FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

On Worst-Case Execution Time Analysis and Optimization

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Alexander Jordan
Registration Number 0125287

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl-Ing. Dr. Andreas Krall

The dissertation has been reviewed by:

(Andreas Krall) (Martin Schoeberl)

Wien, January 30, 2014

(Alexander Jordan)

Erklärung zur Verfassung der Arbeit

Alexander Jordan
Badgasse 22/12, 1090 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Acknowledgments

First, I would like to thank my parents for their support throughout these sometimes turbulent years of my higher education. Thanks to my advisor, Andreas Krall, who gave me the opportunity to do this research and the freedom to pursue it, the way I saw fit. Thanks to Martin Schoeberl for the insights and discussions on topics that at first seemed alien to me. To all my colleagues and friends, at the Complang group in Vienna and the ESE section at DTU, thanks for making the time I have spent in those places pleasant and fun.

This thesis would not have been written, if it has not been for the guidance and support of Florian Brandner. During the last years, apart from just keeping me motivated in my research, he has been tireless at teaching me new tricks and nudged me into —at times strange— new places in work and life. All of it, I thoroughly enjoyed.

Funding This work was partially funded by the Austrian Science Fund (FWF) under contract № P21842, “Optimal Code Generation for Explicitly Parallel Processors”, and the research project “Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST)” under grant agreement № 288008 funded by the 7th framework programme of the European Commission.

Abstract

Embedded systems have become a prevalent part of our daily lives. We can find them —often more than one, connected to each other— in our cars, phones and appliances. We rely on their availability for convenience, and in other, safety-critical areas, we rely on their correctness, sometimes with our lives. When such an embedded system has to perform a task (e.g. respond) in real-time, its correctness is not limited to functional requirements. In this case, we also want to verify its timing correctness. Timing measurements of the target system given a series of different inputs can lead to an estimation of its *worst-case* behaviors. But for any non-trivial application, it is impossible to reliably trigger the actual worst case. *Static analysis* of the system and its software can provide a safe upper bound of all possible timing results. We refer to this technique as worst-case execution time (WCET) analysis. Tools for static WCET analysis have recently become mature enough to be adopted by the avionics and automotive industries. But as software becomes more complex, the effort of performing static WCET analysis grows as well. This affects the computational effort caused by the analysis, as well as the effort spent by the engineer working with the analysis tool.

In this thesis we describe methods to adapt static WCET analysis in light of above noted problems and needs. With the *Patmos* platform, we have a hardware environment that heeds predictable execution as a primary design goal. This extends to the modeling of caches, which are a major factor of unpredictability in universal computing systems. For a cache architecture dedicated to the program *stack*, we describe the algorithms required to extend WCET analysis to accurately determine its worst-case behavior. Our evaluation reveals that the analysis of such a *stack cache* architecture, scales to large applications with complex calling behavior, while remaining efficient to compute and yielding precise results at the same time.

A worst-case-aware transformation of code is another possibility to reduce the WCET bound. Seeing that compiler support for WCET optimization is in its infancy, it is left to the programmer to avoid generating code that is hard to analyze and to do ad-hoc optimization of regions that analysis has shown to trigger the longest execution time. In this context, we further adapt WCET analysis to anticipate the need of programmers and compilers to get a more complete picture of a program's worst-case timing behavior. Through a meta-analysis approach, we create a novel metric that classifies code by its worst-case *criticality*. We formally define this metric and investigate some of its properties with respect to dominance in control-flow graphs. Exploiting these, we propose

algorithms that reduce the overhead of computation by inference. Experiments using a set of real-time benchmarks show the feasibility of our WCET profiling approach and reveal considerable amounts of highly critical as well as uncritical code in these programs.

From the beginnings of static WCET analysis, research has addressed methods to reduce the amount of *overestimation*. Overestimation is a necessary evil, which is applied in various forms, when the state space that would have to be tracked for precisely analyzing a program's behavior on a specific hardware platform grows too large. With our final contribution, we adapt the inner workings of the state-of-the-art path-based WCET analysis technique to follow a *pruning* approach. Our method is based on the notion of Criticality, we introduced before, and can eliminate potential sources of overestimation. Pruning approaches for program analysis have been proposed before, but to the best of our knowledge, this is the first time, pruning is done on the low-level representation of the program graph. Our first experiments with *graph pruning* use a commercial WCET analysis tool and show that our approach is feasible in practice and can improve the WCET bound up to 6% compared to the commercial tool as a baseline.

Kurzfassung

Die Zuverlässigkeit von eingebetteten Systemen ist durch deren große Verbreitung und Einsatz in sicherheitskritischen Bereichen, zu einem wichtigen Thema in Forschung und Entwicklung geworden. Hierbei spielt nicht nur die funktionale Korrektheit, sondern auch das Zeitverhalten einer Anwendung eine ausschlaggebende Rolle. Aus Messungen des Zeitverhaltens eines solchen Echtzeitsystems können Schätzungen, aber keine garantierten Schranken zu dessen Zeitverhalten in sämtlichen Szenarien abgeleitet werden. Die Verwendung von statischer Analyse zur Bestimmung von garantierten oberen Laufzeitschranken, kurz *WCET Analyse*, wird zum Beispiel bereits in der Auto- und Flugzeugindustrie eingesetzt. Primär steht bei der statischen Analyse die Generierung einer sicheren Schranke im Vordergrund. Sekundär soll die Analyse in möglichst kurzer Zeit ihr Ergebnis liefern und eine möglichst genaue Schranke berechnen, das heißt sich möglichst nah an die tatsächliche längste Ausführungszeit annähern. Ein Problem dieser Analysemethode ist jedoch die stetig steigende Komplexität von Computerprogrammen und Hardwarearchitekturen, auf denen diese Programme ausgeführt werden. Beides wirkt sich negativ auf die Analyselaufzeit, sowie deren Genauigkeit aus. Hohe WCET-Schranken in Relation zum Normalverhalten (programminherent oder durch Überschätzung während der Analyse) bedingen wiederum, dass mehr Leistung für die Ausführung zur Verfügung gestellt werden muss. Hier setzen die Optimierungen, die in dieser Dissertation ausgeführt werden, an.

Die *Patmos* Plattform erlaubt uns Analysen an eine Architektur anzupassen, die für die Vorhersagbarkeit von Ausführungszeiten entworfen wird. Der negative Einfluss von in Hardware implementierten Optimierungen (z.B. Caches, Pipelining), die im allgemeinen Fall zwar die Ausführungszeit steigern können, jedoch die Berechnung von oberen Schranken für diese erschweren, wird mit der Patmos Architektur vermieden. In dieser Dissertation beschreiben wir, wie das neue Konzept eines *Stack Cache*, präzise Analysen ermöglicht, deren Ergebnisse sich einfach in ein bestehendes Berechnungsmodell für WCET-Schranken integrieren lassen. Durch die Trennung von Zugriffen auf andere Speicherbereiche und das vorhersagbare Zeitverhalten des Stack Cache können genauere Schranken berechnet werden.

Die Qualität von WCET-Schranken, die mittels statischer Analyse gefunden werden können, hängt grundlegend mit der Struktur eines Programms und des Maschinencodes, der dafür erzeugt wird, zusammen. Werkzeuge, welche die Entwicklung von Echtzeit-Programmen erleichtern, zum Beispiel dafür optimierte Übersetzer, sind zu

diesem Zeitpunkt noch kaum ausgereift. Zur Unterstützung des Entwicklungsprozesses, sowohl auf Seiten des Programmierers als auch des Übersetzers, entwerfen wir eine Metrik, die über die Längste-Pfad-Sicht, die zur Berechnung einer Schranke herangezogen wird, hinausgeht. Diese Metrik gibt Aufschluss darüber, wie kritisch *alle* Teile eines Programms im Hinblick auf dessen WCET-Schranke sind und ermöglicht eine vollständige Sicht auf dessen Verhalten im schlechtesten Fall der Ausführung. Wir beschreiben die Meta-Analyse zur Berechnung von *criticality* mit Hilfe von effizienten Algorithmen und evaluieren deren Verhalten für Echtzeit-Programme, die als Standard-Testfälle im Bereich der WCET Analyse verwendet werden.

Um präzisere Schranken für die Ausführungszeit jeder Art von Hardware Architektur zu berechnen, beschreibt diese Dissertation schlussendlich ein Verfahren, das auf dem Separieren von Knoten in der Graphen-Darstellung eines Programms basiert. Das Ziel dieser Optimierung ist wiederum, durch das Ausschließen von Interferenzen und unter Erhalt der Sicherheit, eine genauere obere Schranke für das Zeitverhalten berechnen zu können. Unsere ersten Experimente mit dieser Methode, ergeben vielversprechende Verbesserungen der Schranken um bis zu 6%. Hierbei ist anzumerken, dass die Verbesserung der Analysegenauigkeit durch *Graph Pruning* mit der Größe des analysierten Programms ansteigt.

Contents

Chapter 1

Introduction

Embedded systems have become a prevalent part of our daily lives. Their application domains are manifold, including automotive, consumer, medical, military and telecommunication applications. By numbers sold, embedded microprocessors, digital signal processors (DSPs) and microcontrollers have been by far outnumbering desktop or notebook processors for at least a decade. In 2001 only 2% of processor sales accounted for desktop CPUs, while the rest belongs to the embedded market, which is dominated by microcontrollers at 80% of total volume [49]. Since then, as requirements for computing performance increases, we have seen embedded microprocessors displace 16-bit microcontrollers.

Through sensors, motors, or a human-machine interfaces, many embedded systems interact with the physical world or with us directly. Some of them are safety-critical, that is, we depend on their correct behavior. For example, we can find them in industrial applications, medical devices, as part of automotive systems and flight control systems on airplanes. Their failure to operate correctly at all times can cause expensive damage, or even lead to the loss of lives. Correctness does not only concern a program's functionality being in accordance with its specification, it also extends to requirements on its timing behavior. The program's sub-functions (*tasks*) are required to always respond in a timely manner. We can divide timing-sensitive embedded applications —following the definitions in [54]— into those that run on a *soft* real-time system, where missing a deadline leads to degraded service, which is not considered a failure unless it occurs too often. Other applications require a *hard* real-time system, i.e., a system that considers it a failure when one of its tasks fails to meet its deadline. (When tasks with hard and other tasks with soft timing constraints execute on the same system, we call it a *mixed criticality* system.)

We can guarantee the correct timing behavior of a real-time system, when we know the longest time it can take for its tasks to respond to inputs. For this in turn, we need to know the worst-case execution time (WCET) of a program, or parts of it. The WCET can be considered on multiple levels, the whole program, a procedure, or some sequential code, and it represents the longest time it can take to execute this region. The WCET is the opposite concept of the best-case execution time (BCET), the fastest execution through a piece of code that can be achieved. Somewhere in the middle between BCET and WCET, lies the average-case execution time (ACET), which general-purpose systems and even soft real-time systems usually optimize for. The WCET of a program can be far off from its average execution time and finding its exact value may not always be practical or required. Thus, engineers have resorted to determining an *upper bound* for their real-time programs, which may overestimate the actual WCET, but must never underestimate it. It is the goal of worst-case execution time analysis to efficiently find these safe WCET bounds while keeping overestimation at a minimum.

1.1 Worst-Case Execution Time Analysis

A straight-forward way to obtain information about the worst-case behavior of a program, is by stress testing it with inputs that try to trigger the *longest* execution time possible. Since the results obtained in this way are estimates, lacking any safety guarantee, an engineer might then add some slack to the longest measured execution time. This *dynamic analysis* is being employed in industry [53]; it is also known as *end-to-end measurements* in [34]. Tools for dynamic analysis can aid in the generation of program inputs, trying to maximize test coverage, while minimizing the number of measurement runs.

Static analysis, on the other hand, provides a safe bound for the worst-case execution time of a program without the need of executing it. The operation of a static WCET analysis tool can be conceptually separated into two components: (1) Control flow analysis is responsible for finding paths through the program that execution might take. This can be performed on different program representations (e.g., source code, binary code). (2) A lower-level analysis, which relies on a model that abstracts the execution behavior of the targeted hardware either *precisely* or *conservatively*. While it might not be feasible to keep track of all timing anomalies (e.g., pipeline- and cache stalls) precisely, the model must at least be conservative with regards to their timing effects.

Hybrid WCET analysis tools combine high-level static analysis, with measurements performed on individual parts of the program on the target hardware. The resulting WCET bound estimate is more robust than that of simplistic end-to-end testing, but in contrast to static analysis with a sound hardware model, it does not represent a safe upper bound.

Table 1.1 gives an overview of WCET tools and frameworks, classifying them by the algorithm used for WCET computation. It covers both, research and commercial projects that target WCET analysis and has a historical perspective. The early *tree*-based algorithms for WCET computation are followed by *path*-based approaches, which, in contrast to before, could capture the flow of execution more precisely, allowing for infeasible program paths to be excluded, thus reducing overestimation. On the other hand, path-based WCET analysis suffers from *path explosion*, in the same way path-sensitive analysis does in general: the state space grows too large, making the analysis intractable. This thesis focuses on the current state-of-the-art of static WCET analyzers, which are based on the implicit path enumeration technique (IPET).

System	Algorithm	Remarks	Ref.
Euclid	tree	real-time support was included in Euclid language	[87]
Modula/R	tree	Modula-2 extended for timing/memory analysis	[84]
SPARK Ada	tree	Ada subset, context-sensitive annotations	[81]
TAS	tree	annotated C and TAL annotation language	[86]
Timing Tool	tree	C subset, interactive annotation *session*	[85]
PL/IDL	path	both languages to specify (in)feasible paths	[83]
Florida	path	working path-based analysis on top of VPO compiler	[66]
AbsInt **aiT**	IPET	commercial, binary input, abstract interpretation	[57]
Bound-T	IPET	commercial, binary input, mostly user annotated	[68]
Chronos	IPET	SimpleScalar processor model, ILP constraints	[40]
Heptane	IPET	C source- and machine-level annoations combined	[60]
Hume	IPET	functional language using aiT for byte-code analysis	[50]
JOP WCA	IPET	targets time-predictable Java processor (JOP)	[20]
Otawa	IPET	low-level bundled with oRange for high-level analysis	[41]
Platin	IPET	see Section 2.2.1	[102]
Sweet	IPET	powerful high-level loop analysis, processor simulators	[65]
TuBound	IPET	Rose source-to-source C++ compiler, wcetC backend	[33]
WCC	IPET	annotated C, polyhedral loop analysis, integrated aiT	[42]
wcetC	IPET	C dialect with annotations transformed to machine code	[63]
Chalmers	simulated	measurements from cycle-accurate simulator	[72]
RapiTime	hybrid	commercial, measurements combined using IPET	[56]
SymTA/P	hybrid	measure “single feasible path” found by analysis	[59]
Vienna	measured	minimizes effort of test data generation by analysis	[51]

Table 1.1: Overview of WCET analysis systems (originally from [19])

1.1.1 Implicit Path Enumeration Technique (IPET)

The control flow graph of a program can be efficiently modeled using implicit path enumeration through a system of linear equations [78, 74]. Variables represent execution counts of basic blocks and constraints define legal execution paths in the CFG. Loop bounds, as well as further restrictions on program flow —these may be originating from previous analysis phases or annotations provided by a user, we refer to them in general as *flow facts* and specifically in the latter case as *user annotations*— are directly added to the integer linear program (ILP), through supplementary constraints. The ILP is completed by adding weights according to local execution times to nodes (or edges) of the graph, and by the objective function, which maximizes the overall cost (execution path length). If the problem is bounded and feasible, a standard ILP solver determines the WCET and execution frequencies for all basic blocks on *some* WCEP. This result does not immediately correspond to a path. It rather represents a *family* of paths, where each path may contain basic blocks several times, according to their IPET execution counts. It is trivial to construct a concrete path, since IPET ensures that Kirchhoff's law is respected by the execution counts.

Alternatively, dynamic programming [23] allows to directly compute the longest path on acyclic CFGs. Also cyclic CFGs can incorporate dynamic programming, by applying it to their acyclic sub-graphs [64].

1.2 Motivation

The problems that today's static WCET analysis tools face have many different sources. Real-time systems have seen a steady increase in complexity during the last decades [24]. Due to the increased processing power of modern processors, more elaborate algorithms are implemented, new functionality added, and existing functionality *migrated* to software. As the complexity of real-time software grows, analysis of the software becomes more and more demanding. In order for static WCET analyzers to yield results in an acceptable amount of time, they often have to make concessions when it comes to the precision of the analysis, i.e., they trade precision for feasibility. This leads to the *overestimation* of WCET bounds.

The most widely used compiler frameworks, due to the range of their supported languages and architectures, are GCC [96] and LLVM [97]. Both include embedded systems architectures as first class targets and can generate highly optimized code for them. Their optimization goals are limited to minimizing code size and maximizing average-case performance. As general purpose compilers they do not however, provide optimizations that target worst-case execution time, any means to disable optimizations that might impede it, or analyzability in general. We only know of one WCET-specific compiler, WCC from the Dortmund University of Technology. In other cases, research compilers were being adapted to WCET needs (e.g., code placement by Zhao et al. [48] uses VPO, Sweet uses LCC compiler). Judging by the target processor supported by commercial WCET analyzers, like aiT and Bound-T, we can conclude that commercial off-the-shelf (COTS) embedded processor platforms are being used for designing hard real-time systems. These architectures include performance-enhancing techniques, such as caching and speculation, which aim at improving average-case performance, but have been identified as a major obstacle to accurate WCET analysis (see [58]). This is due to *architectural states*, which need to be represented in the analysis problem to avoid introducing too much pessimism into the WCET bound (e.g., by considering every cache access a miss). Writing software for a safety-critical real-time system is no easy task. Besides the issue of correctness, timely responses must be ensured. Static WCET analysis provides timing guarantees and does not rely on measurements in its process. However, the analysis does not work as a push-button tool. Programmers need to provide annotations for the analysis to be useful and they need to be able to interpret its results. The latter is essential, when the goal is to optimize a program's WCET.

Why is it important that analysis bounds are reasonably tight? When we have to rely on a WCET bound, which is provably safe, but suffering from overestimation due to problems with our program, choice of hardware, or WCET analysis, we still need to accommodate this bound in the system. We then have to dedicate more resources to this overestimated task (e.g. use a faster processor), which will be rarely utilized, if at all, creating an overall inefficient system.

1.3 Contribution

Our goal is to advance today's state-of-the-art static WCET analysis. Being part of the T-CREST project, an EU FP7 funded research project, with the aim of creating a computing platform for hard real-time system from scratch, gave us the opportunity to come up with solutions across all problem areas we exposed above. Given a processor's instruction set and data cache, both designed for predictability, we show how an IPET-based WCET bound computation can avoid the predominant difficulties of cache analysis and precisely bound the worst-case behavior of a stack cache. With programmers and compilers in mind, we further augment WCET analysis, to gain more WCET-relevant information about real-time programs. On the same basis we tackle the problem of analysis tools having to face increasingly large programs. These contributions can be summarized as follows:

- We describe the analysis problems and algorithms necessary for accurate static analysis of worst-case behavior for a stack cache architecture. To derive execution time bounds, we show how to tightly integrate these results into the IPET phase of a static WCET tool.
- With *criticality* we present a novel metric related to worst-case execution time. It allows creating WCET-relevant program profiles, which may be used to better understand and optimize a real-time program, by a programmer, compiler, or by WCET analysis itself.
- By evaluating *criticality* on a set of programs, considered standard benchmarks in the field of WCET analysis, we learn more about the properties of these real-time programs. We present our findings in detail.
- Graph-pruning for refining WCET analysis is a complementary approach to existing pruning techniques, which target flow analysis or pruning of infeasible paths. Compared to the latter, graph pruning allows all static WCET phases to focus on relevant code parts.

The contributions in this thesis have been presented at international conferences, are pending publication, or have been published:

[1] Florian Brandner and Alexander Jordan. "Refinement of Worst-Case Execution Time Bounds by Graph Pruning". *(under submission)*.

[2] Alexander Jordan. "Evaluating and Estimating the WCET Criticality Metric". In: *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*. ODES '14. *(accepted for publication)*. ACM, 2014.

[5] Florian Brandner, Stefan Hepp, and Alexander Jordan. "Criticality: static profiling for real-time programs". In: *Real-Time Systems* (Oct. 2013). *(published online, pending print)*.

[6] Alexander Jordan, Florian Brandner, and Martin Schoeberl. "Static Analysis of Worst-case Stack Cache Behavior". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. RTNS '13. ACM Press, 2013, pages 55–64.

[9] Florian Brandner, Stefan Hepp, and Alexander Jordan. "Static profiling of the worst-case in real-time programs". In: *Proceedings of the International Conference on Real-Time and Network Systems*. RTNS '12. ACM Press, 2012, pages 101–110.

The elementary algorithms for worst-case analysis of stack cache behavior have been described in a conference paper [6].

Criticality as a novel metric for real-time programs, has been developed together with Florian Brandner and Stefan Hepp. It was first presented by the author of this thesis at RTNS 2012 [9] in Pont à Mousson, France, where it was awarded that years Best Paper Award. *Criticality* has been further extended in [5] and [2]. With regard to co-authorship of the criticality metric, the individual contribution of the author of this thesis covers its first implementation using LLVM and AbsInt `a3`, its evaluation environment, as well as the pruning and estimation methods. Florian Brandner and Stefan Hepp were active co-authors of the publications [9] and [5].

Following up on our work on *criticality*, an article describing the graph-pruning approach for refinement of WCET analysis is currently under submission [1].

1.4 Thesis Outline

This thesis is structured so that any of the main chapters can be read and understood on its own, as long as the reader is familiar with the theory and notation introduced in Section 2.1. An outline of the whole thesis is below:

- In Chapter 1 we give a short introduction to the field of worst-case execution time analysis, its current state-of-the-art, and an overview of methods and tools available today. We describe the various problems that current state-of-the-art tools face, which at the same time constitute the motivation of this thesis.

- Chapter 2 provides a more formal presentation of concepts from static WCET analysis and graph-theoretical background, which we rely on in later chapters. We also describe our experimental evaluation setup: the target processors we use and the set of real-time benchmarks, which we use throughout this thesis.

- Chapter 3 introduces the stack cache architecture, defines the analysis problems with respect to its worst-case behavior and specifies the algorithms to solve these.

Later in the chapter, we describe how these results can be integrated into an IPET-based WCET analysis in order to compute an overall WCET bound without the loss of any information collected during stack cache analysis.

- The *criticality* metric, its theoretical foundations and efficient algorithms to compute it are the topic of Chapter 4. We furthermore studied the behavior of real-time benchmarks based on *criticality* and report our findings in the same chapter.
- In Chapter 5 we present a graph pruning technique that aims to lessen the analysis gap introduced by interference from non-critical code during low-level analysis of worst-case timing behavior.
- Chapter 6 concludes this thesis by summing up the results from our contributions that aim at closing the gap between analyzed WCET bounds and actual worst-case behavior of a system. In the end, we give a list of future directions for research, which we identified while working on the subject at hand.

The following chapter gives definitions and explains notation, which we are going to use throughout the rest of this thesis.

Chapter 2

Preliminaries

In this introductory chapter we begin by defining the control-flow graph (CFG) and other related graphs, which together form the program representation that all our analyses are built on. We go on to describe the class of WCET analysis of tools, which we target by our adapted analysis algorithms. In Section 2.2 we first explain our experimental setup for performing evaluation and the target platforms (i.e. processors) that we are using in the process. We then give an overview —and a short history— of the real-time benchmarks used for evaluation.

2.1 Definitions

Here we define the basic concepts of control-flow graphs —to be exact, node-labeled control-flow graphs— that are used as the intermediate program representation for our analyses. We also detail the functionality of the class of WCET analysis tools that we target with our techniques and seek to optimize.

2.1.1 Program Representation

Directed Graph

A directed graph is given by the pair $G = (V, E)$, with V being a set of vertices (or nodes) and E a set of directed edges between them. We define the following utility functions common to all graphs: the predecessors of a node v in G are given by $\texttt{Pred}(G, v) = \{u \mid (u, v) \in E\}$. In the same way, we define for successors $\texttt{Succ}(G, u) = \{v \mid (u, v) \in E\}$.

Control-Flow Graph

A *control-flow graph* (CFG) is a directed graph given by the quadruple $G = (V, E, r, t)$, where nodes in V denote instructions of the input program, edges in E represent the potential flow of execution, and $r, t \in V$ represent distinguished nodes called root and sink. (For programs with multiple exit points an artificial sink node can be inserted, which is connected to all exit nodes.)

Critical Edge

A *critical edge* is an edge $(u, v) \in E$ of CFG G, where the source node u has multiple successors and the destination node v has multiple predecessors, i.e., $|\texttt{Succ}(G, u)| > 1 \land |\texttt{Pred}(G, v)| > 1$. If a CFG G is free of critical edges, we refer to it as G_{cf}.

Path

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

Longest Path

Given a CFG $G = (V, E, r, t)$ and a weight function $\mathcal{W} : V \cup E \to \mathbb{R}$, representing local worst-case execution times assigned to the nodes and edges in the CFG, we can define the length, or total weight, of a path $p = (v_1, \ldots, v_n)$ by $\bar{\mathcal{W}}(p) = \sum_{0<i\leq n} \mathcal{W}(v_i) + \sum_{0<i<n} \mathcal{W}((v_i, v_{i+1}))$. A path p is the longest path in the CFG, if there exists no other distinct path q such that $\bar{\mathcal{W}}(p) < \bar{\mathcal{W}}(q)$. In cyclic CFGs, flow constraints limit the set of paths to finite length. Since this only reduces the number of paths it can be safely ignored in the context of this work.

Weighted Control-Flow Graph

A CFG for which a weight function $\mathcal{W}$ exist, consequently is a weighted CFG. It can be represented by the tuple $(V, E, r, t, \mathcal{W})$.

Dominance

A CFG node u *pre-dominates* a CFG node v, written as u dom v, iff all paths from the root node r to v pass through u. Note that r trivially pre-dominates all nodes in the CFG. A node u strictly dominates a node v, written as u sdom v, iff $u \neq v$ and u dom v. Similarly a node *u post-dominates* a node v, iff all paths from v to the sink node t pass through u, we then write u pdom v and u spdom v. Note that the sink node t trivially post-dominates all nodes in the CFG. Both kinds of dominance can be represented by respective pre- and post-dominator trees.

Strongly Connected Components

A *strongly connected component* (SCC) in a directed graph G, is a set of nodes such that a path exists from every node to every other node within the set.

Basic Block vs. Single-Instruction Graph

We use node-label basic block graphs as the default CFG representation. I.e., nodes in V represent basic blocks, which are a maximal sequence of instructions without outgoing or incoming branches. When a fine grained view of a program is beneficial, we make use of the node-labeled single instruction CFG G_{SI}, whose nodes represent individual instructions. All graph properties stated above are valid regardless of the node representation of the CFG.

Call Graph

The call graph $CG = (V, E, s, z)$ is a directed graph, consisting of nodes in V, the distinguished source and sink nodes $s, z \in V$, and directed edges E. Nodes represent functions of a program, while each edge $(u, v) \in E$ represents an individual call (site) from function u to v. The source and sink nodes exist for convenience and are established by connecting s to the entry function (e.g., `main`), while all functions containing a call-free path, i.e., a path through the function's CFG that does not contain a call instruction, are connected to the sink node z.

2.1.2 Worst-Case Execution Time Analysis

Worst-Case Execution Time

The WCET of a program is the maximum time it takes for its computation to complete on a specific processor and platform. It is generally not feasible to compute the actual WCET, we thus usually seek a (tight) estimation, the WCET bound. For the purpose of brevity, we will use the term *WCET*, when we actually mean to refer to the *WCET bound*.

Worst-Case Execution Path

A worst-case execution path (WCEP) is a path whose total (estimated) execution time is equal to the WCET (bound). Note that multiple paths can potentially cause the same WCET. Thus, the WCEP is not always unique.

Local Execution Time

The timing behavior of an individual CFG node or CFG edge is computed using program analysis techniques that derive information on potential program and processor states at all relevant program points. This information is combined to compute an upper bound on the local execution time for nodes and edges.

Context-Sensitive Analysis

When an analysis extends its view of execution history beyond the control flow, to include information on the call chain and on previous iterations of a loop, we call it *context-sensitive* analysis. WCET analysis handles inter-procedural (calling) contexts and loop contexts through duplication of regions of the CFG. If WCET analysis does not keep track of contexts, it has to consider the worst-case behavior over all contexts. We then refer to it as *context-insensitive*.

Longest Path Search

Given the local execution times, a weighted CFG is constructed and longest path search is performed to find the global WCET. This problem is solved via integer linear programming (ILP) using the implicit path enumeration technique (cf. Section 1.1).

WCET Analysis Tool

We assume WCET analysis using an IPET-based analysis tool. As proposed by Theiling et al. [69], it proceeds in two general phases: (1) the calculation of local worst-case execution times of individual basic blocks and control-flow edges, followed by (2) a longest path search over the CFG, where nodes and edges are weighted using the local worst-case execution times computed before.

2.2 Analysis Evaluation

The landscape of WCET analysis tools (cf. Section 1.1) is heterogeneous and a direct comparison between analyzers is difficult at best. Program input and annotation formats differ and so does the support for hardware architectures. Recent efforts have the goal, to make it easier to compare WCET analysis tools (see Section 2.2.2). With AbsInt aiT as part of `a3` we have access to one commercial state-of-the WCET analyzer, which we make use of during evaluation. Where possible, we built improved analyses on top of aiT and not a possibly inferior tool of our own. In chapters 3 and 5, evaluation is performed by comparing our augmented analysis algorithms against a standard analysis.

In Chapter 4 we define the novel criticality metric for which we anticipate several areas of use. Chapter 5 goes on to demonstrate an approach in one of these areas. We are also able to evaluate the efficiency of the algorithms, which we propose for computing *criticality* and do so in Section 4.5. Its further evaluation, especially with regard to software engineering aspects, is beyond the scope of this thesis. However, we detail our ideas for future work applying *criticality* in Section 4.6.

On the other hand, the nature of static analysis gives us an advantage when it comes to the soundness of experimental results. Our primary results are WCET bounds computed through static analysis, which relies solely on the input programs, analysis configuration and deterministic models of processors and memories. These results are thus not subject to measurement error. Our secondary results concern the analysis efficiency and computational overhead from analysis. We do measure analysis runtimes on actual workstation computers here, but the significance of the results is in its order of magnitude (i.e., analysis terminates in seconds, minutes, days).

2.2.1 Target Processors

With Patmos[15, 101], we have access to a processor designed for predictability. Together with its framework consisting of an LLVM-based compiler toolchain and tightly integrated WCET analysis, it serves as the testbed to evaluate our stack cache analysis.

For our purely analysis-based evaluation, we selected the PowerPC MPC5554 [95, 94], which represents a processor not specifically designed for predictability but one that is being used in real-time systems today.

Patmos

The Patmos processor has been developed as part of the T-CREST [103] project. T-CREST aims at building a time-predictable multicore platform, with Patmos as the core, suited for hard real-time systems, combining efficiency and a predictable, low WCET. Time-predictability is pursued (1) in hardware by the choice of architectural features in the processor (e.g. in-order issue, cache design), a predictable memory controller, and communication with external (as in off-chip) resources through a statically scheduled network-on-chip. It is also respected during code generation, using the Patmos backend within the LLVM compiler framework, which (3) during compilation, collects and generates information to be consumed by WCET analyzers eventually. The processor, compiler and analysis tool for Patmos are open-source and available online [102].

Instruction Set Architecture Patmos has been designed as a dual-issue VLIW architecture. This means that two instructions form a VLIW-bundle, and at most two instructions per cycle can execute. Delays from non single-latency operations must be respected in the instruction stream (e.g., by placing NOPs). The Patmos instruction set is fully predicated, i.e., a predicate value can nullify the result of virtually every operation.

Registers and Functional Units Patmos has 32 general purpose (GP) integer registers, which are 32-bit wide and 8 single-bit predicate registers. Its functional units can be conceptually summarized as: an integer unit for arithmetic, logic and shifting operations (ALU), a load-store unit (MEM), a multiplier (MUL) and a branch unit (BR). Note that only ALU instructions can execute in both pipelines, while instructions executed by all other units are limited to the primary pipeline.

Memory and Caches Memories are strictly divided into *local* (caches and scratchpad) and *global* (SRAM) memories, which both have predictable delay. Furthermore, the split cache architecture does not only separate data and instructions, it can also make the distinction between different types of data. Thus, Patmos supports caches, which can adapt to the program's usage patterns. For one, the instruction stream can be cached via a *method* cache, which holds whole methods (functions or procedures) at a time. The other specific cache is the *stack* cache that we target with our static analysis in Chapter 3. There, in Section 3.1.1, we also present the functionality of a stack cache in detail.

WCET Analysis The customized LLVM compiler backend for Patmos exports the fundamental program description for later WCET analysis as a side product of compilation. This information is kept in the PML format, which is Platin's unified interchange format (based on YAML [93]). It is also used to encode the program's instruction at different levels, as well as flow facts, provided by the user or other analysis sources. (See Figure 2.1 for the simplified workflow). AbsInt's aiT has been retargeted to support Patmos from its binary format. It requires annotations for certain code structures, which are also provided by the analysis information generated during compilation.

