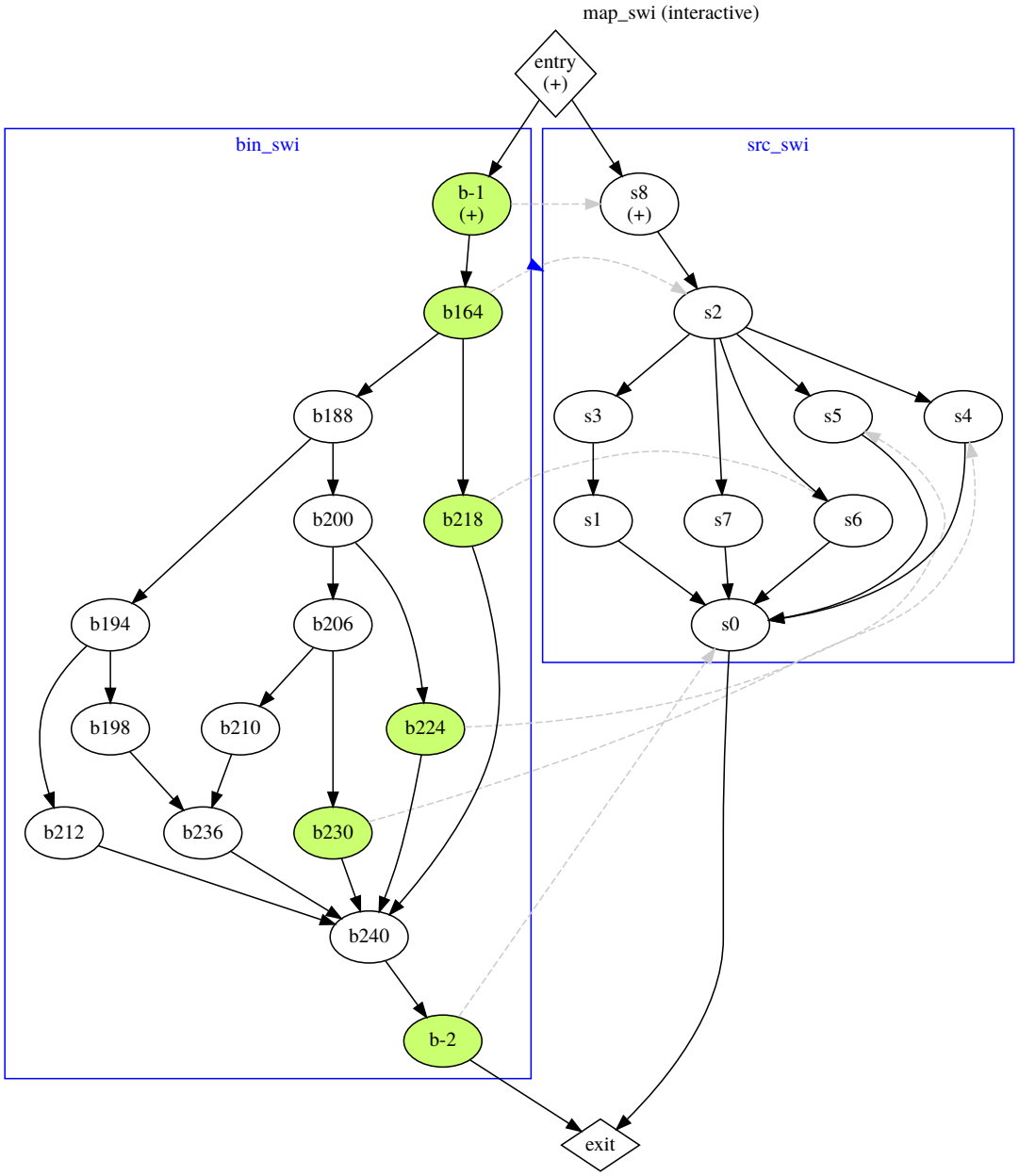


Figure 6.7: Mapping difficulties for binary search implementation of switch-case.

itself. For example, in Fig.6.7, $bb164$ is the binary BB of the switch header, and $b224$ the BB corresponding to one case. The timing of all BBs on $path(b164, b224)$ can be summed up and attributed to $b224$. Similarly, the timing on $path(b164, b230)$ can be attributed to $b230$. Two aspects should be noted. First, the paths can be overlapping, thus this is different from lumping, where we had "moved" the timing in the CFG; here we rather "duplicate" the timing. Second, this sort of mapping evades any dominator relationship, and therefore cannot be handled by either of the mappers. The algorithm is given in Alg.2 and Alg.3. The WCET estimate with this explicit switch-case handling for *cover-50* is then reduced to 3,524 at O0, and to 1,359 at O1, i.e., maximally precise.

Algorithm 2: Switch-case processing (identification).

```
Input: source CFG G_s, binary CFG G_b, debug info D
begin
    V_s ← find_switchheads_source(G_s)
    M_loc ← find_switch_bbbs(G_b, V_s, D)        // maps location to set of bin.BBs
    foreach v ∈ V_s do
        l ← D.get_location(v)
        if l ∈ M_loc then
            V_b ← {v | v ∈ M_loc(l) ∧ ¬D.is_case(v)}        // get decision BBs only
            G_b,sw ← G_b.subgraph(V_b)
            if is_binary_search(G_b,sw, D, l) and is_weakly_connected(G_b,sw) then
                handle_switch_binary_search(l, G_b,sw, G_b)                // see Alg.3
```

Y-Structures. Another weakness of the mapping are Y-structures in the CFG. Neither of the two joining paths is a dominator of the following code, therefore not a control dependency. Consequently, nodes after such joins remain unmapped. This could be enhanced using dominator *fronts*.

Such specific cases can easily be handled by adding further stages to our mapping pipeline. However, we believe that the strategies can and should be generalized, to handle more flow differences, and to avoid that compiler-specific patterns are encoded.

6.6.2 Threats to Safety

In this work, we require the *line table* of the debug information to be correct, yet it is allowed to be incomplete. Note that mismatches as in Fig. 6.3 are merely imprecise, but still correct. We only consider line entries erroneous if a source BB has a corresponding line table entry that does *not overlap* with its line span in the source. In the presence of such an error, our labels and therefore the map entries could become incorrect, which might result in an underestimation of the WCET. In the context of WCET analysis this seems an acceptable requirement, as tools usually have to be certified whenever timing analysis requires a certain level of assurance [FFL+11].

Under optimization. Assuming the entries in the line table are still correct under optimization, the four general optimization effects identified earlier are covered: (1) Change in execution order is tracked by debug locations, and otherwise safely overapproximated. (2) A change of execution conditions can either lead to less BBs being precisely mapped, or, if

Algorithm 3: Switch-case processing (attribution).

Input: location l, binary subgraph $G_{b,sw} \subseteq G_b$, binary CFG G_b
begin
  if $|G_{b,sw}| = 1$ **then**
    return // nothing to gain
  // Compute check paths:
  $B \leftarrow$ topological_sort($G_{b,sw}$)
  $v_h \leftarrow B[0]$ // switch head BB
  $B = B[1 \dots]$
  $M_a : V \rightarrow \emptyset$
  foreach $v \in B$ **do**
    if $G_{b,sw}$.in_degree(v) == 1 **then**
      $p \leftarrow u$, s.t. $(u, v) \in G_{b,sw}$
      if $p \neq v_h$ **then**
        $M_a(v) \leftarrow M_a(p) \cup \{p\}$ // update M_a entry for value p
  // attribute paths to case BBs:
  foreach $v \in B$ **do**
    foreach $c \in \{x \mid (v, x) \in G_b\}$ **do**
      if $c \notin B$ **then**
        foreach $x \in M_a(v)$ **do**
          G_b.time(c) += G_b.time(x) // add time from BB x to case c
        G_b.time(c) += G_b.time(v)
  foreach $v \in B$ **do**
1   G_b.time(v) $\leftarrow 0$ // zero time from decision BB

debug info is available, is detected and carried to the new binary location. (3) Blocks that are optimized out are obviously not part of the mapping domain, and thus supported. (4) Last but not least, duplication, although not requiring any action, can be detected if debug info allows. We therefore conclude that the proposed mapping is safe for WCET analysis, under the assumption of correct line table entries. We have further manually inspected the computed mappings in all upcoming experiments, and not found any contradictions, inconsistencies or apparent errors, suggesting that the line table is a reliable source of information in practice.

6.6.3 Further Improvements

Beyond the already mentioned refinements in completing the mapping and handling Y-structures, further improvements are possible. For example, this mapping assumes the worst case for time-variable instructions. This mainly concerns branch instructions, which often vary in their timing depending on which edge was taken. To precisely annotate such behavior, the branching instruction needs to be analyzed for its polarity, and considered during edge matching. However, the bigger challenge is to annotate this in the source code, since not all binary edges have equivalent source edges/locations.

More improvements can be made by allowing ourselves to alter the source code. Flow differences do not necessarily have to be overapproximated, but could also be "lifted back" into the source, by modeling the unmappable instructions using source statements. This, in principle, is decompilation. It would increase the source complexity, but reduces overestimation, thereby offering an opportunity for a trade-off.

6.6.4 Open Problems

6.6.4.1 Dynamically Resolved Flows

The CFG has to be computed to perform the mapping. Although this is in principle orthogonal to the mapping, it can become difficult when the compiler makes use of indirect addressing, resulting in either unknown edges in the CFG (target BB unknown), or a superset of possible edges (cannot decide which one). Among others, indirect addressing can be caused by switch statements (jump tables), function pointers and virtual functions (C++). The instructions then do not contain absolute addresses for branches and jumps, but instead refer to the current value in some register. Hence, without performing a register value analysis, the next instruction is unknown, which prevents constructing the CFG. Such a value analysis is commonly performed in the traditional, binary-level WCET analysis, and also required here. However, an incomplete mapping could be used to limit the number of possibilities for pointers and jump targets.

6.6.4.2 Conditional instructions

Compilers may perform "if conversions" if the target supports conditionally executed instructions. These cannot be efficiently handled with the presented methods, since they do not appear as branches in the binary CFG. Such instructions can be found in the ARM Instruction Set Architecture (ISA), and the compiler may use those even when optimization is turned off.

6.7 Comparison to Related Work

We have adopted and corrected methods from the Virtual Prototyping domain to solve the graph mapping problem, since no equivalent work existed in the WCET domain. However,

there are some related attempts to solve the mapping problem to a different extent, or with methods that we consider inadequate. We summarize such work in the following.

Compiler Modifications. There are many approaches that rely on extensions to the compiler to obtain a partial mapping between source and binary, to be used to compute flow facts. A complete mapping is however not computed. Some modify existing optimization passes to maintain better debug information [KPP10], whereas others propose methods to transform meta-information among different levels of program representation, such as in the T-CREST project [SAA+15]. In general, many WCET analyzers desire some integration with the compiler, which however always creates a lock-in to a specific tool chain.

Simplifying the Mapping Problem. Some approaches modify the compiler such that source and binary flows are guaranteed to be isomorph [BGP09]. Others, in both the WCET and VP areas [SAA+15, BEG+15], change the original program by inserting markers that propagate through to the binary. These approaches are orthogonal to this work, since the effect of markers is conceptually similar to having better debug information, and compiler modifications can always be used to prevent certain optimizations, or enhance their tracability.

IR-level Mapping. Several groups from the VP community have proposed mappings from binary to compiler IR (thus, only considering back-end optimization) [CZG13, MP18, WH09a], or only from IR to source (thus, only considering high-level optimization), as in [WH12]. Both of these approaches have their place next to a full binary-to-source mapping as proposed here, since they can be used as fallback solutions or intermediate steps.

Model Identification. Another line of work avoids solving the mapping problem for each program individually by deriving a source-level model once, and subsequently obtaining timing annotations from this model only [AGLS16]. The timing behavior is measured on a set of training programs, and then used to derive a timing model for source constructs. Only the model is subsequently used to analyze and annotate new programs. Similarly, there are approaches based on machine learning [MGG17]. We did not consider them for WCET analysis, because there is no guarantee for the correctness of such timing models.

6.8 Chapter Summary

We have proposed a method to compute a mapping from machine instructions to source code which is compiler- and target-independent and can tolerate optimization. Towards that, we have built on methods from the Virtual Prototyping domain, which however were not safe for WCET analysis initially. After proposing fixes to have these methods compute a sound mapping, we combined them into a hierarchical, general-purpose mapping infrastructure. We first compute binary and source CFGs, and establish a structure-based mapping for the BBs on each source line separately, to disambiguate the debugging information. A modified dominator homomorphism was used to this end, which no longer produces unsafe results. Second, we established the overall mapping individually on pairs of subflows, as given by their loop nesting trees, using a control dependency-preserving mapping that is sound thanks to the preceding disambiguation. Finally, we have completed the partial mapping by "lumping" unmapped binary BBs into their pre- or postdominators (whichever was able to better capture their execution count). The WCET estimates based on this generic mapping are quite promising, several times outperforming traditional WCET analysis, yet lacking behind

the mapper from the previous chapter, which was specifically crafted for one compiler and flag setting. The generic mapping is however able to work for different compilers and flag settings, and therefore of greater interest to make WCET analysis portable to different targets. It has shown a broadly similar precision for optimization level O1, without any changes to the mapping algorithm. Additionally, we have identified several causes where the generic mapper is inferior and suggested improvements for them.

In summary, this chapter demonstrates that a generic, compiler-independent back-annotation with sufficient precision is possible, especially under moderate optimization, where source-level analysis apparently can identify more infeasible paths than binary analyzers.

Chapter 7

Microarchitectural Source-Level Models

***

The approach presented in previous chapters only accounts for simple processors, where instruction timing can be assumed quasi-constant. While this is sufficient to support 8-bit microcontrollers which are still widely used today [Lew17], it cannot capture the timing behavior for larger system-on-chips, for example the popular embedded processors ARM Cortex and Intel Atom. Such processors make use of various performance-enhancing hardware features and complex bus systems, which can introduce large variances to the instruction timing. To assume their worst case is no longer a viable option, since that could lead to an overestimation of several orders of magnitude [WEE+08]. Consequently, we need to develop extensions to the proposed source-level analysis to support such features.

Unfortunately, performance-enhancing features in general-purpose processors have been conceived to increase the *average-case* performance, but make WCET estimation more complex and less tight. In some cases they can even decrease the actual worst-case performance (not only the estimate), through their complex interactions [WEE+08, uC18a].

This chapter presents an extension for the most important performance-enhancing processor feature, namely for caches, and further discusses the feasibility of modeling other features commonly found in embedded processors, such as branch predictors and prefetchers.

Processor Pipeline. We consider a classic RISC-like microarchitecture with a pipeline of four stages, as illustrated in Fig. 7.1. The processor has both instruction- and data caches, which speed up all access to the slower backing memory (e.g., Dynamic Random Access Memory (DRAM) or Electrically Erasable Programmable Read-Only Memory (EEPROM)). The pipeline consists of the following four steps:

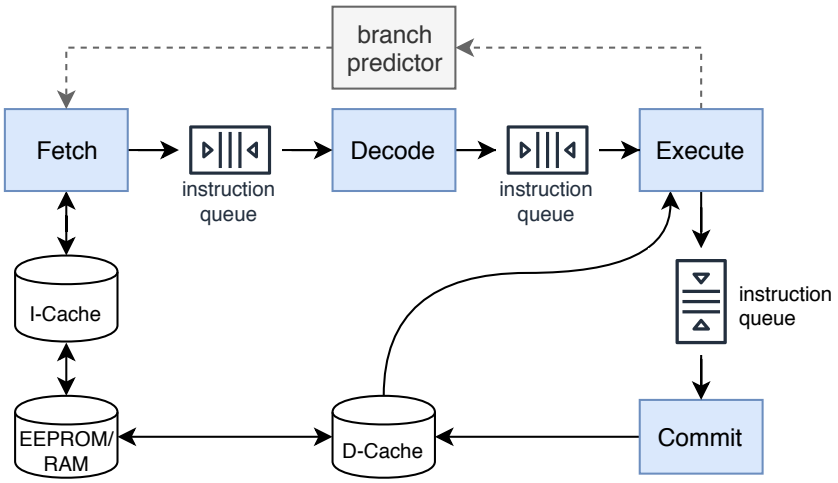


Figure 7.1: Considered microarchitecture for this chapter.

1. **Instruction fetch**: the next instruction is read from instruction cache (I-cache). If the instruction is cached, it can be forwarded to the decoder within one clock cycle, otherwise it must be loaded from the slower backing memory, causing a delay.
2. **Instruction decode**: the instruction is interpreted, and forwarded to the execution stage. Interpretation might entail decoding microcode and reading memory contents for indirect addressing.
3. **Execute**: the instruction is performed according to its semantics. For example, this could be to compare two numbers, or to move data between registers.
4. **Commit**: the results are written back to the processor registers and become visible for subsequent instructions.

A branch predictor can be employed by the fetch stage to anticipate and load upcoming instructions before the actual branch outcome is known, and thereby to hide their memory access latency.

The classic RISC pipeline has a fifth stage (Memory Access) between execute and commit, which however is mainly used to accommodate for the additional time required for memory accesses. This is therefore not explicitly modeled, but instead captured in different execution times for each instruction.

We further assume *in-order* execution, that is, the instructions are reaching the execution and commit stages in the same order as given by the program counter. Additionally, we assume the pipeline is *scalar*, i.e., it only fetches, decodes and executes one instruction at a time at each stage. This is in contrast to high-end processors, which use *out-of-order* pipelines that can change the instruction order to parallelize work, and issue multiple instructions at each clock cycle [Fog18, uC18a].

With such an architecture, there are different ways in which timing could be measured and predicted. One way is to count the number of clock cycles between instruction fetches, i.e., how long it takes to get an instruction from memory to the processor. A WCET prediction could then add the execution time to that fetch time. Yet this might be deceiving, since the processor may occasionally fetch an instruction based on a branch prediction that was incorrect. Another way is to count the cycles between commits, i.e., the rate at which instructions are completed. This is the chosen metric in our experiments, which however only makes sense for in-order pipelines. In out-of-order pipelines, some instructions may execute *and* never reach the commit stage, due to *speculative execution*.

7.1 Caches

The *Memory Wall* [WM95] is arguably one of the biggest performance bottlenecks in modern computer systems. This term stands for the unpreventable speed difference between fast processors and slow memory in current computer technology, verbally illustrating the performance wall that we run into when a fast processor requires slow memory access. This speed difference is currently several orders of magnitude large and still diverging, and thus it is ever more important to consider memory accesses in a timing analysis.

Caching is the omnipresent approach to counteract this issue, by preloading or latching all data in faster, smaller memories, and exploiting the *principle of temporal locality*. That is, data that has been recently accessed, is likely to be accessed soon in the future again. It thus makes sense to buffer recently used data in caches. Every time a data item is requested, we check for the desired data in the faster cache, before accessing the slow memory. If we find the data, this event is called a *cache hit*, and we have circumvented waiting for the slow memory. If the cache does not contain the data, called a *cache miss*, we have to pay the penalty for accessing the slow memory (usually DRAM). After this, the data is placed in the cache for future reference.

Since caches have to be small to be fast, there are inevitably situations when data is not in the cache, and needs to be loaded from slower memory. Even if we were able to perfectly predict what data (including instructions) is needed, then there are still compulsory misses on first access. As a result, execution can be slowed down by one or two orders of magnitude even with fast caches and very high hit ratio. For example, let us assume each instruction takes one cycle, and that each cache miss costs five extra cycles. Then, even with $p = 95\%$ hit ratio, a program with X instructions would take $X + X(1 - p)5 = 1.25X$ cycles, i.e., experience a 25% slowdown.

Consequently, caches are the most important processor feature to be supported in a WCET analysis method, since otherwise orders of magnitude of overestimation can be expected, rendering the estimate useless. However, the limited body of pre-existing work in source-level timing analysis has never looked at caches before. There is some work in the Virtual Prototyping community, as mentioned below, yet these models are incomplete and not sound, and thus not usable for WCET analysis. We therefore introduce a novel source-level cache model in the following.

7.1.1 Mode of Operation

Caches are organized in units of *lines*, typically 32 to 64 bytes in size. Data between caches and the slower backing memory can only be transferred in multiples of these units. A *cache block* is a contiguous chunk of memory that has been loaded into a cache line; therefore it has the same size as one line. The *associativity* defines in how many different lines a given block may be stored. If we only allow one line for each block ("direct-mapped" cache), then the lookup of a block is easy and fast, but the cache is possibly used inefficiently since we cannot avoid evictions whenever another block is already occupying its line, even if other lines are unused. On the other hand, allowing A different locations ("A-way associative"), then the lookup becomes more complicated and thus slower, but some evictions can be avoided, resulting in better performance. Typical associativities lie between one and eight in practice.

We now briefly describe the mode of operation of a single level of cache. Without loss of generality, let us consider an A-way associative cache with the following specifications:

- n cache lines,
- line/block size of b bytes,
- a total capacity of $b \cdot n$ bytes, and
- associativity A,

for instructions or data. For implementation efficiency in hardware, all these properties are usually a power of two.

Cache Block and Offset. Programs may need to access memory addresses that are not multiples of the block size. For example, suppose we require the memory contents at address m. Since we can only access entire blocks, we first compute the address where the *cache block* starts as

$$block(m) = \left\lfloor \frac{m}{b} \right\rfloor b. \tag{7.1}$$

In practice, where s is a power of two, this is done by masking the $ld(b)$ least significant bits of m. Therefore, when accessing any memory address m, the cache is only concerned with the block address. In general $m \neq block(m)$, resulting in an access with $o(m) = m - block(m)$ bytes into the cache block, where $o(m)$ is called the *offset*. The CPU has to take care of slicing the returned data accordingly.

Cache Sets. By definition of its associativity, there are exactly A different cache lines out of the n lines where a given memory block could be stored. Vice versa, the cache is logically partitioned into n/A groups $F = \{f_0, \ldots, f_{(n/A)-1}\}$, called *cache sets*[1]. Specifically, let m be the memory address of that shall be accessed, then the index of its cache set is computed as [FW99]

$$set(m) = f_i, \text{ where } i = m \,\%\, \frac{n}{A}, \tag{7.2}$$

where "%" is the modulo operator. Again, computing the set can be achieved efficiently in practice by bitmasking.

Replacement Policy. All blocks of a cache set compete for its A cache lines. Hence, if more than A blocks map to the same cache set, then these are *conflicting* with each other, and a replacement policy is used to decide which one is evicted in favor of a new access. Commonly used policies are

- Least Recently Used (LRU): maintains a queue of blocks by age. Newly accessed blocks are promoted to (if already contained) or inserted at (if not in cache) the head, pushing older ones towards the tail. The queue maintains a maximum length of A by dropping blocks at the tail.

[1] How the logical cache sets are mapped onto the physical cache lines is a matter of implementation, yet not relevant for our analysis.

- First-In First-Out (FIFO): Similar to LRU, but blocks already in the queue are not promoted if accessed again. Therefore, whichever block was accessed first is also evicted first.
- Most Recently Used (MRU): Only keeps track of the youngest block, evicting any of the others (usually the one with the lowest index) when required.
- Pseudo-LRU (PLRU): a tree-based approximation of LRU that is cheaper to implement, but does not always evict the least recently used element.
- Random Replacement (RR): Any block in the set can be discarded. Such a policy is amenable for probabilistic analysis if the randomness is "good enough".

Hits and Misses. Accessing a block that is currently residing in the cache results in a cache hit and thus low access time. Otherwise, a miss is happening, and the block has to be fetched from the slower backing memory into the cache, possibly triggering an eviction if the set is full. Note that only blocks mapping to the same set can conflict with each other, and thus we can model and analyze each cache set separately as a smaller, fully associative cache.

Unaligned Access. If the CPU requests an access at address m with an offset of $o(m) > 0$ and with a length of more than $b - o$ bytes, then this requires accessing two memory blocks, since the requested data extends beyond the end of one cache line. This is called *unaligned* access, and obviously has an adverse impact on performance. While compilers try to avoid this situation, a small amount of unaligned access can still happen if the data cannot be aligned to cache lines for some reason.

7.1.1.1 Instruction Caches

The instruction cache buffers recently used machine instructions to avoid access to the slower backing memory, which would stall the pipeline and delay program execution. The addresses being accessed correspond to the *load addresses* of the instructions, which in embedded systems are often known statically as the *text address*, due to the use of static linking and absence of an operating system (implications otherwise are discussed in Chapter 8).

For analysis, the instruction stream can be clustered into *line blocks* (for short, *L-blocks*), which are non-overlapping subsets of the basic blocks delimited by cache line boundaries [LM97]. In other words, each BB is chunked into a sequence of L-blocks, and the cut points are the ends of cache blocks/lines. As soon as the first instruction in the L-block is accessed, the remaining instructions of the same L-block are guaranteed to be a cache hit. It therefore suffices to analyze instruction cache behavior at the granularity of L-blocks.

Let $\mathcal{L}_i = \{v_{i,1}, v_{i,2}, \dots\}$ be the set of L-blocks belonging to a BB v_i, then the execution time of v_i is given by

$$t(v_i) = \sum_{v_{i,j} \in \mathcal{L}} c_{i,j} + \begin{cases} 0 & \text{if } block(addr(v_{i,j})) \text{ in cache} \\ t^{\text{miss}} & \text{else} \end{cases}, \tag{7.3}$$

where $addr(v_{i,j})$ is the lowest address of the L-block, and $c_{i,j}$ is the time that the instructions in L-block $v_{i,j}$ take to execute, once they are fetched from memory. Four important properties can be observed.

1. The timing corresponding to term $c_{i,j}$ is a constant lower bound for $t(v_{i,j})$ which cannot be undercut by any execution, since all L-blocks in a BB must execute once the BB starts. This term therefore corresponds to our `TIC(x)` macro from earlier chapters.

2. There can be multiple cache misses per BB. In the presence of an instruction prefetcher (discussed in Section 7.6), this can often be reduced to only one miss per BB, following the first observation.

3. If $o(addr(v_{i,j})) > 0$, it might be the case that the predecessor BB falls into the same cache block, such that even the first access to a BB may result in a hit. In fact, it is even likely, since compilers try to put successive BBs closely together in memory, to exploit the principle of locality [S+89].

4. Another important property is that the L-blocks of one BB usually fall into different cache sets, and therefore are not evicting each other. In rare cases, however – and one of these cases is the *fdct* benchmark used later – it can happen that a BB comprises L-blocks which can evict each other. This either happens for very long BBs, such that $set(m)$ wraps around, or when the BB is non-contiguous in the address space by virtue of unconditional jumps introduced by the compiler, such that $set(m)$ incidentally evaluates identically for two different m. This property requires us to model the eviction also *within* BBs, and not only between them.

Finally, note that unaligned access does not have to be considered at L-block level, which by definition does not extend beyond one cache line. Instead, we must consider for unaligned access when clustering the instruction stream into L-blocks, since (rarely, and only on some microarchitectures) a single instruction might break into two L-blocks.

7.1.2 Traditional Cache Analysis

Cache analysis is part of the microarchitectural analysis, and traditionally performed using AI (see Section 3.4) on the CFG. The original approach of integrating the cache behavior into the ILP system [LMW95] was more precise, yet suffered from bad scalability [Wil04]. AI is used to compute invariants on the cache state over the program locations. For example, it can determine which accesses always result in cache hits or misses. These accesses need no longer be explicitly modeled, and can directly be encoded as a longer/shorter BB execution time. Memory blocks which cannot be classified are usually assumed to behave as the worst case, and also not modeled. Therefore, the scalability of the path analysis is not affected in the presence of caches in the traditional approach.

We now briefly review the traditional AI-based cache analysis, summarizing how these invariants are established. Since the AI framework is sound, it can be used to establish cache invariants in the context of WCET analysis. Its approximative nature has been claimed to be precise enough in practice [Wil04], i.e., not introducing much overestimation of the WCET. For reasons of clarity, we now briefly recapitulate the original version of AI-based cache analysis, as introduced by Ferdinand and Wilhelm [FW99], although some flaws have been identified recently [HJR11], and although a more precise encoding meanwhile has been proposed [TMMR19] – the principles remain the same: An abstraction function $\alpha : C \to A$ maps a set of concrete cache states $c \in C$ to an abstract, possibly approximative representation $a \in A$. The invariants are established by computing fixed points on the CFG location in the abstract domain.

7.1.2.1 Concrete Semantics

Traditional cache analysis, with CFG $G = (V, E, I)$, uses AI as follows [FW99]. Let M be the set of all cache blocks in the program, and *env* describe a *concrete cache state* in which it maps each cache set $f_i \in F$ to its concrete state $s \in S$ as

$$env : F \to S. \tag{7.4}$$

Further, let the *concrete set state* s capture which memory block is currently being stored in each line L in this set as

$$s : L \to M', \tag{7.5}$$
$$M' := M \cup \{\square\}, \tag{7.6}$$

with $\square$ indicating that no address is stored, i.e., an empty line. In this model, the lines also indicate the age of the contained blocks, that is, line l_1 holds the youngest block, and l_A the oldest.

The concrete semantics $f : C \to C$ is given by the change of state of *env* when an address m is accessed, which is formalized by the *concrete set state update* and *concrete cache state update* functions. The details are not relevant here, since they merely model the used cache replacement policy. For example, for an LRU cache, the update function models that $block(m)$ is the youngest block in the set (i.e., $s(l_1) = m$), and that all blocks there were younger than m before this access are aging [FW99].

7.1.2.2 Abstract Semantics

In the collecting semantics, each program point l is mapped to the powerset of possible concrete cache states as $Coll : l \to \mathcal{P}_l(F \to S)$. Accordingly, in the abstract domain we have $Abs : l \to F \to \mathcal{P}_l(S)$, where $\mathcal{P}_l$ is constructed by the *abstract set state*

$$\hat{s} : L \to 2^{M'}, \tag{7.7}$$

which maps the lines in one cache set to sets of possibly held memory blocks. It further holds

$$\forall l_a, l_b \in L.\ \forall m \in M.\ m \in (\hat{s}(l_a) \cup \hat{s}(l_b)) \Rightarrow l_a = l_b, \tag{7.8}$$

which means that a memory block m can occur at most once in an abstract set state.