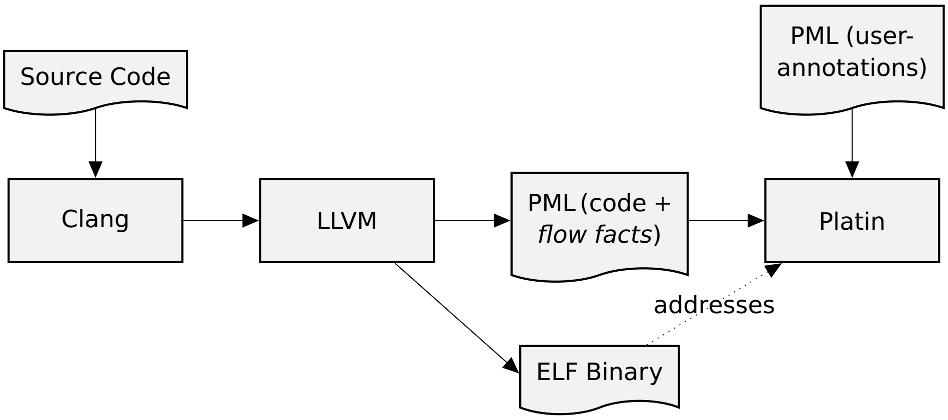


Figure 2.1: Workflow of LLVM/Platin-based WCET analysis (Patmos)

PowerPC 5000 (MPC5554)

The MPC5554 represents a class of embedded microprocessors, which have replaced microcontrollers for real-time control tasks that have complex computational requirements. (The semiconductor company Freescale targets it at automotive applications, e.g., engine control.) It is part of a family of PowerPC based embedded microprocessors, which branched off from PowerPC's main line of development for server and desktop processors. While the MPC5554 and its close relatives implement the same instruction set as the POWER5, which is aimed at the high-end servers, their micro-architecture is simple in comparison. This makes the MPC5554 a feasible target for WCET analysis.

Instruction Set Architecture The MPC5554 is a 32-bit implementation of the PowerPC instruction set, which, following the RISC philosophy, consists of short and simple instructions. Latencies are hidden from the instruction stream and while other PowerPC implementations may enable parallelism through a superscalar design, the MPC5554 issues a single instruction at a time.

Registers and Functional Units The MPC5554 has 32 general purpose (GP) integer registers that all ALU instructions see as 32-bit wide. An 8-bit condition register is used for testing and branching. All memory operations are performed by the load-store unit (LSU). The branch unit (BR) handles branching and is involved in branch prediction. Most instructions execute in a single cycle, notable exceptions are multiplies, loads and stores, which have 3-cycle latencies, divides take even longer to complete. A buffer for branch targets and a 2-bit flag for dynamic branch prediction are aimed at avoiding a 3-cycle branch latency. Further functional units extend the instructions set with signal processing instructions (multiply-accumulate and vector instructions) and floating point operations.

Memory and Caches For this specific model, instruction cache and data cache are combined into a 32 KiB unified cache, controlled by a unified memory management unit (MMU). A further 64 KiB SRAM is available on-chip, while all other memory is external and needs to be connected through a bus.

WCET Analysis The binaries compiled for the MPC5554 serve as the input to WCET analysis using AbsInt's aiT. In this case, the control flow graph is reconstructed from the binary representation. The configuration of the abstract processor model (e.g., cache- and memory timings) and all further annotations are passed to aiT in external files. Both are depicted in Figure 2.2 by the XML-based APX file and the text-based AIS file, respectively. The challenge that the MPC5554 poses for WCET analysis comes from its pipeline behavior, including the effects from branch prediction, as well as potential cache conflicts. All of these need to be considered by the analysis tool for the calculated WCET bound to be safe and precise.

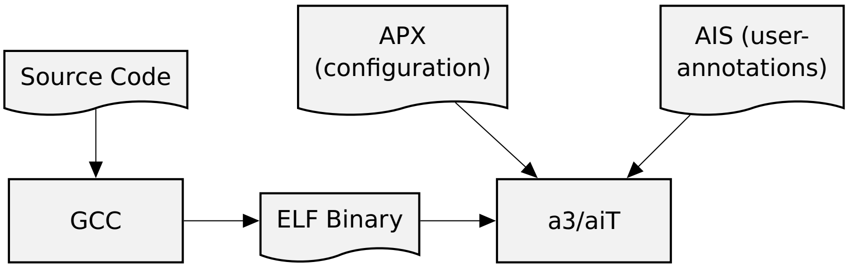


Figure 2.2: Workflow of a3-based WCET analysis (MPC5554/GCC)

2.2.2 Real-Time Benchmarks

In real-time systems and especially WCET analysis, we do not have a well-established collection of benchmark suites available, as researchers and developers —most of the time concerned with some kind of average-case behavior— do, in fields like programming languages, hardware design, and compilers. To reproduce the result of a WCET benchmark, assuming the same analysis tool, we have the following dependencies:

1. program *(with compiler and compiler configuration)*
2. target machine configuration *(how the hardware model behaves)*
3. problem definition *(exactly what the analysis goal is)*
4. annotations *(known restrictions on input data, loops and control flow)*

For a representative set of real-time WCET benchmarks, we refer to the WCET Tool Challenge, which was instituted with the goal to study the results of different WCET analysis tools on a common basis that allows comparison and facilitates exchange between research groups working on WCET analysis. It has seen three editions so far, (WCC'06 [30], WCC'08 [31], and WCC'11 [12]) and has created an initial collection of programs and specifications suitable for benchmarking WCET analyzers. For our evaluation we adopt the benchmark programs and analysis specifications from the three WCET tool challenges, as far as they were available to us and documented, respectively. To reproduce results in the field of WCET analysis, one might require —besides the benchmark program itself— the same or similar compiler, the analysis tool, all of their configuration options, as well as the user annotations that were initially provided for the analysis. Where possible, we are making this information available online [101, 103]. A short description of the available benchmark suites follows, a more detailed discussion can be found in [17].

The **Mälardalen WCET Benchmarks** are maintained by the Mälardalen WCET research group and consist of 35 programs, all fundamental algorithms found in real-time applications, with a focus on math functions and signal-processing. 15 of these programs were selected for the 2006 WCET Tool Challenge. We compiled all benchmarks from their C source code without optimizations (`-O0`, since the programs are small and we want to stress analysis in our evaluation) and added annotations, such as control-flow hints and loop bounds, from the information partly available online [98]. Three programs, which are not part of the WCET Challenge subset, could not be successfully analyzed with our setup.

PapaBench [45] is an open-source flight control software used in autonomous aircraft. The analysis problems that relate to different tasks in the software and annotations required for analysis were taken from WCC'11.

The **Debie1** benchmark [67] is based on the on-board software of a satellite instrument and is available under a special license. In this case we relied on a pre-compiled binary, which is part of the official program distribution. Loop bounds and annotations again were taken from WCC'11.

We could not include an automotive application by Daimler or the `rathijit` benchmark programs in our evaluation. The former is not available to the public, while the latter could not be analyzed out-of-the-box due to their synthetic nature (macro generated code) that targets the cache analysis of a WCET tool.

Chapter 3

Precise Stack Cache Analysis

Utilizing a stack cache in a real-time system can improve predictability of the data cache by avoiding interference that memory traffic to the heap causes. While stack loads and stores are guaranteed cache hits, explicit operations are responsible for managing the stack cache. The worst-case behavior of these operations can be analyzed statically. In this chapter we present algorithms that derive worst-case bounds on the latency-inducing operations of a stack cache. We furthermore describe how these results can be used in the IPET phase of a static WCET analysis tool. In the following section we provide necessary background on a stack cache enabled architecture. In Section 3.2, we formally describe the analysis problems and algorithms we have devised for them. Sections 3.3 and 3.4 describe the integration of our stack cache analysis into state-of-the-art WCET analysis and a generalization of our analysis model, respectively. Finally, we evaluate our approach in Section 3.5, before discussing related work in the last section.

The algorithms for precise worst-case analysis of stack cache behavior in Section 3.2 are published in [6] and have been presented at RTNS 2013.

3.1 The Stack Cache

Caches pose a particular challenge with regard to WCET analysis, as a potentially huge state space reflecting a long execution history of the program has to be tracked. A solution to this problem is splitting the cache according to access patterns. For instance, it is recommended to split data and instruction caches to avoid interference [16]. A data cache can be split in a similar fashion, to adapt the caching strategy to the access patterns of different memory regions, such as those for the program stack, constants and static fields [8].

Memory accesses to the program stack via a standard data cache can be hard to analyze, when their addressing is not based on a single authoritative stack pointer. The cache analysis needs to identify potential address ranges for the stack. These inherently depend on the nesting of function calls and their analysis require high levels of context-sensitivity. Likewise, pointers to the stack area have to be identified as such, in order for all stack accesses to be considered. Having a closer look at typical usage patterns of stack data, one finds that data in the stack frame of a function is predominantly exclusive to that function and only accessed while the function is active. This motivates a caching strategy that follows the nesting of function calls so that the stack frame of the active function is readily available in the cache. Recent work proposed such a *stack cache* using a rather simple ring buffer [4]. Even though simple, this cache design handles up to 75% of the dynamic data accesses of embedded benchmarks. In the remaining chapter, we explore analysis techniques for this stack cache design.

The stack cache is explicitly controlled by the compiler (or the programmer when writing assembly code) using dedicated instructions to *reserve* and *free* space on the stack cache for the stack frames of functions. During a reserve operation the requested space might exceed the cache's capacity and cause *spilling* of (parts of) the stack cache's content. If the stack frame of a function has been spilled to memory it can be reloaded, or *filled*, using an *ensure* operation, e.g., when a function becomes active after returning from another function. Using these stack control primitives, a sliding window of cached data is realized that follows the nesting of function calls and ensures that all accesses to the stack data of a function are guaranteed hits with constant latency, independent of the precise address of the access and independent of the current value of the stack pointer. This results in a considerably simpler analysis model.

3.1.1 Stack Cache Implemented in Hardware

In this section, we describe the general functionality of a stack cache. A detailed description of the implementation in the Patmos processor can be found in [4]. Compared to accessing the program heap storage —normally located in external memory and accessed through a cache— a stack cache's load and store operations complete with a small and predictable latency. The operations address values on the stack relative to a base address, the *stack pointer*. Furthermore, the model we consider herein uses three primitives to manage the stack cache. These are responsible for *reserving* space for stack-allocated data, *freeing* the same space again, and *ensuring* that data, which will be subsequently used, is available.

Conceptually, the stack in question grows towards lower addresses and is divided into a set of equally sized blocks (see Figure 3.1). *The number and actual size of the blocks depend on the implementation.*

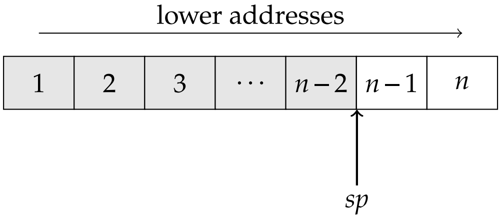


Figure 3.1: Stack cache of size $|SC| = n$ ($n-2$ blocks occupied)

Each operation takes an argument representing a block count. In detail, the semantics for stack cache manipulation are as follows:

Reserve: `sres` k

Allocates an area of k blocks in the stack cache and sets the stack pointer to the beginning of this region. If k and the number of currently reserved blocks counted together exceed the capacity of the stack cache, some blocks need to be *spilled* (i.e., saved to external memory). The stack cache always selects a minimal number of blocks from the earliest reserved (highest address) blocks for spilling.

Free: `sfree` k

Discards the k most recently reserved (lowest address) blocks and adjusts the stack pointer accordingly. The contents of the stack cache are not changed.

Ensure: `sens` k

If not all of the k blocks starting at the current stack pointer are available in the stack cache, (only) the missing blocks are *filled* (i.e., loaded from external memory).

Only an `sres` or `sens` operation may access external memory and can cause a variable-time latency. Particularly, WCET analysis benefits from this characteristic, as it is sufficient to know (an upper bound on) the number of blocks in a transfer, to statically calculate its latency. Our model assumes *uniform cost* for the transfer of every stack cache block, but this can be trivially adapted to the behavior of any implementation (as long as it remains predictable). *This stack cache model can be implemented in several ways. In the concrete case of the Patmos processor, all three stack cache operations are exposed in the instruction set and hardware is responsible for spilling data to and loading data from external memory. The argument k may represent another unit of allocation, however, for simplicity of description, we abstract it to blocks. Also, without any impact on the analysis we present, some (or all) functionality could be shifted to software.*

Splitting the `sens` from the `sfree` operation is a straight-forward optimization for a generalized *tail call*, i.e., a function call towards the end of a parent function's code, after which the stack is not being accessed until the parent returns. Filling the stack cache after the child returns, would cause unnecessary loading from external memory.

The spill and fill operations inherently provide a WCET bound. In the most pessimistic manner, we could assume that upon every `sres` and `sens` all k of the argument need to be spilled and filled respectively. In the following, our analyses shall reduce this worst-case bound.

Example 1. *We are using the program in Figure 3.2 as a running example throughout this chapter. Figure 3.3 visualizes the operation of a stack cache with* 4 *blocks for this program, between the call to* `B` *in line* a_3 *and the call to* `C` *in line* a_5*. At the top of Figure 3.3b,* `B` *reserves its stack space, which causes partial spilling of* `A`*'s stack frame. Calling* `C` *then spills the rest of* `A`*'s frame and partly evicts the stack space of* `B`*, while the remainder of the stack cache is allocated to* `C`*'s data. Towards the end of Figure 3.3b, after* `B` *and* `C` *have returned, the stack cache becomes completely empty and at line* a_6*,* `A` *has to reload all of its previously evicted stack data through an ensure.*

3.2 Stack Cache Analysis

In order to determine the worst-case behavior of a real-time program utilizing a stack cache, two analysis problems have to be solved: (1) for every `sres`-instruction in the program, the worst-case *spilling* behavior has to be computed and (2) for every `sens`-instruction, the worst-case *filling* behavior has to be computed. We refer to these two problems as the *reserve* and *ensure* analyses respectively.

Definition 1. *Stack cache operations can be placed at arbitrary points in the program (with some inherent rules for legal programs). Commonly they appear around function calls, i.e.,* `sres` *is placed after function entry,* `sfree` *before the return, and if required, the caller places* `sens` *immediately after the call. In the following, we present an analysis that assumes the simple case of placement around function calls. A generalization is discussed in Section 3.4. We further assume that every function reserves a value k less than, or equal to the size of the stack cache. This requirement is guaranteed by the compiler.*

Reserve analysis During reserve analysis we have to consider the (maximum) stack cache occupancy on all potential executions leading up to an `sres`-instruction. This information cannot be computed using local information alone and thus requires a context-sensitive analysis. The worst-case behavior of an `sres` instruction depends on the stack cache fill level (*occupancy*) of the current function, which in turn depends on the occupancy at the invoking call instruction, thus on the occupancy of the function surrounding the call, and so on until the program's entry point is reached. In order to compute this information efficiently, the analysis has to account for the change of stack occupancy that occurs between the entry of a function and each call instruction. Note that the stack occupancy not necessarily increases here. If, for instance, parts of the stack cache are intermediately spilled, the occupancy may also decrease. Here, a key observation is that the worst-case occupancy at call sites can be bounded by a function-local analysis, which is based on the *minimum displacement* occurring between the function entry and the call. This displacement determines how much of the allocated stack space

```
a1: func A() {
a2:   sres 2;
a3:   B();
a4:   sens 2;
a5:   C();
a6:   sens 2
a7:   sfree 2;
a8: }
```

(a) Code of A

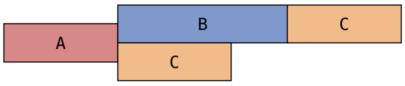
```
1B  func B() {
2B    sres 3;
3B    C();
4B    sens 3
5B    C();
6B    sens 3
7B    sfree 3;
8B  }
```

(b) Code of B

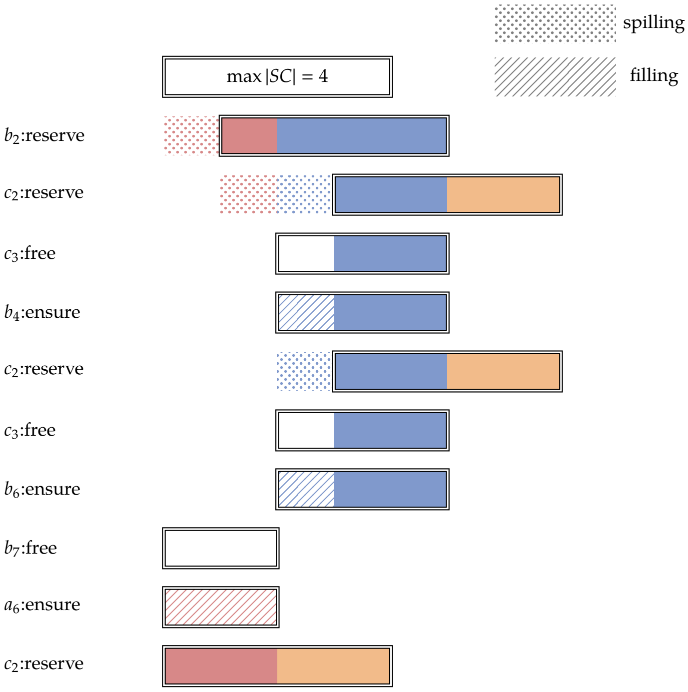
```
1C  func C() {
2C    sres 2;
3C    sfree 2;
4C  }
```

(c) Code of C

Figure 3.2: Program consisting of 3 functions, reserving, freeing and ensuring space on the stack cache.



(a) Logical stack frames (`A` calls `B` and `C`, `B` calls `C`)



(b) Dynamic stack cache behavior between first call to `B` and last call to `C`

Figure 3.3: Sliding window visualization of the stack cache for program in Figure 3.2

remains allocated in the worst-case when calling a function. Using the results of this preliminary analysis, a function-local data-flow analysis can propagate *worst-case occupancy bounds* along all paths from a function's entry to all call sites, initially assuming a fully occupied stack cache. The final step of reserve analysis is an inter-procedural propagation of the context-sensitive stack occupancy bound on the program's call graph.

Ensure analysis The analysis of ensure instructions similarly relies on the stack displacement at function calls. In contrast to reserve analysis, however, the maximum displacement is required, i.e., how much of the stack space allocated by the current function is spilled to external memory by another function in the worst-case. Based on the maximum displacement, the ensure analysis can be formulated as a function-local data-flow analysis racking the amount of stack space belonging to the current function as it undergoes spilling and reloading.

Example 2. *Consider a stack cache of size* 4 *and our example program from Figure 3.2. The stack cache occupancy at the entry of function* `C` *depends on its three calling contexts (a_5, b_3, b_5). We are only interested in the last calling context (b_5) for now. By assuming a full stack cache at the entry of the calling function* `B` *and examining the path leading to the call instruction, we realize that the minimum displacement along this path can be used to bound the worst-case occupancy for this context. Since the* `sres`*-instruction at the entry of* `B` *cannot further increase the occupancy, the first call to* `C` *(b_3) displaces* 2 *blocks from the full stack cache, resulting in a worst-case occupancy of* 2 *when returning. The* `sens`*-instruction (b_4) again raises it to* 3*, which becomes the occupancy bound at b_5.*

The ensure analysis similarly first pre-computes the maximum displacement of function calls and then propagates information within functions. Function `A` *for instance, reserves* 2 *blocks on the stack cache. The displacement due to the call to function* `B` *(and its calls to* `C`*) is determined to be* 4*. The ensure analysis thus finds that the entire stack space of* `A` *may have been evicted after returning from* `B`*. The* `sens`*-instruction a_4 thus has to fill* 2 *blocks from external memory in the worst-case.*

Combined Analysis The reserve and the ensure analysis both rely on related underlying computations and can be combined. The combined analysis then consists of three main phases: (1) the pre-computation of the minimum and maximum displacements on the call graph, followed by (2) the function-local data-flow analyses for the reserve and ensure analysis, and finally (3) the context-sensitive reserve analysis on the program's call graph. We will discuss each of the phases in the following sub-sections.

Algorithm 1 Algorithm to compute the displacement at call sites (`ComputeMinimumDisplacement`, `ComputeMaximumDisplacement`).

Input: $ACG = (V, E, s, z)$ … An annotated call graph.

Output: The minimum/maximum displacement at each call site is returned.

```
foreach $n \in V$ do
    ▷ Sub-graph with zero costs.
    $V_0 = \{v_0 \mid v \in V, v \neq z\}$
    $E_0 = \{(u_0, v_0, 0) \mid (u, v, w) \in E, v \neq z\}$
    ▷ Weighted sub-graph.
    $V_W = V \setminus \{s\}$
    $E_W = E \setminus \{(u, v, w) \in E \mid u = s \lor v = s\}$
    ▷ Edges to transition between sub-graphs.
    $E_T = \{(u_0, n, 0) \mid (u, n, w) \in E\}$
    Let $ACG' = (V_0 \cup V_W, E_0 \cup E_W \cup E_T, s_0, z)$ in
        $D[n]$ = ComputePathOver($ACG', n$)
return $D$
```

3.2.1 Stack Cache Displacement

Computing the minimum displacement of a call site, as required by the reserve analysis, corresponds to a shortest path search in the call graph, where edges are annotated with weights representing the amount of stack space reserved by the calling function. We will later see that ensure analysis depends on the maximum displacement, which can be computed in the same way, but performing a *longest path* search. Assuming the restricted placement of ensure and free instructions of Definition 1 for now, this technique can easily be extended to the larger class of *well-formed* programs as described in Section 3.4.

Definition 2. *The annotated call graph* $ACG = (V, E, s, z)$ *is a call graph, with weighted edges* $(u, v, w) \in E \subseteq V \times V \times \mathbb{N}^0$*. The weight* w *represents the stack space reserved in function* u*. The edge connecting* s *to the entry function is assumed to have weight* 0*, while edges incident to* z *are annotated, like ordinary edges, with the stack space reserved in the respective functions.*

Definition 3. *For a call site* (u, v, w) *of an annotated call graph* $ACG = (V, E, s, z)$ *the minimum (maximum) displacement is given by the shortest (longest) tail from* v *to the sink node* z*, for any path of the form* $(s, \ldots, u, v, \ldots, z)$.

Acyclic Call Graphs

In the case of acyclic call graphs, i.e., programs without recursion, the minimum and maximum displacement of all nodes can be computed using dynamic programming [23] in linear time ($O(|V| + |E|)$). The nodes are traversed in reverse topological order, computing each node's displacement from the displacement of their respective successors in the graph.

Call Graphs With Recursion

Shortest and longest path searches for programs with cyclic call graphs, i.e., with recursion, can be modeled using integer linear programming. The technique resembles the IPET approach from WCET analysis. Instead of plainly searching for the shortest (longest) path though, the path with the shortest (longest) tail, where the tail starts with a specific node, needs to be computed. The reason for this will become clear when we later introduce *user constraints*.

We model the computation of these paths on a transformed call graph, which is constructed by duplicating the original graph twice (see Algorithm 1 and Figure 3.4). One duplicate represents the paths' tails and is thus associated with the weights of the original graph (l. 6). The other duplicate is associated with zero costs (l. 3) and represents the heads of the paths. The two sub-graphs are connected only at the node whose displacement is to be computed, i.e., edges lead from the node's duplicate in the sub-graph with zero costs to the respective duplicate in the weighted sub-graph. The shortest or longest path search is then performed on this transformed graph (l. 11). Note that not all nodes and edges need to be duplicated in practice. Only the nodes and edges that may appear on a path from s to z and passing through the currently considered node need to be considered.

For the path search `ComputePathOver` (Algorithm 1, l. 11) takes the transformed call graph ACG' and a target node n as arguments and constructs an ILP, which models the nesting of function calls that can be observed (in the worst-case) when executing the program. Each ILP variable represents the number of times a function has been called, or more precisely, how often a specific call site was used to call a function, in this nesting. The ILP variables can be seen as representing *flow* that has to meet constraints. For instance, the flow entering a function has to leave that function again, i.e., the sum of the adjacent ILP variables needs to be equal.

The (flow) constraints of the ILP for the transformed call graph $ACG' = (V', E', s', z')$ are formally defined as:

$$\left.\begin{aligned} \mathcal{V}(v) &= \sum_{e=(u,v,w)\in E'} \mathcal{V}(e) \\ \mathcal{V}(v) &= \sum_{f=(v,u,w)\in E'} \mathcal{V}(f) \end{aligned}\right\} \quad \forall v \in V' \tag{3.1a}$$

$$\mathcal{V}(s') = 1 \tag{3.1b}$$

$$\mathcal{V}(z') = 1 \tag{3.1c}$$

$$\mathcal{V}(n) > 0 \tag{3.1d}$$

With integer variables:

$$\mathcal{V}(e), \mathcal{V}(v) \in \mathbb{N}^0 \quad \forall e \in E', \forall v \in V' \tag{3.1e}$$

Variables (3.1e) are created for every node and every edge in the transformed call graph and the functions $\mathcal{V}(n)$ and $\mathcal{V}(e)$ map nodes and edges of the graph to their respective ILP variables. For each node, incoming and outgoing flow has to match (3.1a). In order to get a legal nesting of function calls including the node n, three more flow constraints have to be added. These force the flow at the source (3.1b) and sink node (3.1c) to 1 and the flow over the target node (3.1d) to be non-zero.

The optimization objective is either to minimize the objective function, when the shortest path is to be computed:

$$\min \sum_{e=(u,v,w)\in E'} w\mathcal{V}(e), \tag{3.1f}$$

or maximize the same function for the longest path search:

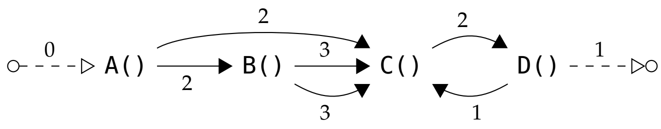
$$\max \sum_{e=(u,v,w)\in E'} w\mathcal{V}(e) \tag{3.1g}$$

The ILP formulation presented above expresses all possible nestings of function calls that might potentially be observed when the program is executed. However, in particular when the objective function is to be maximized, many of these nestings cannot actually occur in practice, e.g., because recursion in the program is limited to a certain depth. This information can be supplied by the user and expressed as an additional constraint for the affected ILP variable. More complex *user constraints* can be expressed by linear equations using different variables of the ILP.

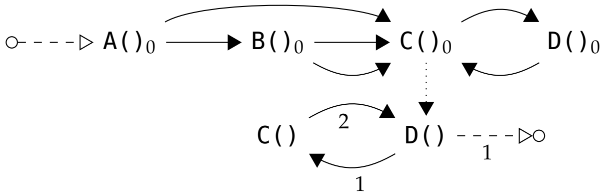
Example 3. *Consider the annotated call graph in Figure 3.4a. Function* `D` *has a call-free path and is thus connected to the sink. Assuming that the displacement of* `D` *shall be determined, Figure 3.4b shows two duplicates of the graph, where the sub-graph with zero costs is above the weighted one. The edge that allows to transition between the two sub-graphs* (E_T) *is highlighted using a dotted line. (Only nodes and edges on a path from the source to the sink and passing through* `D`*, as well as non-zero edge weights are shown.)*

Assuming that function `C` *can appear at most* 10 *times on any legal nesting of function calls, we can add a user-specific constraint such as:* $\mathcal{V}(C()) + \mathcal{V}(C()_0) < 11$. *Since* `D` *can only be invoked by* `C`*, this implies a maximum displacement for* `D` *of* $9 \cdot 1 + 9 \cdot 2 + 1 = 28$. *Because the constraint on* `C` *only defines an upper bound, it is easy to see that the minimum displacement for* `D` *(shortest path) is* 1.

Certainly, as long as a user constraint does not interfere, shortest path search can be solved efficiently on the original graph (e.g., using *Dijkstra's algorithm*). Also note that the approach for cyclic and acyclic graphs can be combined by collapsing the nodes of the individual *strongly connected components* (SCCs) of the original call graph into representative nodes. This results in an acyclic graph that can be traversed as described in Section 3.2.1. Whenever the representative of an SCC is visited during the traversal, an ILP is constructed as shown by Algorithm 1 based on the sub-graph induced by the SCC in the original call graph.



(a) Call Graph



(b) Transformed Graph

Figure 3.4: A recursive call graph and the corresponding transformed graph to compute `D`'s displacement.

3.2.2 Data-Flow Analyses

Once the minimum and maximum stack cache displacements have been computed (see Section 3.2.1), the stack cache occupancy can be bounded from both directions by an intra-procedural data-flow analysis. This is a prerequisite for reserve analysis and it immediately enables ensure analysis.

Data-flow Analysis Framework Data-flow analysis [35, 27] is used to gather information about the behavior of a program without executing it. For a specific property (e.g., variable liveness) the effect of every instruction (i.e., every node in some CFG) and its dependencies on other instructions are examined. A common way to perform data-flow analysis is to attach equations that relate input and output information to every node, then solving these equations for every node repeatedly, until their results no longer change (i.e., a *fixpoint* for the whole system is reached). Two parts of our stack cache analysis closely resemble this concept and are thus best described in the standard way for data-flow analyses.

We are interested in bounds for the stack cache occupancy at certain points in the program. I.e., a value from a finite subset of $\mathbb{N}^0$ ($\mathbb{N} \cup \{0\}$) bounded from above by the number of blocks in the stack cache $|SC|$. Formally, the value domain for the analysis is $\mathcal{D} = \{0, \ldots, |SC|\}$. To set up the system of equations, every instruction i in the CFG is associated with two variables, $IN(i)$ and $OUT(i)$, which can take values from $\mathcal{D}$ and represent the occupancy bound before and after the instruction respectively. Data-flow equations (also known as *transfer functions*) between the variables define (1) the change of the worst-case occupancy bound induced by instructions and (2) how to merge those bounds at control-flow joins. The iterative algorithm to solve these equations initializes the IN-states once, then repeats updating OUT and IN states until a fixpoint is reached. The latter is guaranteed by the monotonic update of the occupancy bound, in the bounded domain $\mathcal{D}$.

Bounding the Stack Cache Occupancy

Assuming a full stack cache at function entry, the analysis propagates an upper bound on the stack occupancy (o_{bound}) along all paths from the entry to the call sites within the function. The minimal displacement of call instructions as well as `sens`-instructions along the path may have an impact on this bound, while all other instructions can safely be ignored.

A standard data-flow analysis based on the framework defined at the beginning of this section, with domain $\mathcal{D}$ is then performed on the CFG. The transfer functions for an instruction i are:

$$d_{loc}(i) = \min(|SC|, d_{min}(i)) \tag{3.2}$$

$$OUT(i) = \begin{cases} \max(IN(i), k) & \text{if } i = \texttt{sens k} \quad (3.3a) \\ \min(IN(i), |SC| - d_{loc}(i)) & \text{if } i = \mathit{call} \quad (3.3b) \\ IN(i) & \text{otherwise} \quad (3.3c) \end{cases}$$

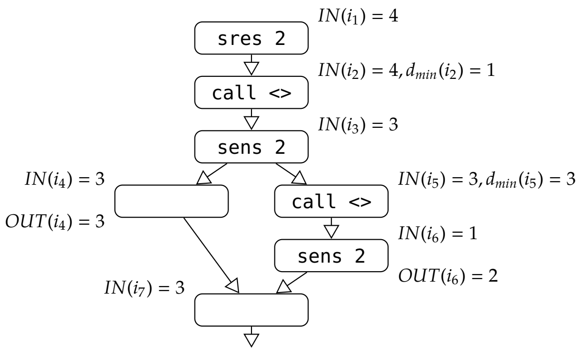


Figure 3.5: Propagation of stack cache bounds in a function-local control-flow graph.

When i is an ensure instruction (`sens` k) with its argument $k \in \mathbb{N}^0$, the current upper bound is increased to k (3.3a). The transfer function of a call i depends on the size of the stack cache $|SC|$ and the minimal displacement $d_{min}(i)$ of the functions that are potentially invoked by the call (3.3b). ($d_{min}(i)$ is computed as described in Section 3.2.1.) For all other kinds of instructions the transfer function is the identity function (3.3c).

The o_{bound} also needs to be propagated between instructions, i.e., from the *OUT*-values of the predecessors of an instruction to an instruction's *IN*-value. This is done using the transfer functions:

$$IN(i) = \begin{cases} |SC| & \text{if } i = r & (3.4a) \\ \max\limits_{p \in Pred(i)} (OUT(p)) & \text{otherwise} & (3.4b) \end{cases}$$

At joins in the CFG, the maximum (*max*) of all incoming values is used as the *meet operator* (3.4b). In order to model a full cache at the entry of the current function, the *IN*-value of the first instruction in the function's CFG r is initialized with the full size of the stack cache $|SC|$ (3.4a). For the remaining initial values at each program point, we use the value $0 \in \mathcal{D}$ which is the neutral element with respect to max.

Example 4. *Given the control-flow graph shown in Figure 3.5 and a stack cache size of* 4*, the analysis starts by associating the IN-value of instruction* i_1 *with the full stack cache size* ($|SC| = 4$)*. Since* i_1 *is an* `sres`*-instruction the identity function leaves the stack cache occupancy bound* (o_{bound}) *unchanged for the following instruction.* i_2 *is a call with a minimal displacement of* 1 ($d_{min}(i_2) = 1$)*. Applying the corresponding transfer function (see Eq. 3.3b) yields a stack occupancy of* 3 *after the call, which is propagated to the IN-value of instruction* i_3*. Applying the transfer function from Eq. 3.3a leaves* o_{bound} *unchanged. A stack occupancy of* 3 *is thus propagated to its two successors in the CFG. The successor on the right,* i_5 *is a call instruction with a displacement of* 3*, which results in an occupancy bound of* 1 *after the call. The instruction following the call is again an ensure. As the current* o_{bound} *value is smaller than the stack space ensured by this instruction* $OUT(i_6)$ *is assigned a value of* 2*. Finally, the analysis determines* $IN(i_7)$ *by selecting the maximum among the values* $OUT(i_4)$ *and* $OUT(i_6)$*.*

Worst-Case Filling of Ensures

To analyze the filling behavior of `sens`-instructions, we formally define `AnalyzeEnsures`, which has to solve a similar data-flow analysis problem to that before. The algorithm takes two arguments: the function's CFG G and a mapping from call sites to their respective maximum displacement D_{max}. The analysis is based on the framework defined at the beginning of this section, with domain $\mathcal{D}$. The transfer functions are as follows:

$$d_{loc}(i) = \min(|SC|, d_{max}(i)) \tag{3.5}$$

$$OUT(i) = \begin{cases} k & \text{if } i = \texttt{sres k} \\ \max(IN(i), k) & \text{if } i = \texttt{sens k} \\ \min(IN(i), |SC| - d_{loc}(i)) & \text{if } i = \mathit{call} \\ IN(i) & \text{otherwise} \end{cases} \tag{3.6}$$

$$IN(i) = \begin{cases} 0 & \text{if } i = r \\ \min_{p \in Pred(i)}(OUT(p)) & \text{otherwise} \end{cases} \tag{3.7}$$

Opposite to the upper bound for stack cache occupancy, we now aim to find a *minimum* bound. Thus the meet operator in (3.7) accordingly changes to min and the initial value is $|SC|$.

With the solution of the data-flow analysis above, we can compute the worst-case filling cost for every `sens`-instruction (assuming constant per-block fill cost $\widehat{c_f}$):

$$fillcost(i) = \widehat{c_f} \cdot (OUT(i) - IN(i)) \tag{3.8}$$

Example 5. *Consider function* `A` *in Figure 3.2a and a stack cache with size* $|SC| = 4$*. The ensure analysis starts by applying the transfer function (Eq. 3.6) of the* `sres`*-instruction* a_2*. This causes a minimum stack occupancy of* 2 *to be propagated to the following call* a_3 *to* `B`*. From the analysis of the maximum displacement it is known that* `B` *might spill the entire content of the stack cache since (*$d_{max}(a_3) = 5$*) including what* `A` *reserved. The minimal stack occupancy after the call thus has to be assumed to be* 0 *in the worst-case, which is propagated to* $IN(a_4)$ *of the following ensure. As the stack cache might be empty, the ensure has to reload the* 2 *blocks specified as its argument and its OUT-value thus becomes* 2*. The next call instruction* a_5 *invoking* `C` *has a maximum displacement* $d_{max}(a_5) = 2$*. The analysis determines that* $|SC| - 2 = 2$ *and thus equal to the minimum occupancy (Eq. 3.6), which therefore does not change (i.e., the content of* `A`*'s and* `C`*'s stack frames both fit into the stack cache). Consequently it is not necessary for the final ensure instruction* a_6 *to fill any data. The instruction could even be removed without any side-effect.*

3.2.3 Worst-Case Spilling

Given upper bounds for the stack cache occupancy, we can finally define `AnalyzeReserves`, which computes the worst-case spilling of reserve instructions within the program. This analysis problem is context-sensitive, i.e., it depends on the nesting of function calls. More precisely, an `sres`-instruction will spill when the occupancy before the reserve is too high so the requested space is not available. As we have noted before, the maximum occupancy at function entry is specific to a calling-context and depends on the accumulated occupancy of the nested function calls leading to the entry. On the other hand, we need to locally account for spilling caused by calls to other functions that may have evicted parts of the stack cache before the execution reaches the respective reserve. Given the o_{bound} value (see Section 3.2.2) for a call site, the context-dependent analysis of its maximum stack occupancy becomes simple: when a new stack occupancy is derived for the entry of the enclosing function, either (1) the *new occupancy* value including the locally reserved space is propagated to the call site or (2) the *bound* is propagated to the call site, whichever is smaller. From the context-dependent stack occupancy a graph can be constructed that can be used to represent the spill costs of the `sres`-instructions in individual contexts:

Definition 4. *The* spill cost analysis graph *is a directed graph* $SCA = (ACG, V_c, E_c)$ *consisting of nodes in* V_c *representing occupancy-annotated calling-contexts and edges in* $E_c \subseteq V_c \times V_c$ *that correspond to call sites of the annotated call graph. The nodes are pairs* $(n, o) \in V_c$*, where* n *is an ACG node and* $o \in \mathbb{N}^0$ *is the context's stack cache occupancy.*

There exists a $1 : n$ mapping between call sites in the call graph and edges in in the SCA graph, i.e., one edge representing a call site translates to one or more SCA edges. We write $e_{acg} \cong e_{sca}$ when the pair (e_{acg}, e_{sca}) from both graphs are *call*-related in this way.

Algorithm 3 constructs an SCA graph from an annotated call graph (cf. Definition 2) using a simple work list. The analysis starts at the sink node s, which is assumed to have a stack occupancy of 0 (l. 3–2). From this initial context other SCA contexts are derived

Algorithm 2 Main steps of the stack cache analysis for both, the ensure and reserve analysis problems.

Input: *ACG* … The call graph of the program.
CFGs … The CFGs of all functions.

Output: Annotate `sens`- and `sres`-instructions with their worst-case filling and spilling behavior.

```
▷ Minimum/maximum displacement at call sites.
$D_{min}$ = ComputeMinimumDisplacement($ACG$)
$D_{max}$ = ComputeMaximumDisplacement($ACG$)
foreach $G \in CFGs$ do
    ▷ Ensure analysis.
    AnalyzeEnsures($G, D_{max}$)
    ▷ Bound worst-case occupancy at call sites.
    $O_{bound} = O_{bound} \cup$ BoundOccupancy($G, D_{min}$)
▷ Reserve analysis.
AnalyzeReserves($ACG, O_{bound}$)
```

Algorithm 3 Constructing the Spill Cost Analysis Graph (SCA), as part of `AnalyzeReserves`.

Input: $ACG = (V, E, s, z) \ldots$ An annotated call graph.
$O_{bound} \ldots$ The occupancy bounds of call sites.
Output: Context-sensitive stack cache occupancy derived for the $SCA = (ACG, V_c, E_c)$.

```
1:  ▷ Initialize the SCA graph and work list
2:  $E_c = \emptyset$; $V_c = \{(s, 0)\}$
3:  $W = \{(s, 0)\}$
4:  ▷ Iteratively derive new SCA contexts
5:  while $W \neq \emptyset$ do
6:      ▷ Process some context from the work list
7:      Let $c = (u, o) \in W$ in
8:          $W = W \setminus c$
9:          foreach $e = (u, v, w) \in E$ do
10:             ▷ Derive a potentially new SCA context
11:             Let $c' = (v, \min(o + w, O_{bound}[e]))$ in
12:                 ▷ Update the work list
13:                 if $c' \notin V_c$ then
14:                     $W = W \cup c'$
15:                 ▷ Update the SCA graph
16:                 $V_c = V_c \cup c'$
17:                 $E_c = E_c \cup (c, c')$
18: return $SCA$
```

by processing one context from the work list at a time (l. 7). The context is removed from the work list and new contexts are constructed considering the current occupancy and the weighted call sites associated with the corresponding call graph node of the context (l. 11). Note the use of the occupancy bound that was computed before (see Section 3.2.2). If the so discovered contexts were not yet known, they are added to the work list (l. 14). Finally, the SCA graph is updated to cover the newly discovered contexts (l. 16).

Using the occupancy information of the SCA graph, the spill costs of the individual reserve instructions in the program can immediately be derived. Assuming constant per-block spill cost $\widehat{c_s}$ and a context $c = (n, o)$ of an `sres`-instruction i that reserves k blocks, the spill cost is:

$$spillcost(i, c) = \widehat{c_s} \cdot \max(0, o + k - |SC|) \tag{3.9}$$

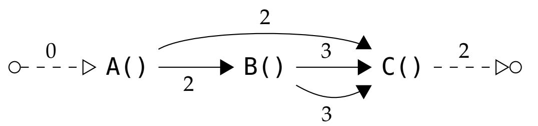
Example 6. *Assuming a stack cache with* 4 *blocks, the spill cost analysis graph shown in Figure 3.6 is constructed from the example program from Figure 3.2. While only a single context is constructed for the functions* `A` *and* `B` *respectively, three different contexts are created for* `C`*. The stack occupancy of these contexts are* 4*,* 3*, and* 2 *blocks. This results in a worst-case spilling of*

1 *and* 2 *blocks respectively for the first two context. No spilling is performed in the last context. Note that the edges in the SCA graph correspond to edges in the call graph (shown in Figure 3.6a).*

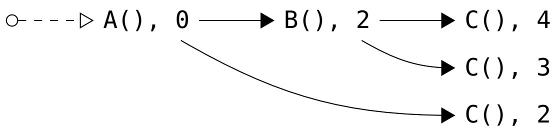
3.2.4 Combining the Analyses

From the individual analyses above, a simple algorithm that solves both stack cache analysis problems at the same time can be devised (see Algorithm 2). The algorithm takes a call graph and a set of control-flow graphs (one for each function in the program) as input and associates every `sens`- and `sres`-instruction with information on their respective worst-case filling and spilling behavior. It proceeds by first computing the minimum and maximum displacement of the call sites within the program using the input call graph (l. 2-3). Next, the worst-case stack occupancy at call sites is bounded (l. 8) using the previously computed minimum displacements (D_{min}) for each function separately. The ensure analysis (l. 6), which relies on the maximum displacements (D_{max}), is similarly performed for each function individually. Information on the minimum stack occupancy is propagated locally from call sites to the ensure instructions. Finally, the reserve analysis (l. 10) is performed on the call graph. It uses the occupancy bound (`BoundOccupancy`, l. 8) and propagates context-dependent information on the stack occupancy to the individual reserve instructions.

Computational Complexity Examining the computational complexity of the individual analysis phases, `AnalyzeReserves` is bounded by the number of possible contexts, i.e., the number of functions times the constant number of stack cache states. The data-flow analysis problems of `BoundOccupancy` and `AnalyzeEnsures` are similarly linear, but in the number of CFG nodes (due to the constant and typically low $|\mathcal{D}|$, the number of iterations until the analysis fixpoint is sub-polynomial). The remaining two functions are those that compute the minimum and maximum displacement. Their underlying path search problems are polynomial and NP respectively, when considered without user constraints. But also when relying on an ILP formulation, the problems have shown to be efficiently solvable even for graphs larger than the ones encountered here (the number of functions in a program being naturally low). The shortest and acyclic longest path searches are quadratic and linear, respectively.



(a) Call Graph



(b) Spill Cost Graph

Figure 3.6: Annotated call graph and spill cost graph of the program from Figure 3.2.

SCA Graph Pruning During the SCA graph construction (cf. Algorithm 3), several contexts might be created for the same function whose spill costs evaluate to 0, but have different occupancy values. When analyzing the worst-case spilling behavior of a program, these contexts are equivalent and can be merged. Note that this merging could also be done during graph construction, at the expense of conservatively collapsing descendant contexts having different costs.

Furthermore, potentially infeasible contexts may be created, e.g., when the recursion depth of a recursive function is limited, which can be eliminated during a post-processing phase or during graph construction. The maximum displacement, for instance, can be used to prune parts of the graph while processing it.

In addition to the loss-less pruning opportunities mentioned above, the size of the graph can also be reduced by merging contexts and annotating the merged context with the larger occupancy and higher spill cost. This would, for instance, allow us to reduce the complexity of any subsequent analysis that takes spill cost into account. The advantage of this approach is that the degree of context merging can be decided on-demand, which allows to trade analysis precision against computational complexity.

3.3 IPET Integration

In the previous analysis steps, we have computed worst-case bounds for spilling and filling of the stack cache at `sres` and `sens` instructions. We now want to integrate the latencies imposed by the access to external memory into the overall timing analysis. Since we rely on implicit path enumeration for WCET analysis, our goal is to appropriately add the worst-case cost related to every *spill* and *fill* site to the ILP, which is maximized to find the worst-case execution path. The IPET problem can be formulated to be context-sensitive (cf. Section 2.1.2) by duplicating ILP variables for each context they should be considered in, and adapting the ILP constraints accordingly. Ensure analysis is context-independent, thus its results can be added directly to the worst-case timing cost of the respective code in the ILP, which contains the `sens` instruction. This procedure can be applied to a context-insensitive IPET problem in the exact same way as to a context-sensitive one. Integrating spill cost requires more care, as the spill cost analysis graph is fully context-sensitive with regard to the observable stack cache state. The contexts considered by the reserve analysis might differ from the contexts used by the IPET analysis. Two options can be considered to overcome this problem. The first option is to merge the SCA contexts (as described above) so they match the contexts of the IPET analysis. The merging might then cause some loss of precision. The other option is to account for the full context information encoded in the SCA graph by introducing additional ILP variables representing the nodes and edges of the SCA graph. ILP constraints then connect these ILP variables to those of the original IPET problem. In the following we demonstrate how to integrate the stack cache analysis results into a context-*insensitive* IPET formulation: (1) by completely merging stack cache spill contexts and (2) retaining all stack cache context by adding the SCA graph to the IPET problem.

We consider the ILP for the IPET graph $G = (V, E)$, with nodes $v \in V$ representing basic blocks in the program and edges $e \in E$ representing the flow of control between them. For our extension, we represent the nodes of the SCA graph as C (contexts) and its edges as transitions between contexts as $T \subseteq C \times C$. The ILPs below will use the variables

- x_i for the amount of flow on the edges of the IPET graph and
- y_j for the frequency of transition edges in the SCA graph.

Furthermore, the constant values in the ILP, which we decorate with a hat (e.g. $\hat{c}$) are

- $\hat{t}_i$ the cost (weight) of an edge in a program's IPET graph,
- $\hat{s}_i$, $\hat{s}'_i$ IPET edge weights augmented with stack cache related cost, and
- $\hat{r}_i$ spill cost attached to an edge in the SCA graph.
- $\hat{a}_{ij}$, $\hat{a}'_{ij}$ and k are constant factors that allow to constrain the program flow.
- $\hat{Bf}$ is a local upper bound for the frequency of a block, that is, within its function (e.g., given node frequencies x_i, a function entry $r \in V$ and a node $u \in V$, the relation $x_u \leq \hat{Bf} x_r$ holds).

The basic ILP formulation has structural (3.11) as well as flow constraints with relations in $\circ \in \{\leq, \geq, =\}$ (3.12):

$$\text{WCET} = \max \sum_{i=1}^{|E|} \hat{t}_i x_i \tag{3.10}$$

$$\text{s.t.} \sum_{e_m=(u,v)} x_m = \sum_{e_n=(v,w)} x_n \tag{3.11}$$

$$\sum_{e_j \in E} \hat{a}_{ij} x_j \circ \sum_{e_j \in E} \hat{a}'_{ij} x_j + k \tag{3.12}$$

$$x_i \in \mathbb{N}^0 \tag{3.13}$$

For edge $e_i = (u, v)$, the constant timing value $\hat{t}_i$ represents the (longest) execution between the start of block u and the start of block v. The decision variable x_i counts the execution frequency for the same edge. The structural constraints (3.11) ensure that the sum of control flow entering and leaving a node is equal, thus ensuring that no flow gets added or lost, except at the distinguished start and end nodes. Equations (3.12) represent arbitrary constrains on the control flow of the program, provided by prior data-flow analysis or as user annotations. Maximizing the weighted sum for all edges is the objective function (3.10) of the ILP and gives the total WCET bound of the program.

To extend the IPET model with spill and fill cost, we need to map `sres` (`sens`) instructions to their containing blocks and their worst-case cost to edges of the IPET eventually. Thus, we define `SRES`(u) as a function that returns the single reserve instruction of the

IPET block u, or of the SCA context u. Likewise, we define $\texttt{SENS}(v)$, to return the set of ensure instructions in block v. Also, let $F \subset E$ be the IPET edges, which represent function calls.

Model 1: Merging Stack Cache Contexts

We begin with the simple extension to the ILP model, which involves a context-insensitive approximation of stack cache spill cost. In doing so, we augment the constant timing cost $\hat{t}_i$ for edges in the IPET graph, with results from the previously described analysis:

$$\text{Let } e_i = (u, v) \text{ and } k = \texttt{SRES}(u) \text{ in :}$$

$$\hat{s}_i = \hat{t}_i + \sum_{j \in \texttt{SENS}(u)} \mathit{fillcost}(j) + \max_{c \in C} \mathit{spillcost}(k, c) \tag{3.14}$$

For every block in the CFG, which contains analyzed `sens` or `sres` instructions, Equation (3.14) adds the fill or spill cost to the cost of the outgoing IPET edges. In the former case, the cost is made up of a sum of $\mathit{fillcost}$ values, since more than one ensure can be found in a block. In the latter case, we need to select the worst-case cost over all contexts from the SCA graph. In order to add the stack cost to the ILP, we only need to replace the constant cost in the objective function:

$$\text{WCET} = \max \sum_{e_i \in E} \hat{s}_i x_i \tag{3.15}$$

Due to the modified objective function (3.15), the ILP now computes a valid upper bound for the program's execution time, considering latencies caused by the stack cache.

Model 2: Integrating Stack Cache Contexts

With the previous model, we have not taken advantage of the stack cache context that is contained in the SCA graph. To do so, we introduce nodes and edges from the SCA graph in the ILP as contexts C and transitions between contexts T, respectively. The IPET graph thus becomes the quadruple $G = (V, E, C, T)$. Spill frequencies in the SCA graph are modeled by adding a variable y_j for each edge $e_j \in T$. We add the following equations to the original IPET problem (Equations (3.10) to (3.13)).

$$\forall e_i \in F : \quad x_i = \sum_{e'_j \in T : e'_j \cong e_i} y_j \tag{3.16}$$

$$\forall e_n = (v, w) \in T : \quad \sum_{e_m = (u,v) \in T} \hat{B^f}\, y_m \geq y_n \tag{3.17}$$

$$y_j \in \mathbb{N}^0 \tag{3.18}$$

Equation (3.16) links a function call edge in the original IPET to those edges that represent the same call in the SCA graph. Recall that the relation $e_{acg} \cong e_{sca}$ means that two edges in our composite graph correspond to the same call site. The constraint forces the accumulated frequency of the call site to be equal in both graphs. Similar to constraints that appear in a standard IPET ILP, Equation (3.17) is a structural constraint for the SCA graph. Different to the IPET graph though, the semantics of the SCA graph require that flow originates at the root node and flows towards the leaves. Thus, we can constrain the incoming flow of an SCA node to be greater or equal to its outgoing flow. In the case of a call site within a loop, the bound $\widehat{B^f}$ expresses that the (looping) call site's IPET frequency does not translate into more than one SCA context transition. This follows directly from the semantics of stack behavior in loops. For call sites outside of loops, $\widehat{B^f}$ is simply 1. Note that with this constraint, we currently do not support programs with recursive function calls. At this time, this is generally not supported by Platin's WCET analysis.

Generally speaking the constraints (3.17) prevent solutions of the ILP, where (through maximization), an expensive spill context is selected in the SCA graph, although the stack cache occupancy could not have been exceeded (i.e., there is no path of active edges from the root of the SCA graph to this node). Further, we have to compute the constant timing cost for every `sres` instruction i, for each spill context $v \in C$ of the SCA graph and translate it to the incoming call site of i:

$$\begin{aligned} &\text{Let } e_j = (u, v) \in T \text{ and } i = \mathtt{SRES}(v) \text{ in :} \\ &\qquad\qquad \hat{r}_j = spillcost(i, v) \end{aligned} \tag{3.19}$$

Finally, we reuse the $fillcost$-augmented edge cost from the first model and modify the ILP's objective function to include both, the context-independent fill cost and context-sensitive spill cost at the same time:

$$\begin{aligned} &\text{Let } e_i = (u, v) \text{ in :} \\ &\qquad \hat{s'}_i = \hat{t}_i + \sum_{j \in \mathtt{SENS}(u)} fillcost(j) \end{aligned} \tag{3.20}$$

$$\text{WCET} = \max \sum_{e_i \in E} \hat{s'}_i x_i + \sum_{e_j \in T} \hat{r}_j y_j \tag{3.21}$$

3.4 Well-Formed Programs

The presented algorithms are based on the simple program model described in Definition 1, which assumes a single `sres`-instruction at the entry of a function and a single `sfree`-instruction at function exit. However, the approach is easy to extend to the larger class of *well-formed* programs. We define a well-formed program in terms of the paths through the program's functions:

Definition 5. *A program is well-formed when all functions in the program are well-formed.*

Definition 6. *A function with CFG $G = (V, E, r, t)$ is well-formed, if every path of nodes $p = (n_1, \dots, n_m)$, where $\forall\, 0 < i < m\colon (n_i, n_{i+1}) \in E$, satisfies one of the conditions:*

1. *No instruction $n_i \in p$ is an `sres`- or `sfree`-instruction.*
2. *Two indices $0 < i_r < i_f \leq m$ exist, such that n_{i_r} is the first `sres`-instruction and n_{i_f} is the last `sfree`-instruction on the path, and the amount of space reserved by n_{i_r} is equal to the amount freed by n_{i_f}, and the path $p' = (n_{i_r+1}, \dots, n_{i_f-1})$ is empty or well-formed.*

The above definition can also be extended to cover `sens`-instructions. Note that the definition of well-formed paths is based on *all* possible paths through the CFG, including potentially infeasible paths. Well-formed programs have the nice property that allows them to be analyzed using the previously described algorithms with only minor modifications:

Lemma 1. *Given the CFG $G = (V, E, r, t)$ of a well-formed function and a node $n \in V$, the accumulated amount of space reserved on the stack cache locally within the function is identical at n for all paths $(r, \dots, n)$, i.e. reaching n from the root node.*

Proof. Consider two distinct well-formed paths $p_1 = (r, \dots, n_r^1, \dots, n, \dots, n_f^1, \dots t)$ and $p_2 = (r, \dots, n_r^2, \dots, n, \dots, n_f^2, \dots t)$, where n_r^1 and n_r^2 are the last `sres`-instructions on their respective paths before n, such that the sub-paths $p_1' = (n_r^1, \dots, n, \dots, n_f^1)$ and $p_2' = (n_r^2, \dots, n, \dots, n_f^2)$ are well-formed.

Since all paths in G are well-formed, also the path $p_3 = (r, \dots, n_r^1, \dots, n, \dots, n_f^2, \dots t)$ has to be well-formed. This implies that the amount of space reserved by n_r^1 and n_r^2 has to be equal.

This property can be extended to sequences of `sres`-instructions, preceding n on a path, whose matching `sfree`s succeed n on the path. Since these sequences have to match across all paths passing through n, the accumulated amount of space reserved locally within the function has to be identical over all paths. ■

The previous lemma allows us to adapt the definition of the annotated call graph (cf. Definition 2) to weight the edges in the graph using the accumulated amount of stack space reserved locally within functions. The algorithms to compute the minimum and maximum displacement from Section 3.2.1 can then be used without modification. The data-flow analyses defined in Section 3.2.2 have to be adapted to account for the nesting of `sres`-instructions, but otherwise proceed as before. The SCA graph also needs to be adapted to capture reserve instructions within functions explicitly using *logical* contexts. This can easily be done as the propagation rules to construct logical contexts from reserves are the same as those for calls. It is even possible to model the effect of loops explicitly in the SCA graph by loop peeling, i.e., several logical contexts may be constructed for one reserve instruction.

Benchmark	Total WCET Simple	Total WCET SCA	Stack-Cache Simple	Stack-Cache SCA
mdh-adpcm	2,972,821	2,972,821	43,697	43,697
mdh-cnt	3,811	3,811	13	13
mdh-compress	21,952	21,952	169	169
mdh-crc	52,164	52,156	16	8
mdh-edn	49,864	49,864	10	10
mdh-expint	78,350	77,750	1,209	609
mdh-fdct	1,114	1,114	8	8
mdh-jfdctint	2,316	2,316	8	8
mdh-matmult	98,498	98,490	21	13
mdh-minmax	102	102	6	6
mdh-ndes	46,202	46,202	170	170
mdh-select	28,132	28,132	12	12
mdh-statemate	660	660	9	9
mdh-ud	34,616	34,580	712	676

Table 3.1: WCET bounds including stack cache overhead as computed by Platin

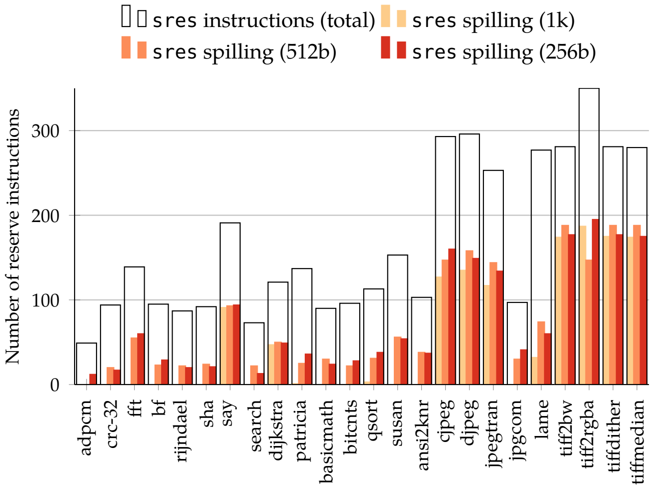


Figure 3.7: Total number of `sres`-instructions and number of `sres`-instructions potentially spilling in the worst-case (lower is better).

3.5 Evaluation

We evaluated our approach using the LLVM-based compiler and analysis framework of the Patmos processor. Note that the compiler automatically allocates stack data on the stack cache, but in a conservative manner, which results in low stack cache utilization. Platin with its internal IPET-based longest-path search is used as the WCET analysis tool. It processes the program binary and all relevant analysis information generated by LLVM and the user (cf. Section 2.2.1).

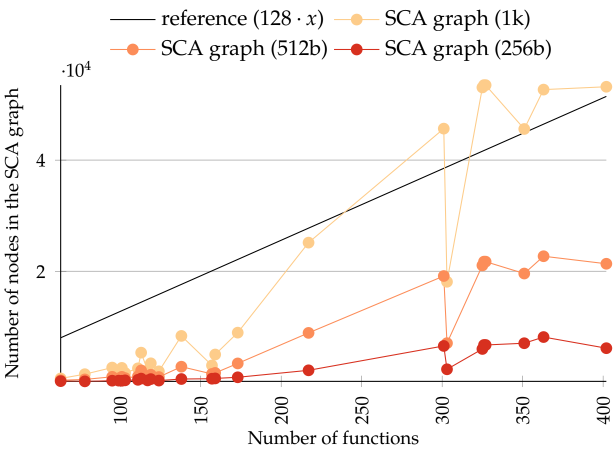


Figure 3.8: The size of the Spill Cost Analysis graph is linear in the number of functions (lower is better).

We first evaluate stack cache analysis integrated into Platin's WCET analysis for the Mälardalen benchmark suite. Each program in the Mälardalen benchmark suite represents an algorithm, which would appear as part of an actual application. Therefore, to see the effects from stack cache spills and fills, we assume its size to be only 256 bytes (`256b`) and its occupancy full at analysis start. Also because of the small size of the benchmarks, we enabled only basic optimizations (`-O1`) in the Clang/LLVM compiler, and disabled function inlining. Table 3.1 contains the results from integrated WCET analysis. A comparison between the "Simple" integration model with context-insensitive stack cache cost and the "SCA" model, which integrates the stack cache context using the *spill cost analysis graph*, shows a mild improvement of the WCET bound. The program `mdh-expint` benefits most, but in general, the Mälardalen benchmarks don't exhibit prominent calling behavior. For the programs analyzed, only up to 5% of worst-case cycles are caused by the stack cache. The overall analysis overhead including the stack cache analysis described in Section 3.2 and solving of the integrated IPET from Section 3.3 took less than one second per benchmark. This limited experiment was the first we could perform using the Platin analysis framework, which is still being developed at this time.

Programs from the MiBench benchmark suite [62] were selected to evaluate whether stack cache analysis can scale to large programs with more complex calling behavior. They were compiled using aggressive optimizations (`-O3`) and subsequently analyzed using our technique from Section 3.2. Since MiBench programs cannot be analyzed by Platin at this point, these stack cache analysis results could not be applied to compute a total WCET bound. The analysis assumes stack cache sizes of 1024 (`1k`), 512 (`512b`), and 256 (`256b`) bytes with 4 byte blocks. The compiled benchmarks contain between 9550 and 74291 instructions, of which 0.3 - 0.5% are `sres`- and 0.1 - 5.5% are `sens`-instructions. While it does not reflect how a Patmos system will eventually be configured for a real-word scenario, this experimental setup allows us to evaluate the behavior of stack cache analysis for up to 256 blocks. *Realistic configurations are more likely to have* 2 *to* 4 *KiB of stack cache, a larger block size of e.g.* 16 *bytes (resulting in up to* 256 *blocks), and —enabled by an additional analysis in the compiler— more data allocated to the stack cache.*

Figure 3.7 summarizes the result of the reserve analysis (Section 3.2.3). It shows the total number of `sres`-instructions in the benchmarks (white bar) as well as the number of potentially spilling instructions among theses (colored bars). For the `1k` configuration the analysis can prove that almost no spilling will occur at runtime as the benchmarks have a rather shallow nesting of function calls. Even for the smallest cache configuration, the analysis finds that on average, only about 37% of `sres`-instructions may cause worst-case spilling. Note that for the combination of a small stack configuration and large benchmark programs (e.g. `tiff2rgba`) the compiler needs to avoid stack frames from exceeding the stack cache size. This results in different displacement patterns and can result in fewer active `sres`-instructions. The analysis results reflect the observed spilling behavior at runtime reported in previous work [4].

More detailed numbers are presented in Table 3.2. In total 728 ILPs are generated (column ILP) during the displacement analysis (Section 3.2.1), 94 of which are due to `djpeg`. Note that the shortest path search was also performed using ILP to disambiguate calls through function pointers.

The number of contexts computed by the reserve analysis (Section 3.2.3) before pruning grows linearly with the stack cache size (columns O_{1k}, O_{512b}, O_{256b}), while the growth is much smaller for the pruned graphs (columns P). The highest number of contexts is initially computed for the `1k` configuration, where up to 53487 distinct contexts (`tiffdither`) are computed. Most of these are irrelevant for the worst-case spilling and can thus be eliminated. The pruned graphs only retain 15% of the original contexts in the mean (ignoring empty graphs). For the `256b` configuration fewer contexts are computed, at most 8198 distinct contexts for `djpeg`. However, a larger number of contexts is retained during pruning (29%).

The ensure analysis (Section 3.2.2) finds that very few of the ensure instructions may cause filling in the worst-case (columns F_{1k}, F_{512b}, F_{256b}). Up to 99% (`adpcm`) of the ensures never perform any filling for the `256b` configuration (in the mean 79%, column Ratio_{256b}).

It is worth noting that the number of nodes in the SCA graphs grows not only in relation to the number of functions in a program, but also with the size of the stack cache. This effect can be seen in Figure 3.8. It can be explained by the fact that the graph has to keep track of a higher number of distinct occupancy contexts. Once the stack cache size exceeds a point, where a considerable part of functions cannot cause any spilling at all, this would counteract the graph growth.

As most parts of the analysis are linear in the program size, the computational overhead of the entire analysis is low. Using an unoptimized executable, the average analysis time was 1.30s, 0.75s, and 0.46s for the `1k`, `512b`, and `256b` configurations respectively.

3.6 Related Work

In this section, we give an overview of related data cache and stack-specific analyses. We also describe scratch-pad memory, an alternative to the hardware overhead that comes with dynamic caching. Embedded systems use scratch-pad memory for reasons of efficiency as well as predictability.

Data Cache Analysis

Static analysis [75, 71] of caches typically proceeds in two phases: (1) potential addresses of memory accesses are determined, (2) the potential cache content for every program point is computed. Depending on factors such as size and replacement strategy of the cache, an analysis needs to keep track of a potentially huge state space reflecting a long execution history of the program. The stack cache allows for a simpler analysis model that does not require the precise knowledge of addresses. This eliminates a source of complexity and imprecision. The hardware states of the stack cache can, furthermore, be summarized using the stack occupancy. The analysis is thus simplified drastically, allowing us to compute fully context-dependent cache states. This information can be encoded on-demand into the actual timing analysis. This is further supported by the observation that the stack cache serves up to 75% of the dynamic memory accesses [4].

Register Window Analysis

Tidorum Ltd.'s WCET analysis tool Bound-T supports the analysis of over- and underflow of the register-window mechanism of the SPARC architecture [22, Section 2.2]. They compute bounds on the number of register windows that are pushed to and popped from the stack and use these bounds to classify the corresponding `save` and `restore` instructions as trapping or non-trapping. The analysis imposes several limitations on the program structure and operating system's trap handler. Additionally, it is unclear how calling contexts and recursion are handled.

Stack Depth Analysis

Our approach to compute the stack cache displacement has some similarity to techniques used to statically analyze the maximum stack depth [39, 22]. Instead of only finding the maximum stack depth for the program's entry function , i.e., a longest path on a weighted call graph, the displacement information is required for every function. We duplicate the call graph, such that one copy represents the cost-free head and the second copy the weighted tail of the desired path.