The goal is to identify three types of invariants on the abstract set states, summarized in Table 7.1. Each of these invariants requires a separate analysis, and thus an individual instance of the AI framework with respective domains and semantics.

The first is called *MUST* analysis, and produces assertions on cache hits (AH - always hit) at a given program location. The second one is *MAY* analysis, and produces assertions on cache misses (AM - always miss) at a given program location. Last but not least, there is *PERS* (persistence) analysis, which tries to establish which blocks are never evicted once they are loaded (FM - first miss). We now look into them in more detail.

MUST Analysis. This analysis is designed to find guaranteed cache hits. That is, for each m in the set under analysis, if $m \in \hat{s}(l_i), i = 1, \ldots, A$ at a program location, then m must also be in the concrete set state. Towards that, the abstract semantics are defined as follows. For LRU caches, the LUB/JOIN function $LUB(\hat{s}_1, \hat{s}_2) = \hat{s}$ is defined for each line l_x in the cache

Table 7.1: Invariants on cache states.

invariant	meaning	analysis
always-miss	L-block is never in cache when accessed	MAY analysis
always-hit	L-block is always in cache when accessed	MUST analysis
no-evict/first-miss	L-block is never evicted once loaded/always missed on first access	PERS analysis
unclassified	any other case	–

(and therefore for each age x) according to

$$\hat{s}(l_x) = \{m \mid \exists l_a.l_b \text{ with } m \in \hat{s}_1(l_a) \wedge m \in \hat{s}_2(l_b) \text{ and } x = max(a,b)\}. \tag{7.9}$$

In words, the elements in the joined state are the intersection of those in the inputs, and the new age of each element is the maximum age among the inputs.

MAY analysis. This analysis is designed to find guaranteed cache misses. Towards that, only the LUB changes; for LRU caches it is defined as

$$\hat{s}(l_x) = \{m \mid \exists l_a, l_b \text{ with } m \in \hat{s}_1(l_a), m \in \hat{s}_2(l_b) \text{ and } x = min(a,b)\} \tag{7.10}$$
$$\cup \{m \mid m \in \hat{s}_1(l_x) \wedge \nexists l_a \text{ with } m \in \hat{s}_2(l_a)\} \tag{7.11}$$
$$\cup \{m \mid m \in \hat{s}_2(l_x) \wedge \nexists l_a \text{ with } m \in \hat{s}_1(l_a)\}. \tag{7.12}$$

In words, the elements in the joined state are the union of the elements in the inputs, and the new age of each element is the minimum age among the inputs.

Remark: Note that with the proposed abstract semantics, the overapproximation in both MUST and MAY analysis only comes from the JOIN function (it discards the relations between the elements). Precise alternatives have been proposed, see [TMMR19], but are not discussed here.

Cache analysis for loops. Loop-like constructs have the effect that some BBs are executed multiple times. This gives rise to the possibility that cache accesses during the first loop iteration result in a miss, whereas in subsequent iterations the required L-blocks may already be loaded, resulting in a hit. Neither MAY nor MUST analysis can classify such L-blocks in subsequent iterations.

There are at least three notions in literature to classify such repeated accesses [BC08]:

1. **first-miss:** In a program trace, accessing an L-block can only result in one miss globally. Eviction is possible if the L-block is re-loaded incidentally before it is accessed another time, by accessing another L-block that shares the same cache block.

2. **no-evict:** A miss at some point in the trace implies that for all later points in the trace the L-block resides in the cache set.

3. **first-access:** The L-block was never accessed earlier in the trace, and therefore it must result in a miss. This does not classify consecutive accesses.

None of them is dominant w.r.t. precision, and there are also different strategies for the first memory reference, i.e., different assumptions on the classification of the first access itself.

For example, the initial approach was to unpeel the first iteration of loops, to handle first-miss with the previous two analyses, However, the first access does not necessarily happen during the first loop iteration, leaving L-blocks unclassified [AFMW96]. A later approach was called *persistence* (PERS) analysis, and used a separate abstract domain with an element that indicates that a cache block was loaded, but then evicted. Therefore, this is closer to the second notion.

The details of the analysis methods are not discussed here, since they are not relevant for the following experiments. However, it should be noted that AI can be conducted on partial traces, i.e., loops can be analyzed in isolation, disregarding preceding and subsequent code. Some more advanced cache analyses for loops exploit that fact into a more precise and generic solution, as in OTAWA [BCRS10]. They identify the widest-possible loop scope in which the invariant holds, referred to as "multi-level persistence analysis" [BC08]. When entering a nested loop, the abstract cache state is reset, such that only the evictions/conflicts in the current loop are considered. When the loop is left, the result is merged with the abstract cache state when the loop was entered. This enables a multi-level persistence analysis on the fly, without unrolling. On the downside, this method does not support irreducible loops, where it leads to unsafe results [uC18b]. Therefore, all methods have shortcomings.

In the general case, each cache state invariant for loops is associated with a scope, which, in nested loops, can be any level in the nesting. Consequently, and unlike L-blocks classified as always-hit and always-miss, those L-blocks classified as first-miss/no-evict must be linked to path analysis. The traditionally used ILP system is extended to increment the timing at most once for all the visits of the associated scope. The complexity increase is in the worst case only linear with the number of instructions in the program, and thus no longer critical for analysis.

Unclassified Blocks. For some L-blocks none of the above invariants hold true, for one of two reasons. Either the access pattern of the L-block truly does not satisfy the MUST/-MAY/PERS definitions, or the analysis is too imprecise (incomplete) to deduce the precise access pattern. In both cases such are usually assumed to be misses, and subsequently not modeled [WEE+08, BCRS10]. Some methods have been proposed to classify some of these L-blocks with more precise analysis methods, as in [CR13], but WCET analyzers rarely implement such refinements.

7.1.3 Problem Setting and Challenges

First, to leverage the precision of source code analysis for a tighter WCET estimate, the timing behavior of the caches should be made visible to the source code analyzer, which implies a back-annotation of a cache model into the source.

The prerequisites for our cache model are as follows:

- **Cache:** We consider an A-way set-associative *instruction* cache, since they are harder to model than data caches, due to their dependency on flow differences.

- **Policy:** We consider the LRU replacement policy in the following, because LRU is a near-optimal caching policy, often used in practice [STW92], and because it has been shown to be the most amenable caching policy for verification [RGBW07]. In principle, our approach could also model the other replacement policies, yet with different analysis complexity. The only significant deviation would be that the invariant computation with standard AI tools in Section 7.3 would not work, since the JOIN function would

not be appropriate. Other than that, we expect FIFO and RR policies to have less complex models, whereas PLRU would be more complex.

- **Mapping:** We assume the instruction-to-source mapping has already been established, as proposed in previous chapters, and that it satisfies the following properties.
  - The mapping must not imply any execution order of the BBs that is infeasible in the actual binary execution. This forbids that cache accesses can rejuvenate the wrong cache blocks, which would lead to unsound differences in the encountered evictions between binary and source model. Nevertheless, the mapping may still drop some ordering relations if a binary flow cannot be represented in the source.
  - The mapping must preserve or overapproximate execution counts observable in the binary, since otherwise the source model might fail to capture feasible evictions, possibly leading to underestimation.

  Both of these properties are satisfied with the mapping strategy from Chapter 6.

Challenges. The requirements to the previously computed mapping are necessary in practice to enable a mapping in the first place, but unfortunately rather mild for cache analysis, posing a number of challenges towards computing a sound and precise source-level model.

- Flow differences imply approximations, which can make it impossible to precisely model the original cache behavior of the binary control flow in the source. While we could always resort to pessimism and assume that such approximations leave their accessed cache sets as unknown, this would quickly result in an unacceptable overestimation. Like for the mapping itself, decompilation would be an obvious workaround, yet this is not considered here since it is tedious and unreliable in practice, and in theory always available as backup approach.
- When multiple binary BBs map to the same source BB, then their possible ordering in the binary might not be representable in the source code, but it needs to be accounted for in the cache model. Otherwise, the modeled cache behavior would differ from the actual behavior. This is challenging when entire binary subflows are mapped to a single source BB, since we need to work out all the possible cache state transitions.
- For WCET analysis, it is essential to avoid modeling any behavior that is infeasible on the binary *and* leads to an underestimation of execution time. Specifically, this implies that we must not model cache accesses in the source which are not feasible in the binary ("spurious accesses"). For example, consider the case where the mapping adds more cache accesses than feasible (due to dominator lumping into loop headers), then this might rejuvenate a cache item under LRU policy such that it does not get evicted, although it would in reality. As a consequence, the analysis may conclude a cache hit when the item is indeed accessed again, whereas in reality only a miss is possible. Clearly, this situation can cause underestimation.
- Lastly, and perhaps most important for the viability of source-level analysis, we need to address the scalability problem that arises when cache states are modeled exhaustively. We should expect an exponential increase in computational complexity [Wil04], which would clearly not scale well. An efficient encoding of the cache model or source transformations like introduced in Chapter 5 are insufficient to mitigate such an increase in complexity. We therefore have to develop other means to keep source-level WCET analysis tractable.

7.2 A Source-Level Cache Model

We now introduce a source-level model for instruction caches. Our goal is to annotate each source BB not only with the time of the execution of the instruction, but also with the time required to fetch the instructions from the cache or backing memory, such that all possible executions of the binary are represented in the model.

Towards that, memory accesses must be tracked along with the source statements, together with the states of their associated cache sets. For the remainder of this chapter and without loss of generality, we assume that the memory addresses $addr(v_{i,j})$ for all L-blocks $v_{i,j} \in \mathcal{L}_i$ of a BB v_i are known. This is usually the case for embedded software with static linking [WEE+08]; exceptions are discussed in Section 8.5.

The differences between source and binary CFGs are the main concern for modeling the cache accesses soundly. In absence of such differences, the model is straightforward. We therefore focus on these differences, starting by how to detect them, followed by a model to represent them at source level in the presence of such differences.

7.2.1 Detecting Flow Differences

As explained earlier, flow differences can make it impossible to annotate the caching behavior in source precisely, and annotations can lead to underestimation if execution count and order cannot be preserved. Consider the example shown in Fig. 7.2. Four binary BBs map to the same source BB, thus their precise flow cannot be represented in the source code. Additionally, nodes v_2 and v_3 do not necessarily execute. In order to model their timing and impact on the cache state soundly, the source model has to allow accesses according to any possible execution in the binary flow, and has to prevent for possibly spurious execution in the model, to avoid causing spurious cache hits downstream (this case rarely occurs in practice, but is contained, inter alia, in the *adpcm* benchmark later on).

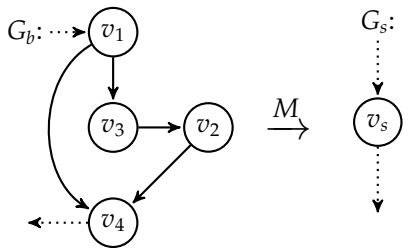


Figure 7.2: Flow difference in mapping.

The first task towards handling such flow differences is to detect them. Given the requirements to the mapper, it follows that flow differences cannot span multiple source BBs. If they were, preservation of execution order or count would be violated by the mapping. Therefore, a flow difference is present if one of the following properties holds true. Given the mapping M and one source BB v_s, let $V_b = \{v \mid v \in G_b \wedge M(v) = v_s\}$, then:

1. There exists a flow difference for v_s if $\exists v \in V_b \wedge f(v) < f(v_s)$. This property is evaluated using the metadata from the mapping. In our case, we compare the mapping at different stages in the mapping pipeline (see Fig. 6.4 on page 126). All binary BBs which are not in the map after the precise stages (e.g., control dependency mapping) are known to be possibly overapproximated.

2. There exists a flow difference for v_s if $|V_b| > 1$, i.e., if multiple binary BBs map to the same source BB. Note that this is a conservative condition, since straight-line lumping (see Section 6.4.5) can be seen as a precise mapping. However, we handle such special cases as part of the generic solution which does not loose any precision for such cases.

Since flow differences are delimited to individual source BBs, it suffices to describe the source-level model for a single source BB.

7.2.2 Ontology of the Source-Level Cache Model

The source cache model defines how to represent cache access of the instructions mapped to a single source BB. It covers the most general case, which is the handling of flow differences. All simpler cases naturally fall into this case, and are thus covered in this unified model.

Figure 7.3 shows the general ontology for the cache model. Each source BB is associated with one *source block modeling function*, which contains exactly one *flow difference*. A flow difference, in turn, consists of one or more binary BBs (those that have been mapped to the source BB), and a set of flow constraints between them. For example, their order of execution, their mutual exclusion, and possibly different execution frequencies. Last but not least, each binary BB is decomposed into one or more L-block accesses, each of which may result in a cache hit or miss, which is determined by the cache state and decided later during analysis.

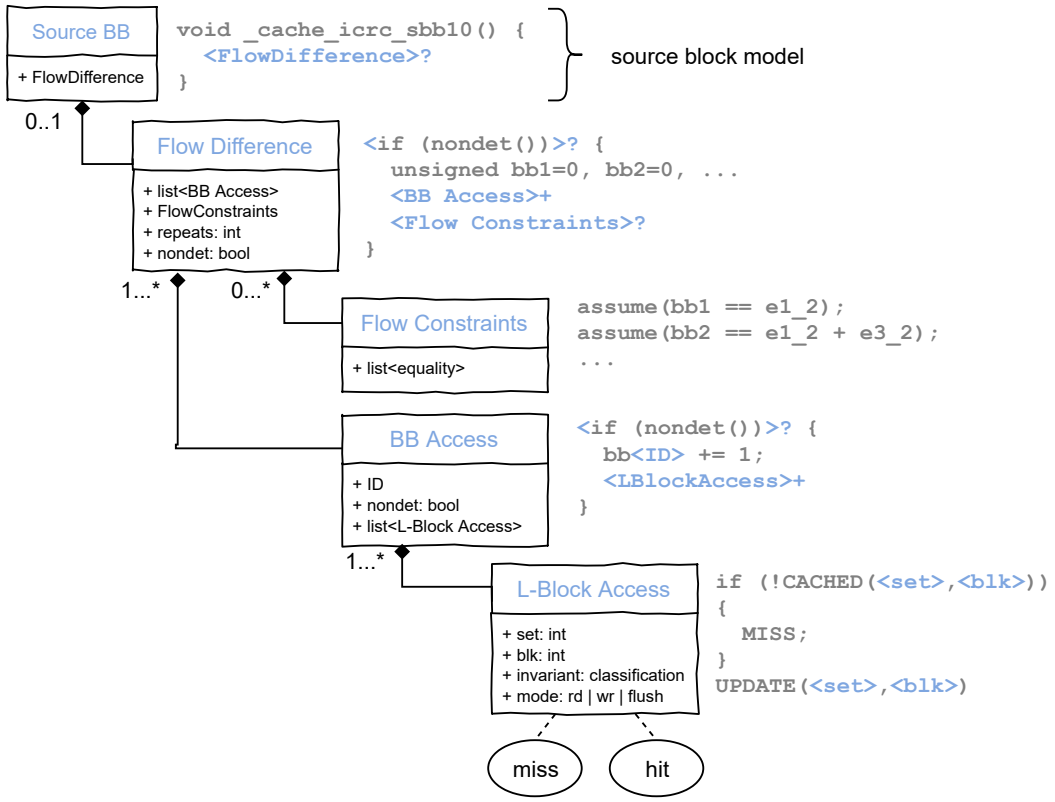


Figure 7.3: Ontology of the source cache model.

Next, we describe the cache model from the bottom up in this ontology, starting with the representation of a single L-block access.

7.2.3 Modeling L-block Access

Consider an L-block starting at address m, with block `blk` $= block(m)$ and `set` $= set(m)$ as defined in Eqs. (7.1) and (7.2). To model its cache access, we define the *cache model functions*:

1. `CACHED(set,blk)`: returns *true* if `blk` is present in `set`, otherwise *false*.
2. `UPDATE(set,blk)`: updates the state of `set` when `blk` is accessed. Models evictions, if any, according to the replacement policy.
3. `FLUSH(set,blk)`: evicts `blk` from `set`. Required for first-miss model.
4. `MISS`: increments the time variable by the miss penalty.

For each L-block access, we can select one of the three modes *read*, *write* or *flush*. For instruction caches, we only require support for *read* and *flush*, whereas the latter one will be used later to approximate some L-block accesses.

The source model for a single L-block is then defined as follows:

```
1 if (!CACHED(set,blk)) {
2   MISS;
3   UPDATE(set,blk);
4 }
```

If some a-priori knowledge is available, this L-block model can be simplified. For example, in case of a known "always-hit", we can drop lines 1 and 2. The call to `UPDATE`, however, must be kept to model the effect on other L-blocks in the same cache set, unless their states are already known, as well.

7.2.4 Modeling Binary BB Access

Each binary BB consists of one or more L-blocks. By the definition of a basic block, the contained L-blocks always execute together and in the same statically known sequence. Consequently, modeling the cache effects of a single binary BB is a concatenation of L-block accesses, given by iterating over the L-blocks in instruction order.

7.2.5 Modeling a Flow Difference

For each source BB which has binary BBs mapped to it, we capture the cache effects as a flow difference, since this is the generic case. We can distinguish between two fundamental cases created by the mapping:

1. Exactly one binary BB maps to the source BB. Technically this may or may not be a flow difference according to the conditions in Section 7.2.1. Since there is only one BB, there are no flow constraints.

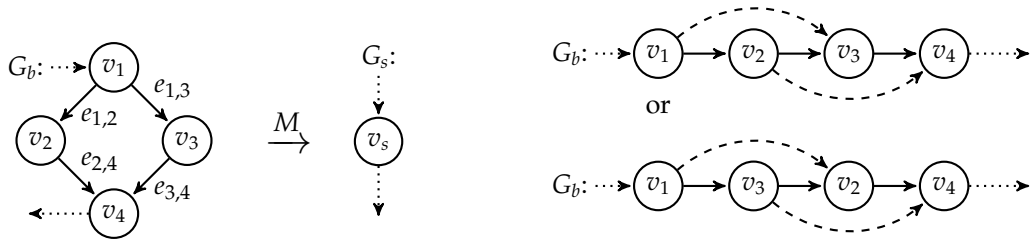
2. More than one binary BB maps the source BB.

   a) First, we get rid of the many different execution paths that might be possible through all involved binary BBs, and create a flat list of cache accesses. For example, consider the flow difference and possible linearizations depicted in Fig. 7.4. Note that both linearizations are valid w.r.t. execution order, since there is no ordering between v_2 and v_3 for being located on mutually exclusive paths. However, such a linearization would mean that we model spurious cache accesses, since only one of them can execute, and spurious accesses in the model can lead to underestimation. To solve this problem, we will consider such nodes as non-deterministically executed: The dashed edges allow skipping such nodes in the linearized sequence, and will be subject to constraints.

One of the potentially many valid linearizations can be efficiently computed using a *topological sort* on the involved binary BBs. Note that unmatched binary loops are therefore currently not supported, but can be handled in principle by cutting back-edges and inserting loop statements into the model.

b) Give the linearization, we iterate over the list binary BBs, and model each one as discussed before. Each binary BB which executes only conditionally (identified by post-dominator analysis) is marked as non-deterministically executed by wrapping it into `if (nondet()){...}`, thereby avoiding underestimation. This effectively creates the dashed edges in Fig. 7.4.

c) Last but not least, to exclude infeasible paths, we compute flow constraints for the dashed edges similar to the IPET approach in [LM97]. As a result, the source analysis implements a flat list of accesses, yet it considers all possible paths through from the original flow.



(a) flow difference **(b)** linearized acccess

Figure 7.4: Flow difference and possible linearizations.

Note that this flow difference handling improves over the previous chapter. Whereas with the earlier lumping technique we assumed that binary BBs always execute, here we only increment the time if the constraints allow their execution.

The source model of the flow difference from Fig. 7.4 is shown in Fig. 7.5. Note that to enable the flow constraints (line 24), the edge variables (line 3) remain uninitialized, which (for *cbmc*) creates free variables to be chosen by the source analyzer. In this example, the constraints therefore ensure that the binary BBs wrapped in the nondet functions are picked in mutual exclusion, and that exactly one of them gets picked at all times.

Last but not least, dominator lumping in the mapper is still allowed and may lead to an overapproximation of the execution count, which may cause spurious cache accesses in the model. As a consequence, we must account for the real execution, as well. Subflows whose execution count has been overapproximated are therefore flagged as non-deterministically conditionally executed. The `nondet` in Fig. 7.5 line 4 accounts for the fact that this particular subflow was lumped into another binary BB, and therefore its execution might not actually be feasible. It ensures that the analysis considers *not* executing this subflow, to capture the fact that the cache might not have been accessed and thereby avoid underestimation.

Lines 24 through 30, together with the `nondets` in lines 10 and 15 encode the flow constraints between all the binary BBs involved in this subflow. Similarly to our analysis in earlier chapters, the `assume` statement is recognized by the source analyzers and cuts all execution paths which do not fulfill these given predicates, therefore forbidding the analyzer to consider any path that violates the flow constraints.

A necessary condition for correctness of the linearization is that all involved binary BBs must form a *weakly connected* subgraph, i.e., there must be a path between every pair of nodes

```
1  void _cache_icrc_sb10(void) {
2    unsigned bb1=0, bb2=0, bb3=0, bb4=0; /* BB count */
3    unsigned e1_2, e1_3, e3_4, e2_4; /* edge count */
4    if (_nondet()) { /* linearized access: */
5      { /* bb1: */
6        bb1 += 1;
7        TIC(2); /* insn time of bb1 */
8        if (!CACHED(2,33568)) { MISS; } UPDATE(2,33568); /* set 2, l-block 8324 */
9      }
10     if (nondet()) { /* bb2: */
11       bb2 += 1;
12       TIC(4); /* insn time of bb2 */
13       if (!CACHED(4,33600)) { MISS; } UPDATE(4,33600); /* set 4, l-block 8340 */
14     }
15     if (nondet()) { /* bb3: */
16       bb3 += 1;
17       TIC(1); /* insn time of bb3 */
18       if (!CACHED(4,33600)) { MISS; } UPDATE(4,33600); /* set 4, l-block 8344 */
19     }
20     { /* bb4: */
21       bb4 += 1;
22       ...
23     }
24     /* flow constraints: */
25     assume(bb1 == e1_2 + e1_3); // bb1 out
26     assume(bb2 == e1_2); // bb2 in
27     assume(bb2 == e2_4); // bb2 out
28     assume(bb3 == e1_3); // bb3 in
29     assume(bb3 == e3_4); // bb3 out
30     assume(bb4 == e2_4 + e3_4); // bb4 in
31   }
32 }
```

Figure 7.5: Example of source cache model in the presence of flow differences.

when disregarding the edge direction. In other words, the mapping must not have "holes", since otherwise the model would lack cache accesses. This condition is trivially true as a result of our requirements to the mapper.

Last but not least, it is also possible that binary BBs with different execution frequencies have been lumped into the same source BB. These cases are handled by a combination of flow constraints and insertion of loop statements, but not further discussed here.

Alternatives. Flow differences could also be modeled by decompilation (thus, mimicking the binary flow with equivalent source statements). The resulting complexity would however be similar to the proposed approach, unless the decompilation goes beyond structure, and carries over the specific semantics of the assembly. Another solution could be an overapproximation by assuming that all BBs might execute, which effectively means dropping the constraints as proposed above, and thus even decreasing complexity.

7.2.6 Encoding the Cache Model Functions

The details of the cache model functions (UPDATE, FLUSH and CACHED) have not been discussed so far, because they are orthogonal to the proposed ontology. There are at least two different ways of implementation, which we refer to as *encoding*. The most efficient way depends on the source-level analyzer being used. We have experimented with two different encodings:

1. *set-wise:* An explicit, close-to-hardware model. The model consists of one array CSET for each cache set, and its elements indicate memory blocks being in the cache. Further, let

their order reflect their last access time ("age"). For each potential memory access, we update the contents and ordering of CSET to reflect the age of the blocks in the set, and remove a block from the array if it gets evicted. The amount of variables grows linearly with the cache size.

2. *block-wise:* In analogy to age-based analysis in Section 7.1.1.1, we spend one variable for each cache block (not L-block!). The amount of variables grows linearly with program size (as given by the number of instructions in the text segment).

A full example of block-wise and set-wise encodings is given in Appendix B. We found that the block-wise encoding scales better with our chosen source analyzers, and therefore use this encoding in the experiments. Furthermore, an age-based model allows to couple binary and source analyses directly, which we exploit next, when investigating options to prevent complexity explosion.

7.3 Preventing Complexity Explosion

The proposed model facilitates a maximally precise annotation of the caching behavior and timing to the source code w.r.t. the mapping. While this would allow for a precise source-level WCET estimate and does not suffer from any of the difficulties in binary-level analysis (cache analysis for loops, call context separation, etc.), the source now contains both microarchitectural and logical information. Asking any source analysis method to determine the value of the time variable and thus to estimate the WCET, means that it must handle the additional complexity incurred by the microarchitecture, on top of the already complex path analysis. This increase in complexity is known to be exponential [Wil04], and thus a state-space explosion must be expected. Unlike in earlier chapters, source code transformations are not considered here (but could still be applied), since they had not proven to be effective enough to counter an exponential complexity increase.

To address the state-space explosion, we can identify invariants on the microarchitectural state at each program point, specifically on the cache state before each access is modeled. If, for example, we could infer that at a certain program location, an access is a guaranteed hit, then we could drop the expensive cache model at this location. This is in analogy to MUST/MAY/PERS analysis from the traditional WCET estimation approach [WEE+08], where all cache accesses are classified (see Tab. 7.1) and subsequently most of them removed from the model, realizing an effective separation between microarchitectural analysis and path analysis.

For our time- and cache-annotated source code, we have two options to identify such microarchitectural invariants:

1. *Invariants from binary:* Perform traditional cache analysis on the binary, and only annotate the timing of the cache accesses to the source code, according to their classification. The functions UPDATE/FLUSH/CACHED are therefore never expressed or annotated in the source code, which results in a similar scalability than in previous chapters.

2. *Invariants from source:* Back-annotate the full cache model and compute invariants on the cache state at source level, taking into account the precise program semantics. The source cache model can subsequently be simplified according to the invariants, similarly to the previous approach.

7.3.1 L-Block Invariants from Binary

This solution is similar to the traditional approach of WCET analysis. We perform an AI-based MUST/MAY/PERS analysis on the binary as explained in Section 7.1.1.1, which therefore separates microarchitectural analysis from path analysis done later on the source. Computing the invariants on the binary does not leverage the semantic information from the source for microarchitectural analysis, but it provides an interesting data point for our source-level analysis. When we have a precise mapping, then all improvements in the WCET estimates can be attributed to the semantic information in the source code which is otherwise lost during the compilation.

7.3.2 L-Block Invariants from Source

If we annotate the full cache model to the source code and then use source analysis methods to establish microarchitectural invariants, we could maximally exploit the semantic information available in the source. However, source-level tools are not customized for this kind of analysis, therefore we have to carefully chose an approach which can compute invariants at least equivalent to the binary-level classification.

Again, we can choose between MC, AI or DV as analysis technique. With our encoding, choosing MC or DV implies to perform one verification run for each L-block, which would increase the computational effort proportional to program size (in the address space). Since these methods are themselves of exponential complexity (see Chapter 4) and additionally would have to work on the now enlarged state space, the net reduction of computational effort is expected to be asymptotically zero in the best case.

Using AI to establish microarchitectural invariants appears to be a better approach, and equivalent to what binary-level analyzers do. However, standard source-level AI tools do not implement the same abstract domain as traditional cache analysis. For example, Frama-C implements an abstract domain of ranges. As a consequence, the specific encoding of the cache model decides what kind of invariants we can establish.

To determine the suitability of a typical range analysis like in Frama-C, we must compare the two AI operators UPDATE and JOIN between this and binary-level cache analysis. Since UPDATE is obviously equivalent and sound (it must merely model the cache policy correctly, and AI itself is also sound), we focus on the JOIN function of the three analyses in the following, which must consolidate abstract states from joining paths in the CFG.

7.3.2.1 MUST/MAY Analysis

With our proposed encoding, the JOIN function for MUST and MAY analysis of an LRU are both soundly contained in the JOIN function of a standard range-based AI domain. To prove this statement, we formalize our block-based model to a similar structure as the traditional cache analysis as follows.