Scratch-Pad Memory

In the context of embedded systems, scratch-pad memory (SPM) is an alternative for the use of caches. SPM is a fast local memory that avoids the logic required to keep caches synchronized with the external memory they are mapped. An SPM is explicitly managed in software (or sometimes by a memory management unit). Which data is allocated to SPM can be decided statically or dynamically, by the programmer or the compiler, and is an optimization problem dependent on the memory usage pattern of a specific application. A software solution using an SPM for stack data is described in [26]. Its circular management of stack frames within the SPM is similar to our stack cache. Yet its replacement strategy is different, because it is implicit: a "software SPM manager" handles overflow and underflow of the SPM's stack region and transfers frames to and from external memory. An optimal ILP-based model for the placement and transfer of stack data kept in scratch-pad memory is described in [25]. Their model has a similar restriction to our stack cache design, namely that a whole (not partial) stack frame of the currently active function must be in the SPM. While, similar in concept, the SPM solutions above, target efficiency, in the sense of average-case performance and energy efficiency, and do not consider the WCET bound.

In a soon to be published paper [3], Kim et al. describe a stack allocation model for SPMs, which optimizes for worst-case execution time. Like us, the authors assume a predictable architecture, in their case the PRET [32] architecture. Their optimization model is solved to optimality using an ILP solver and heuristically for those programs that the model does not scale to feasibly.

			SCA graph size (original/pruned)						Number of `sens`-instructions (filling/non-filling)						
Benchmark	Fun.	ILP	O_{1k}	P_{1k}	O_{512b}	P_{512b}	O_{256b}	P_{256b}	F_{1k}	NF_{1k}	F_{512b}	NF_{512b}	F_{256b}	NF_{256b}	$Ratio_{256b}$
adpcm	63	12	755	0	428	0	270	46	0	99	0	99	1	98	0.99
crc-32	103	2	1750	0	821	62	415	80	0	358	4	354	68	290	0.81
fft	159	14	5060	0	1751	258	761	207	0	842	10	832	182	660	0.78
bf	111	14	2574	0	1136	66	509	104	0	368	2	366	64	304	0.83
rijndael	95	2	2710	0	1051	119	352	61	0	358	7	351	78	280	0.78
sha	101	2	2655	0	1048	97	338	76	0	360	5	355	73	287	0.80
say	217	20	25170	2422	8945	1781	2218	700	17	1907	77	1847	271	1653	0.86
search	78	2	1521	0	581	75	231	29	0	311	6	305	67	244	0.78
dijkstra	138	16	8402	483	2868	410	653	197	2	586	8	580	167	421	0.72
patricia	157	16	3100	63	1565	133	706	159	2	629	7	624	164	467	0.74
basicmath	99	2	1691	0	780	92	363	82	0	839	18	821	85	754	0.90
bitcnts	117	16	2193	0	930	76	407	77	0	365	5	360	67	298	0.82
qsort	124	4	2037	14	1010	103	373	110	2	588	8	582	166	424	0.72
susan	173	14	8989	0	3455	411	964	311	0	760	10	750	101	659	0.87
ansi2knr	119	14	3478	0	1424	139	639	173	0	397	11	386	76	321	0.81
cjpeg	351	50	45599	17434	19615	9570	7086	4061	24	1669	487	1206	666	1027	0.61
djpeg	363	94	52677	14783	22725	8619	8198	3771	158	1388	421	1125	588	958	0.62
jpegtran	301	90	45664	17605	19168	9427	6597	3870	119	1306	438	987	614	811	0.57
jpgcom	113	14	5394	0	2196	168	730	211	0	387	7	380	72	315	0.81
lame	303	14	18134	237	7098	1196	2403	574	4	3898	70	3832	322	3580	0.92
tiff2bw	327	72	53482	16436	21748	9215	6771	3522	82	1449	269	1262	392	1139	0.74
tiff2rgba	402	72	53189	17542	21371	5806	6195	3111	129	2022	395	1756	518	1633	0.76
tiffdither	326	72	53487	17254	21753	9213	6776	3520	76	1457	263	1270	386	1147	0.75
tiffmedian	325	72	53065	17410	21079	8943	6055	2980	79	1407	267	1219	336	1150	0.77

Table 3.2: The number of functions, the number of ILP runs, the SCA graph size before and after pruning, as well as the number of ensure instructions that are potentially filling or are certain to not cause any filling.

Chapter 4

Criticality

The criticality metric expresses which parts of a program are relevant with regard to the global WCET. The metric can be derived for every basic block in the CFG and may take any value in the range 0 to 1, where lower values indicate less critical code (0 would indicate unreachable or dead code), while code on the WCEP is assigned a criticality of 1. Note that it would be possible to derive criticality values for CFG edges, however, we limit ourselves to basic blocks in this work. Since criticalities are derived for all basic blocks, they provide a more complete view than the WCET or even its associated WCEP alone. They thus provide ideal information for a programmer or software development tool aiming at improving the WCET of a program.

In the following two sections we introduces the concepts of our metric, and the algorithms we devised for its computation. Our ideas regarding an estimation approach for *criticality* and how to visually present its results to a programmer can be found in sections 4.3 and 4.4. We present an evaluation of the behavior and feasibility of our metric in Section 4.5. This part sheds some light on the properties of standard WCET benchmarks we described in Section 2.2.2. With a discussion of the possible application areas of the criticality metric in Section 4.6 and an overview of related work in Section 4.7 we end this chapter.

Criticality has been first published in [9] and presented at RTNS 2012. An in-depth description together with further algorithms for its computation can be found in a journal article [5] that has been published online and is pending print publication. A paper [2] describing the estimation approach for criticality computation and our ideas for visualization of criticality profiles has been accepted for publication. It will be presented at the upcoming ODES workshop as part of CGO 2014.

4.1 The Criticality Metric on Control-Flow Graphs

We will first give a definition of *Criticality* using the worst-case execution time induced by paths passing through a basic block and their relation to the global WCET. The notation for graph-related properties used in the following has been defined in Section 2.1.1.

Definition 7. *Given a weighted CFG* $G = (V, E, r, t)$ *and a node* $u \in V$*, the length of the longest path* p *passing through* u *is denoted by* $WCET(u) = \bar{\mathcal{W}}(p)$*, i.e.,* p *is the longest path such that* $u \in p$ *and there exists no other path* q*,* $u \in q$ *where* $\bar{\mathcal{W}}(p) < \bar{\mathcal{W}}(q)$*.*

Note that the length of the longest path from the root node to the sink node, and thus the WCET, is given by the longest path passing through the root r or sink t, i.e. WCET = WCET(r) = WCET(t).

Using the previous definition, and assuming a non-zero WCET, we can now formally define the criticality of an individual basic block as follows:

Definition 8. *The* criticality *of a basic block and its associated CFG node* u *is defined using the WCET over all paths passing through* u *and the global WCET as:*

$$Crit(u) = \frac{WCET(u)}{WCET}$$

Example 7. *Consider the paths of the weighted CFG in Figure 4.1. Assuming all edges have a weight of zero, the longest path of length* 15 *is given by* $(r, \mathtt{BB0}, \mathtt{BB2}, \mathtt{BB4}, \mathtt{BB5}, t)$*, followed by* $(r, \mathtt{BB0}, \mathtt{BB1}, \mathtt{BB4}, \mathtt{BB5}, t)$*, having a length of* 11*, and finally* $(r, \mathtt{BB0}, \mathtt{BB1}, \mathtt{BB3}, \mathtt{BB5}, t)$ *with a length of* 9*. Clearly,* `BB0`*,* `BB2`*,* `BB4`*, and* `BB5` *have to be considered critical since these blocks appear on the WCEP. Even though* `BB1` *is not on the WCEP itself, it has to be considered critical, since a path of length* 11 *passes through it. The remaining block* `BB3` *on the other hand is comparatively unimportant since none of the highly* critical *paths pass through it. This is reflected by the different criticalities assigned to the various blocks:* 1 *for blocks appearing on the WCEP,* 0.73 *for* `BB1`*, and* 0.6 *for* `BB3`*.*

Following the preceding definition of criticality for basic blocks, it is apparent that the WCET analysis has to be performed several times, once for every basic block of the program. It is, however, not necessary, to repeat all of it. The results of the data-flow, cache, or pipeline analyses remain valid throughout, thus it suffices to perform only the longest path search.

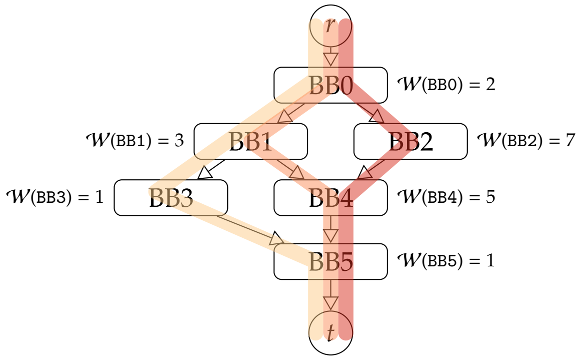


Figure 4.1: An example control-flow graph annotated with basic-block-local execution times and all potential execution paths highlighted.

4.1.1 Properties of Criticality

Since *criticality* is tightly bound to the concept of paths, i.e., the longest path passing through a CFG node, it is not surprising that structural properties of the CFG related to paths affect *criticality*. The properties we are specifically interested in, regard the relation between criticality values of different nodes in the CFG. Pre- and post-dominance in the graph provide information that can be exploited during the calculation of basic block criticalities.

Lemma 2. *Given a CFG* $G = (V, E, r, t)$*, two nodes* u *and* v *in* V*, and a weight function* $\mathcal{W}$*, the criticality of* v *is less than or equal to the criticality of* u*, if* u *pre-dominates* v*:*

$$u \; dom \; v \implies Crit(v) \leq Crit(u)$$

Proof. Suppose that u pre-dominates v and that the criticality of v is greater than the criticality of u, i.e., $\text{Crit}(v) > \text{Crit}(u)$ and consequently $\text{WCET}(v) > \text{WCET}(u)$. Since $\text{WCET}(v) > \text{WCET}(u)$, there exists a path passing through v, but not through u, that is longer than any path passing through u. The existence of this path implies $u \; \neg\text{dom} \; v$, which contradicts the initial assumption that u pre-dominates v. ■

Lemma 3. *Given two nodes* u *and* v *of a weighted CFG* G*, the criticality of* v *is less than or equal to the criticality of* u*, if* u *post-dominates* v*, i.e.:*

$$u \; pdom \; v \implies Crit(v) \leq Crit(u)$$

Proof. Analogous to Lemma 2. ■

The previous two lemmas provide upper bounds for CFG nodes, which are pre- or post-dominated by some other node. When the difference between very low criticality values is of no interest, this can be used to stop computing it for nodes, once the criticality of a pre- or post-dominating node is known to be sufficiently low (see Section 4.2.4 for the algorithm designed around this concept).

The dominance properties further allow us to derive the criticality of dominating nodes. However, for this we have to ensure that all paths passing through the dominator also pass through one of its dominated nodes. Control-flow graphs that are free from critical edges fulfill this property. Note that critical edges can be eliminated by *edge-splitting*, i.e., by placing additional basic blocks between the respective source and destination nodes of the edges.

Theorem 1. *Let* $S = \{v : u \; sdom \; v\}$ *be the set of nodes strictly pre-dominated by a node* u *of a weighted CFG* G_{cf} *that is free from critical edges. The criticality of* u*, strictly pre-dominating at least one node, is equal to the maximum criticality among the nodes in* S*:*

$$|S| \geq 1 \implies Crit(u) = \max_{v \in S} Crit(v)$$

Proof. The proof consists of two steps. We will first show that, in a CFG that is free from critical edges, all paths passing through a CFG node u, pass through one of the nodes in S. We will then prove the theorem itself.

Since the CFG is free from, critical edges, we may encounter two situations. In the first scenario, we assume CFG node u has multiple outgoing edges. Since no critical edges exist, all successors of the node have a single incoming edge and are thus pre-dominated by u. It trivially follows that all paths through u also pass through one of its successors.

In the second scenario, we assume CFG node u has a single outgoing edge leading to its successor s. We now have to show that u cannot strictly pre-dominate any node $n \neq s$ unless u strictly pre-dominates s. Assume that u strictly pre-dominates some node $n \neq s$, but does not strictly pre-dominate s. This implies that all paths passing through n pass through u and s (as s is the only successor of u). It follows that a path from s to n exists that does not pass through n. Since s is not pre-dominated by u, a path can be constructed that passes through s and n but not through u. This contradicts the initial assumption that u pre-dominates n. u thus either strictly pre-dominates its successor or does not strictly pre-dominate any node. This completes the first step of the proof.

Lemma 2 already shows that the criticality of all nodes pre-dominated by some CFG node u has to be smaller or equal to the criticality of u. It thus remains to show that u's criticality cannot be greater than the maximum criticality of any of the nodes it strictly pre-dominates. Suppose that some CFG node u exists whose criticality is greater than the criticality of any node strictly pre-dominated by u, i.e., $\forall v \in S : \mathrm{Crit}(v) < \mathrm{Crit}(u)$ and consequently $\forall v \in S : \mathrm{WCET}(v) < \mathrm{WCET}(u)$. Thus, there exists a path passing through u, but not through any node strictly pre-dominated by u. However, this is impossible since we showed in the first step of the proof that any path passing through u also passes through a node in S. ■

Theorem 2. *Let $S = \{v : u \; spdom \; v\}$ be the set of nodes strictly post-dominated by a node u of a weighted CFG G_{cf} that is free from critical edges. The criticality of u, strictly post-dominating at least one node, is equal to the maximum criticality among the nodes in S:*

$$|S| \geq 1 \Longrightarrow Crit(u) = \max_{v \in S} Crit(v)$$

Proof. Analogous to Theorem 1, using Lemma 3. ■

Corollary 1. *Two nodes u and v of a weighted CFG have the same criticality, if u pre-dominates v and, at the same time, v post-dominates u:*

$$u \; dom \; v \wedge v \; pdom \; u \Longrightarrow Crit(u) = Crit(v)$$

Proof. This follows immediately from Lemma 1 and 2. ■

The previous two theorems allow us to reduce the overhead for the computation of criticality for a whole program by exploiting the pre-dominance and post-dominance relation. It is thus not necessary to perform individual longest path searches for all basic blocks of a program. Instead it is sufficient to consider only a subset of the basic blocks and derive the criticality of the remaining blocks by simply traversing the pre- or post-dominator trees (see Section 4.2 for the concrete algorithm). Also note that all proofs hold in the presence of flow constraints (cf. Section 1.1), as these can only reduce the number of valid paths through the CFG. Dominance relations derived from the structure of the CFG thus cannot be invalidated. However, new dominance relations might emerge when only valid paths are considered. Taking these additional dominators into account, could further reduce the computational overhead of the algorithms presented later.

Example 8. *Consider the CFG node* `BB0` *in Figure 4.1, which pre-dominates its two successors* `BB1` *and* `BB2`*. It thus follows from Lemma 2 that the criticality of the two successors is smaller or equal to that of* `BB0`*, which evaluates to* 1*. Indeed, as shown in Example 7, the criticality of* `BB1` *is* 0.73*, while that of* `BB2` *is* 1*. The same also applies to* `BB5`*, which post-dominates its predecessors* `BB3` *and* `BB4` *(Lemma 3). In both cases, the criticality of the pre-/post-dominating node equals the maximum of the criticality of their respective successors/predecessors. Furthermore, since* `BB1` *and* `BB5` *pre-dominate and post-dominate each other, their criticality values have to be equal.*

4.1.2 Invariant Code

A very interesting property is related to the notion of *invariant paths*, first proposed by [28]. They showed that certain CFG nodes will always remain on the WCEP even when the weights of other nodes in the graph are changed, e.g., due to a modification of the program that preserves the original structure of the CFG. The corresponding code blocks are thus guaranteed to have a criticality of 1, independent from other code in the program. The original definition [28] is very informal. Using the properties proved in the previous sub-section, a clean (and correct) definition of *invariant code* can be stated as follows:

Definition 9. *A CFG node v of a weighted CFG G is* invariant *when its criticality evaluates to* 1*, independent from the node and edge weights of the CFG.*

Every CFG has at least two *trivial* invariant nodes, the root and the sink node. Furthermore, Corollary 1 shows that every pair of CFG nodes pre-dominating and post-dominating each other have the same criticality. Since the root of the CFG pre-dominates *all* nodes it follows that all CFG nodes post-dominating the root node are invariant.

Corollary 2. *In a weighted CFG G all nodes post-dominating the root node r are invariant.*

Proof. This follows immediately from Corollary 1 and the fact that the root node r pre-dominates all other nodes in the CFG. ■

Note that, in contrast to Lokuciejewski et al., this definition refers to invariant code blocks instead of an invariant path. The set of invariant nodes in a CFG usually does *not* constitute a path for two reasons. Firstly, a CFG node can be invariant even when none of its predecessor and successor nodes are invariant. Secondly, two independent nodes might be invariant depending on the calling context of a function. In addition, our definition is independent from any particular WCEP, opposed to the original definition, and strictly refers to structural CFG properties.

The fact that invariant code solely depends on the structure of the CFG is particularly interesting, even beyond the scope of our work on *criticality*. It shows that certain code parts are known to be on some WCEP (and thus influence the WCET) without performing an actual WCET analysis.

4.2 Algorithms for Computing Criticality

Given the definition from before, we may now ask how to compute the actual criticality for each basic block of a program. We will examine two scenarios: longest path search (1) using dynamic programming on acyclic graphs and (2) using implicit path enumeration (IPET) on general graphs. We require IPET, which is flexible enough to be adapted for computing *criticality* and at the same time powerful enough to handle cyclic control flow. Due to the high computational cost it sometimes incurs, we envision an approach combining both, IPET and dynamic programming, to improve the computation of the criticality metric (see Section 4.6).

4.2.1 Dynamic Programming on Acyclic Graphs

The longest path in a weighted, acyclic CFG can be efficiently computed using dynamic programming [23, Chapter 24] by discovering nodes reachable from the root node using paths of length 1, then nodes reachable by paths of length 2, and so on, until no longer path can be discovered. It is important to note that this approach not only yields the global longest path, but also the longest paths from the root to every node in the CFG. The criticality of all blocks, can then be computed by performing the longest-path search *twice*. The first search considers only paths starting at the root node leading to its sink node. The second pass performs a longest-path search on the *reversed* CFG, i.e., only paths starting at the sink leading to the root are considered. The partial path lengths, from the root to a node and from the node to the sink, can then be combined to determine the longest path passing through the individual CFG nodes.

Algorithm 4 DAGLongestPaths(G): Calculate the weights of the longest paths to every node in an acyclic graph.

```
Input: $G = (V, E, r, t) \ldots$ An acyclic directed graph with weights $\mathcal{W}$.
Output: $d[v]$ contains the weight of the longest path to $v$ for all $v$ in $G$.
▷ Set initial distance values
for all $v \in V$ do
    $d[v] = \mathcal{W}(v)$
for all $v \in V$ in topological order do
    for all outgoing edges $e = (v, w) \in E$ do
        $d[w] = \max(d[w], d[v] + \mathcal{W}(e) + \mathcal{W}(w))$
return $d$
```

Algorithm 4 calculates the distance $d[v]$, i.e., the length of the longest path, from the root to every nodes v in the CFG. The algorithm first initializes $d[v]$ (line 3) and then visits each node in a topological order (l. 4). When a node v is visited, $d[v]$ holds the maximum distance from the root to v. Based on $d[v]$ the maximum distance to all of v's successors is computed using the respective edge weights (line 6).

Algorithm 5 computes all criticality values by first performing a regular longest-path search on the CFG (line 2), followed by a longest path search on the edge-reversed graph G^R (line 4–5). Note that both $d_r[t]$ and $d_t[r]$ contain the length of the longest path from the root to the sink node and thus $d_r[t] = d_t[r] = \text{WCET}$. Combining the results for a node v yields the longest path passing through that node. The criticality $\text{Crit}(v)$ can then be calculated by dividing the result by the overall WCET $d_r[t]$ (l. 7).

This algorithm can also be used to perform the criticality computation on cyclic CFGs using the techniques of [64]. A cyclic CFG here is decomposed into acyclic regions, so-called *scopes*, where the longest path search within each scope is performed using dynamic programming (Algorithm 4). The overall program structure is represented by a scope graph, which interconnects the various scopes and allows to propagate information about the longest path across scopes. Since scopes are acyclic, our approach can immediately be applied to them. However, care has to be taken to also account for the longest path leading to/from a scope from the root node and to the sink node respectively. This can be done during the scope graph traversal. Also note that certain flow facts can be handled by this approach using graph transformations and virtual scopes.

Correctness We start by considering Algorithm 4 and show that it computes the longest path from the root node to every node in an acyclic, weighted CFG.

Lemma 4. *Algorithm 4 calculates the distance $d[v]$, i.e., the length of the longest path, from the root node r for every node v of an acyclic graph G.*

Proof. See [23, Chap. 24] for a detailed discussion. ■

Using this result, we can now show the correctness of Algorithm 5.

Algorithm 5 Compute criticalities for all CFG nodes of an acyclic CFG.

```
Input: $G = (V, E, r, t)$ ... A weighted, acyclic control-flow graph.
Output: The criticality of every node in $G$ is computed.
1: ▷ Compute the longest paths from the root node
2: $d_r$ = DAGLongestPaths($G$)

3: ▷ Compute the longest paths to the sink node $t$
4: $G^R$ = ReverseGraph($G$)
5: $d_t$ = DAGLongestPaths($G^R$)

6: for all $v \in V$ do
7:     $\text{Crit}(v) = (d_r[v] + d_t[v] - \mathcal{W}(v))/d_r[t]$
8:
```

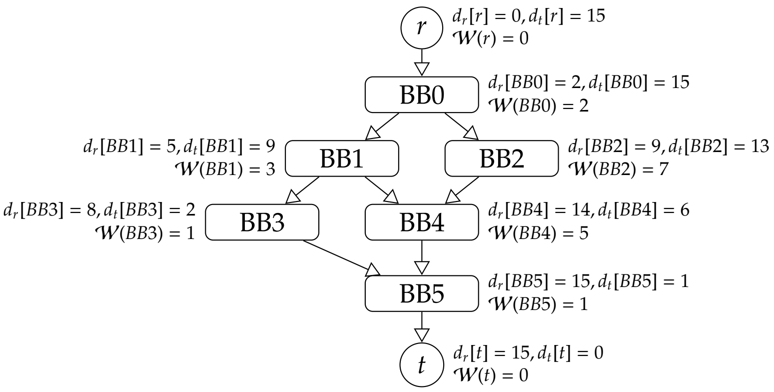


Figure 4.2: Example control-flow graph annotated with longest path weights.

Theorem 3. *Algorithm 5 calculates the criticalities Crit(v) of all nodes v of an acyclic control-flow graph G.*

Proof. Lemma 4 ensures that $d_r[v]$ contains the length of the longest path from the root to some node v and that $d_t[v]$ contains the length of the longest path from v to the sink node. Concatenating these two paths gives the longest path passing through v. The expression $d_r[v] + d_t[v] - \mathcal{W}(v)$ (Algorithm 5 line 7) yields the length WCET(v) of this path. Note that the weight $\mathcal{W}(v)$ appears in both $d_r[v]$ and $d_t[v]$ and thus needs to be subtracted once. Given WCET(v), the node's criticality Crit(v) can be calculated immediately. ■

Complexity The computational complexity of Algorithm 4 performing a longest path search on an acyclic CFG is linear in the size of the graph in $O(|V| + |E|)$ [23, Chap. 24.].

Computing the criticality for each node in Algorithm 5 is linear in the number of CFG nodes (lines 6–8). Constructing the reverse graph (l. 4) can be performed in $O(|V| + |E|)$ by iterating over all outgoing edges of all nodes of graph G. However, depending on the graph representation, this step can be eliminated. The overall complexity of Algorithm 5 is consequently determined by the longest path search and hence in $O(|V| + |E|)$.

Example 9. *We use the control-flow graph from Figure 4.1. Since it is an acyclic graph, we can use Algorithm 5 to calculate the criticalities.*

First we need to run a longest path search on the CFG and on the reversed CFG. Figure 4.2 shows the weights of the CFG nodes and results of the longest path searches. d_r denotes the weight of the longest path from the root to a block, while d_t denotes the weight of the longest path from a block to the sink node t. The WCET for the CFG is given by $d_r(t) = 15$. The weight of the longest path over a node v, WCET(v), is calculated as $d_r(v) + d_t(v) - \mathcal{W}(v)$. $\mathcal{W}(v)$ has to be subtracted once, since it has been added to both $d_r(v)$ and $d_t(v)$ previously.

For the nodes `BB0`, `BB2`, `BB4` *and* `BB5`, *WCET(v) evaluates to 15. Their criticality is therefore* $15/15 = 1$. *The weight of the longest path over* `BB1` *is* $7 + 11 - 5 = 13$, *its criticality is therefore* $11/15 = 0.73$. *For node* `BB3` *WCET(v) is* $8 + 2 - 1 = 9$ *and its criticality is* $9/15 = 0.6$.

4.2.2 Path Enumeration on Cyclic Graphs

For arbitrary graphs, and even for inter-procedural program representations, an implicit enumeration of paths using ILP [78, 74] can be performed to find the WCEP. The computation of the criticality of all basic blocks in a program might require a separate IPET run for each basic block in the worst case. However, we have already seen that the number of blocks that have to be considered can be reduced using dominance properties (see Theorem 1 and 2). We will now present an algorithm that exploits the *basic block dominator graph*, or short dominator graph, initially introduced by [80]:

Definition 10. *Given a CFG G, the dominator graph $\mathcal{D} = (V, D)$ is defined by the CFG nodes in V and dominance edges in $D \subseteq V \times V$ that are derived by unifying the set of edges of the pre- and post-dominator tree of G.*

Note that, in contrast to the pre- and post-dominator tree, the dominator graph usually contains cycles, e.g., the root node of the CFG pre-dominates the sink node while the sink node post-dominates the root, which results in at least one cycle in the dominator graph.

Using the results of the previous section, it is clear that the criticality of non-leaf nodes in the dominator graph can be computed from the criticality of adjacent nodes in the graph. It thus suffices to perform an IPET run for leaf nodes only and to subsequently derive the criticality of all nodes in the graph. Algorithm 6 shows the main steps of the criticality computation given a CFG that is free from critical edges.

First, the dominator graph is constructed and its strongly connected components are computed (`DominatorGraph`, `StronglyConnectedComponents`). Each SCC is then collapsed into a unique node representing the respective SCC (`CollapseSCCs`). All edges entering, respectively leaving, an SCC are redirected to the corresponding representative. Note that the collapsed dominator graph is then free of cycles, i.e., a directed acyclic graph.

Before computing the actual criticalities, the algorithm determines the global WCET of the input program (Algorithm 6 line 9). The function `ComputeWCETOverNode` determines the longest of all paths in a given CFG that pass through a given CFG node (see Definition 7). Implicit path enumeration can easily be adopted to perform this longest path search *over* a specific node. In the simplest case, a single additional constraint has to be added to the ILP, forcing the respective ILP variable to a non-zero value. Note that even though we assume an IPET-based approach, the longest path search can also be performed using any other algorithm.

Following these preparatory steps, the actual criticalities can be computed (lines 11 – 18) by traversing the collapsed dominator graph in post order. When leaving a node during the traversal, the criticality of that node is either (1) computed directly from the longest path passing through that node (l. 14) or (2) derived from the node's successors in the dominator graph (l. 18). The direct computation is required for all leaf nodes of the dominator graph (`isLeaf`), while for other nodes only the adjacent nodes in the graph are considered (`Succ`).

Algorithm 6 Compute criticalities for all CFG nodes by exploiting dominance properties.

Input: $G = (V, E, r, t) \dots$ A weighted control-flow graph.

Output: The criticality of every node in G is computed.

```
 1: ▷ Eliminate critical edges by placing basic blocks between them
 2: SplitCriticalEdges($G$)
 3: ▷ Compute the dominator graph and collapse its SCCs
 4: $\mathcal{D}$ = DominatorGraph($G$)
 5:
 6: $\mathcal{S}$ = StronglyConnectedComponents($\mathcal{D}$)
 7: $\mathcal{D}'$ = CollapseSCCs($\mathcal{D}, \mathcal{S}$)
 8: ▷ Compute the global WCET of the input program.
 9: WCET = ComputeWCETtoverNode($G, r$)
10: ▷ Compute criticalities within the dominator graph.
11: for all $v \in \mathcal{D}'$ in post order do
12:     if isLeaf($\mathcal{D}', v$) then
13:         ▷ Directly compute criticality of the current node.
14:         Crit($v$) = ComputeWCETtoverNode($G, v$) / $WCET$
15:
16:     else
17:         ▷ Derive criticality using dominance.
18:         Crit($v$) = $\mathtt{max}_{u \in \mathtt{Succ}(\mathcal{D}', v)}$Crit($u$)
19: ▷ Propagate criticalities within the SCCs.
20: for all scc $\in \mathcal{S}$ do let$u$ be the node representing the SCC scc in $\mathcal{D}'$
21:     for all $v \in$ scc do
22:         Crit($v$) = $Crit(u)$
23:
```

At this point all possible criticalities are known. What remains is to propagate the criticality among nodes of SCCs using the value of the representative in the collapsed dominator graph (l. 22).

Example 10. *In order to apply the algorithm to the CFG in Figure 4.1, we must first split the critical edge* (`BB1,BB4`) *and introduce a new block* `BB14` *on this edge. Calculating the dominator graph and collapsing the strongly connected components into single nodes results in the dominator graph that is shown in Figure 4.3. For the leaf nodes* `BB3` *and* `BB14` *we need to calculate the longest path over those nodes in the CFG (line 14). The criticalities evaluate to* 0.6 *for* `BB3` *and to* 0.73 *for* `BB14`*. The criticality of node* `BB1` *is derived as the maximum criticality of its children (l. 18), which evaluates to* 0.73*. Continuing the traversal of the dominator graph in post order, we find a criticality of* 1 *for leaf node* `BB2` *by using a longest path search. The criticality of node* `BB4` *can be derived from its children again, no path search is required. Finally, we derive*

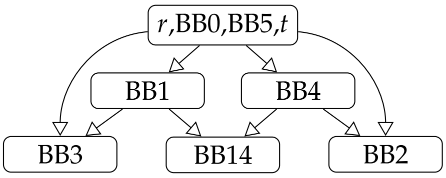


Figure 4.3: Dominator graph for the CFG in Figure 4.1 after splitting critical edges and collapsing SCCs.

a criticality of 1 *for the SCC in the dominator graph from the criticalities of its children, which is then propagated to the CFG nodes* `r`, `BB0`, `BB5` *and* `t` *(l. 22). Note that since the nodes* `BB0`, `BB5` *and* `t` *are in the same SCC as node* `r`, *they are invariant, i.e., their criticality is always* 1 *regardless of the weights in the CFG.*

The basic algorithm described above exploits dominance properties, but does not take advantage of any further pruning opportunities. The simplest technique, which at the same time does not impact the precision of the results, relies on the following corollary: a basic block known to be part of the global WCEP has criticality 1 (by definition). We can easily extend Algorithm 6 to reflect this: the set of blocks belonging to the global WCEP is available from the initial WCET computation (line 9). All blocks contained in this set need no further computation and can be filtered at the computation stage (l. 11 – 18).

Correctness The correctness of this approach follows immediately from the theoretical properties of the criticality metric discussed in the previous section.

Theorem 4. *Given a weighted CFG G that is free from critical edges, Algorithm 6 correctly computes the criticality of all CFG nodes according to Definition 8.*

Proof (sketch). If a node is a leaf in the collapsed dominator graph $\mathcal{D}'$, its criticality is computed directly. The algorithm is thus trivially correct for all leaves in $\mathcal{D}'$. The algorithm visits all other nodes of that graph during a post-order traversal. This ensures that the criticalities of all successors of a node have been computed before it is visited. The correctness of the propagation then follows immediately from Theorem 1 and Theorem 2. Finally, a node in $\mathcal{D}'$ directly corresponds to a CFG node (or a set of CFG nodes forming an SCC in $\mathcal{D}$) with equal criticality. It follows that the criticality of all CFG nodes is computed correctly. ■

Complexity Ignoring `ComputeWCETоverNode` for now, the complexity of the algorithm is mostly influenced by the computation of the dominator graph (`DominatorGraph`). A standard dominator tree can be constructed in linear time in the number of nodes and edges of the CFG [55]. A simpler algorithm is used in most modern compilation and analysis tools [90]. The complexity of that algorithm is in $O(|E| \log |V|)$, where $|E|$ and $|V|$ represent the number of CFG edges and nodes respectively. Since the dominator graph is derived by merging the pre- and post-dominator tree, the complexity remains in $O(|E| \log |V|)$.

SCCs (`StronglyConnectedComponents`) can be computed in linear time $O(|V|+|D|)$, where $|D|$ represents the number of edges in the dominator graph(see [92]). Collapsing the SCCs into unique nodes is likewise linear. Note that the number of edges in the dominator graph is bounded by the number of edges in E.

The algorithm then traverses the dominator graph in post order, visiting each node and edge of the graph exactly once. The first for-loop thus executes in $O(|V| + |D|)$ time. The second for-loop finally visits every SCC in the graph and every CFG node once – it thus executes in $O(|V|)$ time.