Concrete Semantics. Let $\mathbb{N}_{\leq A} = \{0, 1, \ldots, A\}$. Since our model is block-centric instead of set-centric, we define a *concrete block state* as

$$a : M \to \mathbb{N}_{\leq A}, \tag{7.13}$$

where a value of A indicates that the block is not present in the cache set, and otherwise $a(m)$ yields its relative age. The *concrete set state* s_i for a cache set f_i is not explicitly modeled, but

can be computed as

$$s_i : L \to M, \text{ with} \tag{7.14}$$

$$s_i(l_x) := \begin{cases} m & \text{if } \exists m \in M \text{ s.t. } set(m) = f_i \wedge a(m) = x, \\ \square & \text{else} \end{cases} . \tag{7.15}$$

Abstract Semantics. Since we use a standard value range analysis, the *abstract block state* is defined as

$$\hat{a} : M \to [\hat{a}_{\text{low}}, \hat{a}_{\text{high}}] \text{ with } \hat{a}_{\text{low}}, \hat{a}_{\text{high}} \in \mathbb{N}_{\leq A} \text{ and } \hat{a}_{\text{low}} \leq \hat{a}_{\text{high}}.$$

However, the range $[\hat{a}_{\text{low}}, \hat{a}_{\text{high}}]$ implicitly represents a set $\{x \in \mathbb{N}_{\leq A} \mid \hat{a}_{\text{low}} \leq x \leq \hat{a}_{\text{high}}\}$, hence we continue to use the set notation for the sake of clarity. Thus, the abstract block state is defined as

$$\hat{a} : M \to 2^{\mathbb{N}_{\leq A}}, \tag{7.16}$$

and the JOIN function as

$$LUB(\hat{a}_1, \hat{a}_2) = \hat{a}, \tag{7.17}$$

$$\hat{a}(m) := \{x \mid \min(\hat{a}_1(m) \cup \hat{a}_2(m)) \leq x \leq \max(\hat{a}_1(m) \cup \hat{a}_2(m))\} . \tag{7.18}$$

The *abstract set state* of a cache set f_i (again, only tracked implicitly) can be computed as

$$\hat{s}_i : L \to 2^{M'}, \text{ with} \tag{7.19}$$

$$\hat{s}_i(l_x) := \{m \mid x \in \hat{a}(m) \wedge set(m) = f_i\} . \tag{7.20}$$

We can now show that the JOIN function of range analysis subsumes both MUST and MAY analysis as follows. Since $m \in \hat{s}(l_x)$ in the traditional analysis implies that $x \in \hat{a}(m) \wedge x < A$, we can rewrite the JOIN operation of traditional MAY analysis as

$$\hat{s}_i(l_x) = \{m \mid [a \in \hat{a}_1(m) \wedge a < A] \wedge [b \in \hat{a}_2(m) \wedge b < A] \text{ and } x = \min(a, b)\} \tag{7.21}$$

$$\cup \{m \mid [x \in \hat{a}_1(m) \wedge x < A] \wedge [\hat{a}_2(m) = A]\} \tag{7.22}$$

$$\cup \{m \mid [x \in \hat{a}_2(m) \wedge x < A] \wedge [\hat{a}_1(m) = A]\} \tag{7.23}$$

Since $y < A \wedge z = A$ implies $y < z$, the two last lines can be represented with the same minimum operator, such that

$$\hat{s}_i(l_x) = \{m \mid [a \in \hat{a}_1(m) \wedge a < A] \vee [b \in \hat{a}_2(m) \wedge b < A] \text{ and } x = \min(a, b)\} . \tag{7.24}$$

Finally, in contrast to the traditional abstract set state, this is abstract set state not unique yet, since we have a range of ages for each memory block m, as opposed to only one age in Eq. (7.8). Traditional MAY analysis chooses the minimum, thus

$$\hat{s}_i(l_x) = \{m \mid x = \min(\hat{a}_1(m) \cup \hat{a}_2(m))\} , \tag{7.25}$$

which is trivially contained as the minimum in Equations (7.18) and (7.20). In words, MAY analysis requires a JOIN to follow "min+union" semantics, and in a sound range-based analysis, the union and minimum are trivially preserved, and therefore MAY analysis is contained in the value range analysis as the minimum value of the abstract range.

Similarly, for MUST analysis we can rewrite Eq. (7.9) as

$$\hat{s}_i(l_x) = \{m \mid [a \in \hat{a}_1(m) \wedge a < A] \wedge [b \in \hat{a}_2(m) \wedge b < A] \text{ and } x = \max(a,b)\}\,. \tag{7.26}$$

Remember that we encode "absent" as A, and that this value is the maximum in the abstract domain, such that the max-operator only returns an age $< A$ if m is present in both abstract states. Thus, this follows the required "max+intersection" semantics for MUST analysis, and we can simplify to

$$\hat{s}_i(l_x) = \{m \mid x = \max(\hat{a}_1(m) \cup \hat{a}_2(m))\}\,. \tag{7.27}$$

This again is trivially contained in Equations (7.18) and (7.20) as the maximum of the range-based analysis. It therefore follows that MUST and MAY analysis are subsumed by a standard range analysis, and therefore can be performed with, for example, Frama-C's value plugin.

Obtaining PERS invariants can in principle be done with a range-based analysis, as well, since it merely establishes an "always-hit" w.r.t. to a specific scope. However, it is not straight forward because of the required scope separation, and therefore discussed separately next.

7.3.2.2 Persistence Analysis

Ideally, we want to obtain invariants equivalent to those from the multi-level persistence analysis described earlier, and therefore compute "first-miss" invariants w.r.t. the widest-possible loop scopes. This poses a challenge with standard source-level analyzers, since we have no control over the AI execution, and thus the multi-level analysis method from [BC08] cannot be followed. Additionally, the analysis must also consider inter-procedural loops. For example, in the *crc* benchmark (see Interprocedural CFG (ICFG) in Fig. 7.6) there are L-blocks in the callee (`icrc`) which are persistent w.r.t. a loop in the caller (`icrc`), i.e., the persistence invariant crosses function barriers. Ignoring such wider scopes can result in weak invariants and large overestimation of the WCET.

A naive solution would be to introduce one counter variable for each pair of L-block and scope, which is reset when the scope is entered and counts up for each miss or eviction. For example, to test for a first-miss property of one L-block within the loop "scope icrc1.bb2" (lower right in Fig. 7.6), we would need one variable for each of the scopes "icrc1.bb2" and "icrc.bb4", and choose the outermost scope in which the counter variable is zero. However, that would drastically increase the model complexity and also be pessimistic: Even if an L-block is evicted, it could be re-loaded because a different L-block sharing the same cache block is accessed earlier. Therefore, we need to register the evictions on a *cache block* basis to avoid this pessimism. Note that the proposed source model is based on cache blocks anyway, and therefore naturally enables this approach. Whether an L-block is first-miss or not then boils down to computing its cache block number, and evaluating this block's eviction counters. As a result, we could introduce one eviction counter per cache block and scope to establish persistence invariants, which would be better than the naive approach, but still adds complexity the source model in the order of $n_l \cdot m$, where n_l is the number of L-blocks contained in loops, and m the number of surrounding loops for each.

Computing "first-miss" from Eviction Events. Instead, we propose a method to obtain "first-miss" classifications for L-blocks during a source-level analysis as follows. Given a single cache block, if we know the program locations of all its potential evictions (for short, its "evictors"), we could deduce the optimal first-miss scope as the outermost scope in the

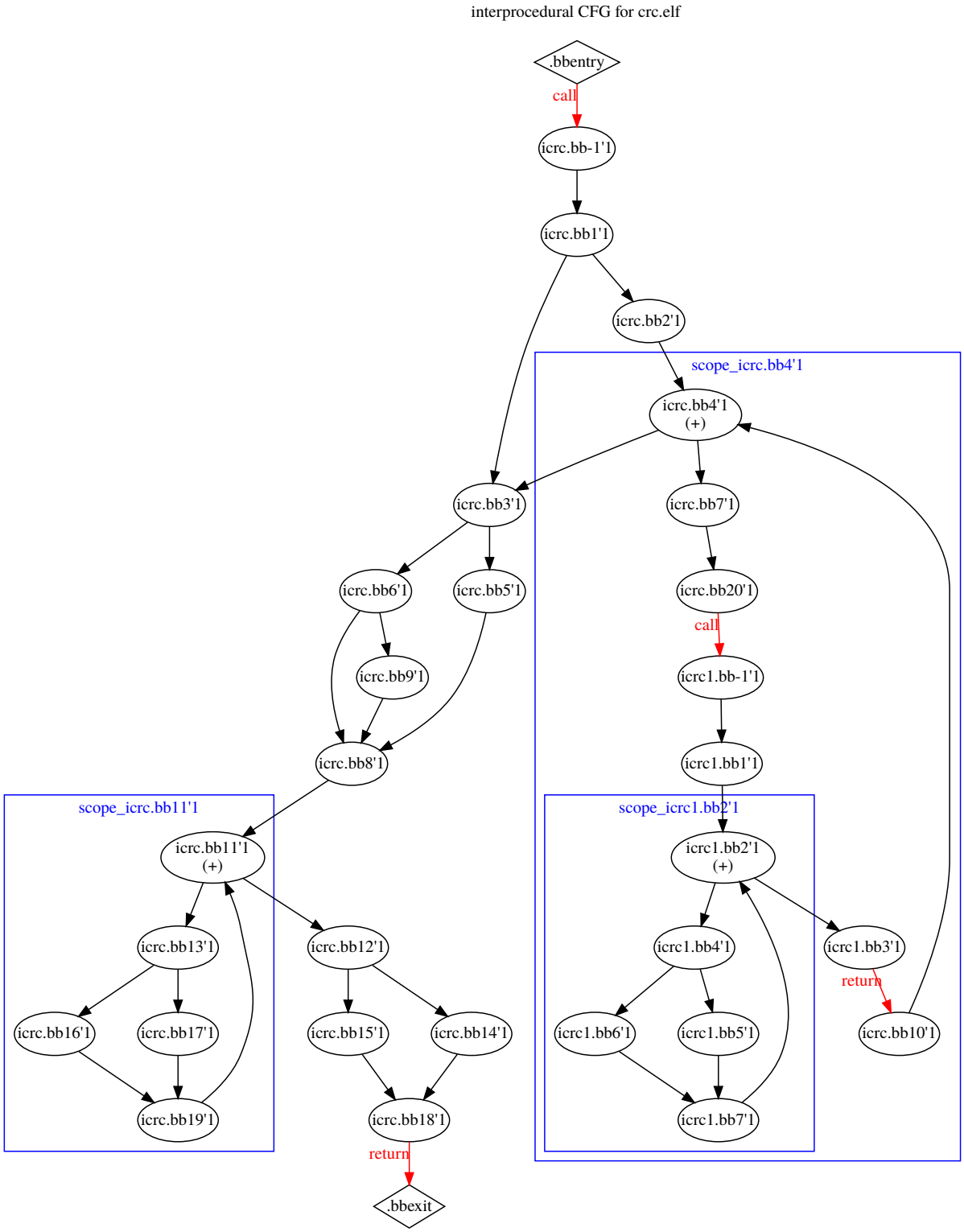


Figure 7.6: Interprocedural loops in the *crc* benchmark.

ICFG in which no eviction occurs. Towards this, we first compute the ICFG as the one in Fig. 7.6, and then perform a loop analysis as in [Hav97] to obtain an interprocedural loop nesting tree. Note that the ICFG has to separate different call contexts to avoid irreducible loops, as they would otherwise not allow to uniquely compute the loop nesting tree. Lastly, we identify the outermost interprocedural loops which does not contain evictors of the respective L-block, which are then used as the scopes for its first-miss invariant.

Obtaining Eviction Events. To obtain all possible evictors, we exploit the characteristic of Frama-C's value analysis, which tracks values precisely until the set of values exceeds a certain size, before reverting to the more familiar range representation. We use this behavior by adding to our model one additional variable for each cache block, which tracks L-blocks that caused its eviction. More generally, the evictors could also be computed without adding any overhead to the source model, by inspecting the results of the age value before and after each possible eviction. This could either be done programmatically on the results of the value analysis (yet Frama-C lacks such an interface at the moment), or on the fly, since AI iterates over the CFG, the (overapproximated) effect of each cache access can be monitored, together with the path in the call tree. This would result in similar evictor information as before, and could subsequently use the same ICFG approach to pinpoint the first-miss scopes.

As a result, this solution is able to identify at least the same first-miss invariants as state-of-the-art multi-level persistence analysis, but only uses standard source-level range analysis.

7.3.3 Approximating Unclassified L-blocks

One remaining problem in both approaches stems from L-blocks for which no invariants can be established. Since in general we cannot model and back-annotate to the source only the unclassified blocks without also modeling all the other L-blocks in the affected cache sets, we either have to model the full sets, or to approximate all unknown accesses as cache misses. We choose the latter variant, since it is the equivalent to what OTAWA is doing, and enables a direct comparison of the estimates.

7.3.4 Remaining Complexity Challenges

Once invariants on microarchitectural states have been established and the models been simplified, there still exist two sources of complexity increase in our approach, caused by microarchitectural states. Specifically, for a source-level cache model, the following two issues are remaining.

7.3.4.1 L-blocks classified "first-miss"

As mentioned earlier, to model accesses to L-blocks classified as "first-miss", they must be linked to path analysis, since the invariants only hold for specific scopes. We either have to introduce one variable to register when the first access has been made, thereby increasing the complexity of the path analysis problem, or we can overapproximate the block access by paying the miss penalty as soon as the associated scope is entered. The latter version would suffer from overestimation if the L-block is accessed only conditionally, but results in no fundamental increase in complexity. On the other hand, modeling each first-miss precisely increases the number of variables linearly with the program size, and the complexity increase is exponential (since each access may or may not result in a hit).

Solution. To counter the exponential increase in complexity, the overapproximation is required in practice, and will later be shown cause only negligible overestimation.

7.3.4.2 Exponential complexity explosion from flow differences

The handling of flow differences as proposed can cause exponential increase in complexity w.r.t. the source program. Consider the following snippet from the *crc* benchmark:

```
1 for (j=0; _cache_icrc_sb10(), j<=255;j++) {
2   icrctb[j]=icrc1(j << 8,(uchar)0);
3   ...
4 }
```

The cache model function `_cache_icrc_sb10()` is interpreted 256 times during analysis. The associated source model is similar to the one shown on Fig. 7.5, with a `nondet` enclosing the cache model functions. As a consequence, this snippet produces 2^{256} paths to be analyzed, an exponential state-space explosion compared to the original program. Note that we cannot remove the surrounding `nondet` if at least one L-block's state is possibly changed. If non-execution is feasible in the binary flow but we only consider execution, then this could cause a spurious cache hit to be experienced downstream, leading to underestimation. On the other hand, always assuming that it never executes may fail to model an L-block being loaded in the cache that might feasibly be loaded, causing a spurious miss downstream.

Solution. Once microarchitectural invariants have been established for all L-blocks in a flow difference, the source model for each such block no longer influences other L-blocks in the same set, and consequently the surrounding `nondet` can be dropped. Since modeling a spurious L-block access to a set does no longer change the set state as seen by other L-blocks in the same cache set, it is sound to overapproximate such accesses by removing the surrounding `nondet`. This that means we possibly overapproximate the WCET by one miss penalty for all contained L-blocks. Interestingly, this action alone does not produce overestimation, because a sound source-level analysis would anyway have to choose this path, since globally there no longer exists any other path which could be influenced by this decision *and* produces a worse timing globally. One exception seem to be L-blocks classified as "first-miss", if not at approximated as discussed before. However, the same reasoning applies here, such that in conclusion any flow difference with fully classified microarchitectural states can be simplified in this way.

Moreover, we can now locally compute the longest path through each flow difference, since the invariants remove any kind of timing anomalies. This does not loose any more precision, since the model checker would have to pick this path anyway. Towards this, we formulate an ILP/IPET instance as in traditional WCET analysis, and subsequently only encode the longest path without any non-determinism. The entire algorithm for the cache model is sketched in Alg.4.

In summary, the exponential increase of complexity due to microarchitectural states can be prevented by computing invariants, followed by removing non-deterministic execution and a local longest path substitution on the involved binary BBs.

7.4 Experiments

We have evaluated our source-level cache model on the known benchmark programs to identify potential shortcomings, and to obtain a comparison against traditional, binary-level WCET analysis. The experiments explore three of the options discussed before, namely:

Algorithm 4: Algorithm to encode the cache model as source code

```
Input: Mapping M, list of functions
Output: <source code>
foreach func in functions do
    foreach sbb in func.src.bbs do
        fdiff = FlowDiff()
        involved_bbs = {bbb | M(bbb) = sbb}
        subg = f.src.subgraph(involved_bbbs)
        topo = topological_order(subg) /* Sect. 7.2.5                    */
        cons = []
        foreach bbb in topo do
            cons.append(make_constraints(bbb, subg)) /* Eq.(1.4)          */
        if invariants available then
            /* find and model only longest path:                          */
            longest_path = lp_solve(involved_bbbs, cons)
            for bbb in topo do
                if bbb∈ longest_path then
                    must_execute = True
                    fdiff.add_bb(model_bbb(must_execute)) /* Sect. 7.2.4 */
        else
            /* model the full flow difference:                            */
            is_uncond = func.bin.postdom.test_dominance(bbb, topo[0])
            must_execute = not possibly_overapproximated(bbb) and is_uncond
            foreach bbb in topo do
                fdiff.add_bb(model_bbb()) /* Sect. 7.2.4                  */
            fdiff.add_cons(cons)
            fdiff.set_nondet(|involved_bbs| > 1)
        encode_sbb_model(fdiff) /* write source code, see Fig. 7.5        */
encode_globals() /* Section 7.2.6                                         */
```

1. **Analyzing the full source model.** This setup yields the maximum precision for any source-level WCET analysis, since it uses every bit of available semantic information, and includes the full microarchitectural state, as precise as it can be represented in the source code. Although, this is not expected to be useful in practice due to the expected state space explosion, it demonstrates how much WCET analysis can benefit from source semantics.

2. **Analyzing the model with binary-level microarchitectural invariants.** This setup prevents the exponential complexity increase from microarchitectural states by computing invariants of them on the binary, i.e., using the traditional cache analysis methods. This method is expected to scale well, and demonstrates how much the path analysis part of WCET analysis can benefit from source semantics.

3. **Analyzing the model with source-level microarchitectural invariants.** This setup also prevents the exponential complexity by computing microarchitectural invariants. In contrast to the previous setup, the invariants are computed at source level. Therefore, this setup is also expected to scale well, and demonstrates how much the microarchitectural analysis part of WCET analysis can benefit from source semantics.

As before, we have chosen MC for path analysis, which – since MC itself does not loose any precision – enables us to attribute all overapproximation to our models and therefore to investigate the quality of the cache model. As additional data point, we have also performed an AI-based WCET analysis, which scales better than MC due to its abstractions, and demonstrates how useful and precise this approach would be.

7.4.1 Assumed Processor and Compiler

For our experiments, we have assumed a hypothetical in-order ARM processor with an instruction cache, an ideal branch predictor, and no other performance-enhancing features. The specifications are as follows:

- ARMv5 ISA,
- 4-stage, scalar in-order RISC pipeline,
- single-level LRU instruction cache with $A = 2$, block size $b = 16$ bytes, $n = 32$ lines, total size 512 bytes and a miss penalty of $t^{\text{miss}} = 10$ clock cycles, and
- backing memory DRAM with 10 cycles access latency.

Note that cache miss penalty and access latency to DRAM are one and the same. The I-cache configuration has been empirically chosen to cause some interesting effects over all benchmarks, such that all discussed L-block classifications and eviction effects are covered.

As earlier, we have used an unmodified compiler, this time the `gcc-4.9` from the GNU ARM Embedded Toolchain 2014-Q4, based on newlib 2.1.0 and hard floats.

7.4.2 Tools and Simulation Baseline

To obtain a baseline for our source-level analysis with more complex processors, we also need to change the WCET analyzer and simulator compared to earlier chapters.

WCET analyzer. We switched to the OTAWA toolbox [BCRS10], which implements the traditional IPET/ILP approach, and models caches with the traditional analysis from Section 7.1.1.1. It also implements the multi-level persistence analysis described earlier, yielding

tight WCET estimates even for programs with loops. However, OTAWA currently does not attempt to identify any infeasible paths. While it attempts to deduce loop bounds by a source-level analysis with its tool *orange*, it occasionally fails and requires some user inputs. OTAWA was also used to obtain the binary-level invariants.

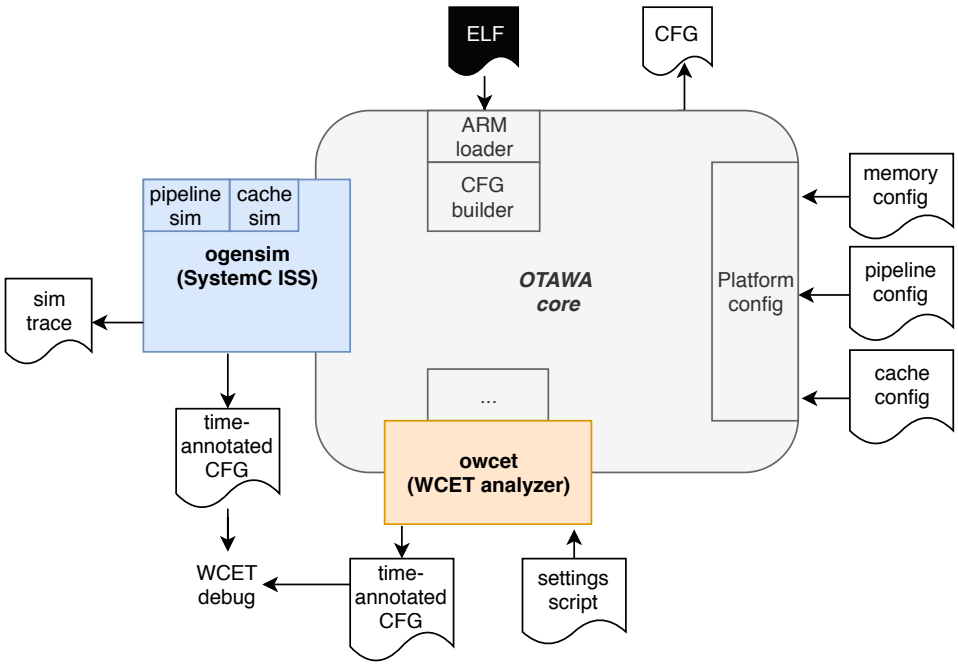


Figure 7.7: Developed setup for evaluation of advanced architectural models.

Simulator. To evaluate the tightness of all WCET estimates, we developed a generic and configurable SystemC simulator for the microarchitecture shown in Fig. 7.1, building on OTAWA's core facilities, see Fig. 7.7. The simulator performs a cycle-accurate simulation of binary executables while precisely modeling the behavior of caches, and yields both a simulation trace and a binary CFG, annotated with execution count, cache and timing events, as shown in Appendix B.4. Our simulator therefore gives us full introspection about when and where certain timing-related events occur. As a result, we are able to compare our source-level WCET estimates with both the traditional ILP+AI estimates from OTAWA and with a precise simulation, and can identify any differences or errors in the analyses. A similar tool has meanwhile been developed by the OTAWA authors, but not been made available, and further lacks a cache model [JHM+18].

7.4.3 Benchmarks

From the benchmarks introduced in Chapter 5, we have chosen a subset that covers some properties to thoroughly test our proposed cache model:

- *adpcm* contains flows such that spurious accesses in the cache model can lead to underestimation, see Section 7.2.1. It also contains flow differences between source and binary. It has multiple callsites for several functions.
- *crc* is a multi-path program with flow differences, and reasonably complex. Source-level analysis can potentially outperform binary-level analysis in precision.

- *cnt* is a small program with loops, which therefore serves as a good test for first-miss invariants.
- *fibcall* and *insertsort* are small and short programs for which we should be able to estimate a WCET on the full source-level model.
- *fdct, jfdctint* are single-path programs, where binary-level analysis should be precise.
- *matmult* is a long single-path program. It has therefore higher complexity than the previous ones.
- *ndes* has flow differences and a high complexity. Its control flow depends on bit operations. Source-level analysis can potentially outperform binary-level analysis in precision. It has multiple callsites for one function.
- *ns* is a small program with many loops, therefore a good test case for first-miss properties.
- *nsichneu* contains a large number of flow differences, due to the bad debugging information discussed in the previous chapter. It is also a very large program, testing the scalability of our approach.

Note that we distinguish between *small* and *short* programs. Whereas the first refers to the number of locations in the program (or, the size of the text segment) and therefore is proportional to cache fill level, the second one refers to the number of instructions executed. These two are not correlated in general, since a small program can be long-running due to loops. In analogy, we distinguish *large* and *long* programs.

For each benchmark we have chosen one entry function below `main`, such that the execution context is not fully defined for both binary and source-level analysis. Naturally, the simulation baseline requires an execution context, and therefore uses the known worst case inputs. Additionally, our simulator clears the caches when the function under analysis is entered, to avoid that warmed-up caches skew the results.

Excluded Benchmarks. *PapaBench* could not be included, since it contains *asm* code specific to the AVR microprocessor. We further had to omit all benchmarks which make implicit library calls, since we currently do not model their cache effects. This shortcoming does not refute the validity of our approach for these programs, and is discussed in detail in Section 8.5.

Manual Inputs. Benchmarks which could not be automatically bounded by OTAWA, namely *crc* and *fibcall*, have been bounded manually to enable a comparison between binary and source-level WCET estimates. In contrast, our source-level analysis required no manual inputs for any of the benchmarks.

Source transformations and abstractions. We did not use any of the scalability-enhancing methods from Chapter 5, as to not skew the upcoming results in the aspects of computational complexity and precision. These transformations can still be used, but our primary concern here is to evaluate the raw precision and complexity of the cache model, before we improve its scalability.

Analyzer and backend. For all experiments we remain with the bounded Model Checker *cbmc*, in the same version as before. As SMT backend we have only used *yices*, because it was consistently among the fastest solvers in our earlier experiments.

7.4.4 Results

The results are shown in Table 7.2, giving the WCET estimates for each method, and in Table 7.3, quantifying the computational complexity. Additionally, we have visualized the estimates in Fig. 7.8.

Most benchmarks could be solved during the allotted time of 1 hour, and in fact often much quicker. Source-level WCET analysis failed completely on the benchmark *ndes* due to a timeout on all model versions. Furthermore, we were not able to obtain source-level invariants for *nsichneu* and *ndes* for similar reasons. Last but not least, the full source model, only used as a reference point, was only tractable in 8 out of the 11 benchmarks, where it resulted in extremely tight WCET estimates that nearly reach the simulated value.

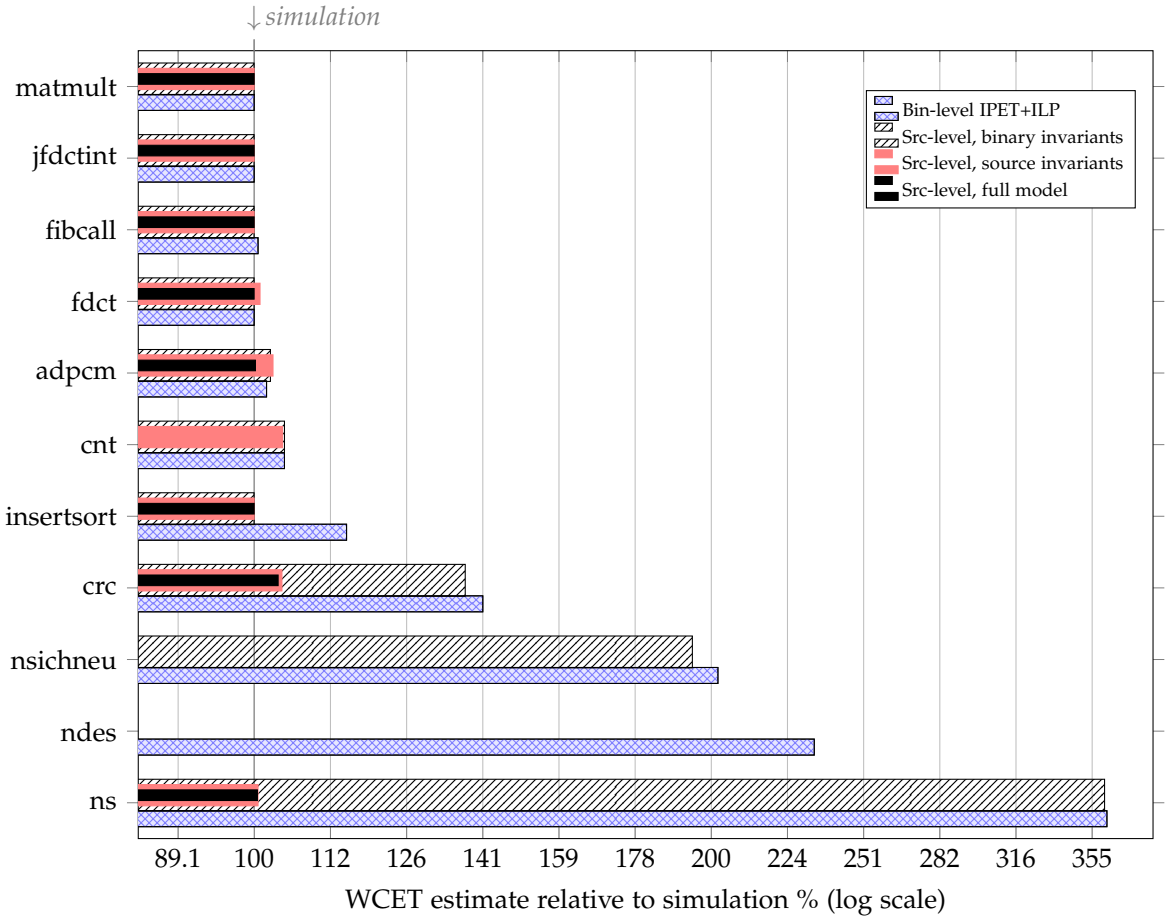


Figure 7.8: Best WCET estimates using different analysis methods.

7.5 Discussion

The experiments show that source-level analysis dominates precision in all benchmarks. As expected, the (full) source-level model including a precise annotation of cache behavior is only tractable for short programs. The computed microarchitectural invariants have kept our model from a state space explosion, and therefore the source-level analysis at a similar complexity than in previous chapters, although some precision was lost in this process as indicated by the even tighter results where a full model was tractable.

Table 7.2: Best WCET estimates using different analysis methods, timeout 1 hour.

			Sim.	Bin.-Level (ILP)		Src.-Level (MC)						Src.-Level (AI+)	
						full model		bin. invariants		src. invariants		full model	
benchmark	D?	S?	WCET≥	WCET	Δ%	WCET	Δ%	WCET	Δ%	WCET	Δ%	WCET	Δ%
adpcm	✓	×	6,502	6,632	+1.9	**6,516**	**+0.2**	6,686	+2.0	6,696	+2.9	7,090	+9.0
cnt	×	×	3,161	3,310	+4.7	timeout	–	3,311	+4.7	3,301	+4.4	**3,171**	**+0.3**
crc	✓	×	53,206	75,232	+41.3	**55,213**	**+3.7**	73,263	+37.6	55,553	+4.3	78,153	+46.8
fdct	×	✓	19,422	19,442†	≈ 0	**19,422**	**0**	19,442†	≈ 0	19,602	+0.9	**19,422**	**0**
fibcall	✓	✓	643	647	+0.6	**643**	**0**	**643**	**0**	**643**	**0**	653	+1.5
jfdctint	×	✓	18,188	18,278	≈ 0	**18,188**	**0**	18,278	≈ 0	18,198	≈ 0	**18,188**	**0**
insertsort	×	×	1,679	1,931	+15.0	**1,679**	**0**	**1,679**	**0**	**1,679**	**0**	1,689	+0.5
matmult	×	✓	335,133	**335,133**	**0**	**335,133**	**0**	**335,133**	**0**	**335,133**	**0**	375,637	+12.0
ndes	✓	×	146,515	341,717	+133.2	timeout	–	timeout	–	–	–	timeout	–
ns	×	×	21,439	77,837*	+263.0	**21,557**	**+0.5**	77,557	+261.7	21,587	+0.6	21,567	+0.5
nsichneu	✓	×	29,101	58,685	+101.6	timeout	–	**56,465**	**+94.0**	–	–	timeout	–

D=flow differences, S=single path Δ=overestimation, *=multi-level PERS analysis failed, †=bug

Table 7.3: Computational Effort for WCET estimation using different analysis methods.