Using Lengauer and Tarjan's dominator algorithm [90], the overall complexity of the algorithm is thus in $O(|E| \log |V|)$. The bound is not affected when including the critical edges splitting, which is linear in the CFG size.

The complexity of the IPET approach (`ComputeWCETYoverNode`), is NP-hard in general. Its feasibility depends on the size of the underlying ILP problem, i.e., the size and complexity of the program under analysis, and the efficiency of the ILP solver. Section 4.5 gives insights on the overhead that can be expected from repeatedly solving IPET problems for a program using a commercial WCET tool.

4.2.3 Handling Critical Edges

The previously presented algorithm requires critical edges to be split as a pre-processing step, which introduces new basic blocks to the CFG. This can have an adverse effect since adding nodes to a large graph may impact the solver's performance. We may also end up computing criticality values when it is effectively redundant to do so. We thus describe the changes to Algorithm 6, which are necessary to compute criticalities correctly, even in the presence of critical edges.

Since the criticality properties of a node with regard to dominance, are invalidated by an incident critical edge, the `isLeaf` check (Algorithm 6, line 12) alone is not sufficient. Assuming that nodes incident to a critical edge are marked beforehand (i.e. marked SCCs in the collapsed graph that contain such nodes), we additionally compute the criticality of the current node v if `isMarked`(v) is true. This alone is sufficient to maintain the correctness of Algorithm 6, but can be improved further. Algorithm 7 shows how to mark nodes for calculation, only when it is not possible to find at least one other node in the same SCC that can be used to derive the SCC's criticality. First, only SCCs that contain a node with an incident critical edge (`HasCriticalEdge`) need to be considered further (l. 4). For all other SCCs, the algorithm tries to find a node, which either pre-dominates all its CFG-successors outside the current SCC (l. 8) or likewise post-dominates all its CFG-predecessors (l. 10). If such a node cannot be found, the SCC is subsequently marked (l. 13).

Example 11. *Considering our example from Figure 4.1, the dominator graph after collapsing SCCs (but with critical edge* (`BB1,BB4`) *intact) is similar to Figure 4.3, with node* `BB14` *and its incoming edges removed. Instead of the artificial node* `BB14`*, it is now necessary to explicitly compute the WCET over the non-leaf node* `BB1`*, since it is not on the global WCEP and in this simple example cannot be derived from other nodes in its SCC. Efficiently handling critical edges*

Algorithm 7 Mark nodes that require criticality calculation due to a critical edge

```
Input: $G = (V, E, r, t) \dots$ A weighted control-flow graph.
       $\mathcal{S} \dots$ Strongly connected components of the dominator graph.
       $\mathcal{D}' \dots$ Dominator Graph with collapsed SCCs.
Output: Nodes in $\mathcal{D}'$ requiring calculation are marked True.
▷ Iterate over all SCCs.
for all $v' \in \mathcal{D}'$ do
    letscc = CorrespondingNodes($S, v'$)
    if $\forall v \in$ scc : HasCriticalEdge($v$) = False then
        Mark($v'$) = False
        continue
    ▷ scc is affected by a critical edge but may be derived from successors or predecessors.
    if $\exists u \in$ scc : $\forall w \in$ (Succ($G, u$) \ scc) : $u$ dom $w$ then
        Mark($v'$) = False
    else if $\exists u \in$ scc : $\forall w \in$ (Pred($G, u$) \ scc) : $u$ pdom $w$ then
        Mark($v'$) = False
    else
        Mark($v'$) = True
```

during criticality computation (as described in Algorithm 7) is more complicated, but advantageous when a large number of critical edges connect few nodes in the CFG (e.g., due to the use of switch statements with fallthrough cases).

4.2.4 Pruned Criticality Computation

The algorithms in the previous section are very simple to implement and efficient when the criticality value for all code blocks needs to be computed. However, in some situations it might be sufficient to restrict the computation to only those code blocks whose criticality is above a given threshold. For example, when an optimization wants to shift instructions from highly critical code, it would be enough to know about neighboring regions with criticality below a certain threshold (inferred, for instance, by the compiler). The individual criticality values within these regions itself would be uninteresting and need not be computed. As for Algorithm 6, pre- and post-dominance can be exploited to reduce the runtime overhead of computing pruned criticalities. This time, however, the dominator graph is processed starting from the root node down towards the leaves of the graph. The algorithm iteratively computes the longest path over pre- or post-dominators until it finds a dominator node whose criticality is below the given threshold (cf. Lemma 2 and 3).

Algorithm 8 presents a simplified variant of a worklist algorithm taking a weighted CFG G and a criticality threshold MinCrit as input. As before, an acyclic dominator

Algorithm 8 Compute all CFG nodes with a criticality above a given threshold.

Input: $G = (V, E, r, t) \dots$ A weighted control-flow graph.
MinCrit … Threshold for criticality computation.

Output: The criticality is computed for nodes in G whose criticality is above the threshold; for other nodes the criticality is bounded.

```
▷ Compute the dominator graph and collapse its SCCs.
$\mathcal{D}$ = DominatorGraph($G$)

$\mathcal{S}$ = StronglyConnectedComponents($\mathcal{D}$)
$\mathcal{D}'$ = CollapseSCCs($\mathcal{D}, \mathcal{S}$)
▷ Unmark all nodes in the dominator graph.
for all $v \in \mathcal{D}'$ do
    Mark($v$) = False
▷ Initialize global WCET.
WCET = 0
▷ Discover nodes that have not been processed so far.
while $\exists v \in \mathcal{D}'$: Mark($v$) = False do
    ▷ Get some WCEP over any of the root nodes in the dominator graph.
    letRoots = {$v \in \mathcal{D}'$ | Mark($v$) = False $\wedge$ $\forall p \in$ Pred($\mathcal{D}', v$): Mark($p$) = True}
    P, WCET$_P$ = ComputeWCETOverAny($G$, Roots)
    ▷ Get global WCET (only relevant for first iteration).
    WCET = max(WCET$_P$, WCET)
    ▷ Compute criticality of the current path $P$.
    letCrit$_P$ = WCET$_P$ / $WCET$
    if Crit$_P$ < MinCrit then
        ▷ Bound criticality for all unmarked nodes in the dominator graph.
        for all $v \in \mathcal{D}'$: Mark($v$) = False do
            Crit($v$) = Crit$_P$
        break
    else
        ▷ Mark all nodes of the current path and set their criticality.
        for all $v \in P$: Mark($v$) = False do
            Crit($v$) = Crit$_P$
            Mark($v$) = True
```

graph is constructed from the CFG where all strongly connected components are collapsed (line 2). The algorithm then processes the nodes of that graph in some topological order, depending on the marking of nodes. Nodes for which a criticality has been calculated are marked (see `Mark` on l. 7 and l. 28). When all predecessors of a node have been marked, the node is added to the worklist. Note that the worklist in Algorithm 8 is not explicit, but maintained by this gradual marking in the form of the `Roots` set (l. 13). During each iteration, a new WCEP together with its associated WCET is discovered by using `ComputeWCETtoverAny` (l. 14). `ComputeWCETtoverAny` is similar to `ComputeWCETtoverNode`, except that it determines the longest path through *any* of the nodes in its argument set by forcing the sum of the respective ILP variable to a non-zero value. Thus, the newly discovered WCEP is known to be one of the longest paths covering at least one unmarked node. The length of this WCEP gives the criticality of all unmarked nodes on that path (l. 18) and, furthermore, allows to prune the criticality computation (l. 19). In case the current criticality is below the given threshold MinCrit, the algorithm bounds the criticality value of all remaining unmarked nodes with the criticality value of the current WCEP. The iterative processing can then be stopped immediately (l. 23). When the current criticality is above the threshold, the criticality value is assigned to all unmarked nodes on the current WCEP. At the same time, all nodes on the WCEP are marked and a new iteration starts.

The algorithm is based on the following key observations. Due to Lemma 2 and 3 it is ensured that at any moment during the processing (1) marked nodes have a criticality greater or equal to any node in the worklist and (2) nodes in the worklist have a criticality greater or equal to all other nodes, i.e., nodes that are neither marked nor in the worklist. Consequently, the newly discovered WCEP is the longest yet undiscovered path that includes at least one unmarked node from the worklist. Furthermore, every node on the WCEP is either pre- or post-dominated by a node in the worklist.

Note that, in contrast to Algorithm 6, critical edges are not an issue here and need no further attention. The reason for this is that the algorithm relies on the weaker Lemma 2 and 3, which only ensure that a node has a greater or equal criticality in relation to any node it is pre- or post-dominating.

Also note that by restricting which nodes are put on the worklist, the scope of Algorithm 8 can be changed from program- to function-level, without any further modifications. This is useful if we are only interested in the criticality of CFG nodes belonging to a certain function at a time.

Example 12. *Assuming the dominator graph shown in Figure 4.3, which was derived by splitting the critical edge* (`BB1,BB4`) *in the CFG in Figure 4.1, the first iteration of the algorithm processes the SCC node, representing r,* `BB0`*,* `BB5`*, and t. This forces the first invocation of* `ComputeWCETtoverAny` *to derive the longest path through the CFG. This path has a length of* 15 *and covers the root node r,* `BB0`*,* `BB2`*,* `BB4`*,* `BB5`*, and the sink node t. Since all nodes were initially unmarked and the current criticality value trivially evaluates to* 1*, the algorithm marks all the nodes of the path and assigns* 1 *as the criticality to each of them. In the second iteration the only node without unmarked predecessors is* `BB1`*, the second invocation of* `ComputeWCETtoverAny` *thus delivers the longest path passing through that block. The resulting path has length* 11 *and* *thus yields a criticality of* 0.73*. It covers the root node* r*,* `BB0`*,* `BB1`*,* `BB14`*,* `BB4`*,* `BB5`*, and the sink node* t*. All nodes, except* `BB1` *are already marked and thus are known to have a criticality higher than* 0.73*. On the other hand, all nodes not marked so far are known to have a criticality lower than this value. The algorithm could, for instance, now stop the iterative processing of the worklist, e.g., if the specified criticality threshold is assumed to be* 0.9*. The algorithm would then bound the criticality of the unmarked blocks* `BB1` *and* `BB3` *by the current criticality value.*

Correctness The correctness of this approach follows immediately from the theoretical properties of the criticality metric discussed in Section 4.1.1.

Lemma 5. *Given a weighted CFG G and a criticality threshold MinCrit, Algorithm 8 correctly computes the criticality of all CFG nodes whose criticality is greater or equal to the threshold.*

Proof (sketch). Assume a CFG node v has criticality c that is greater than or equal to MinCrit, but Algorithm 8 delivered some criticality $w \neq c$. We then have to distinguish two cases: the criticality value w is assigned to v (1) on line 22 or (2) on line 27. In both cases, a pre- or post-dominator d of v has to be in the worklist when the criticality is assigned to v.
For case (1), the invocation of `ComputeWCETоverAny` returns a path with criticality w that is below the threshold. Independent of whether this path contains d or not, it follows that the longest path passing through d yields a criticality smaller or equal to w and thus smaller than MinCrit. From Lemma 2 and 3 it follows that $c <$ MinCrit. This is impossible according to the initial assumption.
For case (2), the invocation of `ComputeWCETоverAny` returns a path that covers both v and d and yields criticality $w \geq$ MinCrit for both nodes. Obviously, the actual criticality c of v cannot be smaller than w. However, c also cannot be greater than w, since w is derived from the longest path passing through d. Lemma 2 and 3 thus imply $w = c$. This contradicts the initial assumption. The algorithm is thus correct for all nodes whose criticality is above the given threshold. ■

Lemma 6. *Given a weighted CFG G and a criticality threshold MinCrit, Algorithm 8 yields a correct upper bound for the criticality of all CFG nodes whose criticality is below the threshold.*

Proof (sketch). Assume a CFG node v has criticality c that is smaller than MinCrit but greater than the bound b assigned to v by Algorithm 8 (l. 22). Bound b has to be associated with a path p that is computed on the last iteration of the algorithm. Furthermore, some pre- or post-dominator d of v has to be in the worklist before p is computed. Independent of whether p covers d or not, the longest path passing through d is at most as long as p and thus cannot yield a criticality greater than b. Lemma 2 and 3 then imply $c \leq b$. This contradicts our initial assumption. The algorithm thus gives correct bounds for those CFG nodes whose criticality is below the given threshold. ■

Note, setting the threshold MinCrit to zero implies that a criticality value is computed for all CFG nodes.

Theorem 5. *Given a weighted CFG G and a criticality threshold MinCrit, Algorithm 8 is correct.*

Proof. Since the algorithm discovers a new path covering at least one unmarked node in the dominator graph on every iteration, termination is guaranteed. The correctness of the computed criticality values follows from Lemma 5 and 6. The algorithm is thus correct. ■

Complexity As before, the runtime of the pruned criticality computation can be expected to be dominated by the longest path search performed by `ComputeWCETboverAny`. We thus compare the number of longest path searches performed by the full criticality computation from the previous section in relation to the pruned criticality computation when the threshold is set to zero. We already noted that Algorithm 8 computes a new path on every iteration that covers at least one unmarked node v from the worklist. In addition, it is guaranteed that either v is a leaf in the dominator graph or, due to dominance, at least one unmarked leaf is also covered by the path. It follows that the pruned criticality computation performs at most as many longest path searches as the full criticality computation. However, the pruned criticality algorithm presented in this section is harder to parallelize, since the content of the worklist for the next iteration depends on the outcome of the current longest path search. In addition to this, the pruned approach adds larger and more complex flow constraints to the ILP problem, which is likely to increase the ILP solving times.

The pruned criticality computation again requires the computation of the dominator graph (`DominatorGraph`) and its SCCs (`StronglyConnectedComponents`). Using the dominator algorithm of [90], the complexity for these pre-computations is bounded by $O(|E| \log |V|)$ (see Section 4.2.2 for a more detailed discussion).

The number of iterations required to process the worklist can be bounded by the number of nodes in the dominator graph which is known to be at most $|V|$. On each iteration a longest path search is performed followed by the criticality assignment for all nodes on the newly discovered path (Algorithm 8 line 26) and an update of the worklist (l. 13). The loop for the criticality assignment can be bounded in $O(|V|)$, since the path may cover (almost) all nodes in the CFG on each iteration. To update the worklist all edges of the dominator graph have to be visited, in the worst case resulting in a bound of $O(|D|)$. Since the dominator graph is constructed by merging the pre- and postdominator tree, $|D|$ is in the order of $O(|V|)$. The complexity of the iterative processing is thus in $O(|V|^2)$.

Combining the complexity results for the construction of the dominator graph and the iterative processing gives an overall bound in $O(|V|^2 \log |V|)$, since $|E| \leq |V|^2$. In conclusion, pruned criticality computation is, in practice, dominated by the longest path search that is part of the iterative processing.

4.3 Estimating Criticality

In the previous sections, we have shown how to make criticality computation more efficient by reducing the number of expensive computations, i.e. reducing the number of ILPs that need to be solved during iterative longest path search. Nonetheless, the computational overhead remains closely related to that of IPET-based WCET analysis for a specific program, which again is proportional to the level of analysis precision. In some situations, for example when we use *criticality* purely as a profiling method, we might be willing to trade precision, for analysis speed, effectively admitting a criticality *estimation*.

The overhead can be attributed for one part to data-flow analysis and the analysis of cache and pipeline states, which is performed once when criticality computation starts. The required effort largely depends on the size and complexity of a program.

The larger part of the computational overhead comes from repeatedly solving the IPET ILP. For this, the size of the IPET problem (i.e., the number of variables in the ILP, more probable than the number of constraints) is the driving factor. In order to avoid conservative estimations, regions of the program's CFG may need to be considered within different contexts (cf. Section 2.1.2). The latter are introduced to the program representation of the analysis as loop or inter-procedural contexts. The IPET size is directly related to program size and the number of these contexts, as well as the architectural states that the machine model adds.

When an estimation of criticality values is sufficient, we can adapt its computation as follows:

1. During the initial WCET analysis: Ignore all cache effects (i.e. every access to a data- or instruction cache is immediate) and all pipeline effects between basic blocks. In other words, consider only instruction interdependencies within a basic block.

2. Reduce the number of inter-procedural contexts and the number of loop contexts to *small* values.

3. Perform criticality estimation based on the approximate analysis unchanged.

It is clear that with these analysis settings in place, we will either drastically *underestimate* (under optimistic assumptions), or just as drastically *overestimate* (under conservative assumptions) a program's WCET. But as long as this is done uniformly over all worst-case execution paths considered during criticality analysis, there will be little impact on the criticality results compared to precise (and expensive) analysis. The restrictions of item 1 reduce the overhead of data-flow analysis and, depending how much of the information on architectural states is carried over into the IPET problem, also the ILP solving time. Item 2 has an impact on both phases, i.e., if those contexts would have been instantiated for the program under analysis, less data-flow analysis is performed and the ILP size is reduced proportionally to the reduction of contexts.

This approach to estimation is not expected to be useful for all programs that can be analyzed. For simple programs, the analysis overhead for criticality computation is negligible, i.e. it is dominated by the constant effort always incurred by the analysis tool. In such a case, lower analysis precision can only introduce error, while not improving analysis runtime at all. We thus would like to apply estimation only to sufficiently complex analysis problems. Two methods for activating estimation appear feasible: (1) We can fall back onto estimation during criticality computation dynamically, by using a runtime threshold. (2) We can base the decision on a static property of the program and its analysis model, which we have at hand after the initial (data-flow) analysis. For now, we employ the straight-forward second approach, basing our decision on the number of basic blocks in the program and the number of ILP variables in the IPET model.

Correctness All proofs of correctness from Section 4.2 remain valid when using estimation. Since criticality computation itself does not make any assumptions about analysis precision, only the resulting criticality values need to be interpreted with regard to the less precise underlying analysis.

4.4 Visualization

By visualizing profiling information, in our case for worst-case behavior and based on static analysis, we want to enhance program understanding for real-time applications. A detailed view of a program's structure can be given through the control-flow graph on an intra- or inter-procedural level. Compilers and profiling tools make use of this graphical representation for debugging and analysis purposes. For example, LLVM is able to export its intermediate representation at any time in the DOT format; AbsInt's `a3` tool displays and exports analysis information such as results from WCET analysis using the VCG format. (See Section 4.7 for more details.)

Given a mapping of basic blocks to their criticality, we can rank nodes of the CFG by size (i.e., translating criticality to node size) and color (i.e., translating criticality to color shades). This provides visual clues to the programmer, which are superior to a textual representation of the same results. Since we want to rely on open-source tools for visualization in our approach, one of the important tasks of graph visualization, namely the layout of nodes and edges is left to the `dot` layout algorithm and tool from the Graphviz [61] package.

The criticality-annotated CFG representation is ideal to study a real-time program in detail. From this, the programmer may draw conclusions about a program's WCET behavior and possible weaknesses in the static analysis. For larger programs though, it is hard to comprehend its overall structure (e.g., the amount of critical versus uncritical code) at a glance. This would require interactive exploration of the graph using a displaying tool, which at least supports zooming into regions. (Making regions collapsible as supported by AbsInt's VCG viewer aiSee, has also proven useful in this regard.)

In previous work [5], we have divided the $[0, 1]$-range of criticality values into intervals and presented the number of basic blocks within each interval as a column in a table. Since in this way, the number of intervals we can present is low, and the readability of such a table is poor, we resort to a miniature bar chart. These *sparkbars*, after Tufte's sparkline [47], allow us to compactly display criticality distributions. We can place this kind of graphic within a line of text, for example:  represents `debie-4a`. Due to the highly unbalanced distribution of criticality values, the y-axis of a sparkbar graph has logarithmic scale. In Section 4.5.3, we embed these distribution charts in Table 4.8.

4.5 Evaluation

The criticality metric is intended as a supplemental aid for real-time software developers and development tools (compilers) to spot timing-critical code in applications and to guide the tuning of this code under timing constraints. A proper evaluation of the metric and its potential advantages for this purpose would require a large-scale qualitative survey, which is beyond the scope of this thesis. Instead, we focus on evaluating whether the metric can be employed in practice. We designed a series of experiments for this purpose. In Section 4.6, we additionally discuss how the metric could be presented to application programmers, how this information could be exploited to *profile* the worst-case behavior of real-time programs, and where the limitations of the metric are.

For our experiments, we rely on a set of well-known real-time benchmarks, which have been described in Section 2.2.2. The benchmark sizes range between about 30 lines-of-code (LOC) and 4000 for the Mälardalen benchmarks and amount to about 11000 and 14000 LOC for the PapaBench and Debie programs respectively.

The first experiment investigates the code structure of these real-time programs using the LLVM compiler infrastructure (version 3.0). All benchmark programs were compiled to the LLVM intermediate representation (LLVM IR) using the Clang C compiler with optimizations (`-O1`). We then analyzed the structure of the control-flow graphs for each compiled C function. In particular, we studied the structure of the respective dominator graphs. Since no actual WCET analysis is performed, no additional program annotations, such as loop bounds, are required for this experiment.

We furthermore measure the overhead of computing the criticality metric for the Debie WCET benchmark and give a detailed analysis. This experiment is based on a binary version of Debie (part of the official program distribution). The WCET analysis is performed using the `a3` WCET analysis tool (version 12.10i) assuming the default configuration for a PowerPC MPC5554 processor (cf. Section 2.2.1). The respective linear problems are solved using `clpsolve` (a solver based on CLP from the COIN-OR project [99]) on an Intel i5-2520M (2.5GHz) running Linux (kernel version 3.11). Finally we extend the evaluation across the full set of our WCET benchmarks (cf. Section 2.2.2) to learn about their criticality distribution and report on the results when using the estimation approach from Section 4.3.

4.5.1 Code Structure of Real-Time Programs

Since the complexity of the IPET phase easily dominates the runtime of the otherwise fast criticality calculation, we are interested in the number of leaves in the dominator graph for which an actual IPET run (`ComputeWCETоverNode`) is required. We thus compiled all benchmark programs using the Clang C compiler and constructed the dominator graph for each function. When the number of leaf nodes is low, in comparison to the overall number of nodes in the graph, it means that very few IPET runs are required and most criticality values can be derived. Figure 4.4 shows the number of leaves in the dominator graphs for individual functions of the various benchmarks. The general

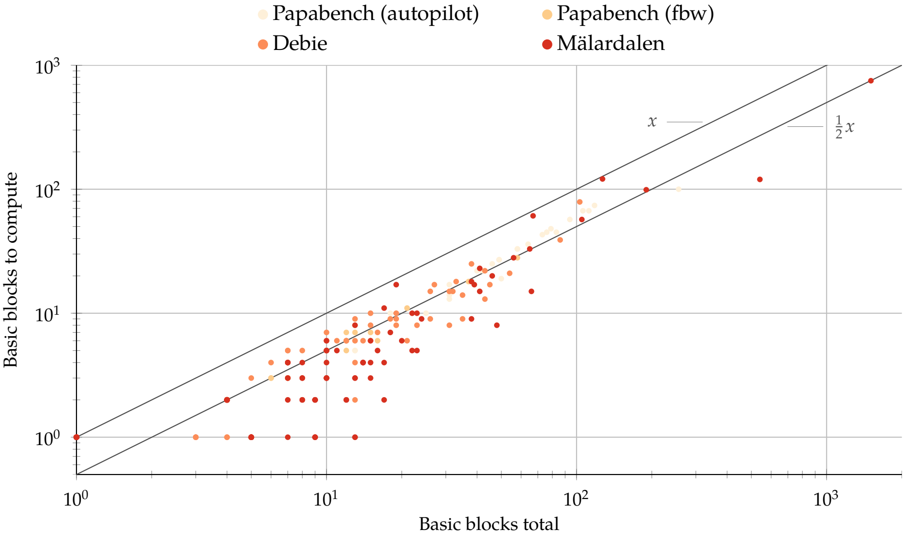


Figure 4.4: Relation between the number of basic blocks in a function and the number of leaves in the corresponding dominator graph. (lower is better)

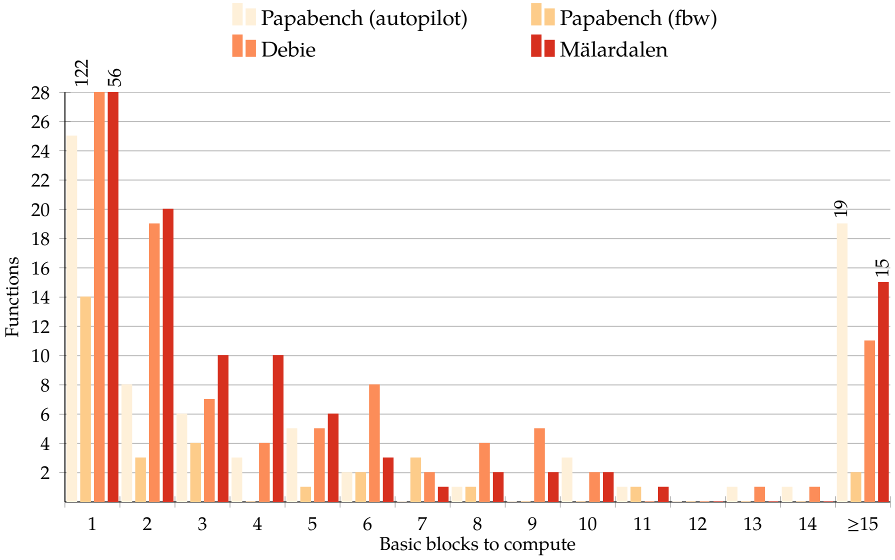


Figure 4.5: Bar chart showing the number of functions (y-axis) with n leaves in the dominator graph (x-axis). (high bars to the left are better)

Module	BBs	SCCs	DomLeaves		Module	BBs	SCCs	DomLeaves	
adpcm	143	31	56	(0.39)	fir	16	4	5	(0.31)
bs	11	3	4	(0.36)	insertsort	7	2	1	(0.14)
bsort100	24	6	9	(0.38)	jfdctint	14	5	3	(0.21)
cnt	25	6	7	(0.28)	lcdnum	27	3	19	(0.70)
complex	17	3	6	(0.35)	lms	59	15	18	(0.31)
compress	118	20	51	(0.43)	loop3	541	121	120	(0.22)
cover	212	6	194	(0.92)	ludcmp	64	17	12	(0.19)
crc	33	7	12	(0.36)	matmult	26	7	6	(0.23)
duff	28	4	12	(0.43)	minmax	28	6	14	(0.50)
edn	57	20	9	(0.16)	minver	110	28	24	(0.22)
exam	42	4	16	(0.38)	ns	21	8	7	(0.33)
expint	24	7	7	(0.29)	nsichneu	1504	128	750	(0.50)
fac	9	3	3	(0.33)	qurt	27	6	11	(0.41)
fdct	10	3	3	(0.30)	select	47	7	21	(0.45)
fft1	59	14	17	(0.29)	sqrt	19	5	7	(0.37)
fibcall	9	2	4	(0.44)	statemate	482	28	252	(0.52)
					Average	119.16	16.53	52.2	(0.44)

Table 4.1: Dominator graph statistics for the Mälardalen benchmarks

Module	BBs	SCCs	DomLeaves		Module	BBs	SCCs	DomLeaves	
ad7714	26	6	13	(0.50)	mainloop	23	1	13	(0.57)
adc	16	4	7	(0.44)	modem	20	5	9	(0.45)
calib	31	7	14	(0.45)	nav	285	31	117	(0.41)
estimator	44	6	20	(0.45)	pid	70	10	36	(0.51)
fbw	19	5	9	(0.47)	spi	5	1	3	(0.60)
infrared	3	0	3	(1.00)	uart	27	5	14	(0.52)
main	1224	94	707	(0.58)	ubx	65	8	26	(0.40)
					Average	132.71	13.07	70.79	(0.53)

Table 4.2: Dominator graph statistics for PapaBench (autopilot)

Module	BBs	SCCs	DomLeaves		Module	BBs	SCCs	DomLeaves	
fbw	19	4	9	(0.47)	servo	58	14	25	(0.43)
main	94	12	50	(0.53)	spi	14	3	7	(0.50)
ppm	74	14	34	(0.46)	uart	18	3	10	(0.56)
					Average	46.17	8.33	22.5	(0.49)

Table 4.3: Dominator graph statistics for PapaBench (fbw)

trend indicates that only about half of the nodes in the dominator graph are leaves, as can be seen by the clustering of data points around the lower line. The only exceptions are three functions of the Mälardalen benchmarks (`swi120`, `swi50`, and `swi10`), which consist of large switch statements and are highly synthetic. This indicates that a considerable reduction in the computation time can be expected when using our algorithm in comparison to a naive approach.

Furthermore, a large number of functions in this scatter plot is clustered at the lower left, representing functions with only a single basic block. Figure 4.5 presents a bar chart showing the number of functions with *n* leaves in the dominator graph. The vast majority of the functions has a single dominator leaf, whereas the number of functions with more leaves quickly declines. This again indicates that the vast majority of the code in these real-time programs has a rather simple structure and thus can be expected to require very few IPET runs in order to compute the criticality of all code.

Tables 4.1 to 4.4 contain statistics from the various benchmarks, broken down per compilation module. The tables show the number of basic blocks (`BBs`), the number of SCCs (`SCCs`) as well as the number of leaves in the dominator graph (`DomLeaves`) and in the last column, the ratio between this number and the total number of blocks. Over all benchmarks, we found the average ratio of dominator graph leaves to be approximately 0.5. This means that the criticality actually needs to be computed at most for half of the total blocks.

However, the share of leaf nodes per function, as reported here, is only an *upper* bound. In practice, additional potential to prune the set of basic blocks for which an actual IPET run is required exists, e.g., by using inter-procedural dominance information to allow additional pruning. This appears particularly interesting due to the high number of functions having only a single leaf in the dominator graph.

Module	BBs	SCCs	DomLeaves		Module	BBs	SCCs	DomLeaves	
class	53	5	31	(0.58)	if	118	21	69	(0.58)
debie	3	0	1	(0.33)	measure	110	20	49	(0.45)
hand	320	47	179	(0.56)	target	4	0	4	(1.00)
harness	382	68	176	(0.46)	telem	76	17	31	(0.41)
health	298	49	151	(0.51)					
					Average	151.56	25.22	76.78	(0.51)

Table 4.4: Dominator graph statistics for the Debie benchmark

4.5.2 Criticality Computation for the Debie Benchmark

With the second experiment we do an in-depth investigation of the actual cost associated with the computation of the criticality metric in WCET analysis setup. We apply the `a3` WCET analysis tool to all analysis problems defined for the application-level Debie benchmark. Thanks to the scripting capabilities of `a3` and the annotation language AIS we were able to force the worst-case path through specific basic blocks of the program, i.e., to implement the function `ComputeWCETOverNode`.

Since, for this experiment, the actual WCEP is known, we can give more accurate numbers on how many basic blocks need to be considered during the criticality computation (see Figure 4.6). The analysis yields that between 14% and 93% of the basic blocks actually are on the WCEP (61% when considering the geometric mean). Combined with the information on leaf nodes in the dominator graph, the percentage of basic blocks for which an actual IPET run is required during the criticality computation is between 43% and only 10% (19% mean). For `debie2a` even all blocks are on the WCEP, thus no computation is required. In fact, a large number of basic blocks throughout all benchmarks does not need any additional computation. Therefore the total time to compute the criticalities for all basic blocks is greatly reduced using our algorithm, as shown by Figure 4.7. In comparison to the *naive* approach, where the criticality is separately computed for every basic block that is not on the WCEP, our algorithm reduces the computation time down to 67% (mean). For `debie2a` no IPET run is required, the computation is thus almost free. The naive approach requires up to 413 seconds (`debie6c`) but only up to 156 seconds (`debie6c`) for our algorithm. The average computation time is reduced from about 42 to 20 seconds.

Algorithm 8 can be used to compute criticality with a threshold value, blocks which fall below it are considered to be uncritical. Table 4.5 shows that this *pruning* method results in less ILPs that need to be solved, and thus shortens the total computation time. Depending on the benchmark, a MinCrit threshold of 0.25 can bring the number of ILPs down to a seventh compared to criticality calculation of all blocks, which reduces computation time to 20% of the original.

Figure 4.9 illustrates the progress of the pruning algorithm as it processes basic blocks in its worklist and successively finds paths with lower criticalities. In this example, enforcing a threshold of 0.25 would result in cutting off the *tail* of low-criticality blocks (on the right) after ten iterations. The particular progression of iterations of the pruning algorithm is obvious, when we compare it to the default (dominator-based) algorithm in Figure 4.8.

Figure 4.10 depicts an example visualization of `debie-1`. With the information provided by a traditional WCET analysis, we can see the WCEP through the program's main function and its single-block leaf procedures, which are also leaves in the interprocedural CFG of Figure 4.10a. After criticality computation, we can annotate the previously unidentified nodes, based on their criticality. Colors and node sizes in Figure 4.10b intuitively represent the criticality values of basic blocks.