	Bin.-Lvl (ILP)		Src.-Level (MC)									Src.-Level (AI+)	
			full model			bin. invariants			src. invariants			full model	
benchmark	time	mem	time	mem	steps	time	mem	steps	time	mem	steps	time	mem
adpcm	0.4	17.8	1,536.0	182.2	74,451	16.1	54.6	4,190	13.4	54.6	4,128	197.1	156.8
cnt	0.1	12.8	timeout	91.3	23,610	32.3	18.6	4,189	32.4	17.3	4,173	1.0	60.2
crc	0.1	13.5	89.0	533.7	434,220	15.9	102.4	63,831	15.3	93.2	60,291	74.2	117.1
fdct	0.1	13.5	13.9	64.1	39,526	2.5	14.8	3,969	3.0	14.8	4,002	11.9	124.5
fibcall	0.1	12.7	0.8	11.7	1,502	0.9	11.2	845	1.27	11.2	833	0.8	54.4
jfdctint	0.1	13.7	14.2	61.3	35,859	2.6	13.8	3,612	2.7	13.9	3,598	9.7	100.1
insertsort	0.1	12.9	1.0	10.1	1,703	1.0	9.0	1,073	1.3	9.1	1,049	0.9	56.4
matmult	0.1	12.9	60.5	339.1	212,836	45.2	186.9	113,769	46.0	186.9	113,765	17.4	164.9
ndes	0.2	17.0	timeout	2020.7	1,307,854	timeout	416.8	130,732	–	–	–	timeout	167.9
ns	0.1	13.1	713.9	1527.2	189,448	215.9	606.4	49,704	171.6	458.8	37,278	5.2	98.6
nsichneu	102.4	65.7	timeout	674.8	–	31.8	61.8	17,926	–	–	–	timeout	2,007.1

memory in MBytes, time in seconds

Using microarchitectural invariants computed at binary level, the source-level WCET estimate is always close to that from a traditional approach, and even gaining some precision during path analysis (see Tab. 7.3). One deviating benchmark is *adpcm*, where the traditional analysis is more precise. A closer look at this benchmark will be necessary.

Even better results were observed with microarchitectural invariants computed from the full source model, indicating that the additional information in the source code is also valuable to constrain microarchitectural analysis. Especially the benchmarks *crc* and *ns* – in both programs the control flow depends heavily on the data flow – emphasize this trend. One disadvantage is that we were not always able to compute them (in *nsichneu* and *ndes*), since the source level analyzers could not cope with the amount of complexity in the model and ran into a timeout. Only the benchmark *fdct* yielded a worse estimate using source-level invariants, which has to be investigated.

We were not able to compute a WCET estimate for *ndes*, since none of the source models were tractable. Nevertheless, this result was expected, since it was also too complex on the simple processors in earlier chapters without any source code transformations. This benchmark is revisited shortly.

The AI-based analysis (again, note that this is not pure AI, but Frama-C's extended version) on the full model also shows promising results, but does not scale better on average than a MC-based approach (see Tab. A.3 in the appendix). Specifically, it ran into timeouts for *nsichneu* and *ndes*.

In the following we inspect some results in more detail, to identify the causes for those benchmarks that showed unexpected results.

7.5.1 Errors in Binary-Level Analysis

We first revisit the case of *fdct*, where source-level invariants unexpectedly lead to less tight estimates than binary-level invariants. The reason turned out to be an error in OTAWA's persistence analysis. This program contains very long BBs, in which the tail of the BB evicts its own head. Moreover, these BBs are located inside a loop, such that every loop iteration results in a cache miss. However, OTAWA classified these blocks incorrectly as always-hit [2]. As a consequence, OTAWA produces a WCET value that, incidentally, is still safe, yet due to the local underestimation yields a tighter WCET estimate than source-level analysis with source-level invariants. Interestingly, the full source-level model yields a similar estimate, but it classifies these accesses correctly and instead gains at other places, which cannot be expressed with the known invariant classes.

The details are intricate, in fact, and confirmed by both the full source model and the cycle-accurate simulation. Each of the long BBs has a dominator prior to the loop, which falls into the same cache block, such that the very first accesses when the respective loops are entered, are indeed hits. The next seven iterations however are misses, because the long BBs evict themselves. Yet, OTAWA fails to detect this and subsequently underestimates the WCET by $2 * 7 * 10 = 140$ cycles, such that the correct estimate would have been $19,442 + 140 = 19,582$. Since we have chosen to approximate first misses in our source-level analysis (except in the full model) such that the first entering of each scope definitely causes a miss, it follows that $19,442 + 140 + 2 * 10 = 19,602$ would have been the correct source-level estimate based on the corrected binary invariants. This agrees with the estimate based on source-level

[2] This flaw in AI-based cache analysis was identified in [HJR11], but we were unaware that this had not been fixed in OTAWA.

invariants, such that we can conclude that source-level invariants are not worse than binary-level invariants in the case of *fdct*, nor in any other program.

No other benchmark was affected by OTAWA's bug. Although *jfdctint* also has long BBs, the cache analysis was not precise enough to assert any hits on those.

7.5.2 Precision

The experiments clearly show that a source-level cache model can enable a more precise WCET analysis than traditional binary-level approaches. In these experiments we mostly had precise or close-to-precise mappings, therefore they highlight the precision of the cache model, as opposed to the mapping. One notable exception was the *nsichneu* benchmark, for which we had only a very imprecise mapping, but nevertheless we obtained a tighter estimate than OTAWA. Similarly, by using MC for path analysis, no precision was lost, such that our data mainly shows the ideal precision of the cache model, as opposed to that of the chosen source analyzer. For latter aspect, consider the column "AI+", which lists the WCET estimates using Frama-C's AI, with all of its additional options (superposed states, etc.).

The data indicates that a source-level analysis is able to deduce more infeasible paths, which have a large impact on the estimate for some benchmarks, viz. *nsichneu*, *crc* and *ns*. Additionally, the imprecision of binary-level architectural invariants has become apparent in the benchmark *crc*. Although *ns* shows the same trend, this benchmark does not permit to draw this conclusion, since we had to disable PERS analysis in OTAWA because it produced an inconsistent result (labeled *bug* in Tab. 7.2).

First-miss approximation. As explained in Section 7.3, first-miss invariants have been approximated by assuming unconditional execution within their respective first-miss scopes. This avoids a state-space explosion, and looses only little precision. Upon inspection of the results, we found that in only three benchmarks the approximation had cause a slightly less tight WCET in comparison to binary-level analysis, namely in *cnt*, *fdct* and *adpcm*. The effect on all benchmarks has been quantified separately in Appendix B.3. This approximation therefore seems justified.

The general trend seems to be that source-level analysis has a precision advantage especially for benchmarks with complex data flows, where the source semantics can help to identify more infeasible paths. Our WCET estimates are often very close to the simulated value, showing that very tight estimates become possible.

7.5.3 Complexity

Computing the WCET estimate on the *full* source model, that is, combining path- and microrarchitectural analysis in one source model, is not recommended. In our benchmarks the computational effort of this full model is approximately one order of magnitude higher than using invariants, as shown in Table 7.3[3]. One extreme example was *nsichneu*, where a precise estimate would have been of high interest, due to the seemingly large overestimation in all other estimates. However, we have spent six hours in an attempt to solve it, during which neither MC nor AI+ were able to yield a WCET estimate. Therefore, source-level invariants were not available in this case. An interesting outcome in this benchmark was, that

[3] Like earlier, we consider the program steps to be a good indicator of complexity, rather than the analysis time, which is subject to possibly randomized solver-specific strategies

our source-level analysis with binary invariants had a lower analysis time than the traditional approach. Here, OTAWA has spent most of its time in loop analysis (the *orange* tool), and only about five seconds for solving the remainder (mainly IPET/ILP).

The complexity of the full model confirms that bounded MC alone is no longer a viable approach for WCET estimation for complex architectures. The additional complexity incurred by a microarchitectural states model – as here, an exponential increase in the state space per cache set [Wil04] – clearly shows an increase in analysis, and at some point is beyond what the solver backends can currently handle.

As a solution, we have proposed to compute microarchitectural invariants, either at binary- or source-level. They effectively avoid a state space explosion during path analysis. Table 7.3 shows that all benchmarks could be reduced to a complexity comparable to our simpler processors, yet while still yielding competitive WCET estimates.

7.5.3.1 Applying Abstractions

One remaining concern is the *ndes* benchmark, which could not be solved even with invariants. Note that in none of the benchmarks we have so far applied our source-level transformations proposed in Section 5.2, to keep the results (especially the complexity metrics in Table 7.3) free from additional effects not caused by the cache model. This benchmark was, however, also not solvable on a simple processor, therefore we must now consider our source transformations again.

After applying loop abstractions, MC was able to compute a WCET estimate based on binary-level invariants within three seconds. Surprisingly, the result (386,202 cycles) was worse than binary-level analysis. The reason is our lacking separation of call contexts during the annotation, which is discussed next, together with the selection of invariants.

7.5.4 Selecting and Annotating Invariants

Source-level invariants should be preferred over binary-level invariants, since they can exploit flow constraints visible in the source. If the source-level analysis is precise enough, in principle we can always compute invariants at least as strong as those at binary-level. Only those flow differences which lead to an overapproximation of the execution count may weaken them, but these can be avoided as explained later.

Obtaining binary-level invariants. We have taken our invariants from the OTAWA analyzer, although these could be obtained by a standalone tool, if a WCET analyzer is lacking. Since traditional cache analysis is based on AI with specific abstract domains, these invariants can be obtained quickly also for large programs. As a disadvantage, they are agnostic to data-dependent flow constraints beyond loop bounds.

Although it has been pointed out that the abstractions used in traditional cache analysis have often little overapproximation [Wil04] and this is true for most programs in our experiments, we also have two cases, *ns* (modulo the PERS inconsistency) and *crc*, where standard cache analysis performs poorly. When feasible control flows heavily depend on the data, then large overestimation can be expected.

On the other hand, source-level invariants can be stronger if the source analyzer is at least somewhat path sensitive, but they come at a cost.

Obtaining source-level invariants. Obtaining microarchitectural invariants at source level did itself take some time, since source analyzers must be (to a high degree) path sensitive. We

use Frama-C as to compute them, while taking advantage of its superposed states. Invariants were taken from the runs that resulted in the AI+ WCET estimates shown in Tab. 7.2, i.e., at the least possible precision level that also yielded a WCET estimate. Note that this is an arbitrary decision, only meant to show how precise source-level invariants can be. In practice, the precision of the WCET estimate and therefore of the source invariants can be modulated by the analysis settings. It is therefore always possible reduce the precision of the source analysis and obtain weaker invariants in less time, up to the point where all path-sensitivity is lost, and they become equal to binary-level invariants.

The observation that source-level invariants are superior to binary-level invariants therefore might change with different tools and precision settings, trading precision against processing time. Nevertheless, our experiments demonstrate that source-level approaches give a choice to select this precision, and that for maximum precision a source-level approach is beneficial.

7.5.5 Revisiting Flow Differences

This section discusses two more approaches to reduce the overestimation caused by flow differences, as observed in *nsichneu* and *adpcm*.

Reducing the number of flow differences. The *nsichneu* benchmark consists of a single function with 378 nested if statements, wrapped in a loop, which has a lot of imprecise debug information (see Chapter 6). As a result, the mapping is rather imprecise, causing numerous flow differences. The differences are mostly two siblings that follow a decision node, which both cannot be precisely annotated to the source code. In such cases, they are both lumped to the preceding decision node and become an instance of the flow difference model proposed earlier. Note that these flow differences preserve the execution count of the siblings, and our flow difference constraints ensure that their mutual exclusion is modeled. However, the model carries no further semantic information about when which branch is taken, therefore it approaches the ILP/IPET solution.

In an attempt to improve the debugging information and thus the mapping, we have reformatted the source code of *nsichneu*, such that debug information and basic block boundaries agree. During this process, we only changed the line breaks in the source code. The resulting mapping is almost fully precise, thus leaving almost no flow differences. The full source model, however, remains intractable with all analysis methods. Using binary-level invariants, we now obtained a WCET estimate of 29,436 cycles, i.e., only +1.1% off the simulated value, vastly outperforming binary-level WCET analysis. The analysis time in that case remained the same as before.

This demonstrates the significance of source code layout on the WCET estimate.

Reducing overestimation in flow differences. The *adpcm* benchmark has one flow difference where the execution count is increased in the source model, resulting in an overestimation compared to a binary-level analyzer. The binary BB b_6 in Fig. 7.9 was not precisely mapped, and as a consequence had been lumped into b_4, which however is a loop header and executes more often. Therefore, the execution time for b_6 is overestimated.

This is one of several similar instances, which caused the source-level analysis with binary-level invariants to be worse than the binary-level analysis in *adpcm*, together with the first-miss approximation. On an architecture without caches (such as in the previous chapter), the mapper could have violated execution order and lumped it into b_8. But with caches this is no longer sound, since it would create a spurious cache access for b_6 after b_8, if edge $c4.5$ was

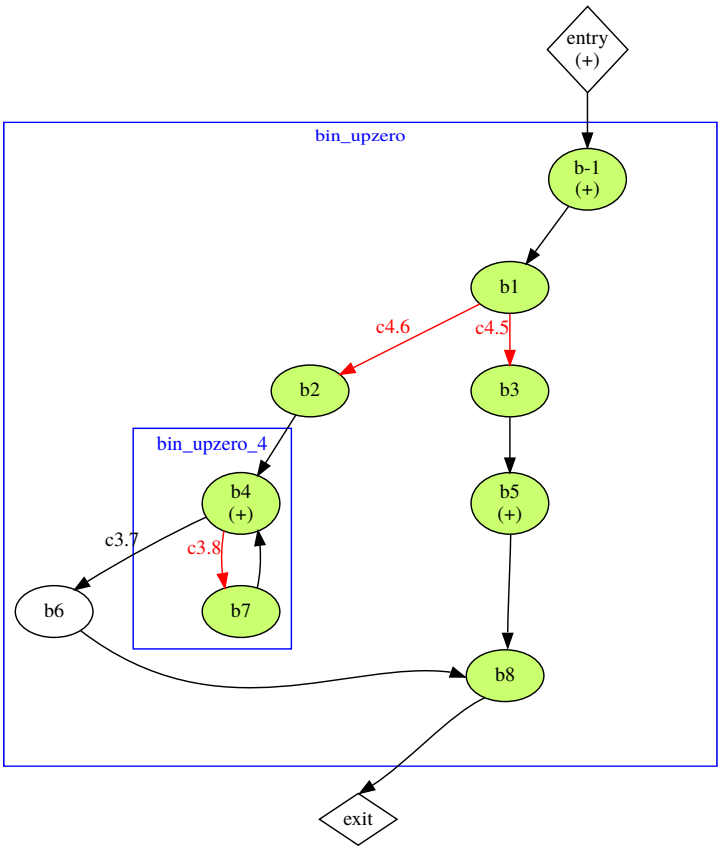


Figure 7.9: Execution count overestimation in *adpcm*.

taken, and thus possibly underestimation (see beginning of this Chapter). There exist at least four ways to prevent overestimation in this case:

1. One way solve this issue is to postprocess the mapping and "move" the b_6 into b_8 where the count is preserved, and then declare its execution nondeterministic, to account for the possibility that b_6 is not reached if edge $c4.5$ is taken. We then only pay the overestimation once when $c4.5$ is indeed taken.

2. Another option is to constrain the execution count in the flow difference to have b_6 be executed at most once when the loop of b_4 is left, but this would increase model complexity.

3. A third option presents itself after invariants have been computed. Since the cache accesses are no longer influencing each other when invariants are used, we could indeed "move" the cost for b_6 to b_8, also paying the overestimation only once when $c4.5$ is taken.

4. A fourth and last solution would be to allow ourselves a rewrite of the source code, by adding a new source block that can model the dangling b_6.

Both of these cases together show that exist are two classes of flow differences: (1) Those that overapproximate the execution count (as in *adpcm*) have an adverse impact on the precision of the WCET estimate, but there are ways to mitigate them. (2) Those that preserve execution count (as in *nsichneu*) become only agnostic to flow facts, which does not necessarily lead to overestimation, but approaches the result of traditional binary-level analysis without infeasible path information. In other words, this second class of flow difference only looses the semantic details of the source code. Furthermore, both classes of differences can be somewhat reduced by reformatting the source code, as described above.

7.5.6 Calling Contexts for Invariants

One last point of discussion arises from the use of microarchitectural invariants. Most WCET analyzers separate calling contexts [WEE+08], especially for low-level analysis. Assume some function f is used from two different callsites $l_1 \prec l_2$ in a program. Then its caching behavior at l_1 might result in all misses, whereas at l_2 there is a chance that some of its instructions still reside in the cache. Therefore, some analyzers establish different invariants for l_1 and l_2.

Similarly, functional verification tools are often context-sensitive, as well, i.e., they consider two difference instances of f when deciding properties with f at l_1 and l_2. This is especially the case for MC, which is fully path- and context sensitive.

As a result, it might be necessary to duplicate the source of f to annotate the different microarchitectural invariants for l_1 and l_2, or to encode the calling context otherwise. Without such means, we can only annotate the weakest invariant among all contexts in the source of f, thereby loosing precision.

We can now revisit our program *ndes*, which, after applying abstractions, resulted in a source-level estimate that was 13% worse than that from binary-level analysis. The reason for this is the mentioned lack of calling context separation in the annotations. OTAWA indeed separates all contexts by virtually inlining all calls, and therefore yields multiple different classifications for some L-blocks (namely, for those in `getbit()`, which has in total 9 callsites). Our current annotator, however, does not duplicate the source code, and therefore can only apply the weakest classification. As a consequence, the overestimation was produced. After duplicating the source of `getbit()` according to the different calling contexts and then

annotating the precise invariants, we finally obtained the same WCET estimate than the binary-level analyzer.

In summary, this demonstrates that calling context separation is necessary when invariants are computed and annotated, since otherwise we loose significant precision in programs that have a high code re-use. As an outlook to the next chapter, this also hints towards the necessity to analyze library calls (meant for high re-use) under call context separation.

7.5.7 Revised Results

Our final WCET estimates are shown in Figure 7.10, after the discussed changes (accommodating for the underestimation in OTAWA, abstractions for *ndes*, source code reformatting for *nsichneu*). It shows that source-level WCET analysis is at least as good as the traditional binary-level approach, except for *adpcm* and *fdct*, where the first-miss approximation creates a small overestimation. Our approach is especially beneficial for programs with complex flows, where it made up to 260% improvements over traditional, binary-level analysis.

The figure also shows the average overestimation relative to the simulation, for all but the full model (since it is clearly not tractable). The dashed vertical lines indicate these averages, which illustrates that a source-level WCET analysis based on binary invariants already yields around 10% tighter estimates, and the source-level approach with source-level invariants is on average 56% tighter than the traditional, binary-level approach.

This comparison is based on the OTAWA tool, which is practically blind to any logical flow constraints. Therefore, some binary-level WCET analyzers may not be outperformed as drastically, although we expect they can only reach a similar precision under nearly ideal circumstances.

7.6 Modeling Other Processor Features

There exist further microarchitectural features that may impact the processor performance considerably. Unfortunately (from an analysis point of view), such features may have intricate interactions, such that they in general cannot be modeled individually. We recently have given an overview about timing effects of various processor features on a full Linux operating system in [uC18a], which describes the interaction of many features in detail. Some earlier discussions of other researchers can be found in [AEF+14], [HLTW03] and [Sch09], analyzing their difficulty for WCET analysis. In the following, we only discuss the most common performance enhancers in embedded systems, and their fundamental feasibility in, and impact on, a source-level WCET analysis.

7.6.1 Data Caches

The modeling of the data cache is analog to our instruction cache model, and therefore feasible and tractable through the use of invariants. The challenge instead lies in determining which addresses are being accessed. As opposed to instructions, where the address is given by their program counter (at least in statically linked programs), data addresses often only become known at run-time. Compilers can make use of indirect addressing – often caused by pointers in the source code – such that it can be hard to determine which address is present at a register at some location, and consequently the cache behavior is hard to predict. In fact, a sequence of A consecutive unknown data addresses, where A is the cache associativity, forces the analysis to consider all previous cache contents evicted, since we cannot limit the

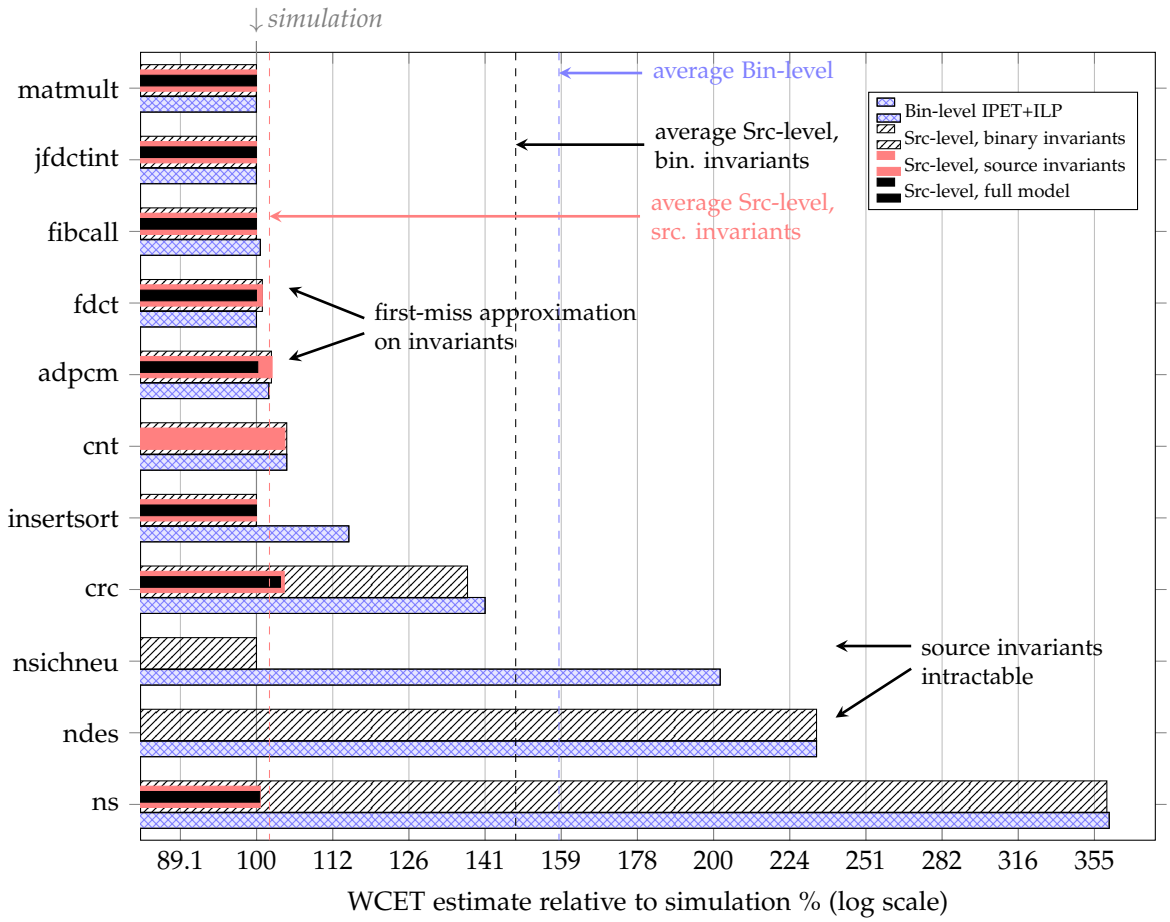


Figure 7.10: Best WCET estimates using different analysis methods after revisions.

access to any set. Analyzing data caches is therefore not only hard, but also vital for a tight WCET estimate. Occasionally, user input is needed to aid this analysis [WEE+08].

Although the issue of value analysis can be considered orthogonal, where standard methods exist [WEE+08], source-level WCET analysis should naturally help to improve the predictions for data caches. The instruction-to-source mapping establishes a link between strong source-level analysis tools that support pointers (as all of our evaluated tools do), and the instruction-level value analysis that may need user inputs. This link could be used to automatically provide such inputs, and with great precision, as well. Indeed, this idea has been successfully implemented in the ALL-TIMES project [LES+13], where a pointer analysis at source code level provided inputs for cache analysis. Although for imprecise mappings there may still be cases where our source-level model cannot decide the precise address being accessed, this will make data cache models most likely more precise and increase the degree of automation.

7.6.2 Multi-Level Caches

Many modern processors use a *hierarchy* of caches. That is, there are multiple caches in a cascade ("levels"), whereas each accesses the next level in case of a miss, before the slower backing memory is accessed. The first level (L1) is the fastest/smallest and usually separate

for data (L1d) and instruction (L1i), whereas higher levels (L2, L3) are larger and slower, and usually unified.

Implementation Sketch. Leaving aside multi-core issues and caches shared with peripherals, our approach can be naturally extended for a cache hierarchy. Each L-block causing a miss would trigger another access at the next-higher cache level, accumulating the time, e.g.,

```
if (!CACHED_L1(4,33600)) {
    MISS_L1;
    if (!CACHED_L2(4,33600)) {
        MISS_L2;
    }
    UPDATE_L2(4,33600);
}
UPDATE_L1(4,33600);
```

The challenge comes again from the complexity the model. As it can be seen, each additional cache level adds one more path on top of the source control flow for each source BB, thus the asymptotic complexity increase due to the cache model (exponential in nature), remains the same. Further, the number of L-blocks also remains the same, since cache lines often have identical sizes at all cache levels [Dre07]. However, updates at one level can trigger updates at other levels as well, depending on the access mode (read or write), and the inclusion policy (inclusive, exclusive). The model of this policy would further increase computational complexity. Memory-wise, and in the most naive implementation, we have to duplicate each age variable for each cache level, although it should be possible to find a better encoding for inclusive caches. The memory requirements for the model thus only grow linearly, therefore leaving us with the already known computational complexity for modeling multi-level caches.

7.6.3 Operand-dependent Timing

As mentioned in Chapter 6, we only consider the worst-case timing for each instruction once it starts executing, but some instructions may exhibit a large time variability depending on their operand values. For example, the timing of a load instruction may depend on the specific address, deciding whether it becomes a slow bus access, or an access to a fast core-coupled memory. A binary-level register value analysis and possibly a source model become necessary. Such a model would realistically have to rely on invariants, since otherwise the complexity would increase exponentially, making it intractable for even the best-scaling analysis. The approach would therefore be similar as for data caches, based on a binary-level value analysis aided through the mapping and source code analysis.

The time-variability of branches and jumps can be attributed to the edges of the CFG and thus can be captured without any overhead, but this creates another problem. Not all edges in the binary CFG have an equivalent in the source CFG. Even worse, a source block must be able to distinguish from where it was entered, calling for additional variables to encode such information.

7.6.4 Pipeline

We assume a scalar in-order pipeline, like many other WCET analyzers [WEE+08]. In such architectures a pipeline model is rarely necessary, since they are free from *timing anomalies*, as proven in [EJ02]. While there exist different notions of that term in literature [CHO12], they all imply that the absence of them allows taking local worst-case assumptions for each instruction to compute a sound overall WCET estimate, and therefore pipeline models can be

kept simple or even omitted. In other words, a timing anomaly means that some instruction that is not taking its worst-case behavior (e.g., experiencing a cache miss) can cause the overall worst-case of the program, which is rather unintuitive.

Notably, we have not made any assumptions on the absence of timing anomalies in our workflow, although this may reduce the analysis complexity. Our cache model is still valid in their presence. Assuming the absence of timing anomalies would in fact prohibit modeling even some in-order processors: An anomaly can be created as soon as the processor implements some form of speculative behavior, for example when it loads likely needed instructions into the cache with the intention of hiding cache misses when the instructions are indeed needed [uC18a, WEE+08]. One such example of speculative behavior is discussed next, because it may occur when a branch predictor is used.

7.6.4.1 Branch Prediction

Many microarchitectures perform some kind of branch prediction, to hide the latency for loading instructions and data after a branch [Fog18, ARM16]. Predictions for both the outcome of two-way branches, and the (possibly multi-way) target address of indirect branches, are being made by a *branch prediction unit*. If the branch is predicted incorrectly, then the pipeline and other resources must be flushed, which means there is a time penalty to fetch and decode the correct instructions.

The difficulty arises for *dynamic* branch predictors, which, unlike their *static* counterparts, predict the branches during run-time, based on the instruction stream passing by. WCET analysis must be able to soundly capture all possible mispredictions, without being too pessimistic. An analysis must therefore model the current state of the predictor (its table entries) up to a certain precision.

Closely related is *speculative behavior*, which can create timing anomalies. This behavior happens in the time window between predicting a branch and learning the actual outcome. The processor continues to feed the pipeline with the instruction at the predicted branch target, and buffers the results until the actual branch target becomes known. Whenever the prediction was incorrect, it flushes the pipeline. If the prediction was correct, the speculatively executed instructions are allowed to be released from the buffer ("retire"), and no time was lost waiting for the branch outcome. A side effect called *cache pollution* [uC18a, Sch09] may then cause timing anomalies as follows. Although the instructions executed during misspeculation are not committed, they can still cause changes in cache and buffer states. These effects do not only manifest during later execution, but crucially, they may sum up to be worse than the original miss that the predictor was trying to avoid. Therefore, a branch predictor model may also interact with the cache model, depending on details in the microarchitecture.

Implementation Sketch. Our source-level analysis in principle allows modeling a branch predictor. First, let us assume the branch prediction does not influence cache states. Then it could be encoded as follows:

```
#define BPRED_PRELOAD(addr) /* nothing */
#define BPRED_PRED(addr) _bpred_predict(addr)
#define BPRED_NOBRANCH _bpred_tar = 0
#define BPRED_EVAL(target) if (_bpred_tar) { _bpred_update(_bpred_src, target); }

void _bpred_predict(unsigned long addr) {
  // ... models the table lookup
  _bpred_tar = ...  // sets the expected target L-block
  _bpred_src = addr; // memorize the branching block
  BPRED_PELOAD(_bpred_tar);
```

```
}

void _bpred_update(unsigned long addr, unsigned long target) {
  // models the update of the table
  if (target != _bpred_tar) {
    TIC(4);  // mispred penalty (pipeline flush)
  }
  ... // update predictor state
}
```

Furthermore, the BB models have to be changed, such that at the beginning of each binary BB we potentially evaluate the branch prediction, and at the end of each binary BB we potentially trigger the branch predictor as follows:

```
void _cache_main_sb7(void) {
  { /* main.bb4 @82b8-82c0: */
    BPRED_EVAL(0x82b8);
    TIC(3); /* insn time of bb4 */
    if (!CACHED(3,0x82b0,0x82b8)) ...
    BPRED_NOBRANCH;
  }
}

void _cache_main_sb8(void) {
  { /* main.bb2 @8340-8348: */
    BPRED_EVAL(0x8340);
    TIC(3); /* insn time of bb2 */
    if (!CACHED(0,0x8340,0x8340)) ...
    BPRED_PRED(0x8340);
  }
}
```

Where BBs without branches are terminated with `BPRED_NOBRANCH` and those with branches with `BRED_PRED`, handing over the PC of the branching instruction and triggering the prediction. As soon as another BB is entered, `BPRED_EVAL` evaluates the prediction (if been made), accumulates a branch misprediction penalty (if it was incorrect), and updates the predictor state.

In case the branch prediction impacts the cache state, then the model is extended as follows:

```
#define BPRED_PRELOAD(addr) do { \
  switch(addr) { \
    case 0x82b8: UPDATE(3,0x82b0,0x82b8); break; \
    case 0x82c4: UPDATE(0,0x82c0,0x82c4); break; \
    // ...
  }; \
} while (0)
```

This updates the cache state together with the prediction, thus entangles cache and branch predictor models further.

This is a straightforward implementation on top of our existing model, but might not be the most efficient one. For example, not every BB requires calling `BPRED_EVAL`, and the definition of `BPRED_PRELOAD` is essentially lookup table with a size of the number of branch targets in the program, and mainly an artifact of our encoding.

Complexity-wise this model generates a lot of additional analysis paths. Every binary BB following a branch or indirect jump would have at least twice the number of paths going through it. While this is not as expensive as the increase from cache analysis (since there every *L-block* twice the number of paths), it is still expected to add too much complexity to be analyzed as is. Especially if the predictor impacts the cache state, we expect that the computation of microarchitectural invariants becomes necessary again. Yet, the details are unclear at this point.

7.6.4.2 Prefetching

A *data prefetcher* circuit can further reduce access times to slow memory. It predicts future data accesses and actively pre-loads the data into caches, to have it when needed. Most prefetchers are triggered by certain access patterns in cache misses [Dre07], and use them to predict future accesses. However, prefetchers may also load data that is never demanded later on, and thereby can have an adverse impact on the caching behavior. Due to these similarities, modeling a prefetcher is expected to be similar to modeling a branch predictor with timing anomalies, except that the prediction follows a different algorithm.

7.6.5 Bus Transfers & Peripherals

In many system-on-chip processors there are peripherals (e.g., serial ports) which are accessed from the core via on-chip buses. Without any behavioral description or at least maxima for access latencies of bus and peripherals, a WCET analysis cannot soundly model their timing effects.

It is hard to predict how such a model could look like, since it strongly depends on the specific microarchitecture. In the simplest form, it may be sufficient to conduct a value analysis and deduce which peripherals are accessed, and then look up their worst-case timing specification. Such a model would be easy to integrate into our source-level analysis. For more advanced processors, a peripheral access may require crossing a number of buses, which themselves could be busy with carrying out other operations, like Direct Memory Access (DMA) transfers, and therefore have a variable timing. This would require a more complex pipeline model, where the recent history of bus accesses is memorized.

Consequently, at this time it is unclear how such features could be modeled in a generic way, since they are highly depending on the specific processor and its detail of documentation.

7.7 Comparison to Related Work

We are not aware of any publication, approach or tool that proposes or implements a source-level model of microarchitectural features for WCET analysis. Some work exists for data caches in the domain of Virtual Prototyping [CZG13, MGG17], yet the safety for WCET analysis would have to be evaluated. Furthermore, architectural models in VP are typically implemented on a simulation basis, and evaluated dynamically. In contrast, static WCET analysis requires those models to be available in the domain of the static analyzer, i.e., not in a separate simulator, but in the source code. On the other hand, our source-level instruction cache model fills a gap in literature, and can be used for data caches, as well.

The computation of invariants that we propose are in direct analogy to the methods in traditional WCET analysis [WEE+08], but adapted towards the capabilities of the used source-level verification tools.

7.8 Chapter Summary

We have introduced a novel source-level model for set-associative instruction caches, and further refined the resolution of flow differences compared to the previous chapter. The model itself is implemented in the source code and referenced by the user code, such that source-level analysis can analyze the microarchitectural behavior along with the user code. This full model yields almost perfectly tight WCET estimates compared to simulation, but as expected its complexity is often too high for MC, which is in line with Wilhelm's analysis [Wil04].

To address the complexity problem, we proposed to use AI to compute microarchitectural invariants before performing path analysis, as done in traditional WCET estimation workflows [WEE+08] to keep the models tractable. As a result, we arrive back at a model complexity similar to that for simple processors from Chapter 5, which we have shown to be tractable even with MC.

Specifically, we have shown two ways to compute these invariants. (1) Obtaining them from a binary-level analysis, as traditionally done, is quick but less precise. The experiments have shown that a source-level WCET analysis with binary invariants can still outperform a pure binary-level analysis, with estimates being on average 10% tighter. (2) On the other hand, we have shown that microarchitectural invariants can also be computed from the full source model with standard tools, which is more complex but also more precise, increasing the precision of the WCET estimate further. Estimates are on average 56% tighter than those from binary-level analysis.

We have further found that an effective use of the invariants requires call context separation and interprocedural analysis, which is an additional task in comparison to analyzing simpler processors. This is a similarity to traditional WCET analysis, and may require code duplication to precisely annotate the microarchitectural invariants to the source code.

With the new flow difference handling, we found that flow differences fall into two classes. (1) Flow differences that are mapped such that they potentially increase the execution frequency of the binary BBs definitely reduce precision, but can be somewhat mitigated by source code formatting. (2) Flow differences which retain the execution frequency can be precisely considered during path analysis, thereby not necessarily leading to overestimation. As a consequence, a precise instruction-to-source mapping is beneficial for a precise estimate, but not a prerequisite.

Our complexity-reducing source code transformation techniques (slicing, acceleration and abstraction) from the previous chapters remain valid, and should be applied after microarchitectural invariants have been annotated. Consequently, the addition of caches increases the computational effort of source-level WCET analysis only by the effort of computing these invariants. AI is the natural choice for such a task, and computationally cheap in comparison to MC. Therefore, MC remains a viable analysis technique for WCET estimation even in the presence of caches, but it needs to be supported by other methods.

We have further discussed modeling data caches, multi-level caches and branch predictors, arriving at the conclusion that all of them can be represented in a source-level model, thereby enabling a much wider range of processors to be supported. The complexity challenge, however, must be expected to stay one concern for modeling these features.

Chapter 8

Discussion and Conclusion

***

This chapter summarizes and reviews the proposed source-level approach to WCET analysis in comparison to traditional WCET analysis, discusses benefits and drawbacks, as well as remaining limitations in a practical setting.

8.1 Complete Workflow for Source-Level Timing Analysis

The complete workflow for a source-level WCET analysis is depicted in Figure 8.1. This illustration focuses on the temporal aspects of the integrated functional and temporal verification workflow shown initially in Figure 1.3 on page 11. As in traditional WCET analysis, the workflow starts with the developer, writing programs incrementally. The following steps are therefore repeated numerous times during the development of an application.

(**1**). **Compile & Link.** The compiler is used frequently to build the program during the implementation of new program parts, which automatically triggers the *immediate feedback loop* as explained in the following.

(**2**). **Flow Analysis.** We first construct the CFGs for both source and binary. For the latter, an architecture-specific ISA decoder is required which can at least identify jumps, calls, returns, branches and their targets (see also Section 6.4.1.2). This construction may further involve a value analysis to determine targets of dynamically resolved calls and jumps. In principle the output of the generic mapper could be used to exploit source-level information to constrain pointer analysis, as in [LES+13] but this is not considered here. The result of this step are the CFGs at both source and binary level.

(**3**). **Instruction-to-Source Mapping.** Using the automatic methods from Chapter 6, we establish a mapping from each binary BB to its associated source BB. If aggressive optimization needs to be enabled, optimization reports from modern compilers [Cla19, Fre19] can be taken into account to support the mapping without requiring user inputs. The result of this step is a complete mapping $M : V_b \rightarrow V_s$ for each CFG, where V_s are the BBs in the source CFG and V_b are the BBs in the corresponding binary CFG, including inlined functions. This mapping further guarantees that the order of execution is not switched and that execution counts are not underestimated, both

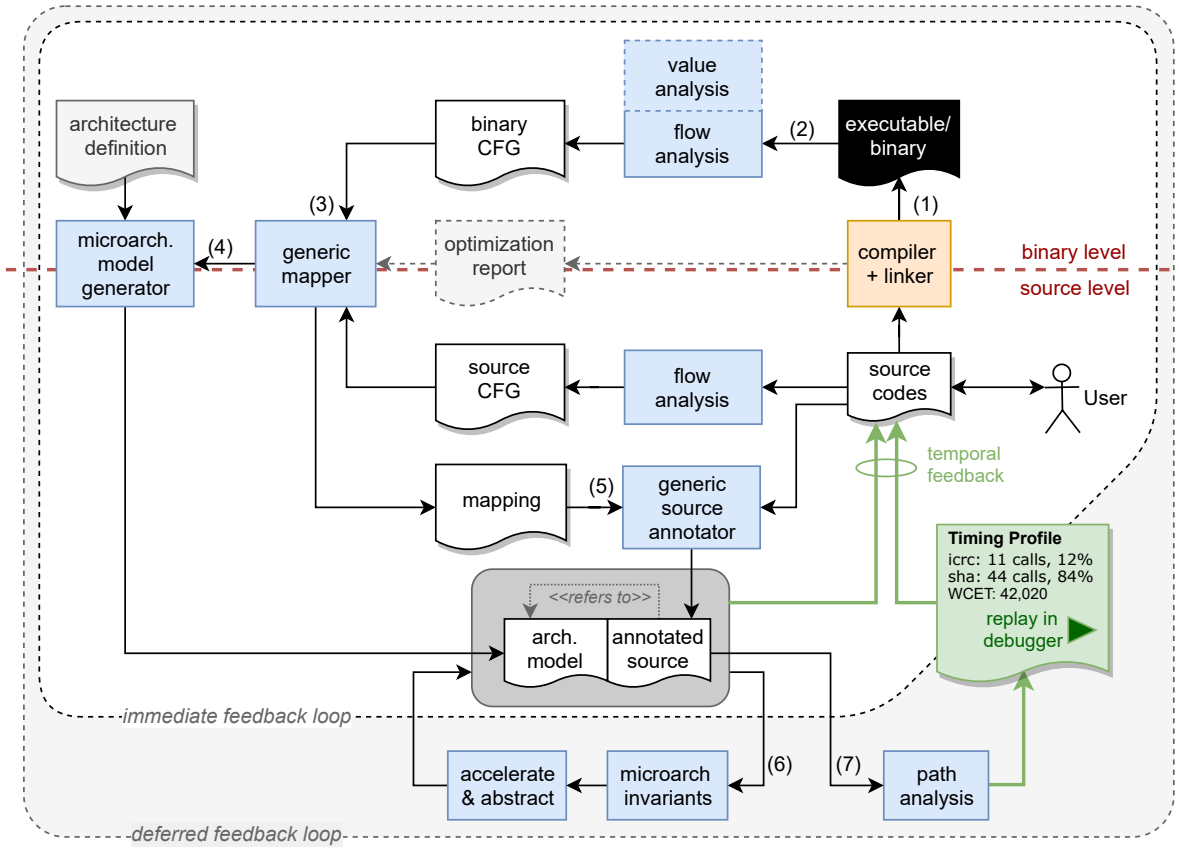


Figure 8.1: Complete workflow for source-level WCET analysis.

essential properties for sound and safe WCET analysis. Flow differences between source and binary can be handled either by lumping (see Section 6.4.5), or by an IPET encoding (see Section 7.2.5).

(**4**). **Microarchitectural Model Generator.** Next, we compute a precise microarchitectural model capturing the effects of every binary BB individually, and combine them to *source block models* as explained in Section 7.2.1. Specifically, these models cluster binary BBs which map to the same source BB. Importantly, these models are *unsolved*, in the sense that the model is just encoded in the source code language, but not evaluated yet. For example, it contains the information about which L-blocks are being accessed when each source BB is executed, but at this point the cache states at each location and hence feasible timing effects are still unknown. This model generator is therefore fast, mainly concerned with how to describe individual architectural effects, and to soundly encode them in the given mapping M. Optionally, this step may take into account results from external analyzers, for example MUST/MAY/PERS information computed by an external WCET tool (see Section 7.3). The result of this step is one additional translation unit containing the microarchitectural model, consisting of the source block models and additional global variables required to model the effects of their execution. These models are subsequently merely referenced by the annotator. After this step, the analysis is solely based on the these source codes.

(**5**). **Generic Source Annotation.** The mapping is then used to annotate source BBs with a call to their respective source block models. The annotator itself is agnostic to architectural properties, and only a language-specific component. It furthermore generates

the driver function (see Section 5.1.3) to set up the analysis context. The result is a source code that is annotated with references to architectural models, describing the local timing properties of each source BB. These properties can be visualized in the programming environment, terminating the *immediate feedback loop*. The annotated source code is used again in the next step.

(**6**). **Accelerate & Abstract.** As we have seen in previous chapters, analyzing the full source code with precise microarchitectural models is rarely tractable. Therefore, now part of the *deferred feedback loop* that is only executed on demand, we apply the introduced scalability-enhancing methods, namely program slicing (Section 5.2.1), loop acceleration (Section 5.2.2), computing of microarchitectural invariants (Section 7.3), and optionally further abstractions (such as loop abstractions, Section 5.2.3). The resulting source code has an equivalent or overapproximated temporal behavior compared to the precise model, but has a complexity that is low enough to apply fully automatic methods such as Model Checking, which is part of the next step.

(**7**). **Path Analysis.** Finally, also part of the deferred feedback loop, is a joint functional and temporal analysis for the WCET path, taking into account the referenced architectural models. This step should be performed using Bounded Model Checking (MC), which can be automated with the search strategy proposed in Section 5.1.4. Alternatively, other automatic off-the-shelf source-level verification tools could be used, such as *Frama-C* [CKK+12], but unlike MC they might lack the necessary precision to bound the WCET, as we have shown in previous chapters. As an optional feedback to the user, we can automatically compute a timing profile from the WCET path as explained in Section 5.4.3, and moreover offer the possibility to launch a debugger session in which the developer can interactively replay and "debug" the WCET. This step concludes the deferred feedback loop.

In the following, we discuss this source-level workflow for WCET estimation regarding its precision and scalability in comparison to traditional binary-level WCET analysis, as well as its usability and limitations, before we draw an overall conclusion.

8.2 Precision

The precision of source-level timing analysis and therefore the tightness of the WCET estimate, mainly depends on the mapping quality. Our source-level model starts with maximally precise models w.r.t. the mapping, and information is subsequently only taken away or abstracted (e.g., slicing or invariants) to increase scalability. Information never needs to be reconstructed, which is in stark contrast to traditional binary-level WCET analysis. The tightness of WCET estimates can therefore only be improved by improving the mapping.

The mapping quality itself depends on the chosen mapper, the target, and the compiler. We noticed that the mapping for ARM was generally more precise than for AVR, partially owing to the larger word width (which did not require splitting decision nodes), and partially due to more hardware capabilities (e.g., all arithmetic shifts could be done in hardware).

8.2.1 Comparison to Traditional WCET Analysis

Given a perfect instruction-to-source mapping, a source-level WCET analysis is generally able to outperform traditional binary-level approaches, as it is able to discover all infeasible paths, and constrain microarchitectural states with maximum precision. However, the two approaches remain incomparable in the presence of compiler optimization, since then both

loose precision. For source-level analysis, the main obstacle are the occurring differences in the control flow between source and binary. The more differences, the more likely become overapproximations, which marks a trend in the direction of less precision with rising optimization level. On the other hand, the same is true for any binary-level WCET analyzer that tries to identify infeasible paths. Although our experimental data shows that the source-level analysis can still outperform binary-level analysis, especially at higher optimization levels, these results should not be generalized with levity to other architectures and compilers.

Furthermore, the precision advantage clearly depends on the program under analysis, and how much information a binary-level WCET analyzer can extract about them. For example, a single-path program like *matmult* is already precisely analyzed by a WCET analyzer that is merely flow-sensitive, such that source-level analysis has nothing to gain from the source semantics. Although more complex programs (e.g., *ns*, *crc*) tend to be more precisely analyzed with our source-level approach in our experiments, consistent with the findings in [BCC+16, LES+13], this might change when comparing against a more sophisticated WCET analyzer which performs a deeper analysis for infeasible paths. In our experiments the source-level approach was on average 17% tighter on cache-less processors, and 56% tighter on a processor with caches (using source-level microarchitectural invariants), in comparison to the chosen tools, and with the selected benchmarks. It is worth noting that the latter comparison is made against a WCET analyzer that has no means to identify infeasible paths, whereas in the former comparison that was not the case. Nevertheless, since our WCET estimates are often very close to the simulation value, this strongly suggests that any binary-level analyzer can only reach this precision under ideal circumstances, and would most likely show some degree of overestimation compared to our source-level estimate.

Lastly, it should be noted that MC intrinsically has full call context separation, since it explores paths in the program individually. This brings a further precision advantage over analyzers which do not attempt such a separation, or only on demand [HS02].

8.2.2 Furthering Precision

We already have proposed a more precise flow difference encoding with the introduction of the source cache model, which has resulted in almost perfectly tight WCET estimates. Still, some binary flows remain challenging due to scarce debugging information. The use of *markers* [SAA+15] might help to increase the mapping precision further. That is, the source code is extended with labels or inline assembly, which eventually can be found again in the binary, therefore directly establishing map entries that can be used as additional fix points. Since this method is orthogonal to the work presented here and our results were already sufficiently precise, it has not been implemented.

Last but not least, we excluded decompilation as means to achieve a mapping, but it might be useful in practice, since it solves two problems at once. First, decompilation resolves flow differences. It would therefore make non-deterministic encoding of flow differences unnecessary, which would reduce the complexity of the analysis (see Section 7.3). Second, a detailed decompilation would alleviate the difficulties of analyzing library calls (see Section 8.5). However, it would not make binary analysis superfluous, since the decompiler itself must analyze the binary.

8.3 Scalability

We have used several methods to reduce the computational effort of a source-level WCET analysis to make it more scalable for larger programs. Specifically, we have proposed program slicing, loop acceleration/abstraction, and computation of microarchitectural invariants, which allowed the use of Model Checking for WCET analysis. All programs could be analyzed in a matter of seconds or mostly minutes, even including a cache model. Although this is a reasonable time span in practice (especially if no further manual work is required), there is room and opportunity for improvements.

8.3.1 Comparison to Traditional WCET Analysis

The overall analysis time of our source-level approach is on average about one order of magnitude higher than the traditional binary-level approach. This may become an obstacle for larger embedded programs. Therefore, additional abstractions should be investigated to reduce the model complexity further.

Nevertheless, we argue that this longer analysis time is compensated by the reduction of user inputs. As practical studies of WCET analyzers and tools have pointed out, users spend most of their time in preparing the program for analysis rather than running the analysis, which includes the task of providing annotations [LES+13, WEE+08].

While much of the inferior analysis time in our approach must be attributed to the chosen analysis method, namely Model Checking, our experiments emphasize that a source-level approach in conjunction with source code transformations is particularly effective to mitigate this weakness, since similar approaches based on Model Checking have a worse scalability, as discussed in Section 5.6. Our analysis time is orders of magnitudes lower than comparable approaches from literature, especially for complex programs.

8.3.2 Furthering Scalability

As a first measure, a per-subprogram modularization could be applied after microarchitectural invariants have been computed, since the timing of subprograms is then independent of the remainder of the program, except for the call context. The latter one could be determined with AI, and declared as an assumption in the driver function. As a result, the models would become smaller, but still not context-insensitive. Furthermore, this would enable parallelization and on-demand timing analysis only parts of the source code that have changed. Additionally, entire function calls could be abstracted when their execution time can be captured as symbolic formulae (as in [VHMW01]).

Another enhancement aims at the SMT back-ends used especially in Bounded Model Checking. A large part of the analysis time is indeed spent in these back-ends. Since they are general-purpose tools, they are not geared towards the analysis of control- and data flows, and therefore not using optimal search strategies. Henry et al. [HAMM14] have shown that SMT-based analysis of WCET problems can be significantly accelerated. They encoded the search problem as an *optimization modulo theory* problem and computed cuts. Those cuts prevent the SMT solver to run into an exponential explosion for specific shapes of the CFG. Among others, they conducted WCET experiments on the *nsichneu* and *PapaBench* benchmarks for an ARM7 architecture, and achieved significant speedups that brought these programs from intractability to solutions within one hour. This demonstrates that the scalability of MC for WCET analysis can be further improved, but also that source-

level analysis itself brings a speedup, since our analysis is about two orders of magnitude faster than their estimation process.

Last but not least, an extreme measure would be to formulate an ILP problem on the source code, similar to binary-level analysis. This would bring analysis times down to traditional approaches, but discard all path information again.

In conclusion, there is much room for improving the scalability of source-level WCET analysis further, with or without sacrificing precision, and especially if Model Checking is used as primary analysis technique.

8.4 Usability & Safety

By shifting the majority of the analysis to source level, users interact with the tooling in their familiar environment, and the capabilities of recent source-level analyzers can be exploited. Ideally, one and the same person develops and verifies a subprogram, therefore a source-level approach can help capturing the developer's implicit knowledge on the program in source annotations. Since this effectively combines functional and temporal verification, any annotation that may be required during analysis serves both purposes, such that no additional efforts are incurred by our timing analysis.

However, unlike precision and complexity, it is mostly meaningless to compare the usability of our current tooling with existing WCET analyzers. Currently, we do not require any user inputs, since we only support simple processors (Atmel AVR) or hypothetical processors (ARM-like) with a limited feature set. We have discussed modelling further processors in Section 7.6, which would likely result in a similar complexity and precision, but undoubtedly some inputs would then be required from the user. In source-level WCET analysis the main concerns are about the construction of the CFGs, and to determine data cache accesses, which both require value analysis in general. Based on the findings in [LES+13], where a traditional WCET analyzer was successfully coupled with source-level analysis to reduce exactly these problems, we argue that source-level analysis should be beneficial to realizing an even better solution. Additionally, we propose to keep all potential annotations in the source code, for it is the best interface between program analysis and developer. Nevertheless, we cannot predict how much manual inputs can be avoided in comparison to traditional WCET analysis.

On the other hand, our source-level approach to WCET analysis has a significant usability advantage over traditional WCET analysis when it comes to infeasible paths. MC does not require any manual inputs to exclude infeasible paths, thereby circumventing the challenges of identifying such paths using external methods [BLH14], and to define an annotating language that can capture them [KKP+07]. Consequently, this part of the toolchain can be highly automated, and avoids human error that may ultimately refute the WCET estimate.

Another advantage over traditional WCET analysis is our debugger-based replay of the WCET. This introduces a novel way of bringing timing closer to the developer, and facilitates a better understanding of where time is being spent, and why. This natural interface has been proven useful even in our synthetic benchmarks, where we have identified a defect that had resulted in a suspiciously high WCET, initially suggesting an analyzer error (see Section 5.4.3). However, the replay of the WCET path had revealed an intricate defect that was causing a loop to be executed many more times than expected. Without the replay we neither would have been able to debug this problem by testing (since the inputs causing this would not have been known), nor would a path information have been sufficient to explain why the loop iterated so often. In fact, it is very likely that in a traditional WCET analysis a user would have specified an incorrect bound, and obtained an unsafe WCET estimate.

Regarding safety, there is one main concern with source-level analysis, which is the correctness of the compiler. Since we only match CFGs but do not inspect whether the binary BBs are functionally equivalent to the source BBs, source-level analysis is not able to safely handle compiler errors. Although this is not the expected behavior and subject to tool certification within a safety-critical context, it is well-known that compilers can make errors, especially under optimization [WZKS13]. Such an error could refute the correctness of source-level analysis as follows. For example, if a defect in the compiler leads to a loop being possibly executed more often than specified by the source code, then our source-level WCET analysis cannot detect this issue and would underestimate the WCET. While most traditional WCET analyzers make use of the source code and thus are somewhat threatened by this issue [WEE+08], source-level analysis arguably is at higher risk. Related to compiler issues, we want to emphasize once more that the absence of undefined behavior is essential for safe WCET estimates, which is discussed in more detail in the next section.

In summary, the source-level approach improves the usability for the developer over traditional approaches, but naturally it cannot eradicate all problems that persist in WCET analysis, tracing back to the decision problem discussed in the beginning. Nevertheless, some long-standing difficulties of WCET analysis can be avoided, and no new usability drawbacks have been introduced.

8.5 Limitations

The work presented here has several limitations, most of which are shared with traditional, binary-level WCET analysis. These are problems with library functions, the constriction to imperative source languages and absence of undefined behavior, the absence of an operating system, and a narrow support for microarchitectures. A detailed description follows now.

8.5.1 Library Functions and Missing Sources

One difficulty for WCET analysis in general, are programs for which no source code is available. Virtually all WCET analyzers refer to the source to derive or specify constraints [WEE+08, LES+13]. One likely scenario is the use of libraries, for which end users usually do not have sources at hand, and libraries which have some functionality implemented in assembly for better performance. Many applications make use of the C library, sometimes even implicitly. For example, compilers may choose to insert library calls to implement certain functionality, without an explicit call being visible or even expected at source level. One such example has been shown in the *adpcm* benchmark in Chapter 5, where arithmetic shifts were implemented using implicit library calls. Such calls to "non-sources", must therefore be supported. For the sake of this discussion, we uniformly refer to all such cases as library functions.

For a source-level WCET analysis, the lack of source code for libraries is even more critical, since it is then agnostic to any behaviors in such parts of the program. We discuss several solution approaches in the following, one of which is already enabled by our flow difference handling introduced in the previous chapter.

Approach 1: Decompilation. One obvious way to circumvent the problem, is to decompile all library functions back to source code. There have been many attempts at decompiling assembly code to source code, both specifically to timing analysis and as a general tool [SRN+18]. However, the decompiled code is typically less precise than the original, due to the information that is lost during compilation [CTL97]. While novel search-based

techniques can reconstruct sources closer to the original, decompilers still introduce some overapproximation due to lacking type info and other contextual knowledge, and thus would weaken the WCET estimate. More importantly, decompilation has been shown to be incorrect for some corner cases [BLSW13], which could refute the WCET estimate.

Approach 2: Substitute by local WCET. A first workaround for library calls has been shown in Chapter 5, where we substituted such calls by their respective WCETs. There are, however, at least three reasons why this is in general not a viable solution.

1. Substituting library functions by their local WCET forces WCET analyzers to assume pessimistic upper bounds on the WCET of such functions, as well as on their side effects on rest of the program. This may lead to a gross overestimation.

2. The WCET of a function may be – practically or truly – unbounded when the call context is not considered. One particular example is a call to `memcpy`, which takes as a parameter the number of bytes n to be copied. If n is unknown, then we must assume the largest possible integer is given, which would lead to an unacceptably high overestimation that can be considered practically unbounded.

3. Last but not least, the global worst case of the program is not necessarily triggered or even permitted by the local WCETs of its constituents, due to *timing anomalies* discussed in Section 7.6.

For processors without timing anomalies and without caches, the first two problems can be solved by computing WCET estimates that are parameterized by their execution context, as proposed by Cerný et al. [CHK+15]. This would allow pre-computing parametric timing formulae for all library functions, which subsequently are used in the source code analysis in lieu of the callees. Such parametric annotations could actually be educational for the developer, offering a succinct description of the library's timing behavior.

In the presence of caches, however, parametric timing analysis is insufficient. Since caches are small by construction, library and user functions both compete for this precious memory, and can cause mutual eviction of their cache entries. Therefore, the WCET of a call to any function depends on the possible cache states at the callsite, which themselves may be a product of a previous call to this function and the remaining program. As a result, timing analysis of the source code and of library code can no longer be separated without making overly pessimistic assumptions, even in the absence of timing anomalies. This mutual interaction of user and library code calls for a joint analysis, as discussed next.

Approach 3: Generalized Flow-difference Handling. A general solution can be attained by regarding library calls as flow differences, and applying our model from Chapter 7. However, a pure source-level analysis cannot generate logical flow facts, therefore only the structural constraints from IPET would be considered. Furthermore, this also means that users have to provide loop bounds for library functions. In summary, this approach would therefore require some user inputs and reduce the precision of the cache model to a similar level as a pure binary-level WCET analysis, but otherwise ignore any additional information that is available in the binary, and thus sacrifice precision.