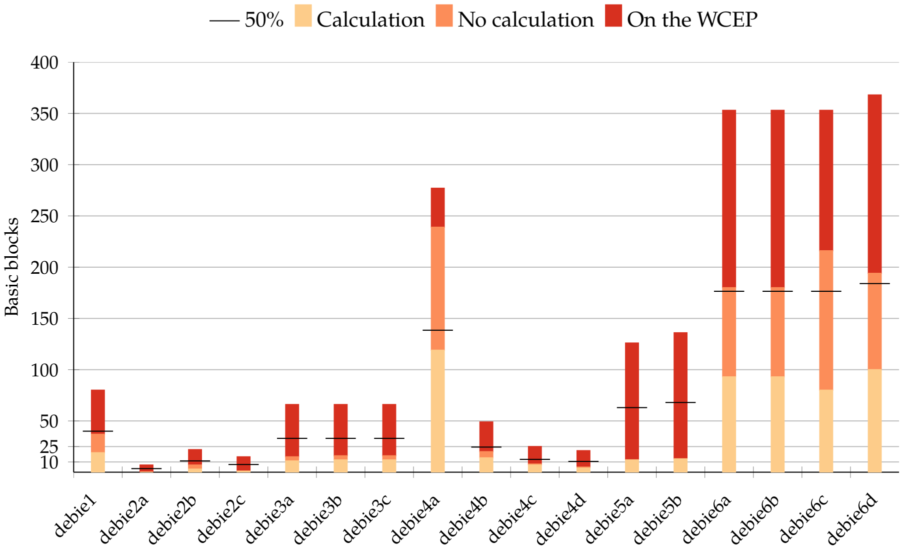


Figure 4.6: Number of basic blocks that (1) require calculation, (2) do not require calculation (leaves of the dominator graph), (3) are on the WCEP (lower value of (1) is better)

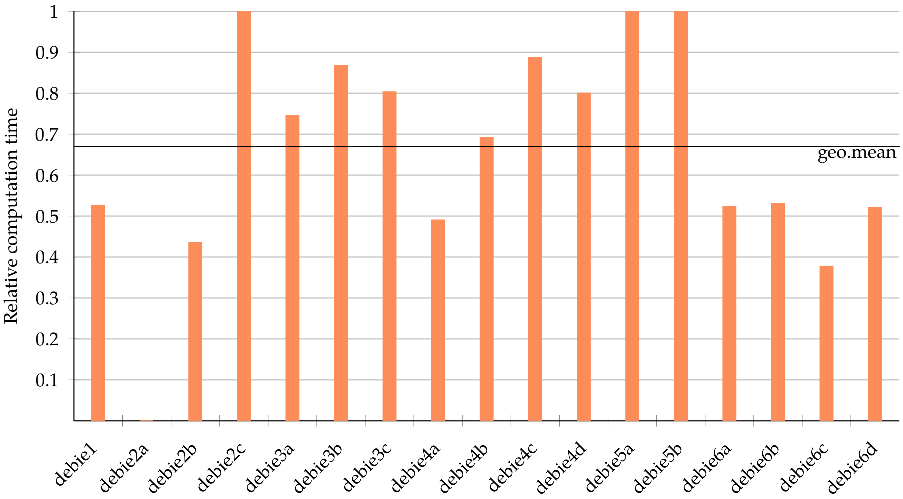


Figure 4.7: Relative reduction of the time required for the computation of criticalities. (lower is better)

	MinCrit = 0			**MinCrit** = 0.25			**MinCrit** = 0.5		
Benchmark	ILPs	BBs	t_0	ILPs	BBs	$t_{0.25}/t_0$	ILPs	BBs	$t_{0.5}/t_0$
debie1	15	80	1.00	14	79	0.97	13	77	0.91
debie2a	1	7	1.00	1	7	1.00	1	7	1.00
debie2b	4	22	1.00	4	22	1.00	4	22	1.00
debie2c	2	15	1.00	2	15	1.00	2	15	1.00
debie3a	7	65	1.00	6	58	0.85	6	58	0.85
debie3b	9	66	1.00	8	59	0.94	8	59	0.94
debie3c	9	66	1.00	8	59	0.93	8	59	0.93
debie4a	71	274	1.00	10	100	0.19	6	68	0.12
debie4b	12	49	1.00	10	46	0.84	7	43	0.57
debie4c	7	25	1.00	7	25	1.00	6	24	0.87
debie4d	4	21	1.00	4	21	1.00	3	20	0.88
debie5a	11	126	1.00	10	125	0.91	10	125	0.91
debie5b	12	136	1.00	9	133	0.76	9	133	0.76
debie6a	30	323	1.00	30	323	1.00	30	323	1.00
debie6b	30	324	1.00	30	324	1.00	30	324	1.00
debie6c	30	324	1.00	30	324	1.00	23	302	0.87
debie6d	32	364	1.00	32	364	1.00	32	364	1.00

Table 4.5: Impact of pruning thresholds (0.25 and 0.5) on criticality computation. (number of ILPs required, basic blocks (BBs) above the threshold and time ratio compared to computation without threshold)

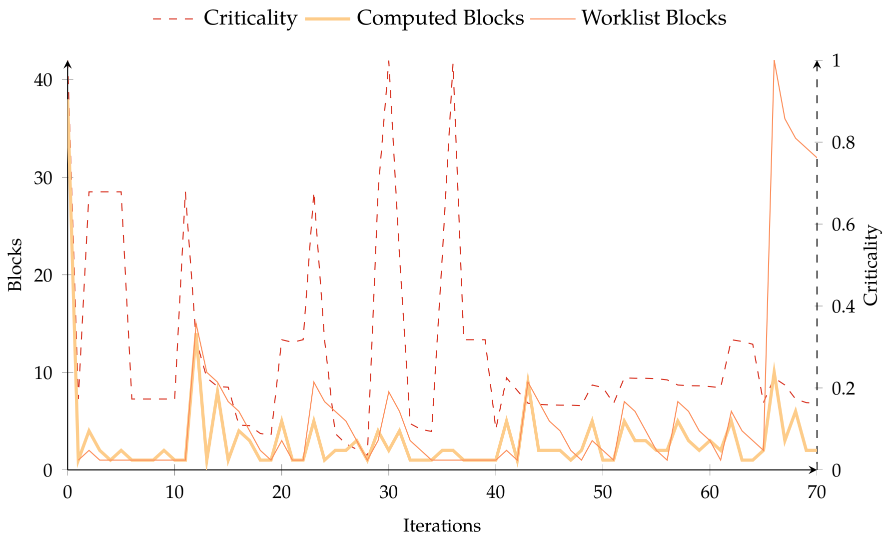


Figure 4.8: Behavior of default criticality computation over time for `debie-4a`, showing for each iteration the computed criticality value, the size of the worklist, and the number of computed blocks.

4.5.3 Criticality Overview of Real-Time Programs

Benchmark	BBs	Inf.	ILP_v	t_{total}
mdh-duff	18	0	1,469	1.54s
mdh-edn	67	6	15,149	32.09s
mdh-fac	10	0	425	0.22s
mdh-fdct	9	0	151	3.84s
mdh-fibcall	7	0	339	0.26s
mdh-insertsort	7	0	767	0.83s
mdh-jfdctint	12	0	669	2.64s
mdh-matmult	27	0	1,248	1.29s
mdh-nsichneu	754	0	4,023	7.72s
debie-2a	23	16	19	0.15s
papa-f1b	285	279	14	0.19s

Table 4.6: Criticality results (C-1: all basic blocks on WCEP)

Benchmark	BBs	Inf.	ILP_v	t_{total}
mdh-adpcm	154	1	2,487	3.47s
mdh-bs	11	0	85	0.18s
mdh-bsort1	22	0	383	0.45s
mdh-cnt	65	0	2,670	1.67s
mdh-cover	29	0	2,571	1.13s
mdh-crc	29	0	741	0.59s
mdh-fir	14	0	2,307	1.85s
mdh-janne_complex	14	0	1,754	0.74s
mdh-ndes	83	0	403	0.83s
mdh-ns	18	0	406	0.27s
mdh-recursion	13	0	2,182	1.32s
mdh-ud	204	0	1,897	3.50s

Table 4.7: Criticality results (C-.99: all basic blocks *critical*)

We finally perform a criticality-based profiling for the entire set of WCET benchmarks, which are available to us, and investigate their criticality distribution. Since the amount of useful information we can compute for each benchmark program varies, it does not make sense to present the following results by benchmark suite. We therefore group them into three distinct categories depending on their criticality properties. Table 4.6 contains those programs, for which all (feasible) basic blocks are part of the global WCEP. This result is revealed after a single WCET analysis run and makes any further criticality analysis obsolete. The table contains the number of basic blocks (BBs) in the program, infeasible blocks thereof (Inf.), size of the IPET integer linear program (ILP_v), and time spent during the (single) WCET analysis (t_{total}). The programs `mdh-edn`

						Analysis Time	
Benchmark	BBs	Inf.	Dist.	C_1	ILP_v	t_{ana}	t_{ilp}
mdh-compress	92	1		77	2,243	1.45s	0.59s
mdh-expint	25	1		17	256	0.28s	0.07s
mdh-fft1	491	22		220	34,154	10.56s	116.81s
mdh-lcdnum	22	0		16	551	0.22s	0.15s
mdh-ludcmp	430	2		177	60,209	15.21s	1,025.86s
mdh-minver	466	3		213	7,063	2.61s	21.35s
mdh-prime	23	0		20	12,239	4.98s	1.32s
mdh-qurt	423	5		162	21,920	4.94s	88.75s
mdh-select	99	0		79	1,236	0.61s	0.46s
mdh-sqrt	518	14		179	28,665	5.52s	126.70s
mdh-statemate	420	11		214	1,876	1.88s	4.20s
debie-1	83	3		43	270	0.22s	0.20s
debie-2b	23	1		15	66	0.17s	0.04s
debie-2c	23	8		14	35	0.14s	0.02s
debie-3a	74	9		51	595	0.51s	0.13s
debie-3b	74	8		50	3,160	1.05s	1.50s
debie-3c	74	8		50	3,160	1.06s	0.79s
debie-4a	285	11		38	2,358	0.94s	2.60s
debie-4b	285	236		29	160	0.33s	0.16s
debie-4c	285	260		16	51	0.29s	0.08s
debie-4d	285	264		16	39	0.29s	0.07s
debie-5a	138	12		114	1,044	0.73s	0.64s
debie-5b	138	2		123	21,212	5.96s	65.72s
debie-6a	376	53		174	13,962	9.18s	7.30s
debie-6b	376	52		174	16,170	9.79s	9.66s
debie-6c	376	52		137	58,122	36.86s	37.01s
debie-6d	376	12		176	15,177	9.02s	8.04s
papa-f1a	285	0		120	1,588	0.75s	3.30s
papa-f2	8	0		4	20	0.08s	0.03s
papa-a1	626	44		231	1,437	0.64s	5.21s
papa-a2a	1,522	535		448	21,891	7.15s	151.19s
papa-a2b	1,522	13		541	163,395	45.16s	2,853.03s
papa-a3a	981	717		143	1,383	0.95s	3.86s
papa-a3b	981	200		384	9,099	2.65s	41.74s
papa-a4	334	52		161	2,454	0.64s	3.32s
papa-a5	438	0		277	926	0.85s	4.65s
papa-a6	682	25		337	5,039	1.85s	20.45s

Table 4.8: Criticality results (C-dist: observable criticality distribution)

and `mdh-nsichneu` demonstrate that neither analysis overhead, nor the number of basic blocks, are suitable for predicting their criticality distribution. The programs in the C-.99 category in Table 4.7 have blocks very close to a criticality of 1.0, i.e., ≥ 0.99. The C-1 and C-.99 categories are mostly made up of programs from the Mälardalen benchmarks suite. This is not surprising since they are described in [17] as executing a single path due to their hard-coded input values. The lack of infeasible blocks for some programs in C-1, tells us though, that given our analysis tool (which may or may not be able to analyze this single-path property in all cases), no other execution paths exist (even considering other input).

Programs, which basic blocks have an observable distribution of criticality values are categorized as C-dist and we give their detailed results in Table 4.8. For these programs we include their criticality distribution as an inline (*sparkbars*) bar chart in column "Dist.". In the next column (C_1) and not included in the sparkbars, is the number of basic blocks on the WCEP. Note that the number of basic blocks varies between analysis problems due to unreachable code, which is eliminated by the analysis early on. Larger benchmarks like `debie-4a` and `debie-6b`, show a particularly interesting distribution, where a substantial amount of critical code has been discovered that is not part of the WCEP. For most of the other benchmarks, considerable amounts of the code have been found to have a very high criticality of more than 0.9. This is valuable information when trying to improve the code in order to meet timing constraints. Using standard WCET analysis, the only information available would be limited to the column C_1, i.e., blocks that are on the WCEP.

4.5.4 Estimation Results

To use our evaluation tool `a3` for an estimation experiment, we configure it to (1) ignore any inter-block pipeline effects and assume a perfect cache. This mode of WCET computation is referred to as "local best-case". We also (2) limit its use of contexts to a single loop and a single calling-context. As we explained in Section 4.3, it is not useful to estimate *criticality* for trivial programs. The choice of one or more program properties to control estimation depends on many factors: the analysis tool, target architecture, analysis settings and performance of the host system performing the analysis. All of these together determine the computational overhead. In the end it is up to the user of the analysis, to decide what is acceptable to her, in terms of analysis runtime. We consider the number of feasible basic blocks and the size of the IPET ILP and for our experiment's configuration, use the following disjunctive predicate to decide, whether a program qualifies for estimation: $\text{ILP}_v > 2000 \vee \text{BBs} > 150$.

Table 4.9 contains the results from comparing precisely computed, to estimated *criticality*. For every program that qualifies for estimation, the maximum error for all criticality values (*max*) is given, and as an average, the root mean square error (*rms*), which gives more weight to greater deviations from the expected value. With the analysis runtime of criticality computation split into data-flow analysis (t_{ana}) and ILP solving (t_{ilp}) times, we also report the overall speedup compared to precise analysis. This speedup can be significant. For example, the total computation time for `mdh-ludcmp`, is reduced from approximately 17 minutes to 6 seconds. The highest amount of error introduced through estimation for a single basic block's criticality is 0.22 (or 22%), with the *rms*-average being .15 (or 15%) for this program. In general, the estimation error is low enough in order for the obtained criticality profiles to be useful.

4.6 Discussion

4.6.1 Application of the Criticality Metric

We demonstrated that the criticality metric can be computed with reasonable overhead and thus, from a technical point, is suitable for use in practice. Since a large-scale qualitative survey of the usefulness of our novel metric would go beyond the scope of this thesis, we limit ourselves to describing three likely scenarios and implement one in the next chapter. All are related to real-time software development, for which we think that our metric is able to complement or improve the state-of-the-art: (1) profiling of worst-case behavior as information for the programmer, (2) supporting optimization decisions during WCET-driven compilation, and (3) guiding WCET analysis in order to yield tighter WCET bounds.

Code Profiling

What can the criticality metric provide the programmer with? It provides a relative ranking of *all* code blocks of a real-time program. A criticality close to 1 indicates that the code needs to be considered when modifying the program to meet timing constraints. This ranking can be presented using tables or code annotations, as done by profiling tools. For instance, critical code lines could be colored in an IDE. The metric does not, by itself, provide information which code blocks consume the most time. This is local information that can be trivially extracted from paths encountered during the criticality computation, e.g., the execution frequency of code on the path determining the code's criticality. It can be included as a secondary measure in the ranking presented to the programmer. Note that the metric provides absolute execution times, as it is derived through normalization. It thus helps highlighting regions of code instead of a singular path. This can make it easier to see underlying problems in the code with regard to analysis, for example the implementation of data structures with unfavorable worst-case behavior.

Guiding Compiler Optimizations

In the area of compiler-based optimization, code transformations need to base their decisions on cost models that reflect the properties, which are sought to be improved. In general, such models are based on the program's CFG, with weights attached to either blocks, edges, or both. An optimization targeting the average case (i.e., ACET optimization) can use execution frequencies derived from typical program runs to calculate these weights. As an example, consider *register allocation* and its *register spilling* subproblem,

	Error		**Analysis Runtime**		
Benchmark	*max*	*rms*	t_{ana}	t_{ilp}	speedup
mdh-adpcm	0.00009	0.00002	1.42s	0.17s	2.20
mdh-cnt	0.00296	0.00071	0.19s	0.15s	4.80
mdh-compress	0.03015	0.00603	0.36s	0.17s	3.91
mdh-cover	0.00052	0.00011	0.13s	0.04s	6.59
mdh-edn	0.00000	0.00000	1.71s	0.02s	18.60
mdh-fft1	0.00920	0.00078	1.64s	13.61s	8.36
mdh-fir	0.00026	0.00007	0.10s	0.02s	14.46
mdh-ludcmp	0.01234	0.00068	0.69s	5.64s	164.36
mdh-minver	0.03520	0.00172	0.96s	3.71s	5.13
mdh-nsichneu	0.00000	0.00000	4.40s	0.03s	1.74
mdh-prime	0.00058	0.00012	0.10s	0.04s	46.63
mdh-qurt	0.00468	0.00033	0.64s	8.38s	10.39
mdh-recursion	0.00083	0.00023	0.07s	0.02s	13.65
mdh-sqrt	0.00679	0.00063	0.40s	5.11s	24.01
mdh-statemate	0.00699	0.00214	1.12s	2.87s	1.52
mdh-ud	0.00051	0.00016	0.33s	0.95s	2.73
debie-3b	0.00181	0.00063	0.42s	0.14s	4.51
debie-3c	0.00175	0.00061	0.42s	0.16s	3.19
debie-4a	0.22401	0.14780	0.72s	1.74s	1.44
debie-5b	0.01901	0.00199	0.89s	0.83s	41.77
debie-6a	0.12512	0.05509	5.90s	3.83s	1.69
debie-6b	0.12512	0.05425	6.61s	4.09s	1.82
debie-6c	0.07545	0.02358	10.80s	7.18s	4.11
debie-6d	0.13494	0.03828	5.78s	3.93s	1.76
papa-a1	0.01136	0.00287	0.52s	5.41s	0.99
papa-a2a	0.00204	0.00046	2.82s	74.14s	2.06
papa-a2b	0.03716	0.01300	10.66s	590.81s	4.82
papa-a3a	0.04403	0.00418	0.63s	2.45s	1.56
papa-a3b	0.04958	0.00622	1.14s	20.75s	2.03
papa-a4	0.07817	0.01410	0.44s	2.56s	1.32
papa-a5	0.00824	0.00110	0.67s	4.88s	0.99
papa-a6	0.06687	0.00828	1.09s	19.89s	1.06
papa-f1a	0.00889	0.00248	0.38s	3.09s	1.17

Table 4.9: Estimation results (estimation error and overhead compared to precise analysis)

which are among the best studied problems in compilation. The state-of-the-art algorithms presented by the research community and those adopted in industrial compilers make use of weighted cost models to minimize the overhead caused by spill code [13, 52]. Using a comparable model, [44] describes an algorithm that optimizes the placement of spill code for callee-saved register at the boundaries between functions. We propose to use criticality as the basis for worst-case specific weights in such optimizations. While this does not reduce the problem of WCET optimization to ACET optimization, our metric's similarity to ACET-style weights benefits reuse of existing algorithms and models.

Supporting WCET Analysis

Another task for which the properties of criticality can prove valuable is WCET analysis itself. Similar to the case of optimizing compilers, a program profile based on *criticality* can provide feedback during WCET analysis, so that different parts of the program can be analyzed differently. This could mean choosing a more appropriate method for analysis or performing it with different levels of detail for different regions of the program.

In Chapter 5, we describe a first technique that is able to improve the precision of static WCET analysis by focusing it on WCET-critical regions of the program. Iteratively analyzing subgraphs we prune parts of the program's CFG that do not influence the global WCET and thus eliminate sources of overestimation. In the process, we use the ranking of basic blocks by criticality to identify candidate regions for pruning and to ensure a safe bound with regard to the original analysis problem.

4.7 Related Work

To the best of our knowledge, the criticality metric is the only existing approach to program profiling, which targets the WCET and is based on static analysis. There is however prior work that is using dominance and flow properties of CFGs in a way similar to ours. We discuss the two instances that share theoretical and practical concepts with criticality analysis. This is followed by a brief review of dynamic profiling and how the information gained through it, is being used. Existing approaches for WCET-aware compilation are discussed at the end.

Dominance-Based Propagation in CFGs

Hilal Agrawal has used dominance in control-flow graphs in a way similar to ours, but targeting improvements for code coverage during testing. The algorithms in [80] and [70] make use of the fact that a test case reaching a basic block also covers all its pre- and post-dominators. This is used to reduce the number of test cases or instrumentation points needed to ensure a certain code coverage. We exploit a stronger property of dominance and prove that the criticality of non-leaf nodes can be derived from its neighbors in the dominator graph.

Dynamic Profiling

Conceptually similar to our approach, Ball and Larus [77] target the efficient collection of dynamic profiles. Their goal is to find a minimal set of instrumentation points, i.e., basic blocks that record their execution, in order to compute *execution counts* for *all* edges, basic blocks or path segments of a CFG. However, they exploit flow laws in combination with maximum spanning trees, which cannot be applied to criticalities.

When it is sufficient to know how a program behaves (e.g., its runtime performance) most of the time, dynamic profiling can give an accurate picture. Unless the machine used for execution has profiling capabilities, it functions by either placing instrumentation code at strategic points in a program to record its execution behavior [88], or by sampling the state of the execution from time to time [82, 37]. During a profile run, the elapsed cycles and the number of executions can be recorded for blocks, edges, or paths up to a certain length. Such profiling techniques are often employed for performance optimizations either explicitly by a programmer during testing or implicitly by a compiler. Since the optimization targets the common case, typical input data, which is easy to come by, simply by executing the program under normal circumstances, can be used for the profile run. This is the fundamental difference compared to the static analyses the criticality metric is based on, which has to assume worst case behavior.

Measurement-Based WCET Profiling

WCET tools that involve a measurement-based based approach can also provide relevant profiling information, as well as code coverage metrics. While a WCET estimate for the program is iteratively computed, profile data is collected from individual executions. Garrido et al. [11] mention this feature in their recent report on using the hybrid WCET tool RapiTime in an experimental setting. The basic functionality of a hybrid WCET tool, puts measurement-based profiling closer to the dynamic common case techniques described above than to our profiling approach.

Profile Visualization

According to a taxonomy of software visualization in [73], *criticality* can be used for *static code visualization*. Profiling data can be aggregated and presented in many different ways. With regard to improving performance, a programmer may be interested in the general timing behavior and identifying critical regions (also referred to as *hot spots*) in the program. The call graph and function-level CFGs are often used to structure profiling results. For example, `gprof` [88], one of the earliest tools, presents a textual call graph profile for each called routine in the program. Graphical profiling tools and development environments visually annotate graphs of the program with execution times or weights that represent time spent during execution. Visualizing profile-based information on the source code level, in a way that is suitable even for large programs, can be achieved by using the line representation technique developed by Ball and Eick [76]. To be effective this would require the adaptation of an editor or graphical development environment. Results from WCET analysis can be visualized in a similar way. For example, `a3` makes use of the Visualization format for Compiler Graphs (VCG) to represent a program graph, which combines the call graph and local CFGs in a nested layout. Results from flow analysis, computed WCET cycles, and the WCEP itself are annotated in the graph on a higher level. Lower-level hardware states are initially hidden through nesting. Nesting and efficient layout of the VCG format [79] make it possible to handle large graphs, folding nested nodes allows interactive exploration. While VCG is also used by the FIRM compiler project [100] for visualization and could be considered a suitable candidate for criticality visualization, its only stand-alone viewers aiSee from AbsInt and yComp from FIRM are closed-source and unavailable without licensing. A recent, visual survey of software visualization techniques is given by Lemieux and Salois [43].

Compiler Support for WCET

Regarding the support for minimizing WCET bounds during compilation, WCC [42, 29, 10] is the only example of a fully WCET-aware compiler so far. Important components are its *flow fact manager*, responsible for keeping control flow information valid across all compiler passes. WCC integrates AbsInt's aiT WCET analyzer and can guide trade-offs in the compiler's decision based on its analysis information. Prantl, Kirner, et al. [18, 19] investigate how program flow constraints can be transformed alongside code in an optimizing compiler. The ability to do this safely allows one to use more elaborate transformations during optimization in the first place. The precision achieved by this transformation is a secondary goal, as together with the benefit of the optimizations, it ultimately allows a lower WCET bound to be proven. Our criticality-based profiling approach can in many cases —except trivial programs or those that have a single path— provide more detail on the worst-case relevant parts of a program, than a single IPET-based analysis (e.g. using aiT). This means that there is also more information for a compiler to base its decisions on. With *criticality* we furthermore want to make it easier to adapt existing compiler optimizations, which are normally guided by an ACET profile, to lower the WCET of program.

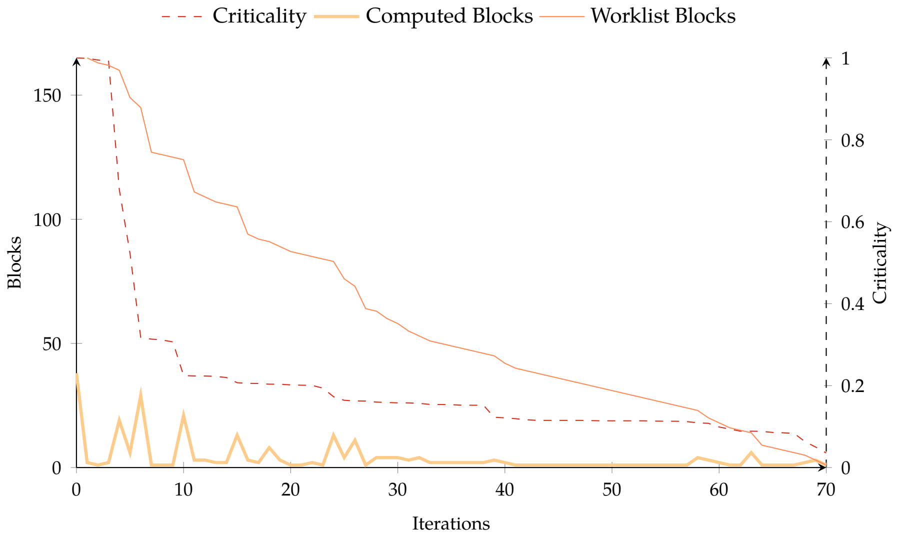
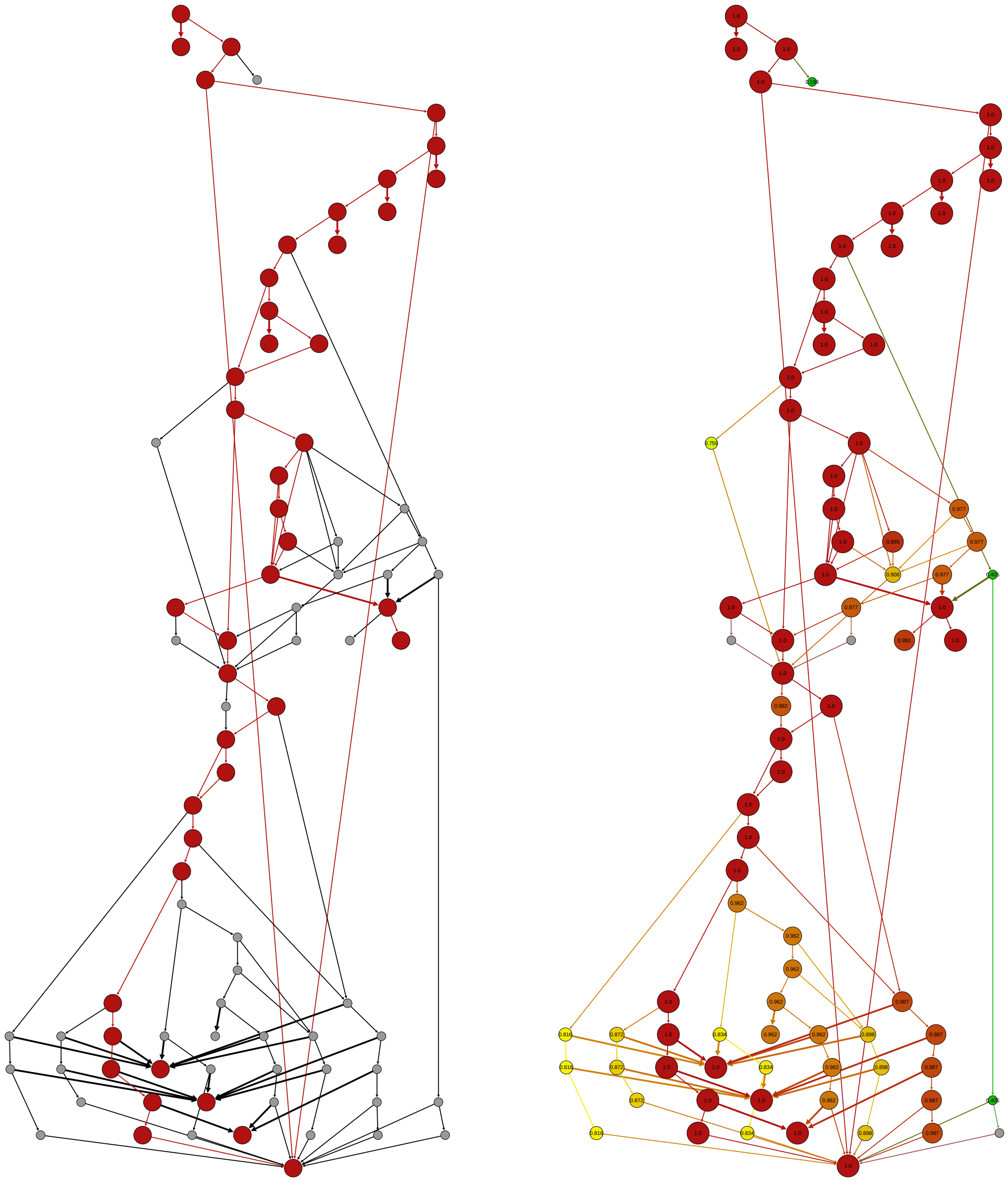


Figure 4.9: Behavior of the pruned criticality computation over time for `debie-4a`, showing for each iteration the computed criticality value, the size of the worklist, and the number of computed blocks.



(a) WCEP View

(b) Criticality View

Figure 4.10: The original WCEP view of `debie-1` and the profiling achieved through Criticality computation (calls are thicker edges, back-edges from procedures are omitted).

Chapter 5

Graph Pruning

In Chapter 4 we have proposed a technique to profile a program based on its WCET bound. The evaluation of this profiling metric showed that a surprisingly large number of basic blocks can be *uncritical* when a complete real-time application is considered. With this observation as a starting point, we now describe a *pruning* technique to improve the precision and computation time of a WCET analysis by excluding the uncritical code parts from the analysis. We start this chapter by looking at the sources of complexity and the problem of overestimation in static WCET analysis. Then we give a detailed description of our graph pruning algorithm, which aims to mitigate these problems. In Section 5.3 we evaluate it on a set of suitable WCET benchmark programs. Section 5.4 ends this chapter with an overview of related techniques that have the same goal of refining WCET bounds.

Our graph pruning approach for WCET analysis and its experimental evaluation has been described in a paper [1] that is currently under submission.

5.1 Sources of Overestimation

Foremost size and complexity of software are causing the analysis overhead to grow rapidly, as the number of potential states of the program under analysis increases. For WCET analysis this effect is amplified since in order to arrive at a safe WCET bound, hardware states need to be considered as well, adding to the number of potential software states. Even when only small portions of a complex software program are relevant for the final WCET estimation, the analysis, has to account for *all* of the program's code to derive a safe bound. Except for trivial analysis problems, this inevitably reduces the precision of the WCET analysis, as irrelevant code parts interfere with the analysis of relevant code parts and lead to unnecessary overestimation of the determined WCET bound compared to the actual worst-case behavior.

Previous work is able to eliminate instructions irrelevant to flow analysis (program slicing) and refine the WCET bound by identifying infeasible paths (see Section 5.4 for details). With Iterative Graph Pruning (`IGP`), described in detail in the following sections, we also aim to limit WCET analysis to relevant parts of a program, but do so on the lower CFG-level. Based on *criticality*, basic blocks are grouped into sets according to the length of their respective paths. The sets are then processed iteratively by decreasing path length. During each iteration a subgraph of the original CFG is formed by unifying the subgraph of the previous iteration with the basic blocks from the currently considered set. A potentially more advanced WCET analysis is then applied to the program represented by the new subgraph. The algorithm terminates, with a possibly refined WCET estimate, as soon as a safe bound, valid for the original program, has been reached.

The benefits from this approach are that (1) the analysis problems defined by the subgraphs at each iteration are much smaller than the original analysis problem. This promises to reduce the analysis overhead, while still providing tight bounds. Furthermore, (2) processing the sets of basic blocks according to their decreasing path lengths, eliminates interference from other basic blocks, whose longest paths are known to be shorter. This improves the precision of the WCET analysis precisely for those code parts of the real-time program that impact the WCET estimate the most.

5.2 Algorithm

In a nutshell, iterative graph pruning (`IGP`), presented as pseudo code in Algorithm 9, performs WCET analysis by iteratively merging a sequence of basic block sets (S_i), terminating when an upper bound (ub_{wcet}) is found to be a safe bound for the input program. The sets S_i are generated by the criticality analysis described in Chapter 4, i.e., by computing the longest path going through every basic block, represented by nodes in the CFG G. The blocks are then inserted into sets S_i based on the path length. The respective path length of a set can later be retrieved by $longestpath(S_i)$.

At every iteration i, the vertex-induced subgraph G' is created from the union of the i first (and most critical) sets S_i (l. 7 – l. 8). G' is then targeted by a full WCET analysis run (`WCETtoverAny` l. 13), which entails abstract interpretation on the program's subgraph G' to generate the weighting function $\mathcal{W}'$, followed by a longest path search. `WCETtoverAny`, additionally, forces the longest path search to only consider those paths passing through blocks in the current S_i. All other paths in G' are uninteresting, since these paths have already been bounded in the previous iterations. Note that G' may not contain any such path that is feasible. `WCETtoverAny` then returns 0 and the current upper bound remains unchanged. If, at the end of an iteration, the current upper bound (ub_{wcet}) is less than the just computed bound (WCET_i), it needs to be updated (l. 10).

The algorithm terminates at the latest when all sets have been considered (i.e., $V' = V$ and G' is the same as the graph of the original program) or before when the termination condition (l. 4) is met. The latter case occurs when the remaining longest path induced by the remaining S_is is shorter than the current WCET bound ub_{wcet}. We will prove that ub_{wcet} is a valid bound for G in the next section.

Example 13. *Consider the CFG shown in Figure 5.1, where weights are shown for each basic block along with the longest paths passing through the respective blocks. Furthermore, assume*

Algorithm 9 Graph-pruning algorithm

```
Input: $G = (V, E, r, t)$  CFG of the input program.
       $S_1, \dots, S_n$  Block-sets sorted by longest path length, from highest to lowest.
$ub_{wcet} = 0$ ▹ Will increase until a safe WCET bound
for $i = 1 \rightarrow n$ do
    ▹ Terminate when no longer paths can exist
    if $ub_{wcet} \geq longestpath(S_i)$ then
        return $ub_{wcet}$

    ▹ Construct and analyze a subgraph
    Let $V' \leftarrow S_1 \cup \cdots \cup S_i$, $E' \leftarrow E \cap V' \times V'$ in
        $G' \leftarrow (V', E', r, t)$
        $\mathrm{WCET}_i \leftarrow$ CalcPrunedWCET($G', S_i, ub_{wcet}$)
        $ub_{wcet} \leftarrow max(\mathrm{WCET}_i, ub_{wcet})$
return $ub_{wcet}$

function CalcPrunedWCET($G', S_i, ub_{wcet}$)
    return WCETonAny($G', S_i$)
```

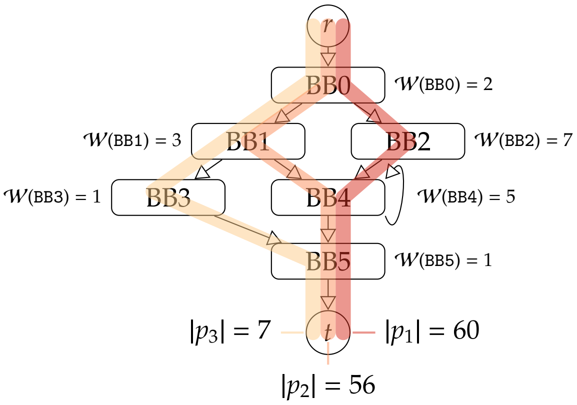


Figure 5.1: Weighted CFG from Figure 4.1 (loop added to `BB4`) with longest path lengths

that the number of loop iterations at `BB4` *depends on a variable* `x` *either assigned to* 10 *in* `BB1` *or* 7 *in* `BB2`.

A first, unmodified WCET analysis discovers both assignments to `x`. *Thus, it derives a loop bound of* 10 *iterations for* `BB4`. *The resulting longest path* (p_1) *has length* 60 *and covers* r, `BB0`, `BB2`, `BB4`, `BB5` *and* t. *These blocks are thus assigned to the basic block set* S_0. *The second longest path passing through a node not covered by* p_1 *is* p_2, *with length* 56. *Since all blocks on* p_1 *are already assigned to* S_0, *the only new block of* p_2 *is assigned to* S_1 = {`BB1`}. *The same* *applies to path* p_3 *and block set* $S_3 = \{\texttt{BB3}\}$.

Our Algorithm starts by reanalyzing subgraph G_1 *induced by* S_0 *shown by Figure 5.2. The analysis now discovers that there is only a single assignment to* `x`*, limiting the number of loop iterations at* `BB4` *to* 7*. In addition, a more precise estimation of the local execution time within the loop is derived, i.e.,* $\mathcal{W}(\texttt{BB4})$ *is now* 4 *instead of* 5*. The initial length of* 60 *for* p_1 *alone was heavily overestimated, since the longest path in* G_1 *is much shorter. This results in an upper bound* ub_{wcet} *of* 38*. The algorithm continues to iterate at this point since the path length associated with* S_1 *is longer than the current upper bound. The second iteration leads to the construction of* G_2 *induced by* $S_0 \cup S_1$ *(see Figure 5.2). A new WCET analysis run is forced to consider only those paths passing through* `BB1` *and finds a loop bound of* 10 *iterations. This results in an upper bound* ub_{wcet} *of* 56*. Since the upper bound now represents a path longer than any that could be uncovered by including the next block set* S_3*, the algorithm terminates with a more precise WCET bound of* 56 *instead of the initial* 60*.*

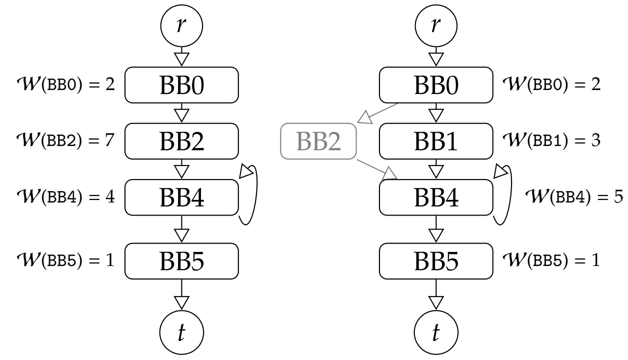
5.2.1 Correctness

To show the correctness of our approach, we have to consider the impact of graph pruning on the typical phases of a WCET analysis run. We assume, without loss of generality, that WCET analysis is performed in two phases [69]. A first phase, based on abstract interpretation [91], delivers local worst-case execution times for each basic block. In the second phase, a longest path search [78, 74] is performed on a weighted CFG, computed from these local execution times.

In our approach, abstract interpretation is applied to a subgraph G' of the original CFG G of the input program. In order to show correctness we thus have to investigate some properties of the subgraph G'.

Lemma 7. *A subgraph* $G' = (V', E', r, t)$ *constructed by Algorithm 9 is connected, i.e., for every CFG node* $v' \in V'$ *a path from* r *to* t*, passing through* v'*, exists.*

Proof. This follows immediately from the way subgraphs are constructed. Remember that the blocks in the subgraph G' of the m-th iteration are computed by unifying all



(a) 1st iteration (b) 2nd iteration

Figure 5.2: Weighted CFGs formed during Iterative Graph Pruning.

the basic block sets S_i, $i \in \{0...m\}$, whose path lengths are longer than the path length associated with S_m, i.e., $V' = \bigcup_{i \in 0..m} S_i$.

Assume G' is not connected, i.e., a CFG node v' exists that is not reachable from the root node r in G'. Since $v' \in V'$ it follows that a corresponding path $p = (r, ..., n', ..., t)$ has to exists in the original graph G. The length of this path corresponds to the path length associated with S_m. As G' is not connected, at least one node of p is not in V'. However, this is impossible, since the existence of p implies that this node either is in S_m or another set S_k, $k < m$. The subgraph G' is thus connected. ■

Lemma 8. *Abstract interpretation applied to a subgraph G' delivers correct results with respect to the potential execution paths through G'.*

Proof (sketch). A sound analysis based on abstract interpretation delivers sound results with regard to potential executions of a program, e.g., represented by its CFG G. When applying the same analysis to a subgraph G', the set of potential executions is reduced (excluding those executions that do not lead to the sink node t). Since the analysis is sound, this approximation trivially remains sound with regard to the executions represented by G'. Note, however, that the analysis result is not guaranteed to be sound with regard to all executions of the original program. ■

The previous two lemmas ensure that, independent of the concrete analysis performed, the local execution times obtained by abstract interpretation in the WCET analysis tool are sound with respect to a subgraph G'. It remains to show that the WCET bound computed during the longest path search by Algorithm 9 is a safe bound.

Lemma 9. *The worst-case execution time bound computed by Algorithm 9 for a subgraph G' is safe.*

Proof. Consider the subgraph G' and its basic block set S_m of the m-th iteration as well as the subgraph G'', with its bound WCET(G''), of the previous iteration.

First, assume that the longest path p through G' represents a valid path in G'', i.e., p does not contain any node in S_m. In relation to the previous iteration, the current WCET estimate for p might increase, due to overestimation of the local worst-case execution times, i.e. WCET(G'') $\leq$ WCET(G'). However, irrespective of the WCET computed for G', the bound established by the previous iteration for G'' still holds. Paths of this structure are thus uninteresting, as the length of all paths in G'' has been bounded by the previous iteration. It remains to bound those paths in G' that do not have a corresponding path in G''. The longest path search thus only needs to consider paths containing at least one node in S_m.

Three cases for the longest path p need to be considered:

1. $|p| >$ WCET(G''):
   Since $|p|$ is longer than the previously established WCET bound, it follows that WCET(G') $= |p|$.

2. $|p| \leq \mathrm{WCET}(G'')$:
   As the length of p is not longer than the previously established bound, it follows that $\mathrm{WCET}(G') = \mathrm{WCET}(G'')$. The longest path through G'' also represents the longest path through G', ignoring any additional overestimation caused by blocks in S_m.

3. No feasible path containing a block in S_m exists:
   This case happens when the abstract interpretation finds that no execution in G' exists that passes through a block in S_m, i.e., none of the conditions of the branches leading to a block in S_m can be satisfied. It follows that $\mathrm{WCET}(G') = \mathrm{WCET}(G'')$. Note that paths over these blocks might become feasible in later iterations, e.g., when code making the, yet unsatisfiable, conditions satisfiable is added.

Using induction, we can finally prove that Algorithm 9 delivers safe WCET bounds for all subgraphs considered during the iterative processing. ■

Theorem 6. *Algorithm 9 computes a safe WCET bound.*

Proof. This follows immediately from the previous lemmas and the termination condition of the algorithm.

The lemmas prove that applying abstract interpretation on subgraphs is sound and that the algorithm computes safe bounds with respect to the subgraphs considered during the iterative processing.

It remains to show that no other paths exist, which could be longer than the WCET bound delivered by the last iteration. This is guaranteed by the order in which the sets of blocks are processed and the termination condition (Algorithm 9, l. 4). Together these ensure that all basic blocks not yet considered by the iterative refinement can only induce paths that are shorter than the computed bound. ■

5.2.2 Complexity

The number of sets S_i is an upper bound on the iterations that will be performed by `IGP`. The former is again bounded by the number of basic blocks in the input program. Since its iterations are linear in the number of blocks, `IGP` is dominated by the complexity of the WCET analysis, i.e., abstract interpretation and longest path search.

The sets S_i, which are assumed as input in Algorithm 9, can be efficiently computed using the criticality algorithms (cf. Chapter 4). This may be performed either during a preprocessing step, or on demand, while the graph pruning algorithm iterates.

5.2.3 Algorithm Variants

It may be the case that the WCET analysis tool targeted by graph pruning, can be configured for different levels of precision. This usually involves a trade-off between tightness (precision) of the WCET bound and longer analysis runtime. `IGP` can be used to incorporate analyses with different precision. Running the higher-precision analysis on a previously pruned graph would be a straight-forward way of further improving the

Algorithm 10 WCET computation function CalcPrunedWCET using two-stage analysis

```
Input: $G' = (V', E', r, t)$ ... A CFG.
       $S_i$ ... The set of newly added blocks.
       $ub_{wcet}$ ... The current upper bound of the WCET.
WCET$_i$ ← WCETover AnyFast$(G', S_i)$
if WCET$_i > ub_{wcet}$ then
    WCET$'_i$ ← WCEToverAnyPrecise$(G', S_i)$
    return WCET$'_i$
return WCET$_i$
```

WCET bound. But Algorithm 9 can also be modified to make use of multiple levels of precision directly. To do this, we replace the CalcPrunedWCET function in Algorithm 9 with the variant given in Algorithm 10. This algorithm performs a second, more precise WCET analysis (`WCEToverAnyPrecise`) to lower the estimate of WCET$_i$ for the current graph G', whenever the imprecise analysis (`WCEToverAnyFast`) would increase the overall WCET bound.

Another valid, but trivial way of incorporating a higher-precision analysis into graph pruning, would be to run `WCEToverAnyPrecise` on the pruned subgraph exactly once after the iterative processing. This would reduce the computational overhead and further lower the WCET bound (down to the path length of the next block set). One could even avoid the iterative processing entirely, by heuristically constructing a subgraph and applying the precise analysis to this subgraph. This would foremost reduce the computational overhead and leave the burden of reducing overestimation on the precise WCET analysis.

5.3 Evaluation

We start by giving a complete example of running iterative graph pruning on the first problem from the Debie1 benchmarks, which has no loops and 83 basic blocks. We then extend our evaluation to a number of WCET benchmarks using standard Iterative Graph Pruning (`IGP`) and its two-stage variant (`IGP-TS`).

5.3.1 Case Study: `debie-1`

We start by analyzing the original program represented by its CFG G. This yields a first valid global WCET bound in cycles (denoted by WCET(G)).

Pre-run: WCET(G) = 1621

At the same time we can perform a basic-block-level WCET profiling according to the definition of *criticality*. This yields, for every block b in the CFG G, the maximum length (a value in the range $[0, \mathrm{WCET}(G)]$) of all paths going through b. Blocks with a value close to the global WCET are considered critical, while those close to 0 are uncritical.

Set	S_i	*longestpath*(S_i)
S_0	43	1,621
S_1	1	1,613
S_2	1	1,608
S_3	5	1,601
S_4	1	1,592
S_5	4	1,585
S_6	8	1,560
S_7	1	1,472
S_8	3	1,456
S_9	3	1,415
S_{10}	3	1,352
S_{11}	3	1,324
S_{12}	1	1,231
S_{13}	2	657
S_{14}	1	321

Table 5.1: Criticality based basic block sets for `debie-1`

Based on their longest paths, we now insert blocks into several sets, so that blocks with the same path length end up in the same set. In our example 15 such sets exist, their path length (*longestpath*(S_i)) and cardinality is given in Table 5.1.

The pruning algorithm kicks off with a WCET analysis, but this time considers only the blocks on the global WCEP, i.e., the blocks in S_0. Using these blocks a vertex-induced subgraph G_1 is created and its WCET bound is analyzed.

Iteration 1: $G_1 = S_0$ WCET(G_1) = 334

WCET(G_1) is drastically lower than the original WCET, even though all blocks of the original WCEP are contained in G_1. Using abstract interpretation the WCET tool has derived that the path is actually infeasible, i.e., no legal execution for the path exists. It is necessary to add more blocks back to the program at this stage. And since we suspect that those blocks with longer paths have a larger impact on the attainable WCET bound, we select the most critical blocks available, namely S_1.

Iteration 2: $G_2 = G_1 \cup S_1$ WCET(G_2) = 334
Iteration 3: $G_3 = G_2 \cup S_2$ WCET(G_3) = 334
Iteration 4: $G_4 = G_3 \cup S_3$ WCET(G_4) = 1423

Adding S_1 to the graph did in fact not change the current WCET bound, neither did S_2. But after adding S_3, a considerably longer path becomes feasible and our current WCET bound jumps to 1423 cycles. Let us pause for a moment and consider what we have done so far. G_4 contains the most critical code ($S_0 \cup \cdots \cup S_3$), it contains 42 basic blocks (compared to 80 for the whole program). Blocks from eleven sets have not yet been added and we have a current bound of WCET(G_4) $<$ WCET(G). Is this already a valid WCET bound for the original problem? No, looking at the remaining sets, we can see there are still blocks in sets $S_4, \dots, S_8$, which may be part of a longer path in an extended subgraph, i.e., $longestpath(S_i) \geq 1423,\ 4 \leq i \leq 8$. With this in mind, we resume adding blocks with S_4.

Iteration 5: $G_5 = G_4 \cup S_4$ WCET(G_5) = 1525
Iteration 6: $G_6 = G_5 \cup S_5$ WCET(G_6) = 1525
Iteration 7: $\underline{G_7} = G_6 \cup S_6$ WCET(G_7) = $\underline{1525}$

When we are about to add set S_7, we notice that in the worst case, it would uncover a path with length 1472. Since the WCET bound changed again in iteration 5, this is below the current value of 1525 and thus would have no impact. This signals termination for our algorithm. At this point, blocks from all sets are either already part of the current graph, or do not contain blocks which could enable a path longer than 1525 cycles. Our result is the pruned graph G_7, which is a WCET-bound-maintaining subgraph of the original program. By using this graph, we have reduced the initial WCET bound of the analysis by 96 cycles ($\sim$ 6%).

5.3.2 Setup for Experiments

We evaluate our approach using AbsInt's aiT tool (`a3` version 12.10i) and the same set of WCET benchmarks (cf. Section 2.2.2), which we already know from Chapter 4. Our target configuration for the PowerPC MPC5554 processor (cf. Section 2.2.1) and analysis settings remain unchanged.

The prototype implementation of graph pruning that we use for this evaluation, treats the WCET analysis tool as a black-box and thus works with an unmodified version of aiT. The graph pruning is realized through scripting and the `AIS` annotations (`IS NEVER EXECUTED` and `FLOW`). This means that due to our setup, *all* information computed by an iteration is discarded and *not* reused by the following iteration. Thus, any analysis runtime results would be dramatically increased. We discuss this shortcoming of our evaluation in Section 5.3.5).

The distribution of basic blocks by their WCET criticality is a side-product of the initial analysis in `IGP` and `IGP-TS`. Thus, we can also rely on our categorization of programs from Section 4.5.3 and exclude those programs (C-1, C-.99), for which no pruning can be performed, from further evaluation. We therefore apply the two graph pruning variants to the 37 analysis problems from the C-dist category and compare the attainable WCET bound with the original result of aiT. The first variant (`IGP`), performs iterative graph pruning using standard IPET (Algorithm 9). The second variant (`IGP-TS`), uses the two-stage WCET computation (Algorithm 9 with the extension in Algorithm 10). Here, the standard computation is potentially refined by a second, more precise, but also more expensive, WCET analysis using aiT's *prediction file* technique [21].

We additionally examine properties of the analysis problem that have a direct influence on overestimation. These are (1) hardware splits, a measure for the amount of duplicated states in the presence of unpredictable hardware behavior (i.e., caches, branch prediction), and (2) the size of the analysis problems at different stages (i.e., subgraph size).

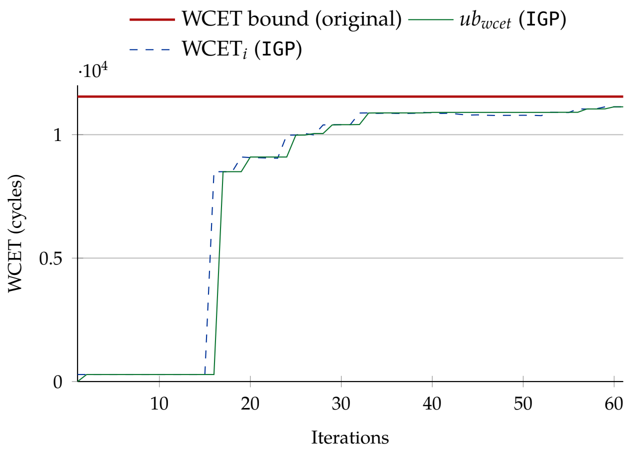


Figure 5.3: WCET bounds per iteration for the `f1a` benchmark using Iterative Graph Pruning. (`IGP`, lower is better)

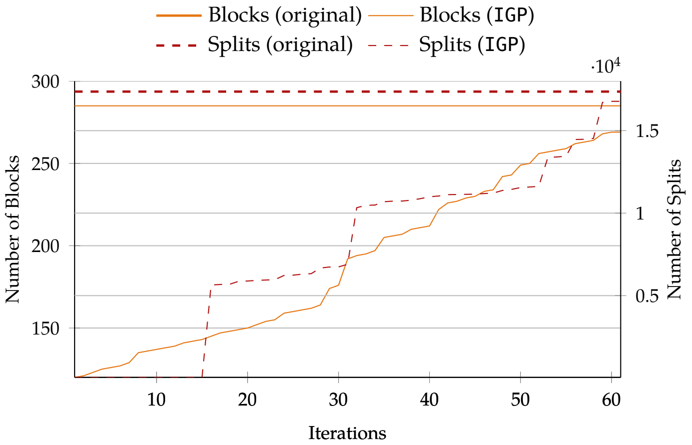


Figure 5.4: Subgraph size and hardware splits per iteration for the `f1a` benchmark using Iterative Graph Pruning. (`IGP`, lower is better)

5.3.3 Iterative Graph Pruning

Table 5.2 summarizes the WCET bound improvement and aggregates it by benchmark suite. The selected Mälardalen programs, especially the ones representing algorithmic cores with almost single-path execution behavior, not surprisingly do not benefit from graph pruning. Their WCET bound improvement is 2% at best. However, the larger application-like programs, Debie1 and PapaBench, benefit from graph pruning and some of their analysis problems have their WCET bound reduced significantly. When we look at `f1a` from PapaBench in detail, we can see in Figure 5.3 that within the first 20 iterations, the upper bound (ub_{wcet}) leaps to a level close to its final value. At this point, the most critical basic block sets have been added to the subgraph and WCET analysis returned a good candidate for the actual global WCEP. Step sizes subsequently decrease

Suite	Contains	Algo.	Improved	WCET Refinement *max*	*min*	*mean*
debie	16	`IGP`	12	−6.43%	0.00%	−1.74%
mdh	11	`IGP`	7	−2.15%	0.00%	−0.58%
papa	10	`IGP`	6	−3.73%	0.00%	−1.95%
debie	16	`IGP-TS`	10	−5.34%	0.00%	−0.82%
mdh	11	`IGP-TS`	5	−1.28%	0.00%	−0.26%
papa	10	`IGP-TS`	4	−2.83%	0.00%	−1.03%

Table 5.2: WCET Reduction using Iterative Graph Pruning (Summary)

and from iteration 35 on, the upper bound (ub_{wcet}) roughly holds while more blocks are added to the `IGP` subgraph (see Figure 5.4).

Another group of problem instances does not exhibit properties that are exploitable by graph pruning. These can be identified in Table 5.3 by their low number of iterations: after one or two iterations, `IGP` terminates since all (feasible) blocks are either on the global WCEP, or there is no interference between WCET-critical and unrelated code.

There is also a third class of benchmark problems, which exhibits a high number of iterations without a significant improvement of the WCET bound. Studying the most severe cases (problems `a2a` and `a2b` from PapaBench), we have found that almost all of the WCEPs found in subgraphs are infeasible on their own (i.e., their feasibility depends on blocks in other block sets). This is likely caused by a particular program structure and the WCET analysis failing to exclude infeasible paths from the longest path search (*flow facts*).

For all non-trivial benchmarks (two `IGP` iterations or more), we see that the minimal (and average) reduction of hardware splits is high (see Figure 5.5). Average subgraph sizes (Figure 5.6) are likewise significantly smaller than their originals. This tells us that overestimation is being effectively addressed by `IGP`. At the same time it can benefit from analysis problems, which are roughly half the original size. In Table 5.3 we present the detailed results from graph pruning using the `IGP` algorithm. It contains the number of iterations in column “Iter.”, the number of unique WCET paths encountered in column “$|\mathcal{P}|$”, and a comparison of graph sizes before and after pruning in the last two columns. Note that the number of infeasible blocks is contained in $|V|$ and will always be pruned. `IGP` improves WCET bounds up to 6% compared to aiT’s result on the original program. The average improvement among non-trivial benchmark problems is 2%.

5.3.4 Two-stage Iterative Graph Pruning

To evaluate the two-stage approach of iterative graph pruning (`IGP-TS`), we make use of the prediction file (PF) based IPET solver in aiT as a second stage analysis (i.e., for the `WCEToverAnyPrecise` invocation in Algorithm 10). When performing PF-based WCET computation, the timing information during longest path search is not restricted to a single WCET for each basic block, but may encompass multiple architectural states [21].

	WCET (cycles)					**Graph size**	
Benchmark	Original	`IGP`	Refined	Iter.	$\lvert\mathcal{P}\rvert$	$\lvert V\rvert$	$\lvert V'\rvert$
debie-1	1,621	1,525	−5.92%	7	6	83	63
debie-2b	566	534	−5.65%	2	2	23	16
debie-2c	489	489	0.00%	1	1	23	14
debie-3a	5,094	5,047	−0.92%	5	5	74	57
debie-3b	28,515	28,451	−0.22%	8	9	74	59
debie-3c	29,227	29,163	−0.22%	8	9	74	59
debie-4a	4,913	4,843	−1.43%	4	4	285	43
debie-4b	1,789	1,674	−6.43%	4	5	285	32
debie-4c	910	891	−2.09%	2	2	285	17
debie-4d	968	968	0.00%	1	1	285	16
debie-5a	6,119	6,047	−1.18%	6	6	138	120
debie-5b	112,538	112,467	−0.06%	7	7	138	130
debie-6a	44,273	44,007	−0.60%	12	10	376	192
debie-6b	44,273	44,007	−0.60%	12	10	376	192
debie-6c	63,133	62,145	−1.57%	13	10	376	155
debie-6d	46,337	46,049	−0.62%	12	10	376	194
mdh-compress	26,697	26,601	−0.36%	8	8	92	84
mdh-expint	793,236	785,573	−0.97%	3	3	25	19
mdh-fft1	651,767	651,249	−0.08%	98	77	491	455
mdh-lcdnum	4,162	4,162	0.00%	1	2	22	16
mdh-ludcmp	973,441	971,502	−0.20%	104	104	430	423
mdh-minver	298,523	292,108	−2.15%	103	104	466	457
mdh-prime	194,136	194,055	−0.04%	3	3	23	22
mdh-qurt	649,934	649,434	−0.08%	101	98	423	408
mdh-select	261,214	260,317	−0.34%	11	12	99	93
mdh-sqrt	219,363	219,074	−0.13%	98	99	518	449
mdh-statemate	24,145	23,657	−2.02%	70	71	420	363
papa-a1	8,551	8,247	−3.56%	101	92	626	484
papa-a2a	81,656	81,187	−0.57%	177	146	1,522	944
papa-a2b	100,939	100,190	−0.74%	199	163	1,522	1,079
papa-a3a	9,138	8,922	−2.36%	65	64	981	255
papa-a3b	28,193	27,691	−1.78%	146	139	981	746
papa-a4	11,104	10,879	−2.03%	44	38	334	253
papa-a5	20,545	20,320	−1.10%	96	97	438	426
papa-a6	29,092	28,006	−3.73%	118	114	682	608
papa-f1a	11,542	11,128	−3.59%	60	47	285	269
papa-f2	237	237	0.00%	1	1	8	4

Table 5.3: Detailed results for Iterative Graph Pruning (`IGP`)

		WCET (cycles)				**Graph size**	
Benchmark	Original	`IGP-TS`	Refined	Iter.	$\vert\mathcal{P}\vert$	$\vert V\vert$	$\vert V'\vert$
debie-1	1,507	1,479	−1.86%	7	9	83	63
debie-2b	520	520	0.00%	2	2	23	16
debie-2c	474	474	0.00%	2	2	23	15
debie-3a	4,787	4,735	−1.09%	6	10	74	58
debie-3b	24,756	24,742	−0.06%	8	15	74	59
debie-3c	25,348	25,334	−0.06%	8	15	74	59
debie-4a	4,810	4,783	−0.56%	4	5	285	43
debie-4b	1,630	1,543	−5.34%	4	5	285	32
debie-4c	877	877	0.00%	4	4	285	20
debie-4d	957	957	0.00%	1	1	285	16
debie-5a	5,784	5,753	−0.54%	8	15	138	123
debie-5b	78,338	78,307	−0.04%	9	18	138	133
debie-6a	42,945	42,680	−0.62%	12	19	376	192
debie-6b	42,945	42,680	−0.62%	12	19	376	192
debie-6c	61,042	60,129	−1.50%	13	14	376	155
debie-6d	44,677	44,390	−0.64%	12	19	376	194
mdh-compress	26,344	26,297	−0.18%	8	15	92	84
mdh-expint	792,298	784,693	−0.96%	3	3	25	19
mdh-fft1	589,137	588,871	−0.05%	103	163	491	468
mdh-lcdnum	4,140	4,140	0.00%	2	5	22	21
mdh-ludcmp	899,460	899,236	−0.02%	105	209	430	426
mdh-minver	275,957	275,805	−0.06%	104	209	466	458
mdh-prime	194,053	194,031	−0.01%	3	4	23	22
mdh-qurt	595,432	595,085	−0.06%	104	200	423	416
mdh-select	222,750	222,215	−0.24%	12	25	99	99
mdh-sqrt	194,497	194,471	−0.02%	127	255	518	503
mdh-statemate	22,173	21,890	−1.28%	98	195	420	409
papa-a1	7,050	7,049	−0.01%	141	267	626	581
papa-a2a	68,641	68,340	−0.44%	200	337	1,522	987
papa-a2b	84,785	84,280	−0.60%	210	347	1,522	1,111
papa-a3a	8,488	8,347	−1.66%	70	136	981	262
papa-a3b	24,063	23,984	−0.33%	164	312	981	780
papa-a4	9,473	9,288	−1.95%	57	100	334	278
papa-a5	19,020	18,965	−0.29%	104	205	438	434
papa-a6	25,045	24,507	−2.15%	122	236	682	635
papa-f1a	10,359	10,066	−2.83%	71	114	285	284
papa-f2	236	236	0.00%	1	1	8	4

Table 5.4: Detailed results for two-stage Iterative Graph Pruning (`IGP-TS`)

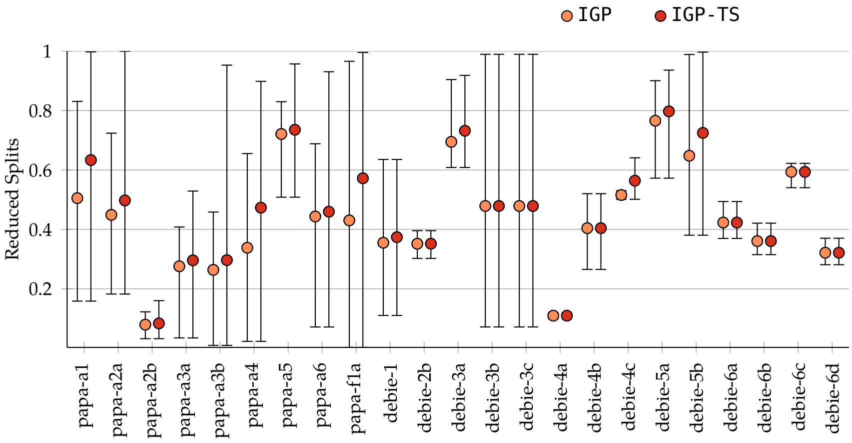


Figure 5.5: Reduction of hardware splits using Iterative Graph Pruning. (`IGP` vs. `IGP-TS`, lower is better. Vertical bars display iterations' min./max., marker is average.)

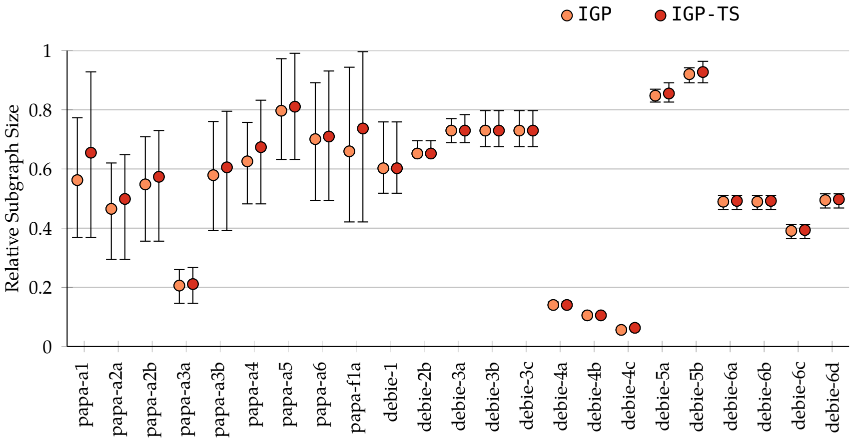


Figure 5.6: Subgraph sizes using Iterative Graph Pruning. (`IGP` vs. `IGP-TS`, lower is better)

Tightening WCET bounds in this way comes at the cost of —depending on program size— much larger ILP problems and thus time needed for solving. (Note that resource demand, i.e., memory, for the larger problems of `papabench` on a related out-of-order PowerPC architecture is bordering on infeasibility.) Fast WCET analysis remains unchanged with regard to `IGP`.

The WCET bound improvement of `IGP-TS`, compared to the PF-enabled analysis as a baseline, behaves similar to that of `IGP`. Benchmark problems, for which the WCET bound was improved by `IGP`, also improve by `IGP-TS`, but the effect is less pronounced (see Table 5.2). We thus conclude that our approach is profitable even compared to a state-of-the-art WCET analysis tool using its most sophisticated analysis technique. How its profitability increases inversely proportional to the use of mechanisms that prevent overestimation.

The reduction of hardware splits and graph sizes measured over all iterations also behaves similar (compare `IGP` and `IGP-TS` in Figures 5.5 and 5.6), although it fails to "cut off" sources of overestimation in the same was as `IGP` does. Furthermore, as we expected, the number of unique WCEPs found by `IGP-TS` is higher (see $|\mathcal{P}|$ in Table 5.3 versus Table 5.4). This is due to the more precise analysis being used. For the same reason, we can see an increase in the number of iterations.