Approach 4: Hybrid Methods. Another general and more precise approach would be to combine binary- and source-level analysis using *path-wise exploration* or *path summaries*, if a binary-level WCET analyzer is available. While the sound integration of these poses

the main challenge, this approach would leverage the available binary analyses, and could produce tighter estimates than a flow difference handling, and moreover spare the user some annotations. We have published a more detailed discussion on such solution approaches in [uMVC19b], and only summarize the results here.

- **Path-wise Exploration.** Using techniques like symbolic execution [KKBC12] and path-wise SAT/SMT formula generation in *cbmc* [CKL04], paths in the program can be explored one by one. To analyze library calls, we could interleave source and binary analyzers along with the executed code. While such a brute force exploration could be maximally precise, its complexity would be exponential in the number of paths, and thus not scale to real world programs. Even if symbolic execution could handle unknown states and thereby avoid enumerating all paths, analysis time would be proportional to the execution time of the program, yet orders of magnitudes slower than native execution [WEE+08].

  To cut down on the number of paths to be analyzed, there have been attempts at analyzing only those with the maximum execution time [NS15]. They typically require a pre-processing stage which identifies and discards infeasible paths as well as those that lead to cache hits, using techniques like AI and MC. The path-wise exploration is then performed only on the remaining paths. Such approaches face two problems: (1) Path exploration is merely shifted to the preprocessing stage, leaving us with the difficulty of selecting a safe subset, and (2) the preprocessing stage suffers from the same problem of having to analyze an obfuscated binary.

- **Path Summaries.** Instead of enumerating all paths in the program, techniques based on AI (see Section 3.4) can be used to compute invariants on the cache states before and after library calls. As opposed to the exponential complexity of path-wise exploration, AI scales better and thus is well suited for larger programs. The workflow would be as follows:

  1. Run source analysis to compute summaries of contexts and cache states at each callsite.
  2. Run binary analysis for each callee as in [FW99], considering the current cache state as the initial one, and back-annotate cache set states upon return to the source.
  3. Use the information computed at the end of Step 2 to resume source analysis, until next callsite is reached.
  4. Repeat Steps 2 and 3 until a fix point is reached, that is, all possible cache states have been identified.
  5. Use the cache states to estimate the WCET of the callers and callees using the respective source and binary analyzers.

  In other words, source and binary analysis are performed alternatingly and iteratively, until all possible cache interactions have been collected. On both sides, we require an analysis technique like AI. This approach can build on existing analyzers, e.g., Frama-C [CKK+12] for source, and OTAWA [BCRS10] for binary level. To improve the invariants, a subsequent refinement using MC is possible [BCR13], but would further increase complexity.

Hybrid approaches require not only an interface between source- and binary-level analyzers to communicate cache invariants, but additionally the overall analysis engine would

need to implement a JOIN operation to integrate new cache states after each iteration. This approach therefore requires a higher implementation effort.

Table 8.1: Qualitative comparison of solution approaches for functions without sources.

	Source			Hybrid	
	decompilation	local WCET	flow difference	Path-wise	Path summary
soundness	?	✓	✓	✓	✓
completeness	?	(b)	✓	×	×
assumptions	?	(I)+(II)	(c)	none	none
precision	high	low	medium	highest	medium
complexity	(integrated)	(b)	(integrated)	exponential	linear to exponential
impl. effort	high	low	medium	high	high
user inputs	none	none	(c)	none	none

(I) no timing anomalies, (II) no caches, (III) completeness threshold known
(b) depends on analysis method, (c) loop bounds

Table 8.1 summarizes all approaches to handle library functions. As it can be seen, there is no clear winner among the presented approaches. A high precision naturally requires spending more analysis effort and user interaction. Since soundness (subject to assumptions) seems attainable in all approaches, precision and complexity are deciding factors in practice.

8.5.2 Source Language and Behaviors

We have developed our source-level WCET analysis for the C language, which is one of the top three languages in software engineering to this day [Ind18], and especially popular in embedded systems. While there are other interesting languages that may even be preferable for embedded systems for their stricter semantics, notably Ada/SPARK (see Section 4.3), the gist is that the leading languages are all *imperative* in nature. This means that the programming paradigm in the source is close to the implementation paradigm at instruction level, having an explicit control flow. Crucially, this property results in source and binary CFGs to share enough structural similarities to meaningfully represent machine behaviors in the source, enabling a mapping that can keep execution order. An extension to object-oriented languages, such as C++, would not require any major changes to our workflow, but introduce additional challenges from virtualization and dynamic dispatch, namely the need to resolve indirect calls. However, this is not a new problem in WCET analysis, and therefore we expect that this can be handled with existing methods.

In contrast, *declarative languages*, such as *Prolog*, *Lisp* and *Lua* (the most popular declarative languages currently ranking 31 to 33 in the 2018 TIOBE index [Ind18]) follow a different programming paradigm. The control flow is only implicit in the source, and mostly given by data dependencies. Although a CFG can still be computed and mapping can also be established, the CFGs could look very different, and hence annotating microarchitectural models would require more overapproximations, likely resulting in unusable WCET estimates. A two-step approach to compute the WCET on the declarative language *Lustre* has been published by Raymond et al. in [RMPC13]. This language is first compiled to a *C* program, and subsequently to machine instructions. They demonstrated that the strict semantics of Lustre simplifies the computation of flow constraints, which can significantly improve the resulting WCET estimate. However, since we use the precise method of Model Checking for path analysis, such a two-step approach is expected to merely reduce analysis time, yet not the

precision, and furthermore only beneficial if the high-level compiler provides a sufficiently precise tracability.

We also observe that some embedded devices move into the direction of scripting languages like, *Python*, *JavaScript* and *Lua*, in the context of the *Internet of Things*. This creates more challenges for WCET analysis in general. Whereas these languages are imperative, the scripted program must essentially be regarded as input data, and the interpreter becomes the actual program to be analyzed. Needless to say, that interpreters have a high complexity for their required capabilities. This entire setup thus bears resemblance to the Halting Problem, with one additional layer of indirection added (we analyze the program that evaluates another program). Unsurprisingly then, there is virtually no literature about WCET estimation of scripted languages.

Specific to source-level WCET analysis there is the need to refrain from constructs that give rise to undefined behavior. As we have seen in Chapter 4, all source-level verification tools then necessarily run the risk of producing unsound results, since by definition it is unknown what functional behavior the compiler is implementing.

Coupled to the source language is the next limitation of our proposed source-level approach to WCET analysis, which is the lacking support for operating systems and multitasking.

8.5.3 Operating Systems, Micro-Kernels and Run-Time Environments

The WCET is by definition the uninterrupted time of a task running on a processor [WEE+08], therefore issues of multi-tasking are usually not considered, but left for *schedulability analysis*. Nonetheless, WCET analysis cannot be agnostic to multi-tasking environments, since different tasks usually share the caches, and therefore mutually influence each other.

In practice, most WCET analyzers either ignore or simplify this problem [WEE+08]. The latter case is enabled by tasking environments that follow some specific task activation pattern, like un-preempted round-robin scheduling of tasks. WCET analyzers then can assume that caches are only modified before and after the program/thread under analysis has completed. Our approach neither takes into account multi-tasking, therefore requiring the same conditions as traditional deterministic WCET analyzers. The selection of a programming language with built-in tasking, such as Ada/SPARK in our case study 4.3 with its run-time environment, can therefore help to establish the required conditions.

On the other hand, micro-kernels like *seL4* [SKH17] meet WCET analysis half way, by making scheduling and other tasking behavior explicitly visible in the source code, which can then be analyzed together with the application software's without any restrictions. An extreme example is the *PapaBench* program derived from the Paparazzi UAV flight controller [NCS+06], which emulates multi-tasking by explicitly implementing a scheduler-like functionality in the user program.

Further complications arise with the use of an Operating System (OS), since it implies the use of virtual addressing on top of multi-tasking. Memory addresses of both instructions and data are only decided at run-time, by the OS' memory allocation algorithms. As a consequence, a cache analysis becomes useless, since we would have to consider all possible memory allocations. Even if the allocation algorithm would be modeled, additional problems would arise from the need to model Translation Look-aside Buffers (TLBs) and their interaction with the data caches [uC18a], since the state of these hardware buffers has a large impact on the latency of address translation. General-purpose hardware and OS are therefore not suitable for WCET analysis. The use of a real-time OS does not change this situation, since it only enables additional preemption points in the kernel, better synchronization mechanisms and less task jitter, yet the non-determinism from virtual addressing remains.

Another issue with OSes is dynamic loading of libraries. First, it requires a start-up phase that can be costly [uC18a], during which the OS searches for the required library and loads it into memory. Second, this very act of loading code at run-time causes the instruction addresses of the library to be statically unknown, which again renders cache analysis useless. Therefore, all WCET analysis is best used with *embedded* systems that make use of static linking, which makes the system analyzable in the first place.

Even without an OS or support for threads, multi-tasking can still be an issue, as seen in our case studies. Interrupts may be necessary to implement certain functionality. Similarly to tasking, this evades the realms of traditional WCET analysis and needs some cooperation between design, OS and hardware. Source-level analysis offers a rudimentary solution for bare-metal platforms, as explained in Section 4.1. In general, however, the source code does not allow considering all preemption points, therefore the analysis could be unsound.

Consequently, WCET analysis in the presence of multi-tasking and OS almost inevitably requires hardware support, e.g., techniques like memory partitioning and explicit round-robin scheduling of tasks, as typically used in avionics [WH09b]. Such special hardware is currently in the focus of academic discussions, and described next.

8.6 Supported Processors and Platforms

Our source-level approach to timing analysis was developed for in-order mono-processors with a scalar pipeline and set-associative caches, which is in line with most current static WCET analyzers [WG14, WEE+08]. This thesis primarily explores the feasibility and effectiveness of a source-level approach to WCET analysis, but does not present a complete, ready-to-use solution. In particular, an explicit pipeline model is not implemented, which might become necessary when a branch predictor is added. A solution strategy has been discussed in the previous chapter, but is subject to implementation and test. Similarly, data caches are currently not modeled, but their modeling is conceptually identical to the instruction cache model presented in Chapter 7. The only additional task is to resolve the target addresses with existing methods from WCET analysis [WEE+08]. Notably, we made no assumptions on the absence of timing anomalies [CHO12], which in principle does allow for processors with speculative instruction prefetch.

8.6.1 Examples of Supported Processors

The following is an incomplete list of commercial processors that we know or expect to be analyzable mainly with the methods presented in this thesis.

- Atmel AVR family (fully modeled in Chapter 5)
- ARM-Cortex-M (partially modeled in Chapter 7)
- ARM Cortex-R (in-order versions, partially modeled in Chapter 7))
- ARM7 TDMI (analog to Chapter 5)
- Analog Devices ADSP-21020 (analog to Chapter 5)
- Renesas H8/300 (analog to Chapter 5)
- SPARC V7 ERC32 (analog to Chapter 5)
- PowerPC e300 (partially modeled, analog to Chapter 7)

Other processors might qualify, or only require minor extensions. A verdict is only possible on a case-by-case inspection of the microarchitecture, due to the variety of proprietary tricks to attain a high processor performance.

8.6.2 Extension to Out-of-order and Multi-Core Processors

For a static WCET analysis, more complex processors pose modeling and complexity challenges that arise from features such as out-of-order processing, speculative execution, cache coherency protocols, bank conflicts, machine clears, and so on [uC18a], regardless of the type of analysis. Such features are not prevalent in architectures used for real-time systems, since their complexity and lack of insight makes it practically impossible to create a usable, or even tractable model [MTT17, Chapter 5.20]. They are typically used in high-end architectures and analyzed with probabilistic methods [CSH+12].

While there exist publications on handling some of these aspects in deterministic WCET analysis, e.g., [LRM06, Gmb19], such tools are an exception. With the presence of hardware threads, there is additional non-determinism caused by resource sharing, most commonly shared caches and buses. One analytic approach for out-of-order multi-core processors has been proposed by Chattopadhyay et al. [CCR+14]. Their work is based on interval analysis, modeling the interactions between shared caches, pipeline and branch predictors, and is also evaluated using the Mälardalen WCET benchmarks. Another one was proposed by Gustavsson et al. based on the model checker UPPAAL [GELP10]. However, both approaches are in practice insufficient to model real-world multi-core processors (they work on hypothetical processors). First, they consider buses with a TDMA policy, which is an uncommon implementation. Second, and more importantly, such high-end processors are typically used with an OS, which comes with the complications described in earlier sections. Last but not least, real processors often lack a sufficient level of documentation to build any model [Pus02, WG14], which is perhaps the reason why hypothetical architectures have been chosen by the authors. At this point, we want to emphasize that this is not to discredit the achievements of the respective authors, but a practical consequence of the steep increase of complexity in applications with such processors. An excellent survey of current timing verification techniques for modern multi-core systems has been published by Maiza et al. in 2019 [MRR+19], to which we want to refer the interested reader. This survey also covers isolation techniques such as the *WCET(m)*-approach, which seek to improve the predictability of modern processors by employing a mix of hardware and software techniques that mitigate temporal interference between tasks and cores.

However, to address the increasingly severe complexity challenge at its root, many researchers are currently active in proposing deterministic designs for high-performance single- and multi-core platforms, as discussed next.

8.6.3 Trends in Real-Time Systems

Recently, various researchers [Pus02, LLK+08, KP08, KP08] and [MTT17, Chapter 5.20] have arrived at the alarming conclusion that a sound timing analysis for modern processors is almost impossible. The complex microarchitectures require considerably large models, which in turn lead to a state-space explosion. Timing predictability has become a real issue in modern processors, even for those made specifically for real-time applications [MTT17]. This development is concerning, and increasingly renders deterministic static analysis either infeasible (due to lacking models), or intractable (due to complexity). A source-level analysis cannot circumvent these problems, and thus suffers from the same limitations.

The need for building predictable, yet high-performance architectures, has lead to a new research direction of designing predictable hardware. An older approach to deterministic multi-core systems comes from the MERASA project [UCS+10], which focused on making tasks independent of each other, to allow a compositional analysis. A newer example of

predictable architectures is the T-CREST [PPH+13] platform, designed to both deliver performance in the worst case (as opposed to average case), and amenable to a deterministic analysis. For example, instructions have constant execution time, bus accesses are time-multiplexed, and caches are managed by software. It builds on the time-predictable processor *Patmos*. Other examples are the CoMPSoC multi-processor architecture [GAC+13] and the Precision Timed Architecture [LLK+08]. There further exist techniques which provide some temporal isolation between the hardware threads, as described in [AEF+14]. The end of this spectrum are also more radical proposals, arguing that everything that threatens analysis must be removed [Sch09].

For future embedded applications requiring timing analysis with guarantees, such potent yet analyzable systems are a prerequisite, and suitable for our proposed source-level analysis. Performance-wise, current technology seems advanced enough to not require high-performance devices where safety and security is important. Indeed, the industry has picked up on these needs and is offering specific product lineups for real-time systems which exclude many of the problematic features, for example the xCore processors [XMO19] or the ARM Cortex-R series [Ltd19], both offering in-order single core platforms that can be analyzed with the methods presented here.

8.7 Concluding Remarks

In this thesis we have introduced a comprehensive approach towards source-level WCET analysis of embedded software, which leverages state-of-the-art source-level analyzers from functional verification. Our motivation were recent advances in functional verification tools, and an increasing adoption of them in general software engineering. This offers a chance to reconsider how timing analysis is done today, which unfortunately is often tedious, takes place at instruction level, and requires user inputs at this level. However, timing analysis should preferably take place at the familiar source code level, where developers are in a better position to reason about their program.

Towards that, we made the temporal behavior of the program visible in the source code, along with its functional specification. We first established a mapping from instructions to source code, and then generated microarchitectural timing models that are called from the source, such that the two are linked explicitly. As primary analysis method for WCET estimation we propose to use Model Checking, which we have shown to be scalable, at variance with the often expressed view that it is too computationally expensive for this purpose. We have introduced scalability-enhancing methods that significantly reduced the computational effort, and the literature suggests that additional significant speedups are possible, without sacrificing precision.

As a result, we produce WCET estimates close to and often tighter than traditional binary-level analysis, especially for complex programs. In comparison to simulation the source-level approach has shown its capability to deliver almost maximally tight estimates. The analysis time is currently inferior to traditional analysis approaches, but still in an acceptable range. In some cases, we were able to analyze programs for which traditional analyzers failed, and could only be applied by deactivating some of their precision-enhancing analyses. In other cases we have observed improvements of over 260%, caused by many infeasible paths that were not detected by binary-level analyzers. In average our source-level approach yields tighter estimates than traditional binary-level WCET analysis. We therefore conclude that source-level analysis can better exploit the program semantics, and yield improved estimates.

This source-level approach also conveniently circumvents the problem of identifying flow constraints and communicating them to the WCET analyzer, which traditionally is a tedious and error-prone task, and has become an entire research area on its own. These constraints are now mostly derived automatically by the model checker, and any user input that is required takes place at the familiar source-level, and is verified by the analysis. The proposed approach is therefore intuitive, reduces human error, and easier to automate.

Although the results are already very precise, we have identified further opportunities for improving the precision. These are mainly revolving around the mapping from instruction to source, which in our approach becomes the primary source for WCET overestimation. We proposed a sound target- and compiler-independent strategy to establish this mapping, which can tolerate moderate levels of optimization. Similarly to traditional binary-level analysis, the tightness of the WCET has been observed to deteriorate with rising levels of optimization, however in our experiments source-level analysis still had an advantage over the traditional approach. The data suggests that a graceful degradation of the tightness with rising levels of optimization can be expected.

Along with our source-level approach to WCET analysis, we have introduced a novel concept of "timing debugging", where developers literally can use a debugger to inspect the timing behavior along with the functional behavior. Specifically, we use an off-the-shelf debugger to dynamically evaluate the results from the verification phase to steer the program into the worst case, during which all capabilities from conventional debugging are available. This therefore provides an explanation of the WCET, including the associated inputs, helping developers to understand why a program is slow, yet without working through abstract traces or binary-level information from WCET analyzers. More generally, this method can also be applied to reconstruct complete counterexamples for any violated functional property, and make it interactively accessible in a debugger.

All methods presented here currently only cover in-order monoprocessors with set-associative caches, which do not make use of an operating system. This allows modeling a fair share of embedded processors, but is insufficient for the high-end segment of embedded systems. Similarly to traditional binary-level WCET analysis, such high-end processors are difficult to model, for they pose an insurmountable complexity challenge and are often not documented in sufficient detail; operating systems further aggravate these issues. One noticeable trend in real-time systems, both in industry and academia, are systems deliberately designed for predictability. Such systems are especially suitable for a source-level approach to timing analysis, since hardware models could be kept simple.

Last but not least, although we had initially considered source-level analysis as an alternative to traditional binary-level WCET analysis, some subtasks (e.g., control flow analysis) still require inspecting the binary in detail. As a consequence, source-level WCET analysis cannot replace the traditional approach completely. Instead, its existing methods can be re-used in our source-level approach, such that we circumvent some challenges in today's WCET analysis tools, and better exploit the program semantics, eventually suggesting a hybrid approach to WCET analysis as the best solution.

8.7.1 Open Problems

This thesis had focused on the main building blocks for a source-level timing analysis, but leaves a number of problems open for future work.

Implementation of Data Caches. A minor open problem is the implementation of a data cache model. Although the proposed instruction cache model can be carried over and thus

the model is not the main difficulty, the addition of data caches requires an instruction-level value analysis to determine the accessed memory addresses. Standard methods for this problem exist [WEE+08], but it is expected that they require user inputs. One study has shown that this can be mitigated by considering source-level information [LES+13], which would integrate naturally with our approach. However, the details of the such an implementation are not entirely clear, and therefore data caches remain an open problem.

Support for Branch Prediction. Since our approach mainly couples the microarchitectural models at instruction level to the source flow, there is no fundamental incompatibility with a branch predictor model. We have sketched such a model in Section 7.6. However, the implementation of such a model will again have an adverse impact on scalability, and likewise on precision. Since this ultimately requires a pipeline model which might change the structure of the proposed source-level models, this is left for future work.

Support for Library Calls. We have discussed the fundamental weakness of source-level analysis when the source code is not available in Section 8.5. Our flow difference encoding from the cache model could be used to model such calls soundly, but would not be able to deduce any logical flow facts, and thus require user inputs similar to traditional binary-level analysis. Hybrid source-binary approaches have been discussed and compared qualitatively, but need to be evaluated in practice, and in particular quantitatively.

8.7.2 Possible Extensions

The proposed source-level timing analysis suggests or opens up several new research directions, which could provide the developer with more non-functional feedback on the program.

Dynamic Testing of Temporal Properties. Since our approach makes the timing visible in the program, methods from general software testing could be applied to calculate input vectors that can approximate the timing *distribution*, instead of just the extreme values. Furthermore, testing could also be conducted to probabilistically explore the timing properties of a program, without the need for the target to be present.

Temporal and Functional Co-Analysis. The integration of functional and temporal analysis suggests at least two new research directions. First, we get the ability to write timing contracts in the program, which could be used for *time budgeting* of functions throughout software development. Second, one could investigate whether there exists a usable correlation between temporal behavior and software defects, which could lead to new methods for identifying functional defects through their timing behavior, and vice versa. Our experiments have already shown one program where such a connection was clearly given.

Back-annotation of other Non-functional Aspects. Exploiting the mapping that we already have established, we could further consider to back-annotate other properties from the machine instructions. One prime candidate is memory consumption, since many applications are hitting the memory wall, preventing them from fully exploiting the computational capabilities of the used processor. This might naturally integrate the analysis of defects like memory overflows and leaks, which are usually analyzed with another set of tools. A second interesting property might be power consumption. While this is even more intricate in terms of the required models and precision, it might integrate well with our approach.

Extension to High-level Models. While many embedded programs are likely developed by writing the code manually, some also make use of model-based design tools. Parts of the program are then generated from a higher-level model, e.g., a Simulink block diagram, which is subsequently compiled and linked together with the main program. Typically, such generators translate their models to C code, which obfuscates the model structure and may discard some information. Source-level timing analysis might however profit from high-level information, as it can provide loop bounds, type info, and so on. Transferring such properties from the model to the source code has been an active research area, and the benefits have been summarized in [AMR13]. An extension of our source-level model could be conceived that leverages such information for even tighter WCET estimates despite the use of abstractions. On a further note, it might be interesting to back-annotate our source-level timing to the higher-level model, to help parameterizing the code generation options.

Appendix A

Statistics for Source-Level WCET Analysis

***

Table A.1 gives the raw data for Figure 5.8 from page 102. The columns denoted as Δ represent the difference between the respective WCET estimates and simulation value, i.e., they give an upper bound of the tightness for both techniques.

Table A.2 gives detailed statistics on the memory usage and analysis time for source-level WCET estimation using MC, on the cache-less processor from Chapter 5. Table A.3 gives similar statistics for source-level WCET estimation using Frama-C's value analysis, which extends classic AI with superposed states.

Table A.1: Tightest WCET estimates for different analysis methods.

	based on Machine Instructions			based on Source Code				
	Simulation	ILP/IPET			Abs. Interpretation		Model Checking	
benchmark	observed	WCET	Δ%	stage	WCET	Δ%	WCET	Δ%
adpcm-decode	48,168	71,575	+48.6	●	⊤	–	69,673	+44.6
				◐	⊤	–	69,673	+44.6
				◖	⊤	–	69,673	+44.6
				◇	69,673	+44.6	69,673	+44.6
adpcm-encode	72,638	113,154	+55.8	●	⊤	–	110,902	+52.7
				◐	⊤	–	110,902	+52.7
				◖	⊤	–	110,902	+52.7
				◇	110,901	+52.7	110,902	+52.7
bs	401	496	+23.7	●	⊤	–	410	≈0
bsort100	788,766	1,553,661	+97.0	●	⊤	–	timeout	–
				◇	⊤	–	797,598	+1.1
cnt	8,502	8,564	≈ 0	●	⊤	–	8,564	≈0
				◐	⊤	–	8,564	≈0
				◇	⊤	–	8,564	≈0
crc	129,470	143,137	+10.6	●	⊤	–	130,114	≈0
				◐	⊤	–	130,114	≈0
				◖	⊤	–	130,114	≈0
				◇	⊤	–	143,426	+10.8
fdct	17,500	17,504	≈ 0	●	⊤	–	17,504	≈0
				◐	⊤	–	17,504	≈0
				◖	17,504	≈0	17,504	≈0
fibcall	1,777	1,781	≈ 0	●	⊤	–	1,780	≈0
				◐	⊤	–	1,780	≈0
				◖	1,780	≈0	1,780	≈0
fir	5,204,167	5,690,524	+9.3	●	⊤	–	timeout	–
				◐	⊤	–	timeout	–
				◖	⊤	–	5,476,023	+5.2
insertsort	5,472	5,476	≈ 0	●	⊤	–	5,476	≈0
jfdctint	14,050	14,054	≈ 0	●	⊤	–	14,054	≈0
				◐	⊤	–	14,054	≈0
				◖	14,054	≈0	14,054	≈0
matmult	1,010,390	1,010,394	≈ 0	●	⊤	–	1,010,394	≈0
				◐	⊤	–	1,010,394	≈0
				◖	⊤	–	1,010,394	≈0
ndes	459,967	470,499	≈ 0	●	⊤	–	timeout	–
				◐	⊤	–	timeout	–
				◇	⊤	–	465,459	≈0
ns	56,409	56,450	≈ 0	●	⊤	–	56,413	≈0
				◇	⊤	–	56,450	≈0
nsichneu	33,199	timeout	–	●	⊤	–	timeout	–
				◇	75,369	+127.0	75,369	+127.0
prime	27,702,943	30,343,092	+9.5	●	⊤	–	timeout	–
				◐	⊤	–	timeout	–
				◇	30,146,785	+8.8	30,146,785	+8.8
ud	35,753	93,487	+161.5	●	⊤	–	38,992	+9.1
				◐	⊤	–	38,992	+9.1
				◖	⊤	–	38,992	+9.1
PapaBench	16,438	53,780	+227.2	●	⊤	–	33,194	+101.9
				◐	⊤	–	33,194	+101.9
				◇	39,209	+138.5	33,762	+105.4

● instrumented, ◐ sliced, ◖ accelerated, ◇ abstracted, Δ upper bound for tightness

Table A.2: Solver statistics for WCET analysis using Model Checking.

benchmark	stage	fastest solver(s)	iter.	time[s]	mem [MB]	prog.size
adpcm-decode	●	C	3	7.4	356	3,537
	◐	C	3	2.8	120	2,416
	◗	C	3	2.9	121	2,134
	◇	C	3	2.5	123	1,65
adpcm-encode	●	C	3	290.6	688	4,911
	◐	C	3	10.4	168	4,313
	◗	C	3	8.5	173	3,986
	◇	C	3	2.6	136	1,832
bs	●	A, B, C, D, E, F	1	0.2	117	372
bsort100	●	timeout	1	timeout	11,572	$1.57{\cdot}10^5$
	◇	A, C	5	2.3	114	5,833
cnt	●	F	1	9.0	240	3,239
	◐	C	3	23.8	100	2,212
	◇	A, B, C, D, E, F	3	0.3	104	373
crc	●	A	3	4.6	407	41,269
	◐	C, D	3	4.7	367	40,519
	◗	A, C, D	3	4.1	268	39,684
	◇	A, B, C, D, E, F	3	1.2	103	11,013
fdct	●	A, B, C, D, E, F	3	0.6	101	989
	◐	A, B, C, D, E, F	3	0.3	96	182
	◗	A, B, C, D, E, F	3	0.3	97	78
fibcall	●	A, B, C, D, E, F	1	0.1	96	591
	◐	A, B, C, D, E, F	1	0.1	96	495
	◗	A, B, C, D, E, F	1	0.1	96	78
fir	●	timeout	–	timeout	6,718	–
	◐	timeout	–	timeout	16,322	–
	◗	A	5	18.5	1,574	50,487
insertsort	●	C	1	2.3	127	1,662
jfdctint	●	A, B, C, D, E, F	3	0.5	101	786
	◐	A, B, C, D, E, F	3	0.3	96	168
	◗	A, B, C, D, E, F	3	0.7	98	100
matmult	●	B	5	22.7	773	70,786
	◐	A	5	6.0	661	62,366
	◗	A, B, C, D, E	5	1.6	101	9,637
ndes	●	timeout	–	timeout	40,867	75,932
	◐	timeout	–	timeout	40,877	75,797
	◇	A, C	2	0.9	167	5,707
ns	●	C	3	28.3	827	22,398
	◇	C	3	6.5	200	8,271
nsichneu	●	timeout	–	timeout	25,491	23,244
	◇	A, B, C, D, E, F	3	0.5	120	84
prime	●	timeout	–	timeout	22,052	–
	◐	timeout	–	timeout	23,532	–
	◇	A, B, D, E, F	5	0.7	99	157
ud	●	D	3	1.2	101	2,381
	◐	A, B, C, D, E, F	3	0.4	98	1,880
	◗	A, B, C, D, E, F	3	0.4	99	1,375
PapaBench	●	C	3	561.0	364	10,753
	◐	C	3	523.0	352	8,963
	◇	C	3	193.8	351	11,585

Step: ● instrumented, ◐ sliced, ◗ accelerated, ◇ abstracted
Solvers: A=minisat, B=mathsat, C=yices, D=z3, E=cvc4, F=glucose

Table A.3: WCET estimates using all of Frama-C's features on the time-annotated source code.

benchmark	simulation	stage	time[s]	mem [MB]	WCET	Δ%
adpcm-decode	48,168	◇	1.1	63.4	69,673	44.6
		◗	1.1	60.8	69,673	44.6
		●	1.3	88.5	69,673	44.6
		◐	1.1	60.6	69,673	44.6
adpcm-encode	72,638	◇	1.1	65.3	110,902	52.7
		◗	2.0	89.0	110,902	52.7
		●	1.6	91.6	110,902	52.7
		◐	2.1	87.9	110,902	52.7
bs	401	●	0.7	53.9	410	2.2
bsort100	788,766	◇	1.0	61.2	797,598	1.1
		●	58.4	128.4	794,312	0.7
cnt	8,502	◇	0.7	51.9	8,564	0.7
		●	0.6	55.7	8,564	0.7
		◐	0.9	54.7	8,564	0.7
crc	129,470	◇	0.8	61.5	143,426	10.8
		◗	2.1	94.8	130,114	0.5
		●	2.3	96.2	130,114	0.5
		◐	2.1	94.8	130,114	0.5
fdct	17,500	◗	0.6	51.5	17,504	0.0
		●	0.7	55.4	17,504	0.0
		◐	0.7	51.7	17,504	0.0
fibcall	1,777	◗	0.6	51.5	1,780	0.2
		●	0.5	52.3	1,780	0.2
		◐	0.6	51.9	1,780	0.2
fir	5,204,167	◗	1.8	85.2	5,476,023	5.2
		●	10.5	190.8	5,476,023	5.2
		◐	7.9	159.9	5,476,023	5.2
insertsort	5,472	●	2.0	65.7	5,476	0.1
jfdctint	14,050	◗	0.6	51.6	14,054	0.0
		●	0.7	55.3	14,054	0.0
		◐	0.6	51.5	14,054	0.0
matmult	1,010,390	◗	0.9	63.5	1,010,394	0.0
		●	2.1	84.8	1,010,394	0.0
		◐	2.8	84.7	1,010,394	0.0
ndes	459,967	◇	1.5	80.4	465,459	1.2
		●	74.4	150.1	460,740	0.2
		◐	99.7	124.6	460,740	0.2
ns	56,409	◇	0.9	59.6	56,450	0.1
		●	2.5	60.6	56,450	0.1
nsichneu	33,199	◇	0.9	55.7	75,369	127.0
		●	596.6	253.1	timeout	–
prime	27,702,943	◇	0.5	52.1	30,146,785	8.8
		●	594.5	339.3	timeout	–
		◐	593.4	338.7	timeout	–
ud	35,753	◗	0.7	54.3	38,992	9.1
		●	0.8	56.7	38,992	9.1
		◐	0.7	54.2	38,992	9.1
PapaBench	16,438	◇	45.8	125.2	39,089†	137.8†
		●	1.6	93.4	40,460†	146.1†
		◐	1.3	69.1	40,460	146.1

● instrumented, ◐ sliced, ◗ accelerated, ◇ abstracted, Δ upper bound for tightness
† requires user annotations

Source-Level Encoding of Cache Model

***

B.1 Set-wise Encoding

Listings B.1 and B.2 show the used source-level model for the benchmark *crc*, using a set-wise, array-based encoding for the instruction cache.

Listing B.1: Header file for set-wise encoding.

```
1  /* Source-level cache model for entry function 'icrc', computed at 2019-04-19
       20:09:21.738785 */
2  #ifndef __CACHE_H
3  #define __CACHE_H
4  /* cache config: {'row_bits': 4, 'block_bits': 4, 'line_size': 16, 'lines': 32, '
       replace': 'LRU', 'way_bits': 1, 'assoc': 2, 'offsetbits': 4, 'sets': 16, '
       miss_penalty': 10, 'size': 512} */
5  /* encoding: set-wise */
6  #include "_tic.h"
7
8  /**********
9   * macros:
10  **********/
11 /* cache params: */
12 #define ASSOC 2
13 /* timing macros: */
14 #define MISS (_miss++, TIC(10))
15 #define ACCESS _access++
16 /* data types: */
17 typedef unsigned long addr;
18 #ifndef bool
19 #define bool __CPROVER_bool
20 #endif
21 bool _nondet(void);
22 extern unsigned long _access, _miss;
23 extern int _age;
24 /* cache macros: */
25 #define CACHED(aset,blk) _access_set##aset(blk)
26 #define UPDATE(cset,block) /* integrated in CACHED */
27 #define LINE_EMPTY 0
28
29 /************************
30  * models for set update:
31  ************************/
32 bool _access_set0(addr);
33 /* ... */
34 bool _access_set15(addr);
35
36 /***********************
37  * called by source BBs:
38  ***********************/
39 void _cache_icrc_sb1(void);
40 /* ... */
41
42 #endif
```

Listing B.2: Body file for set-wise encoding.

```
/* Source-level cache model for entry function 'icrc', computed at 2019-04-19
     20:09:21.738785 */
#include "_cache.h"
/* cache statistics counters: */
unsigned long _access=0, _miss=0;
int _age = 0;

/***************************
 * variables for set states:
 ***************************/
static addr cset0[ASSOC] = {LINE_EMPTY, LINE_EMPTY};
static addr cset1[ASSOC] = {LINE_EMPTY, LINE_EMPTY};
/* ... */

/************************
 * models for set update:
 ************************/
bool _access_set0(addr blk){
  ACCESS;
  addr pre=blk;

  if(cset0[0] == blk) return 1;
  for (unsigned a=0; a<ASSOC; ++a) {
    addr tmp = cset0[a];
    cset0[a] = pre;
    if (tmp == blk) return 1;
    pre = tmp;
  }
  return 0;
}

/* ... */

bool _access_set15(addr blk){
  ACCESS;
  addr pre=blk;

  if(cset15[0] == blk) return 1;
  for (unsigned a=0; a<ASSOC; ++a) {
    addr tmp = cset15[a];
    cset15[a] = pre;
    if (tmp == blk) return 1;
    pre = tmp;
  }
  return 0;
}

/************************
 * called by source BBs:
 ************************/
void _cache_icrc_sb1(void) {
{ /* bb18: */
  if (!CACHED(11,34224)) { MISS; } UPDATE(11,34224);
  if (!CACHED(12,34240)) { MISS; } UPDATE(12,34240);
}
}

/* ... */

void _cache_icrc_sb17(void) {
unsigned _bb7=0, _bb20=0, _bb10=0; /* BB count */
unsigned _e7_20, _e20_10, _e7_20, _e20_10; /* edge count */
if (_nondet()) { /* linearized subflow: */
{ /* bb7: */
  _bb7 += 1;
  if (!CACHED(2,33568)) { MISS; } UPDATE(2,33568);
  if (!CACHED(3,33584)) { MISS; } UPDATE(3,33584);
```

```
67    if (!CACHED(4,33600)) { MISS; } UPDATE(4,33600);
68  }
69  { /* bb20: */
70    _bb20 += 1;
71    if (!CACHED(4,33600)) { MISS; } UPDATE(4,33600);
72  }
73  { /* bb10: */
74    _bb10 += 1;
75    if (!CACHED(4,33600)) { MISS; } UPDATE(4,33600);
76    if (!CACHED(5,33616)) { MISS; } UPDATE(5,33616);
77    if (!CACHED(6,33632)) { MISS; } UPDATE(6,33632);
78    if (!CACHED(7,33648)) { MISS; } UPDATE(7,33648);
79    if (!CACHED(8,33664)) { MISS; } UPDATE(8,33664);
80    if (!CACHED(9,33680)) { MISS; } UPDATE(9,33680);
81    if (!CACHED(10,33696)) { MISS; } UPDATE(10,33696);
82    if (!CACHED(11,33712)) { MISS; } UPDATE(11,33712);
83  }
84  /* flow constraints: */
85  __CPROVER_assume(_bb7 == _e7_20);
86  __CPROVER_assume(_bb20 == _e7_20);
87  __CPROVER_assume(_bb20 == _e20_10);
88  __CPROVER_assume(_bb10 == _e20_10);
89  }
90  }
91
92  /* ... */
```

B.2 Block-Wise Encoding

Listings B.3 and B.4 shows an excerpt of the instruction cache source model for the benchmark *crc*, using an block-wise encoding.

Listing B.3: Header file for block-wise encoding.

```
1  /* Source-level cache model for entry function 'icrc', generated at 2019-05-15
       20:13:20.575921 */
2  #ifndef __CACHE_H
3  #define __CACHE_H
4  /* cache config: total=512B, lines=32, linesize=16B, assoc=2, sets=16, policy=LRU,
       miss penalty=10, raw config={'way_bits': 1, 'row_bits': 4, 'block_bits': 4, '
       miss_penalty': 10, 'replace': 'LRU'} */
5  /* encoding: block-wise */
6  /* dialect: cbmc */
7  /* approx. unclassified: False */
8  /* approx. first-miss: False */
9
10 #include "_tic.h"
11 /* cache params: */
12 #define ASSOC 2
13 /* timing macros: */
14 #define MISS TIC(10)
15 #define ACCESS 0
16 /* data types: */
17 typedef unsigned long addr;
18 #ifndef bool
19 #define bool __CPROVER_bool
20 #endif
21 bool _nondet(void);
22 extern int _age;
23
24 /*****************
25  * cache macros:
26  *****************/
27 #define EVICTED 2
28 #define BLOCKVAR(block) _blk_##block
29 #define BLOCKVAR_CTX(block) _blk_##block##_ctx
```

```
30 #define FLUSH(cset,block,ctx) BLOCKVAR(block)=EVICTED
31 #define UPDATE(cset,block,ctx) do {\
32   unsigned char age = BLOCKVAR(block);\
33   BLOCKVAR(block)=0;\
34   _age_set##cset(age);\
35   BLOCKVAR(block)=0;\
36 } while (0)
37 #define CACHED(cset,block,ctx) (BLOCKVAR(block) < EVICTED)
38
39 /**************************
40  * Source block models
41  **************************/
42 void _age_set0(unsigned char);
43 /* ... */
44 void _age_set15(unsigned char);
45
46 /**************************
47  * State variables
48  **************************/
49 extern unsigned char BLOCKVAR(0x8300), BLOCKVAR(0x8500), BLOCKVAR(0x8400); /* cache
       set 0 */
50 /* ... */
51 extern unsigned char BLOCKVAR(0x84f0), BLOCKVAR(0x82f0), BLOCKVAR(0x83f0); /* cache
       set 15 */
52
53 /**************************
54  * Source block models
55  **************************/
56 void _cache_icrc1_sb1(void);
57 /* ... */
58
59 #endif
```

Listing B.4: Body file for block-wise encoding.

```
1  /* Source-level cache model for entry function 'icrc', generated at 2019-05-15
       20:13:20.575921 */
2  #include "_cache.h"
3  int _age = 0;
4
5  /**************************
6   * State variables
7   **************************/
8  unsigned char BLOCKVAR(0x8300)=EVICTED, BLOCKVAR(0x8500)=EVICTED, BLOCKVAR(0x8400)=
       EVICTED; /* cache set 0 */
9  /* ... */
10 unsigned char BLOCKVAR(0x84f0)=EVICTED, BLOCKVAR(0x82f0)=EVICTED, BLOCKVAR(0x83f0)=
       EVICTED; /* cache set 15 */
11
12 /**************************
13  * Models for set update
14  **************************/
15 void _age_set0(unsigned char thresh){
16   if(BLOCKVAR(0x8300) <= thresh && BLOCKVAR(0x8300) < EVICTED)
17     { BLOCKVAR(0x8300)++; }
18   if(BLOCKVAR(0x8500) <= thresh && BLOCKVAR(0x8500) < EVICTED)
19     { BLOCKVAR(0x8500)++; }
20   if(BLOCKVAR(0x8400) <= thresh && BLOCKVAR(0x8400) < EVICTED)
21     { BLOCKVAR(0x8400)++; }
22 }
23
24 /* ... */
25
26 void _age_set15(unsigned char thresh){
27   if(BLOCKVAR(0x84f0) <= thresh && BLOCKVAR(0x84f0) < EVICTED)
28     { BLOCKVAR(0x84f0)++; }
29   if(BLOCKVAR(0x82f0) <= thresh && BLOCKVAR(0x82f0) < EVICTED)
30     { BLOCKVAR(0x82f0)++; }
31   if(BLOCKVAR(0x83f0) <= thresh && BLOCKVAR(0x83f0) < EVICTED)
```

```
32      { BLOCKVAR(0x83f0)++; }
33  }
34
35  /**************************
36   * Called by source BBs:
37   **************************/
38  void _cache_icrc1_sb1(void) {
39  { /* icrc1.bb3 @82c4-82d4: */
40    TIC(5); /* insn time */
41    if (!CACHED(12,0x82c0,0x82c4)) { MISS; }
42    UPDATE(12,0x82c0,0x82c4); /* unknown: set 12, l-block 82c4 */
43    if (!CACHED(13,0x82d0,0x82d0)) { MISS; }
44    UPDATE(13,0x82d0,0x82d0); /* unknown: set 13, l-block 82d0 */
45  }
46  }
47
48  /* ... */
49
50  void _cache_icrc_sb17(void) {
51  unsigned _bb7=0, _bb20=0, _bb10=0; /* BB count */
52  unsigned e7_20, e20_10; /* edge count */
53  if (nondet()) { /* linearized subflow: */
54    { /* icrc.bb7 @8324-8340: */
55      _bb7 += 1;
56      TIC(8); /* insn time */
57      if (!CACHED(2,0x8320,0x8324)) { MISS; }
58      UPDATE(2,0x8320,0x8324); /* unknown: set 2, l-block 8324 */
59      if (!CACHED(3,0x8330,0x8330)) { MISS; }
60      UPDATE(3,0x8330,0x8330); /* unknown: set 3, l-block 8330 */
61      if (!CACHED(4,0x8340,0x8340)) { MISS; }
62      UPDATE(4,0x8340,0x8340); /* unknown: set 4, l-block 8340 */
63    }
64    { /* icrc.bb20 @8340-8340: */
65      _bb20 += 1;
66      TIC(1); /* insn time */
67      if (!CACHED(4,0x8340,0x8340)) { MISS; }
68      UPDATE(4,0x8340,0x8340); /* unknown: set 4, l-block 8340 */
69    }
70    { /* icrc.bb10 @8344-83b4: */
71      _bb10 += 1;
72      TIC(29); /* insn time */
73      if (!CACHED(4,0x8340,0x8344)) { MISS; }
74      UPDATE(4,0x8340,0x8344); /* unknown: set 4, l-block 8344 */
75      if (!CACHED(5,0x8350,0x8350)) { MISS; }
76      UPDATE(5,0x8350,0x8350); /* unknown: set 5, l-block 8350 */
77      if (!CACHED(6,0x8360,0x8360)) { MISS; }
78      UPDATE(6,0x8360,0x8360); /* unknown: set 6, l-block 8360 */
79      if (!CACHED(7,0x8370,0x8370)) { MISS; }
80      UPDATE(7,0x8370,0x8370); /* unknown: set 7, l-block 8370 */
81      if (!CACHED(8,0x8380,0x8380)) { MISS; }
82      UPDATE(8,0x8380,0x8380); /* unknown: set 8, l-block 8380 */
83      if (!CACHED(9,0x8390,0x8390)) { MISS; }
84      UPDATE(9,0x8390,0x8390); /* unknown: set 9, l-block 8390 */
85      if (!CACHED(10,0x83a0,0x83a0)) { MISS; }
86      UPDATE(10,0x83a0,0x83a0); /* unknown: set 10, l-block 83a0 */
87      if (!CACHED(11,0x83b0,0x83b0)) { MISS; }
88      UPDATE(11,0x83b0,0x83b0); /* unknown: set 11, l-block 83b0 */
89    }
90    /* flow constraints: */
91    __CPROVER_assume(_bb7 == _e7_20); /* bb7 out */
92    __CPROVER_assume(_bb20 == _e7_20); /* bb20 in */
93    __CPROVER_assume(_bb20 == _e20_10); /* bb20 out */
94    __CPROVER_assume(_bb10 == _e20_10); /* bb10 in */
95  }
96  }
```

B.3 First-Miss Approximation

Tables B.1 and B.2 quantify the difference between a source-level WCET analysis using binary-level microarchitectural invariants when modeling "first-miss" precisely, versus approximating "first-miss" by unconditionally paying the penalty when the associated scope is entered.

Table B.1: Best WCET estimates with and without first-miss approximation.

	FM-precise		FM-approx	
benchmark	WCET	Δ%	WCET	Δ%
adpcm	6,666	+2.5	6,686	+2.8
cnt	timeout	–	3,311	+4.7
crc	70,593	+32.6	73,263	+37.6
fdct	19,442	≈ 0	19,442	≈ 0
fibcall	**643**	**0**	**643**	**0**
jfdctint	18,278	≈ 0	18,278	≈ 0
insertsort	**1,679**	**0**	**1,679**	**0**
matmult	**335,133**	**0**	**335,133**	**0**
ns	77,557	+261.7	77,557	+261.7

Δ=overestimation w.r.t simlulation

Table B.2: Computational Effort for WCET estimation with and without first-miss approximation.

	FM-precise			FM-approx		
benchmark	time	mem	steps	time	mem	steps
adpcm	32.2	62.4	6,670	23.6	60.0	4,990
cnt	timeout	91.3	23,379	32.3	18.6	4,189
crc	19.6	109.3	90,527	18.0	121.7	67,998
fdct	2.5	14.9	3,969	2.5	14.8	3,969
fibcall	0.8	11.2	1,066	0.9	11.2	845
jfdctint	2.3	13.8	3,612	2.6	13.8	3,612
insertsort	0.9	9.5	1,653	1.0	9.0	1,073
matmult	56.0	305.1	195,627	45.2	186.9	113,769
ns	214.7	606.4	49,704	215.9	606.4	49,704

memory in MBytes, time in seconds

B.4 Annotated CFG from Cycle-accurate ARM simulator

```
<?xml version="1.0" encoding="UTF-8"?>
<cfg-collection>
  <cfg id="_0" address="_0" label="fib" number="0">
    <property identifier="otawa::INDEX">0</property>
    <property identifier="otawa::ipet::COUNT">1</property>
    <property identifier="otawa::ipet::ICACHE_MISSES">10</property>
    <property identifier="otawa::ipet::TOTAL_TIME">643</property>
```

```
8     <property identifier="otawa::ipet::WCET">643</property>
9     <entry id="_0-0"/>
10    <bb id="_0-1" address="0000821c" size="44" number="1">
11      <property identifier="otawa::ENTRY">fib</property>
12      <property identifier="otawa::INDEX">1</property>
13      <property identifier="otawa::ipet::COUNT">1</property>
14      <property identifier="otawa::ipet::ICACHE_MISSES">4</property>
15      <property identifier="otawa::ipet::TOTAL_TIME">51</property>
16      <property identifier="otawa::ipet::WCET">51</property>
17      <inst address="0000821c" file="fibcall.c" line="48"/>
18      <inst address="00008220" file="fibcall.c" line="48"/>
19      <inst address="00008224" file="fibcall.c" line="48"/>
20      <inst address="00008228" file="fibcall.c" line="48"/>
21      <inst address="0000822c" file="fibcall.c" line="51"/>
22      <inst address="00008230" file="fibcall.c" line="51"/>
23      <inst address="00008234" file="fibcall.c" line="51"/>
24      <inst address="00008238" file="fibcall.c" line="51"/>
25      <inst address="0000823c" file="fibcall.c" line="52"/>
26      <inst address="00008240" file="fibcall.c" line="52"/>
27      <inst address="00008244" file="fibcall.c" line="52"/>
28    </bb>
29    <bb id="_0-2" address="00008274" size="12" number="2">
30      <property identifier="otawa::INDEX">2</property>
31      <property identifier="otawa::ipet::COUNT">30</property>
32      <property identifier="otawa::ipet::ICACHE_MISSES">1</property>
33      <property identifier="otawa::ipet::TOTAL_TIME">100</property>
34      <property identifier="otawa::ipet::WCET">13</property>
35      <inst address="00008274" file="fibcall.c" line="52"/>
36      <inst address="00008278" file="fibcall.c" line="52"/>
37      <inst address="0000827c" file="fibcall.c" line="52"/>
38    </bb>
39    <bb id="_0-3" address="00008280" size="16" number="3">
40      <property identifier="otawa::INDEX">3</property>
41      <property identifier="otawa::ipet::COUNT">29</property>
42      <property identifier="otawa::ipet::ICACHE_MISSES">1</property>
43      <property identifier="otawa::ipet::TOTAL_TIME">126</property>
44      <property identifier="otawa::ipet::WCET">14</property>
45      <inst address="00008280" file="fibcall.c" line="53"/>
46      <inst address="00008284" file="fibcall.c" line="53"/>
47      <inst address="00008288" file="fibcall.c" line="53"/>
48      <inst address="0000828c" file="fibcall.c" line="53"/>
49    </bb>
50    <bb id="_0-4" address="00008290" size="28" number="4">
51      <property identifier="otawa::INDEX">4</property>
52      <property identifier="otawa::RETURN_OF">BB 1 (0000821c)</property>
53      <property identifier="otawa::ipet::COUNT">1</property>
54      <property identifier="otawa::ipet::ICACHE_MISSES">2</property>
55      <property identifier="otawa::ipet::TOTAL_TIME">27</property>
56      <property identifier="otawa::ipet::WCET">27</property>
57      <inst address="00008290" file="fibcall.c" line="60"/>
58      <inst address="00008294" file="fibcall.c" line="60"/>
59      <inst address="00008298" file="fibcall.c" line="61"/>
60      <inst address="0000829c" file="fibcall.c" line="62"/>
61      <inst address="000082a0" file="fibcall.c" line="62"/>
62      <inst address="000082a4" file="fibcall.c" line="62"/>
63      <inst address="000082a8" file="fibcall.c" line="62"/>
64    </bb>
65    <bb id="_0-5" address="00008248" size="44" number="5">
66      <property identifier="otawa::INDEX">5</property>
67      <property identifier="otawa::ipet::COUNT">29</property>
68      <property identifier="otawa::ipet::ICACHE_MISSES">2</property>
69      <property identifier="otawa::ipet::TOTAL_TIME">339</property>
70      <property identifier="otawa::ipet::WCET">31</property>
71      <inst address="00008248" file="fibcall.c" line="56"/>
72      <inst address="0000824c" file="fibcall.c" line="56"/>
73      <inst address="00008250" file="fibcall.c" line="57"/>
74      <inst address="00008254" file="fibcall.c" line="57"/>
75      <inst address="00008258" file="fibcall.c" line="57"/>
76      <inst address="0000825c" file="fibcall.c" line="57"/>
```

```
77        <inst address="00008260" file="fibcall.c" line="58"/>
78        <inst address="00008264" file="fibcall.c" line="58"/>
79        <inst address="00008268" file="fibcall.c" line="54"/>
80        <inst address="0000826c" file="fibcall.c" line="54"/>
81        <inst address="00008270" file="fibcall.c" line="54"/>
82      </bb>
83      <exit id="_0-6"/>
84      <edge kind="virtual-call" source="_0-0" target="_0-1">
85        <property identifier="otawa::CALLED_CFG">fib</property>
86      </edge>
87      <edge kind="taken" source="_0-1" target="_0-2"/>
88      <edge kind="not-taken" source="_0-2" target="_0-3"/>
89      <edge kind="taken" source="_0-2" target="_0-4"/>
90      <edge kind="not-taken" source="_0-3" target="_0-4"/>
91      <edge kind="taken" source="_0-3" target="_0-5"/>
92      <edge kind="virtual-return" source="_0-4" target="_0-6">
93        <property identifier="otawa::CALLED_CFG">fib</property>
94      </edge>
95      <edge kind="not-taken" source="_0-5" target="_0-2"/>
96    </cfg>
97  </cfg-collection>
```

Bibliography

***

[ABC+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[ABRF15] Omar Al-Bataineh, Mark Reynolds, and Tim French. Accelerating worst case execution time analysis of timed automata models with cyclic behaviour. *Formal Aspects of Computing*, 27(5):917–949, 2015.

[Ada15] AdaCore. Ada Drivers Library, 2015. Available online at `https://github.com/AdaCore`.

[AEF+14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.*, 13(4):82:1–82:37, 2014.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In Radhia Cousot and David A. Schmidt, editors, *Proc. Static Analysis, International Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1996.

[AGLS16] Peter Altenbernd, Jan Gustafsson, Björn Lisper, and Friedhelm Stappert. Early execution time-estimation through automatically generated timing models. *Real-Time Systems*, 52(6):731–760, 2016.

[AHP+14] Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quiñones, and Francisco J. Cazorla. On the comparison of deterministic and probabilistic WCET estimation techniques. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 266–275. IEEE Computer Society, 2014.

[AHQ+15] Jaume Abella, Carles Hernández, Eduardo Quiñones, Francisco J. Cazorla, Philippa Ryan Conmy, Mikel Azkarate-askasua, Jon Pérez, Enrico Mezzetti, and Tullio Vardanega. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *Proc. Symposium on Industrial Embedded Systems (SIES)*, pages 39–48. IEEE, 2015.

[AMR13] Mihail Asavoae, Claire Maiza, and Pascal Raymond. Program semantics in model-based WCET analysis: A state of the art perspective. In Claire Maiza, editor, *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 30 of *OASICS*, pages 32–41. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

[ARM16] ARM Ltd. *Cortex-A57 Software Optimization Guide*, ARM UAN 0015B edition, January 2016.

[ATF09] Wasif Afzal, Richard Torkar, and Robert Feldt. A systematic review of search-based testing for non-functional system properties. *Information & Software Technology*, 51(6):957–976, 2009.

[BBGF16] Paul E Black, Lee Badger, Barbara Guttman, and Elizabeth Fong. Dramatically reducing software vulnerabilities. *National institute of Standards and Technology*, 2016.

[BBMZ16] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, pages 470–481. IEEE Computer Society, 2016.

[BC08] Clément Ballabriga and Hugues Cassé. Improving the first-miss computation in set-associative instruction caches. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 341–350. IEEE Computer Society, 2008.

[BC11] Jean-Luc Béchennec and Franck Cassez. Computation of WCET using program slicing and real-time model-checking. *CoRR*, abs/1105.1633, 2011.

[BC16] Carl Brandon and Peter Chapin. The Use of SPARK in a Complex Spacecraft. *HILT*, 2016.

[BCC+16] Armelle Bonenfant, Fabienne Carrier, Hugues Cassé, Philippe Cuenot, Denis Claraz, Nicolas Halbwachs, Hanbing Li, Claire Maiza, Marianne De Michiel, Vincent Mussot, et al. When the worst-case execution time estimation gains from the application semantics, 2016.

[BCR13] Abhijeet Banerjee, Sudipta Chattopadhyay, and Abhik Roychoudhury. Precise micro-architectural modeling for WCET analysis via AI+SAT. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 87–96. IEEE Computer Society, 2013.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: an open toolbox for adaptive WCET analysis. In Sang Lyul Min, Robert G. Pettit IV, Peter P. Puschner, and Theo Ungerer, editors, *Proc. Software Technologies for Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010.

[BDM+07] Guillem Bernat, Robert Davis, Nick Merriam, John Tuffen, Andrew Gardner, Michael Bennett, and Dean Armstrong. Identifying opportunities for worst-case execution time reduction in an avionics system. *Ada User Journal*, 28(3):189–195, 2007.

[BEG+15] Oliver Bringmann, Wolfgang Ecker, Andreas Gerstlauer, Ajay Goyal, Daniel Müller-Gritschneder, Prasanth Sasidharan, and Simranjit Singh. The next generation of virtual prototyping: ultra-fast yet accurate simulation of HW/SW systems. In Nebel and Atienza [NA15], pages 1698–1707.

[Bey14] Dirk Beyer. Status report on software verification. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 373–388. Springer Berlin Heidelberg, 2014.

[BGP09] Aimen Bouchhima, Patrice Gerin, and Frédéric Pétrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In K. Wakabayashi, editor, *Proc. Asia South Pacific Design Automation Conference*, pages 546–551. IEEE, 2009.

[BHJ12] Florian Brandner, Stefan Hepp, and Alexander Jordan. Static profiling of the worst-case in real-time programs. In Liliana Cucu-Grosjean, Nicolas Navet, Christine Rochange, and James H. Anderson, editors, *Proc. International Conference on Real-Time and Network Systems (RTNS)*, pages 101–110. ACM, 2012.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[BJAS11] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A binary analysis platform. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 463–469. Springer, 2011.

[BK11] Doina Bucur and Marta Z. Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):1693–1707, 2011.

[BLH14] Bernard Blackham, Mark H. Liffiton, and Gernot Heiser. Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 169–178. IEEE Computer Society, 2014.

[BLSW13] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, pages 353–368, 2013.

[BMP14] Sandrine Blazy, André Oliveira Maroneze, and David Pichardie. Formal verification of loop bound estimation for WCET analysis. In Ernie Cohen and Andrey Rybalchenko, editors, *Proc. International Conference on Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *Lecture Notes in Computer Science*, pages 281–303. Springer, 2014.

[Bur99] Alan Burns. The Ravenscar Profile. *ACM SIGAda Ada Letters*, 19(4):pp. 49–52, 1999.

[C+10] DWARF Debugging Information Format Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

[Cas11] Franck Cassez. Timed games for computing WCET for pipelined processors with caches. In Benoît Caillaud, Josep Carmona, and Kunihiko Hiraishi, editors, *Proc. International Conference on Application of Concurrency to System Design (ACSD)*, pages 195–204. IEEE Computer Society, 2011.

[CB02] Antoine Colin and Guillem Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, page 50. IEEE Computer Society, 2002.

[CBW96] Roderick Chapman, Alan Burns, and Andy J. Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R.M. Graham et al., editors, *Proc. Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreÉ analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer Berlin Heidelberg, 2005.

[CCR+14] 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.

[CdAJ17] Franck Cassez, Pablo González de Aledo, and Peter Gjøl Jensen. WUPPAAL: computation of worst-case execution-time for binary programs with UPPAAL. In Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare, editors, *Models, Algorithms, Logics and Tools*, volume 10460 of *Lecture Notes in Computer Science*, pages 560–577. Springer, 2017.

[CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[CHK+15] Pavol Cerný, Thomas A. Henzinger, Laura Kovács, Arjun Radhakrishna, and Jakob Zwirchmayr. Segment abstraction for worst-case execution time analysis. In Jan Vitek, editor, *Proc. European Symposium on Programming*, volume 9032 of *Lecture Notes in Computer Science*, pages 105–131. Springer, 2015.

[CHO12] Franck Cassez, René Rydhof Hansen, and Mads Chr. Olesen. What is a timing anomaly? In Vardanega [Var12], pages 1–12.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - A software analysis perspective. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Proc. Software Engineering and Formal Methods (SEFM)*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2012.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKNZ11] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011.

[CKOS04] Edmund M. Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and complexity of bounded model checking. In Steffen and Levi [SL04], pages 85–96.

[Cla19] The Clang Team. *Clang compiler User's Manual*, 2019. Optimization Reports.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, 2000.

[CPHL01] Siddhartha Chatterjee, Erin Parker, Philip J. Hanlon, and Alvin R. Lebeck. Exact analysis of the cache behavior of nested loops. In Michael Burke and Mary Lou Soffa, editors, *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 286–297. ACM, 2001.

[CQV+13] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94, 2013.

[CR79] Stephen A. Cook and Robert A. Reckhow. The relative efficiency of propositional proof systems. *J. Symb. Log.*, 44(1):36–50, 1979.

[CR13] Sudipta Chattopadhyay and Abhik Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems*, 49(4):517–562, 2013.

[CS14] Roderick Chapman and Florian Schanda. Are we there yet? 20 Years of industrial theorem proving with SPARK. *LLCS*, Vol. 8558(March 1987):17–26, 2014.

[CSH+12] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In Robert Davis, editor, *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101. IEEE Computer Society, 2012.

[CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CYL+16] Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perrelle, and Virgile Prevosto. *Frama-C's value analysis plug-in*. CEA LIST, Software Reliability Laboratory, 2016. Aluminium-20160501.

[CZG13] Suhas Chakravarty, Zhuoran Zhao, and Andreas Gerstlauer. Automated, retargetable back-annotation for host compiled performance and power modeling. In *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, 2013.

[DBL11] *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE Computer Society, 2011.

[DCV+15] Priyanka Darke, Bharti Chimdyalwar, R. Venkatesh, Ulka Shrotri, and Ravindra Metta. Over-approximating loops to prove properties using bounded model checking. In Nebel and Atienza [NA15], pages 1407–1412.

[DEL+14] Claire Dross, Pavlos Efstathopoulos, David Lesens, David Mentr, David Mentré, and Yannick Moy. Rail, Space, Security: Three Case Studies for SPARK 2014. *ERTS2 2014*, pages 1–10, 2014.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[DOT+10] Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: modular execution time analysis using model checking. In Lisper [Lis10], pages 113–123.

[DPVZ15] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. In Daniel Kroening and Corina S. Pasareanu, editors, *Proc. 27th International Conference on Computer Aided Verification (CAV)*, volume 9206 of *Lecture Notes in Computer Science*, pages 561–579. Springer, 2015.

[Dre07] Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, Vol.11, 2007.

[EFGA09] Andreas Ermedahl, Johan Fredriksson, Jan Gustafsson, and Peter Altenbernd. Deriving the worst-case execution time input values. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 45–54. IEEE Computer Society, 2009.

[EJ02] Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static WCET analysis. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis, editors, *Proc. Embedded Software (EMSOFT)*, volume 2491 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2002.

[ESE05] Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. Clustered worst-case execution-time calculation. *IEEE Trans. Computers*, 54(9):1104–1122, 2005.

[ESG+07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Rochange [Roc07].

[EY97] Rolf Ernst and Wei Ye. Embedded program timing analysis based on path clustering and architecture classification. In Ralph H. J. M. Otten and Hiroto Yasuura, editors, *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 598–604. IEEE Computer Society / ACM, 1997.

[FBvHS16] Insa Fuhrmann, David Broman, Reinhard von Hanxleden, and Alexander Schulz-Rosengarten. Time for reactive system modeling: Interactive timing analysis with hotspot highlighting. In Alain Plantec, Frank Singhoff, Sébastien Faucou, and Luís Miguel Pinho, editors, *Proc. International Conference on Real-Time Networks and Systems (RTNS)*, pages 289–298. ACM, 2016.

[FFL+11] Ricardo Bedin França, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Towards formally verified optimizing compilation in flight control software. In P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, editors, *Proc. Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop*, volume 18 of *OASICS*, pages 59–68. Schloss Dagstuhl, Germany, 2011.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *Prod. Embedded Software, International Workshop (EMSOFT)*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[FHLS+08] Christian Ferdinand, Reinhold Heckmann, Thierry Le Sergent, D Lopes, B Martin, X Fornari, and F Martin. Combining a high-level design tool for safety-critical systems with a tool for WCET analysis of executables. In *Proc. European Congress on Embedded Real Time Software (ERTS)*. SIA/AAAF/SEE, 2008.

[Fil11] Jean-Christophe Filliâtre. Deductive software verification. *STTT*, 13(5):397–403, 2011.

[Fog18] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. online, Aug 2018. Retrieved 2018 Sept 3rd.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3: Where Programs Meet Provers, 2013.

[Fre19] Free Software Foundation. *Using the GNU Compiler Collection (GCC)*, 2019. GCC Developer Options.

[FW99] Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[GAC+13] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Mariana Molnos, Ashkan Beyranvand Nejad, Andrew Nelson, and Shubhendu Sinha. Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. *SIGBED Review*, 10(3):23–34, 2013.

[GBEL10] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen WCET benchmarks: Past, present and future. In Lisper [Lis10], pages 136–146.

[GELP10] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Lisper [Lis10], pages 101–112.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proc. Real-Time Systems Symposium*, pages 57–66. IEEE Computer Society, 2006.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 375–385. ACM, 2009.

[Gmb19] AbsInt Angewandte Informatik GmbH. Website, 2019. `https://www.absint.com/ait/`, retrieved 2016-May-23.

[Göd31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i. *Monatshefte für mathematik und physik*, 38(1):173–198, 1931.

[HAMM14] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maiza. How to compute worst-case execution time by optimization modulo theory and a clever encoding of program semantics. In Youtao Zhang and Prasad Kulkarni, editors, *Proc. Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, pages 43–52. ACM, 2014.

[Hav97] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[HDZ00] John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[HGB+08] Niklas Holsti, Jan Gustafsson, Guillem Bernat, Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET 2008 – Report from the Tool Challenge 2008 – 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. In Raimund Kirner, editor, *8th International Workshop on Worst-Case Execution Time Analysis (WCET'08)*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-237-3.

[HJR11] Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. Scope-aware data cache analysis for WCET estimation. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)* [DBL11], pages 203–212.

[HK07] Trevor Harmon and Raymond Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *Proc. International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 209–216. IEEE Computer Society, 2007.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[HMWC15] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and GNATprove. *International Journal on Software Tools for Technology Transfer*, 17(6):695–707, 2015.

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, 2012.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa03] Charles Antony Richard Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, January 2003.

[Hol08] Niklas Holsti. Computing time as a program variable: a way around infeasible paths. In Raimund Kirner, editor, *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 8 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[HPP12] Benedikt Huber, Daniel Prokesch, and Peter P. Puschner. A formal framework for precise parametric WCET formulas. In Vardanega [Var12], pages 91–102.

[HRP17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The heptane static worst-case execution time estimation tool. In Reineke [Rei17], pages 8:1–8:12.

[HS02] Niklas Holsti and Sami Saarinen. Status of the Bound-T WCET tool. *Space Systems Finland Ltd*, 2002.

[HSK+12] Trevor Harmon, Martin Schoeberl, Raimund Kirner, Raymond Klefstad, Kwang-Hae (Kane) Kim, and Michael R. Lowry. Fast, interactive worst-case execution time analysis with back-annotation. *IEEE Trans. Industrial Informatics*, 8(2):366–377, 2012.

[HSR+00] Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[Ind18] TIOBE Index. Tiobe-the software quality company. *TIOBE Index— TIOBE–The Software Quality Company [Electronic resource]. Mode of access: https://www. tiobe. com/tiobe-index/-Date of access*, 1, 2018.

[JHM+18] Erwan Jahier, Nicolas Halbwachs, Claire Maiza, Pascal Raymond, Wei-Tsun Sun, and Hugues Cassé. Assessing software abstractions in wcet analysis of reactive programs 3. Technical report, Verimag, February 2018. Research Report TR-2018-2.

[JHRC12] Lei Ju, Bach Khoa Huynh, Abhik Roychoudhury, and Samarjit Chakraborty. Performance debugging of esterel specifications. *Real-Time Systems*, 48(5):570–600, 2012.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Proc. Computer Aided Verification*, volume 5643, pages 661–667. Springer, 2009.

[KHR+96] Lo Ko, Christopher A. Healy, Emily Ratliff, Robert D. Arnold, David B. Whalley, and Marion G. Harmon. Supporting the specification and analysis of timing constraints. In *Proc. 2nd Real-Time Technology and Applications Symposium (RTAS)*, pages 170–178. IEEE Computer Society, 1996.

[KKBC12] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. In *Proc. Conference on Programming Language Design and Implementation*, PLDI '12, pages 193–204, New York, NY, USA, 2012. ACM.

[KKP+07] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET analysis: The annotation language challenge. In Rochange [Roc07].

[KKP+11] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 10(3):411–437, 2011.

[KKZ12] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. Symbolic loop bound computation for WCET analysis. In Edmund M. Clarke, Irina Virbitskaite, and Andrei Voronkov, editors, *Proc. Perspectives of Systems Informatics (PSI), Revised Selected Papers*, volume 7162 of *Lecture Notes in Computer Science*, pages 227–242. Springer, 2012.

[Kli91] George J. Klir. *Facets of systems science*. Springer, 1991. pp.121-128.

[KMMS16] Nikolai Kosmatov, Claude Marché, Yannick Moy, and Julien Signoles. Static versus dynamic verification in Why3, Frama-C and SPARK 2014. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9952 LNCS:461–478, 2016.

[KOS+11] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. Linear completeness thresholds for bounded model checking. In *International Conference on Computer Aided Verification*, pages 557–572. Springer, 2011.

[KP08] Raimund Kirner and Peter P. Puschner. Obstacles in worst-case execution time analysis. In *Proc. International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 333–339. IEEE Computer Society, 2008.

[KPE09] Sungjun Kim, Hiren D. Patel, and Stephen A. Edwards. Using a model checker to determine worst-case execution time. Technical report, Columbia University, 2009. CUCS-038-09.

[KPP10] Raimund Kirner, Peter P. Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1-2):72–105, 2010.

[Kuh98] Claus Kuhnel. *AVR RISC Microcontroller Handbook*. Newnes, first edition, October 1998.

[KW92] Atty Kanamori and Daniel Weise. An empirical study of an abstract interpretation of scheme programs. Technical report, Stanford University, Stanford, USA, 1992.

[KYAR10] Matthew M Y Kuo, Li Hsien Yoong, Sidharta Andalam, and Partha S. Roop. Determining the worst-case reaction time of IEC 61499 function blocks. In *Proc. International Conference on Industrial Informatics*, pages 1104–1109. IEEE, 2010.

[Lat08] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[LES+13] Björn Lisper, Andreas Ermedahl, Dietmar Schreiner, Jens Knoop, and Peter Gliwa. Practical experiences of applying source-level WCET flow analysis to industrial code. *Journal on Software Tools for Technology Transfer*, 15(1):53–63, 2013.

[Lew17] Brandon Lewis. 2017 embedded processor report: At the edge of Moore's law and IoT. *Embedded Computing Design*, 15.1:pp.10–12, Jan/Feb 2017.

[LGG+08] Mingsong Lv, Zonghua Gu, Nan Guan, Qingxu Deng, and Ge Yu. Performance comparison of techniques on static path analysis of WCET. In Cheng-Zhong Xu and Minyi Guo, editors, *Proc. International Conference on Embedded and Ubiquitous Computing (EUC)*, pages 104–111. IEEE Computer Society, 2008.

[Lis03] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. *WCET*, 3:77–80, 2003.

[Lis10] Björn Lisper, editor. *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 15 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[LLK+08] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES)*, pages 137–146. ACM, 2008.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007.

[LM97] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, 1997.

[LMS12] Kun Lu, Daniel Müller-Gritschneder, and Ulf Schlichtmann. Hierarchical control flow matching for source-level simulation of embedded software. In *Proc. International Symposium on System on Chip (ISSOC)*, pages 1–5. IEEE, 2012.

[LMW95] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proc. Real-Time Systems Symposium*, pages 298–307. IEEE Computer Society, 1995.

[LR09] Akash Lal and Thomas Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design*, 35:73–93, 2009.

[LRM06] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227, 2006.

[LS98] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In Frank Mueller and Azer Bestavros, editors, *Proc. Languages, Compilers, and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998.

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[Ltd08] Arm Ltd. *ARM7TDMI Data Sheet*, Doc ARM IHI 0042B edition, April 2008. ABI release 2.06.

[Ltd19] ARM Ltd. Cortex-r7. Online at `https://www.arm.com/products/silicon-ip-cpu/cortex-r/cortex-r7`, 2019. Retrieved 2019-May-30.

[Mar11] Amine Marref. Fully-automatic derivation of exact program-flow constraints for a tighter worst-case execution-time analysis. In Luigi Carro and Andy D. Pimentel, editors, *Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 200–208. IEEE, 2011.

[Met04] Alexander Metzner. Why model checking can improve WCET analysis. In Rajeev Alur and Doron A. Peled, editors, *Proc. 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer, 2004.

[MG19] Daniel Müller-Gritschneder. *Advanced Virtual Prototyping and Communication Synthesis for Integrated System Design at Electronic System Level*. habilitation, Technical University of Munich, 2019.

[MGG17] Daniel Müller-Gritschneder and Andreas Gerstlauer. Host-compiled simulation. *Handbook of Hardware/Software Codesign*, pages 1–27, 2017.

[MLS11] Daniel Müller-Gritschneder, Kun Lu, and Ulf Schlichtmann. Control-flow-driven source level timing annotation for embedded software models on transaction level. In *Proc. Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 600–607. IEEE Computer Society, 2011.

[MP18] Omayma Matoussi and Frédéric Pétrot. A mapping approach between IR and binary cfgs dealing with aggressive compiler optimizations for performance estimation. In Youngsoo Shin, editor, *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 452–457. IEEE, 2018.

[MPC+15] Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet(m) estimation in multi-core systems using single core equivalence. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 174–183. IEEE Computer Society, 2015.

[MRP+17] Claire Maiza, Pascal Raymond, Catherine Parent-Vigouroux, Armelle Bonenfant, Fabienne Carrier, Hugues Cassé, Philippe Cuenot, Denis Claraz, Nicolas Halbwachs, Erwan Jahier, Hanbing Li, Marianne De Michiel, Vincent Mussot, Isabelle Puaut, Erven Rohou, Jordy Ruiz, Pascal Sotin, and Wei-Tsun Sun. The W-SEPT project: Towards semantic-aware WCET estimation. In Reineke [Rei17], pages 9:1–9:13.

[MRR+19] Claire Maiza, Hamza Rihani, Juan Maria Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3):56:1–56:38, 2019.

[MRS+16] Vincent Mussot, Jordy Ruiz, Pascal Sotin, Marianne De Michiel, and Hugues Cassé. Expressing and exploiting conflicts over paths in WCET analysis. In Martin Schoeberl, editor, *Proc. International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 55 of *OASICS*, pages 3:1–3:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[MSSL08] Trevor Meyerowitz, Alberto L. Sangiovanni-Vincentelli, Mirko Sauermann, and Dominik Langen. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In Donatella Sciuto, editor, *Design, Automation and Test in Europe (DATE)*, pages 276–279. ACM, 2008.

[MTFP11] Lorenz Meier, Petri Tanskanen, Friedrich Fraundorfer, and Marc Pollefeys. PIXHAWK: A system for autonomous flight using onboard computer vision. In *ICRA*, pages 2992–2997, 2011.

[MTT17] Tulika Mitra, Jürgen Teich, and Lothar Thiele. Adaptive isolation for predictability and security (Dagstuhl seminar 16441). *Dagstuhl Reports*, 6(10):120–153, 2017.

[MuB+16] Ravindra Metta, Martin Becker, Prasad Bokil, Samarjit Chakraborty, and R. Venkatesh. TIC: a scalable model checking based approach to WCET estimation. In Tei-Wei Kuo and David B. Whalley, editors, *Proc. Conference on Languages, Compilers, Tools, and Theory for Embedded Systems (LCTES)*, pages 72–81. ACM, 2016.

[NA15] Wolfgang Nebel and David Atienza, editors. *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*. ACM, 2015.

[NCS+06] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. PapaBench: a free real-time benchmark. In Frank Mueller, editor, *International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[NS15] Kartik Nagar and Y. N. Srikant. Path sensitive cache analysis using cache miss paths. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 43–60, 2015.

[PB95] Keshav Pingali and Gianfranco Bilardi. APT: A data structure for optimal control dependence computation. In David W. Wall, editor, *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 32–46. ACM, 1995.

[PBCS08] Rodolfo Pellizzoni, Bach Duy Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in cots-based embedded systems. In *Proc. Real-Time Systems Symposium (RTSS)*, pages 221–231. IEEE Computer Society, 2008.

[PK89] Peter P. Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989.

[PPH+13] Peter P. Puschner, Daniel Prokesch, Benedikt Huber, Jens Knoop, Stefan Hepp, and Gernot Gebhard. The T-CREST approach of compiler and WCET-analysis integration. In *Proc. International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, (ISORC)*, pages 1–8. IEEE Computer Society, 2013.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *IEEE Computer*, 24(5):48–57, 1991.

[PS97] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times - A graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In Michael Hanus and Sebastian Fischer, editors, *25. Workshop der GI-Fachgruppe Programmiersprachen und Rechenkonzepte*, number 0811 in Technische Berichte des Instituts für Informatik, pages 117–126, Olshausenstr. 40, D – 24098 Kiel, 2008. Institut für Informatik der Christian-Albrechts-Universität zu Kiel.

[Pus98] Peter P. Puschner. A tool for high-level language analysis of worst-case execution times. In *Proc. Euromicro Conference on Real-Time Systems (ECRTS)*, pages 130–137. IEEE Computer Society, 1998.

[Pus02] Peter P. Puschner. Is WCET analysis a non-problem? – Towards new software and hardware architectures. In Guillem Bernat, editor, *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 89–92. Technical University of Vienna, Austria, 2002.

[Pus03] Peter P. Puschner. The single-path approach towards wcet-analysable software. In *International Conference on Industrial Technology, 2003*, volume 2, pages 699–704. IEEE, 2003.

[Pus05] Peter P. Puschner. Experiments with wcet-oriented programming and the single-path architecture. In *Proc. International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 205–210. IEEE Computer Society, 2005.

[Rei17] Jan Reineke, editor. *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 57 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[RFMdS11] Vitor Rodrigues, Mário Florido, and Simao Melo de Sousa. Back Annotation in Action : from WCET Analysis to Source Code Verification. *Proc. Compilers, Programming Languages, Related Technologies and Applications (CoRTA)*, 2011.

[RGBW07] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007.

[RMPC13] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, and Fabienne Carrier. Timing analysis enhancement for synchronous program. In Michel Auguin, Robert de Simone, Robert I. Davis, and Emmanuel Grolleau, editors, *Proc. International Conference on Real-Time Networks and Systems (RTNS)*, pages 141–150. ACM, 2013.

[RMPV+19] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Erwan Jahier, Nicolas Halbwachs, Fabienne Carrier, Mihail Asavoae, and Rémy Boutonnet. Improving wcet evaluation using linear relation analysis. *Leibniz Transactions on Embedded Systems*, 6(1):02–1, 2019.

[Roc07] Christine Rochange, editor. *Proc. Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 6 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[RS09] Christine Rochange and Pascal Sainrat. A context-parameterized model for static analysis of execution times. *Trans. HiPEAC*, 2:222–241, 2009.

[S+89] Richard M. Stallman et al. Using and porting the gnu compiler collection. *Free Software Foundation*, 51:02110–1301, 1989.

[SAA+15] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil C. Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter P. Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. T-CREST: time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture - Embedded Systems Design*, 61(9):449–471, 2015.

[SBR11a] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Dominator homomorphism based code matching for source-level simulation of embedded software. In R.P. Dick and J. Madsen, editors, *Proc. International Conference on Hardware/Software Codesign and System Synthesis*, pages 305–314. ACM, 2011.

[SBR11b] Stefan Stattelmann, Oliver Bringmann, and Wolfgang Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In Leon Stok, Nikil D. Dutt, and Soha Hassoun, editors, *Proc. Design Automation Conference (DAC)*, pages 486–491. ACM, 2011.

[Sch03] Martin Schoeberl. JOP: A Java optimized processor. In Robert Meersman and Zahir Tari, editors, *Proc. International Workshop On The Move to Meaningful Internet Systems (OTM)*, pages 346–359. Springer Berlin Heidelberg, 2003.

[Sch09] Martin Schoeberl. Time-predictable computer architecture. *EURASIP J. Emb. Sys.*, 2009, 2009.

[SEE01] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In Guang R. Gao, Trevor N. Mudge, and Krishna V. Palem, editors, *Proc. Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 132–140. ACM, 2001.

[SK09] Bastian Schlich and Stefan Kowalewski. Model checking C source code for embedded systems. *International Journal on Software Tools for Technology Transfer*, 11:187–202, 2009.

[SKH17] Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time operating system. *Real-Time Systems*, 53(5):812–853, 2017.

[SL04] Bernhard Steffen and Giorgio Levi, editors. *Proc. International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*. Springer, 2004.

[SPH+05] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *Proc. Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[SRN+18] Eeric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[STW92] Harold S. Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Computers*, 41(9):1054–1068, 1992.

[Sun87] Sun Microsystems Inc. The SPARC Architecture Manual, Version 7. Sun Microsystems Inc., 1987.

[TE14] Piotr Trojanek and Kerstin Eder. Verification and testing of mobile robot navigation algorithms: A case study in SPARK. In *IROS 2014*, pages 1489–1494, 2014.

[TMMR19] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. Fast and exact analysis for LRU caches. *PACMPL*, 3(POPL):54:1–54:29, 2019.

[Tur38] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. *Proc. London Mathematical Society*, s2-43(1):544–546, 1938.

[uC18a] Martin Becker and Samarjit Chakraborty. Measuring software performance on Linux. *CoRR*, abs/1811.01412, 2018.

[uC18b] Martin Becker and Samarjit Chakraborty. Optimizing worst-case execution times using mainstream compilers. In Sander Stuijk, editor, *Proc. Software and Compilers for Embedded Systems (SCOPES)*, pages 10–13. ACM, 2018.

[uC19a] Martin Becker and Samarjit Chakraborty. A valgrind tool to compute the working set of a software process. *CoRR*, abs/1902.11028, 2019.

[uC19b] Martin Becker and Samarjit Chakraborty. WCET analysis meets virtual prototyping: Improving source-level timing annotations. In Sander Stuijk, editor, *Proc. International Workshop on Software and Compilers for Embedded Systems (SCOPES)*. ACM, 2019.

[UCS+10] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quiñones, Mike Gerdes, Marco Paolieri, Julian Wolf, Hugues Cassé, Sascha Uhrig, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, and Jörg Mische. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010.

[uMA+15] Martin Becker, Sajid Mohamed, Karsten Albers, P. P. Chakrabarti, Samarjit Chakraborty, Pallab Dasgupta, Soumyajit Dey, and Ravindra Metta. Timing analysis of safety-critical automotive software: The AUTOSAFE tool flow. In Jing Sun, Y. Raghu Reddy, Arun Bahulkar, and Anjaneyulu Pasala, editors, *Proc. Asia-Pacific Software Engineering Conference (APSEC)*, pages 385–392. IEEE Computer Society, 2015.

[uMC15] Martin Becker, Alejandro Masrur, and Samarjit Chakraborty. Composing real-time applications from communicating black-box components. In *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 624–629. IEEE, 2015.

[uMVC19a] Martin Becker, Ravindra Metta, R. Venkatesh, and Samarjit Chakraborty. Scalable and precise estimation and debugging of the worst-case execution time for analysis-friendly processors. *Journal on Software Tools for Technology Transfer*, 21(5):515–543, 2019.

[uMVC19b] Martin Becker, Ravindra Metta, R. Venkatesh, and Samarjit Chakraborty. WIP: Imprecision in WCET estimates due to library calls and how to reduce it. In N.N., editor, *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2019.

[uNSC15] Martin Becker, Markus Neumair, Alexander Söhn, and Samarjit Chakraborty. Approaches for software verification of an emergency recovery system for micro air vehicles. In Floor Koornneef and Coen van Gulijk, editors, *Proc. Computer Safety, Reliability, and Security (SAFECOMP)*, volume 9337 of *Lecture Notes in Computer Science*, pages 369–385. Springer, 2015.

[uRC17] Martin Becker, Emanuel Regnath, and Samarjit Chakraborty. Development and verification of a flight stack for a high-altitude glider in ada/spark 2014. In Stefano Tonetta, Erwin Schoitsch, and Friedemann Bitsch, editors, *Proc. Computer Safety, Reliability, and Security (SAFECOMP)*, volume 10488 of *Lecture Notes in Computer Science*, pages 105–116. Springer, 2017.

[Var12] Tullio Vardanega, editor. *Proc. International Workshop on Worst-Case Execution Time Analysis (WCET)*, volume 23 of *OASICS*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[VHMW01] Emilio Vivancos, Christopher A. Healy, Frank Mueller, and David B. Whalley. Parametric timing analysis. In S. Hong and S. Pande, editors, *Proc. Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 88–93. ACM, 2001.

[vst09] *Workshop on Verified Software: Theory, Tools, and Experiments (VSTTE)*, Eindhoven, The Netherlands, November 2009. co-located with POPL.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tiluka Mitra, Franke Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7(3):36:1–36:53, 2008.

[Wei81] Mark Weiser. Program slicing. In Seymour Jeffrey and Leon G. Stucki, editors, *Proc. International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE Computer Society, 1981.

[WG14] Reinhard Wilhelm and Daniel Grund. Computation takes time, but how much? *Commun. ACM*, 57(2):94–103, 2014.

[WH09a] Zhonglei Wang and Andreas Herkersdorf. An efficient approach for system-level timing simulation of compiler-optimized embedded software. In *Proc. Design Automation Conference*, pages 220–225. ACM, 2009.

[WH09b] James Windsor and Kjeld Hjortnaes. Time and space partitioning in spacecraft avionics. In *2009 Third IEEE International Conference on Space Mission Challenges for Information Technology*, pages 13–20. IEEE, 2009.

[WH12] Zhonglei Wang and Jörg Henkel. Accurate source-level simulation of embedded software with respect to compiler optimizations. In Wolfgang Rosenstiel and Lothar Thiele, editors, *Proc. Design, Automation & Test in Europe Conference (DATE)*, pages 382–387. IEEE, 2012.

[Wil04] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In Steffen and Levi [SL04], pages 309–322.

[WLH11] Zhonglei Wang, Kun Lu, and Andreas Herkersdorf. An approach to improve accuracy of source-level tlms of embedded software. In *Proc. Design, Automation and Test in Europe*, pages 216–221. IEEE, 2011.

[WM95] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):pp.20–24, 1995.

[WWC+13] Xueguang Wu, Yanjun Wen, Liqian Chen, Wei Dong, and Ji Wang. Data Race Detection for Interrupt-Driven Programs via Bounded Model Checking. *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pages 204–210, 2013.

[WZKS13] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 260–275. ACM, 2013.

[XMO19] XMOS. xcore processors. Online at `https://www.xmos.com/products/general/silicon`, 2019. retrieved 2019-May-30.

[Zel09] Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[ZKW+04] Wankang Zhao, Prasad A. Kulkarni, David B. Whalley, Christopher A. Healy, Frank Mueller, and Gang-Ryung Uh. Tuning the WCET of embedded applications. In *Proc. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 472–481. IEEE Computer Society, 2004.

List of Figures

***

List of Tables

***

Acronyms

***

ACET	Average-Case Execution Time.
ADC	Analog-Digital Converter.
AI	Abstract Interpretation.
AoRTE	Absence of Run-Time Errors.
AP	Atomic Proposition.
AST	Abstract Syntax Tree.
BB	Basic Block.
BCET	Best-Case Execution Time.
BSP	Board Support Package.
CEGAR	Counterexample-Guided Abstraction Refinement.
CFG	Control Flow Graph.
CNF	Conjunctive Normal Form.
DMA	Direct Memory Access.
DRAM	Dynamic Random Access Memory.
DV	Deducive Verification.
DWARF	Debugging Data Format.
EEPROM	Electrically Erasable Programmable Read-Only Memory.
ELF	Executable and Linking Format.
ESC	Electronic Speed Controller.
FCS	Flight Control System.
FIR	Finite Impulse Response.
FPU	Floating Point Unit.
GLB	Greatest Lower Bound.
GPS	Global Positioning Systems.
GUI	Graphical User Interface.
ICFG	Interprocedural CFG.
ILP	Integer Linear Programming.
IMU	Inertial Measurement Unit.
IPET	Implicit Path Enumeration Technique.
ISA	Instruction Set Architecture.
ISR	Interrupt Service Routine.

LRU	Least Recently Used.
LTL	Linear Temporal Logic.
LUB	Least Upper Bound.
MAV	Micro Air Vehicle.
MC	Model Checking.
MCU	Microcontroller.
MEMS	Micro-Electro-Mechanical Systems.
MINT	Minimum Inter-Arrival Time.
MMIO	Memory-Mapped I/O.
MMU	Memory Management Unit.
MRU	Most Recently Used.
OS	Operating System.
PCB	Printed Circuit Board.
PCM	Pulse Code Modulation.
PDG	Program Dependency Graph.
PID	Proportional-integral-derivative (controller).
PLRU	Pseudo LRU.
PPM	Pulse Pause Modulation.
RAM	Random Access Memory.
RC	Remote Control.
RISC	Reduced Instruction Set Computer.
RTS	Run-Time System.
SAT	Satisfiability.
SMT	Satsfiability Modulo Theorem.
SSA	Static Single Assignment.
TDMA	Time Division Multiple Access.
TLB	Translation Look-aside Buffer.
UART	Universal Asynchronous Receiver Transmitter.
UAV	Unmanned Aerial Vehicle.
VC	Verification Condition.
WCET	Worst-Case Execution Time.
WCRT	Worst-Case Response Time.
WP	Weakest Precondition.