5.3.5 Discussion

We have evaluated graph pruning using a state-of-the-art, commercial WCET tool. aiT uses powerful abstract interpretation and is able to produce good WCET bounds on its own. Even so, graph pruning can eliminate sources of overestimation and significantly tighten the WCET bound. While we can configure aiT to analyze subgraphs and extract all results we need from it, our setup is only suitable as a proof-of-concept. The analysis tool is treated as a block box, which leads to needless overhead that could be avoided. We thus do not present detailed measurements of the analysis time here. However, even with these short-comings we observed an increase in analysis time by a factor of 9 on average (tests were performed on an AMD Opteron 8356 at 2.3 GHz, running Linux Kernel version 2.6, with CPLEX version 10 solving the IPET ILP problems).

We expect that most of the analysis overhead can, in fact, be eliminated by designing the WCET analysis to take advantage of the iterative processing. The overhead of performing a complete run of abstract interpretation on every iteration can, for instance, be avoided. Abstract interpretation usually is performed by searching for a fixed-point. Adding basic blocks, as done by our algorithm, can easily be handled by this approach. The fixed-point search can continue from the abstract states computed for the previous iteration to quickly derive a new fixed-point for the current subgraph. Other forms of incremental analysis should equally reduce the overhead of performing a longest path search on structurally similar subgraphs. These techniques, combined with smaller problem sizes (due to smaller subgraphs and reduced hardware splits), promise to even reduce the analysis overhead, compared to a full analysis run using aiT's prediction file technique. We even observed this behavior in our tests for `IGP-TS` and the `a2b` benchmark. Despite an increase in the analysis time by a factor of 14 for the abstract interpretation, a reduction of the ILP solving time lead to an overall reduction of the analysis time of about 15%.

We observed that our technique addresses the problem of overestimation very well, in particular during early iterations. However, we also observed that in many cases the overestimation grew fast, often outweighing large initial gains. The main problem is that the subgraphs steadily grow larger. We could address this problem by restricting the subgraphs to only those nodes reachable from the current basic block set (S_i). However, one could similarly change the strategy for growing subgraphs, e.g., by estimating the impact on the number of hardware splits. In a similar way, neighboring basic block sets may be merged in order to avoid excessive iteration counts.

5.4 Related Work

We divide related approaches for tightening WCET bounds roughly by the main technique they employ. Since most methods —ours included— are complementary to each other and thus can be combined within an integrated analysis, some natural overlap occurs.

Program Slicing

Several pruning techniques, similar in spirit to our technique, have been proposed in the past based on program slicing [89]. The basic idea of program slicing is to improve the precision and the computational overhead of static program analyses by discarding *program statements* that are irrelevant to the goal of the analysis. Consider, for instance, the case when the goal of a static analysis is to determine the value of a given variable in a program. When forming a slice for that particular variable, only those statements are considered during the analysis that directly and indirectly contribute to the computation of that variable. All other statements are ignored. The goal in our approach is to improve the analysis of the WCET itself, our technique thus can be seen as a form of *program slicing on the timing domain*.

Sandberg et al. [46], for example, propose to use program slicing to improve the static analysis of flow facts. They construct program slices based on either all conditions of branches in a program, on all loop-exit conditions, or on the loop-exit conditions of a particular loop. Based on the computed slices, flow facts, such as loop bounds, are computed. In contrast to our work, the focus here is on deriving flow facts only, regardless of the relevance or impact to the final WCET. Their technique can, however, be combined with our approach. This would, for instance, allow to derive flow facts that are only valid with respect to the current subgraph under consideration.

A similar approach is proposed by Lokuciejewski et al. [29]. They combine abstract interpretation, polytope models, and program slicing to derive precise loop bounds. The approach again does not consider the *relevance* of the respective loops under analysis with regard to the final WCET.

Infeasible-Path Elimination

Bang and Kim [36], similar to our technique, propose an iterative approach to refine the attainable WCET using standard IPET. The basic idea is to perform a regular WCET analysis run. The resulting WCEP is subsequently checked for feasibility and, in the case of an infeasible WCEP, additional constraints are added to the IPET problem to exclude the path. This process is repeated until a feasible path is encountered. It is important to note that the presented feasibility checks are conservative and only consider individual basic blocks and pairs or blocks, but not the entire path. The major problem of this approach is that the refinement is based on individual paths through the program, whose number is potentially exponential in the number of conditional branches in the program. The additional constraints are, furthermore, only applied during the final

IPET run. Contrary to our approach, the technique thus cannot improve the precision of previous analysis phases, such as the cache- or pipeline analysis.

Zwirchmayer et al. [7] propose a related scheme called WCET Squeezing. The authors iteratively check the feasibility of the current WCEP using symbolic execution and exclude paths found to be infeasible from the IPET problem. In contrast to Bang and Kim, this technique considers the entire path and may thus potentially derive more complex constraints. The technique similarly does not allow to improve the precision of other analysis phases than the final IPET. To its advantage, WCET squeezing is an *anytime* algorithm, i.e., when interrupted at any time, a possible up-to-then achieved refinement is sound. Our current graph pruning algorithm does not have the anytime property, in fact it needs to run until the WCET bound from a pruned subgraph has been proved valid.

Abstract Interpretation

We evaluated graph pruning using AbsInt's aiT WCET analyzer, which makes use of abstract interpretation for its value analysis. While in this specific setting, the precision of value analysis benefits from the smaller graphs that we provide through pruning, context-sensitive abstract interpretation techniques [91, 38] themselves, in fact, share the same goal with us. They are used for the automatic computation of control-flow bounds (i.e. loop bounds or flow facts) that ultimately aim to tighten the WCET bound. The challenge for abstract interpretation is to cope with an overwhelming combination of program-, pipeline-, and cache states through safe approximation of values withing their domain and the merging of related hardware states. Efficient widening (and narrowing) operations are essential for analysis precision. [14]

Chapter 6

Closing

We are optimistic that today's state-of-the-art static WCET analysis will be able to keep up with the requirements of hard real-time applications in the foreseeable future. In this thesis, we have described ways to adapt WCET analysis to meet this goal.

We have shown how to efficiently analyze the worst-case behavior of a stack cache. Implemented in a real-time system, the stack cache benefits cache predictability and in combination with a tailor-made analysis has the potential to lower the WCET bound for real-time programs. We exhaustively investigated the behavior of *reserve* (spill) and *ensure* (fill) operations of the stack cache. Ensure analysis has proven to be a local problem and can be solved with minimal computational overhead. In general, our techniques combine intra-procedural data-flow analysis with longest (as well as shortest) path search on the inter-procedural call graph. By propagating bounds induced by stack cache size, we can avoid a costly context-sensitive data-flow analysis at program scope. To find the maximum remaining stack depth starting from arbitrary points in the call graph, we augment the graph and enable path search on the weighted tail. We also introduced the SCA graph, which fully models the context-sensitive spill cost of the stack cache and is suitable for integration in IPET-based worst-case timing analysis. Opportunities for pruning the graph with and without loss of analysis precision have been presented. Our accurate analysis technique scales to large analysis problems. It makes use of data-flow analysis and an adapted path search method modeled as an ILP, in an intertwined fashion, which has not been described before. Its applicability extends to caching concepts related to a stack cache, such as the sliding register window of the SPARC architecture.

In Chapter 4 we presented *criticality*, a novel metric for real-time programs. Computing *criticality* for program extends the view beyond that of a single worst-case path, to that of a more complete WCET profile. To the best of our knowledge, this approach is completely new in the field of static timing analysis. We have described graph-theoretic foundations and algorithms related to its computation, which exploit dominance properties and are applicable beyond the scope of *criticality*. An example for this, is the formal definition of invariant code. Our evaluation has shown that many of the WCET programs we use, either consist solely of invariant code, or contain an overwhelming amount of highly critical code. It is too early though, to conclude whether this is a prevalent property of real-time programs. The more likely cause, in our opinion, is that the effect is due to the limited scope of WCET benchmark programs available today.

Our graph pruning approach in Chapter 5 aims to improve WCET analysis independent of any special hardware features or program restrictions. Facing large analysis problems, WCET tools have to continuously restrict problem size, in order to meet space as well as time constraints and maintain feasibility. This results in a loss of precision: an overwhelming combination of program-, pipeline-, and cache states may make simplification necessary, at any stage of the analysis. To mitigate WCET overestimation, we propose to perform analysis on a reduced but safe subset of the program graph. Program slicing on the source code level has before been proposed as a method to reduce analysis overhead. In comparison, we introduce a form of slicing that can be applied to a low-level program representation. For our graph pruning method, we demonstrated that a criticality-based WCET profile is a suitable first guide for searching such reduced graphs.

Directions for Future Work

Fighting Overestimation All three major contributions of this thesis —*criticality* at least in part— can be viewed as pursuits to counteract the central problem of overestimating WCET bounds during static analysis. Graph pruning is our blanket approach for this problem, and while we were able to gain analysis precision by guiding the algorithm with a criticality profile, this represents only one possible way to perform pruning. Exploiting problem structure has proven to be the key to many optimization problems in the past. We believe the structure of a program's CFG and its properties specifically with regard to WCET analysis lend themselves to further improving analysis. Building a hardware platform and its compiler with the requirements of an analysis tool in mind, opens up even more possibilities in the "fight against overestimation".

WCET Analysis Tools Our algorithms often perform an analysis in an iterative manner, which would enable the reuse of analysis information up to a certain degree. For instance, the computational overhead of criticality analysis could be reduced by improving the way the ILP problems are solved for related IPET problems. A more general undertaking would be, to prevent WCET tools at different analysis stages, from throwing away intermediate results that can be reused.

From our experience as users of WCET analysis, we found that tools should make a better effort to adapt to the properties (e.g, size and complexity) of the program under analysis.

WCET Benchmarks Ideally, the programs which make up a benchmark suite are chosen to be representative for the types of applications that exist within the domain that the benchmark suite targets. At this point in time, this is unfortunately not the case in the field of WCET analysis. Besides the lack of benchmarks that represent real-world applications, opposite to micro benchmarks, there is also a lack of meta-data and documentation (e.g. consistent bounds from input data, analysis configuration), which is required to reach some degree of reproducibility. Not to belittle the previous work that has been done in this direction, as part of the WCET Tool Challenges, an ongoing community effort is needed to improve this situation. We look forward to contributing to this effort.

Bibliography

[1] Florian Brandner and Alexander Jordan. "Refinement of Worst-Case Execution Time Bounds by Graph Pruning". *(under submission)* (cited on pages 6, 81).

[2] Alexander Jordan. "Evaluating and Estimating the WCET Criticality Metric". In: *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*. ODES '14. *(accepted for publication)*. ACM, 2014. DOI: `10.1145/2568326.2568331` (cited on pages 6, 43).

[3] Yooseong Kim et al. "WCET-Aware Dynamic Code Management on Scratchpads for Software-Managed Multicores". In: *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS)*. Berlin, Germany, 2014 (cited on page 41).

[4] Sahar Abbaspour, Florian Brandner, and Martin Schoeberl. "A Time-predictable Stack Cache". In: *Proceedings of the Workshop on Software Technologies for Embedded and Ubiquitous Systems*. SEUS '13. 2013 (cited on pages 18, 39, 40).

[5] Florian Brandner, Stefan Hepp, and Alexander Jordan. "Criticality: static profiling for real-time programs". In: *Real-Time Systems* (Oct. 2013). *(published online, pending print)*. DOI: `10.1007/s11241-013-9196-y` (cited on pages 6, 43, 61).

[6] Alexander Jordan, Florian Brandner, and Martin Schoeberl. "Static Analysis of Worst-case Stack Cache Behavior". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. RTNS '13. ACM Press, 2013, pages 55–64. DOI: `10.1145/2516821.2516828` (cited on pages 6, 17).

[7] Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. "WCET squeezing". In: *Proceedings of the 21st International conference on Real-Time Networks and Systems - RTNS '13*. ACM Press, 2013, page 161. DOI: `10.1145/2516821.2516847` (cited on page 97).

[8] Martin Schoeberl, Benedikt Huber, and Wolfgang Puffitsch. "Data Cache Organization for Accurate Timing Analysis". In: *Real-Time Systems* 49.1 (Jan. 2013), pages 1–28 (cited on page 17).

[9] Florian Brandner, Stefan Hepp, and Alexander Jordan. "Static profiling of the worst-case in real-time programs". In: *Proceedings of the International Conference on Real-Time and Network Systems*. RTNS '12. ACM Press, 2012, pages 101–110 (cited on pages 6, 43).

[10] Heiko Falk, Peter Marwedel, and Paul Lokuciejewski. "Reconciling Compilation and Timing Analysis". In: edited by Samarjit Chakraborty and Jörg Eberspächer. Springer, Mar. 2012. Chapter 7, pages 145–170 (cited on page 77).

[11] Jorge Garrido, Daniel Brosnan, and JA de la Puente. "Analysis of WCET in an experimental satellite software development." In: *12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012)*. Wcet. 2012, pages 81–90 (cited on page 76).

[12] Reinhard von Hanxleden et al. *The WCET Tool Challenge 2011*. Technical report 1215. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Oct. 2012 (cited on page 15).

[13] Quentin Colombet, Florian Brandner, and Alain Darte. "Studying optimal spilling in the light of SSA". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '11. ACM, 2011, pages 25–34 (cited on page 75).

[14] Agostino Cortesi and Matteo Zanioli. "Widening and narrowing operators for abstract interpretation". In: *Computer Languages, Systems &Structures* 37.1 (2011), pages 24–42 (cited on page 97).

[15] Martin Schoeberl et al. "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach". In: *Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. PPES '11. 2011, pages 11–20 (cited on page 12).

[16] Christoph Cullmann et al. "Predictability Considerations in the Design of Multi-Core Embedded Systems". In: *Ingénieurs de l'Automobile* 807 (Sept. 2010), pages 36–42 (cited on page 17).

[17] Jan Gustafsson et al. "The Mälardalen WCET Benchmarks: Past, Present And Future." In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Wcet. OCG, 2010, pages 136–146 (cited on pages 16, 72).

[18] Raimund Kirner, Peter Puschner, and Adrian Prantl. "Transforming Flow Information during Code Optimization for Timing Analysis". In: *Real-Time Systems* (2010). DOI: `http://dx.doi.org/10.1007/s11241-010-9091-8` (cited on page 77).

[19] Adrian Prantl. "High-level Compiler Support for Timing Analysis". PhD thesis. Technische Universität Wien, 2010 (cited on pages 3, 77).

[20] Martin Schoeberl et al. "Worst-case execution time analysis for a Java processor". In: *Software: Practice and Experience* 40.6 (May 2010), pages 507–542. DOI: `10.1002/spe.968` (cited on page 3).

[21] Ingmar Jendrik Stein. "ILP-based Path Analysis on Abstract Pipeline State Graphs". PhD thesis. Universität des Saarlandes, 2010 (cited on pages 89, 91).

[22] Tidorum Ltd. *BoundT Time and Stack Analyzer - Application Note SPARC/ERC32 V7, V8, V8E*. Technical report TR-AN-SPARC-001, Version 7. Tidorum Ltd., 2010 (cited on page 40).

[23] Thomas H Cormen et al. *Introduction to Algorithms*. 3rd. MIT Press, 2009 (cited on pages 4, 23, 48–50).

[24] Daniel L Dvorak and Daniel L Dvorak (editor). *NASA Study on Flight Software Complexity*. Technical report. NASA Office of Chief Engineer, 2009 (cited on page 4).

[25] Lovi Gauthier and Tohru Ishihara. "Optimal stack frame placement and transfer for energy reduction targeting embedded processors with scratch-pad memories". In: *2009 IEEE/ACM/IFIP 7th Workshop on Embedded Systems for Real-Time Multimedia*. Ieee, Oct. 2009, pages 116–125. DOI: `10.1109/ESTMED.2009.5336819` (cited on page 41).

[26] Arun Kannan et al. "A software solution for dynamic stack management on scratch pad memory". In: *2009 Asia and South Pacific Design Automation Conference*. Ieee, Jan. 2009, pages 612–617. DOI: `10.1109/ASPDAC.2009.4796548` (cited on page 41).

[27] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. 1st. CRC Press, Inc., 2009 (cited on page 26).

[28] Paul Lokuciejewski, Fatih Gedikli, and Peter Marwedel. "Accelerating WCET-driven Optimizations by the Invariant Path Paradigm - A Case Study of Loop Unswitching". In: *Proc. of the Workshop on Software &Compilers for Embedded Systems (SCOPES '09)*. 2009, pages 11–20 (cited on page 47).

[29] Paul Lokuciejewski et al. "A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '09. IEEE, 2009, pages 136–146. DOI: `10.1109/CGO.2009.17` (cited on pages 77, 96).

[30] Lili Tan. "The worst-case execution time tool challenge 2006". In: *International Journal on Software Tools for Technology Transfer (STTT)* 11.2 (Jan. 2009), pages 133–152. DOI: `10.1007/s10009-008-0095-9` (cited on page 15).

[31] Niklas Holsti et al. "WCET Tool Challenge 2008: Report". In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*. Prague, Czech Republic: Österreichische Computer Gesellschaft, July 2008, pages 149–171 (cited on page 15).

[32] Ben Lickly et al. "Predictable programming on a precision timed architecture". In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems - CASES '08* (2008), page 137. DOI: `10.1145/1450095.1450117` (cited on page 41).

[33] Adrian Prantl, Markus Schordan, and Jens Knoop. "TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis". In: *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*. Prague, Czech Republic: Österreichische Computer Gesellschaft, 2008, pages 141–148 (cited on page 3).

[34] Reinhard Wilhelm et al. "The Worst-Case Execution Time Problem – Overview of Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pages 1–53. DOI: `10.1145/1347375.1347389` (cited on page 2).

[35] Alfred V Aho et al. *Compilers: principles, techniques and tools*. Second. Pearson Education, 2007 (cited on page 26).

[36] Ho Jung Bang, Tai Hyo Kim, and Sung Deok Cha. "An Iterative Refinement Framework for Tighter Worst-Case Execution Time Calculation". In: *Proceedings of the Symposium on Object and Component-Oriented Real-Time Distributed Computing*. ISORC '07. IEEE, 2007, pages 365–372. DOI: `10.1109/ISORC.2007.19` (cited on page 96).

[37] Michael Dunlavey. "Performance tuning with instruction-level cost derived from call-stack sampling". In: *ACM SIGPLAN Notices* 42.8 (2007), pages 4–8. DOI: `10.1145/1294297.1294298` (cited on page 76).

[38] Andreas Ermedahl et al. "Loop Bound Analysis based on a Combination of Program Slicing, Abstract Interpretation, and Invariant Analysis". In: *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*. Edited by Christine Rochange. Volume 07002. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007 (cited on page 97).

[39] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. "Static memory and timing analysis of embedded systems code". In: *Proceedings of Symposium on Verification and Validation of Software Systems*. VVSS '07. Eindhoven University of Technology, 2007, pages 153–163 (cited on page 40).

[40] Xianfeng Li et al. "Chronos: A Timing Analyzer for Embedded Software". In: *Science of Computer Programming* 69.1-3 (2007), pages 56–67 (cited on page 3).

[41] Hugues Cassé and Pascal Sainrat. "OTAWA, a framework for experimenting WCET computations". In: *European Congress on Embedded Real-Time Software*. Toulouse: Société de l'Electricité, de l'Electronique et des Technologies de l'Information et de la Communication (SEE), Jan. 2006 (cited on page 3).

[42] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. "Design of a WCET-Aware C Compiler". In: *2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. Edited by Frank Mueller. Volume 06902. Dagstuhl Seminar Proceedings. IEEE, Oct. 2006, pages 121–126. DOI: `10.1109/ESTMED.2006.321284` (cited on pages 3, 77).

[43] François Lemieux and Martin Salois. "Visualization techniques for program comprehension". In: *Proceedings of the 2006 conference on New Trends in Software Methodologies, Tools and Techniques*. February. 2006 (cited on page 77).

[44] Christopher Lupo and Kent D KD K.D. Wilken. "Post Register Allocation Spill Code Optimization". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '06. Ieee, 2006, pages 245–255. DOI: `10.1109/CGO.2006.28` (cited on page 75).

[45] Fadia Nemer et al. "PapaBench: A Free Real-Time Benchmark". In: *Proc. of the Workshop on Worst-Case Execution Time Analysis*. OCG, 2006, pages 63–68 (cited on page 16).

[46] Christer Sandberg et al. "Faster WCET Flow Analysis by Program Slicing". In: *Proceedings of the conference on Language, Compilers, and Tool Support for Embedded Systems*. LCTES '06. New York, New York, USA: ACM, 2006, pages 103–112. DOI: `10.1145/1134650.1134666` (cited on page 96).

[47] Edward R Tufte. *Beautiful evidence*. Graphics Press LLC, 2006, 213 p (cited on page 61).

[48] Wankang Zhao et al. "Improving WCET by applying worst-case path optimizations". In: *Real-Time Systems* 34.2 (June 2006), pages 129–152. DOI: `10.1007/s11241-006-8643-4` (cited on page 4).

[49] Joseph A Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing: a VLIW approach to architecture, compilers and tools*. Morgan Kaufmann Publishers, 2005, page 671 (cited on page 1).

[50] Kevin Hammond et al. "The Embounded project (project start paper)". In: *Trends in Functional Programming*. Edited by Marko C J D van Eekelen. Volume 6. Trends in Functional Programming. Intellect, 2005, pages 195–210 (cited on page 3).

[51] Ingomar Wenzel et al. "Automatic Timing Model Generation by CFG Partitioning and Model Checking". In: *DATE*. IEEE Computer Society, 2005, pages 606–611 (cited on page 3).

[52] Christian Wimmer and Hanspeter Mössenböck. "Optimized interval splitting in a linear scan register allocator". In: *Proceedings of the International Conference on Virtual Execution Environments*. VEE '05. ACM, 2005, pages 132–141 (cited on page 75).

[53] Susanna Byhlin. "Evaluation of Static Time Analysis for Volcano Communications Technologies AB". PhD thesis. Mälardalen University, 2004 (cited on page 2).

[54] Joseph A Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, Dec. 2004 (cited on page 1).

[55] Loukas Georgiadis and Robert E Tarjan. "Finding dominators revisited: extended abstract". In: *Proceedings of the Symposium on Discrete Algorithms*. SODA '04. Society for Industrial and Applied Mathematics, 2004, pages 869–878 (cited on page 53).

[56] Guillem Bernat, Antoine Colin, and Stefan M Petters. “pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems”. In: *3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*. 2003, pages 21–38 (cited on page 3).

[57] Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. “Convenient User Annotations for a WCET Tool”. In: *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*. Porto, Portugal, July 2003, pages 17–20 (cited on page 3).

[58] R. Heckmann et al. “The influence of processor architecture on the design and the results of WCET tools”. In: *Proceedings of the IEEE* 91.7 (July 2003), pages 1038–1054. DOI: `10.1109/JPROC.2003.814618` (cited on page 4).

[59] Fabian Wolf, Jan Staschulat, and Rolf Ernst. “Associative caches in formal software timing analysis”. In: *DAC '02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, 2002, pages 622–627. DOI: `http://doi.acm.org/10.1145/513918.514076` (cited on page 3).

[60] Antoine Colin and Isabelle Puaut. “A Modular and Retargetable Framework for Tree-based WCET Analysis”. In: *Proc. 13th Euromicro Conference on Real-Time Systems*. Technical University of Delft. Delft, Netherland, June 2001, pages 37–44 (cited on page 3).

[61] John Ellson et al. “Graphviz - Open Source Graph Drawing Tools”. In: *Graph Drawing* 2265 (2001), pages 483–484. DOI: `10.1007/3-540-45848-4` (cited on page 61).

[62] Matthew R Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Proceedings of the Workshop on Workload Characterization*. 2001 (cited on page 38).

[63] Raimund Kirner. *User's Manual – WCET-Analysis Framework based on wcetC*. 0.0.3. Vienna University of Technology. Vienna, Austria, July 2001 (cited on page 3).

[64] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. “Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects”. In: *Proc. of the Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, 2001, pages 132–140 (cited on pages 4, 49).

[65] Jakob Engblom and Andreas Ermedahl. “Modeling complex flows for worst-case execution time analysis”. In: *Proceedings 21st IEEE Real-Time Systems Symposium*. RTSS '00. IEEE, 2000, pages 163–174. DOI: `10.1109/REAL.2000.896006` (cited on page 3).

[66] Christopher A Healy et al. “Supporting Timing Analysis by Automatic Bounding of Loop Iterations”. In: *Real-Time Systems* 18.2/3 (2000), pages 129–156 (cited on page 3).

[67] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. “Using a Worst-Case Execution Time Tool for Real-Time Verification of the DEBIE Software”. In: *Proc. of the Data Systems in Aerospace Conference*. ESA, 2000, page 307 (cited on page 16).

[68] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. "Worst-Case Execution Time Analysis for Digital Signal Processors". In: *European Signal Processing Conference 2000 (EUSIPCO 2000)*. Space Systems Finland Ltd. 2000 (cited on page 3).

[69] Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. "Fast and Precise WCET Prediction by Separated Cache and Path Analyses". In: *Real-Time Systems* 18.2/3 (2000), pages 157–179 (cited on pages 12, 84).

[70] Hiralal Agrawal. "Efficient coverage testing using global dominator graphs". In: *ACM SIGSOFT Software Engineering Notes* 24 (1999), pages 11–20. DOI: `10.1145/381788.316166` (cited on page 75).

[71] Christian Ferdinand and Reinhard Wilhelm. "Efficient and Precise Cache Behavior Prediction for Real-Time Systems". In: *Real-Time Systems* 17.2-3 (1999), pages 131–181. DOI: `10.1023/A:1008186323068` (cited on page 40).

[72] Thomas Lundqvist and Per Stenström. "An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution". In: *Real-Time Systems* 17.2-3 (1999), pages 183–207 (cited on page 3).

[73] John T. Stasko et al. *Software visualization : Programming as a multimedia experience*. MIT Press, 1998, 562 s (cited on page 76).

[74] Peter P Puschner and Anton V Schedl. "Computing Maximum Task Execution Times - A Graph-Based Approach". In: *Real-Time Systems* 13.1 (July 1997), pages 67–91 (cited on pages 3, 51, 84).

[75] Randall T White et al. "Timing Analysis for Data Caches and Set-Associative Caches". In: *Proceedings of the Real-Time Technology and Applications Symposium*. RTAS '97. IEEE, 1997, pages 192–203 (cited on page 40).

[76] Thomas Ball and Stephen G. Eick. "Software visualization in the large". In: *Computer* 29.4 (Apr. 1996), pages 33–43. DOI: `10.1109/2.488299` (cited on page 76).

[77] Thomas Ball and James R Larus. "Efficient Path Profiling". In: *IEEE/ACM Int'l Symp. on Microarchitecture (MICRO-29)* (Dec. 1996) (cited on page 76).

[78] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software using Implicit Path Enumeration". In: *Proceedings of the Design Automation Conference*. DAC '95. ACM, 1995, pages 456–461. DOI: `10.1145/217474.217570` (cited on pages 3, 51, 84).

[79] Georg Sander. "Graph layout through the VCG tool". In: *Graph Drawing* (1995). DOI: `10.1007/3-540-58950-3_371` (cited on page 77).

[80] Hiralal Agrawal. "Dominators, super blocks, and program coverage". In: *Proc. POPL*. 1994, pages 25–34. DOI: `10.1145/174675.175935` (cited on pages 51, 75).

[81] Roderick Chapman, Alan Burns, and Andy Wellings. "Integrated Program Proof and Worst-case Timing Analysis of SPARK Ada". In: *Proc. ACM Workshop on Language, Compiler and Tool Support for Real-time Systems*. June 1994, K1–K11 (cited on page 3).

[82] Michael Dunlavey. *Building Better Applications: A Theory of Efficient Software Development*. 1st. New York, NY, USA: John Wiley and Sons, Inc., 1994 (cited on page 76).

[83] Chang Yun Park. "Predicting Deterministic Execution Times of Real-Time Programs". PhD thesis. Seattle, USA: University of Washington, 1992 (cited on page 3).

[84] Alexander Vrchoticky. *Modula/R – Language Definition*. Technical report 02/1992. Treitlstr. 1-3/182-1, 1040 Vienna, Austria: Technische Universität Wien, Institut für Technische Informatik, 1992 (cited on page 3).

[85] Alan C Shaw. "Reasoning about time in higher level language software". In: *IEEE Transactions on Software Engineering* 15.7 (July 1989), pages 875–889 (cited on page 3).

[86] Moyer Chen. *A Timing Analysis Language – (TAL)*. Dept. of Computer Science, University of Texas. Austin, TX, USA, 1987 (cited on page 3).

[87] Eugene Klingerman and Alexander D Stoyenko. "Real-Time Euclid: A Language for Reliable Real-Time Systems". In: *IEEE Transactions on Software Engineering* 12.9 (1986), pages 941–989 (cited on page 3).

[88] Susan L Graham, Peter B Kessler, and Marshall K McKusick. "gprof: a call graph execution profiler (with retrospective)". In: *Best of PLDI*. Edited by Kathryn S McKinley. ACM, 1982, pages 49–57 (cited on page 76).

[89] Mark Weiser. "Program Slicing". In: *Proceedings of the Conference on Software Engineering*. ICSE '81. IEEE, 1981, pages 439–449 (cited on page 96).

[90] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Trans. Program. Lang. Syst.* 1.1 (1979), pages 121–141 (cited on pages 53, 54, 59).

[91] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the Symposium on Principles of Programming Languages*. POPL '77. ACM, 1977, pages 238–252. DOI: `10.1145/512950.512973` (cited on pages 84, 97).

[92] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. Volume 1. SWAT '71 2. IEEE, Oct. 1971, pages 114–121. DOI: `10.1109/SWAT.1971.10` (cited on page 54).

[93] Clark Evans et al. *YAML Ain't Markup Language*. 2014. URL: `http://yaml.org/` (Retrieved 01/30/2014) (cited on page 13).

[94] Freescale Semiconductor. *e200z6 PowerPC Core Reference Manual Rev. 0 06/04*. 2014. URL: `http://www.freescale.com/` (Retrieved 01/30/2014) (cited on page 12).

[95] Freescale Semiconductor. *MPC5553/5554 Microcontroller Reference Manual Rev. 5.1 03/12*. 2014. URL: `http://www.freescale.com/` (Retrieved 01/30/2014) (cited on page 12).

[96] FSF – Free Software Foundation. *GCC, the GNU Compiler Collection*. 2014. URL: `http://gcc.gnu.org/` (Retrieved 01/30/2014) (cited on page 4).

[97] The LLVM Project. *The LLVM Compiler Infrastructure*. 2014. URL: `http://llvm.org/` (Retrieved 01/30/2014) (cited on page 4).

[98] Mälardalen University. *Benchmarks for WCET Analysis*. 2014. URL: `\url{http://www.mrtc.mdh.se/projects/wcet/benchmarks.html}` (Retrieved 01/30/2014) (cited on page 16).

[99] COIN-OR The Computational Infrastructure for Operations Research. 2014. URL: `http://www.coin-or.org/` (Retrieved 01/30/2014) (cited on page 62).

[100] Programming paradigms group. Firm. 2014. URL: `http://pp.ipd.kit.edu/firm/` (Retrieved 01/30/2014) (cited on page 77).

[101] The T-CREST Project. *Patmos A Time-predictable Processor for Real-Time Systems*. 2014. URL: `http://patmos.compute.dtu.dk/` (Retrieved 01/30/2014) (cited on pages 12, 16).

[102] The T-CREST Project. *T-CREST Github repository*. 2014. URL: `http://github.com/t-crest/` (Retrieved 01/30/2014) (cited on pages 3, 13).

[103] The T-CREST Project. *Time-predictable Multi-Core Architecture for Embedded Systems*. 2014. URL: `http://www.t-crest.org/` (Retrieved 01/30/2014) (cited on pages 13, 16).

Curriculum Vitae

PERSONAL DATA

Name	Alexander Jordan
Date of birth	26.02.1982
Place of birth	Vienna
Nationality	Austrian
Address	Badgasse 22/12, 1090 Wien
email	`ajordan@complang.tuwien.ac.at`

EMPLOYMENT HISTORY

Technical University of Denmark (DTU), Embedded Systems Engineering Section, Department of Informatics and Mathematical Modelling

Research Assistant December 2013 to present

Project: Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST)

Vienna University of Technology, Institute of Computer Languages, Compilers and Languages Group

Research Assistant October 2009 to December 2013

Project: Optimal Code Generation for Explicitly Parallel Processors (EPICOpt)

STMicroelectronics, Compilation Expertise Center, Grenoble.

Internship July to October 2012

StreamUnlimited Engineering GmbH, Vienna, Austria

Software Engineer, Software Architect April 2005 to March 2010

Audio/Video Innovation Center, Philips Electronics, Vienna, Austria

Software Tester, Integration Engineer July 2003 to March 2005

EDUCATION

Vienna University of Technology

Doctoral Studies in Computer Science October 2009 to present

Advised by Andreas Krall

MSc in Computer Science 2005-2007

Field: Software Engineering & Internet Computing with specialization in compilation and virtual machines
Graduated with distinction

BSc in Computer Science 2001-2005

Field: Software Engineering & Internet Computing

SUMMER SCHOOL ACACES 2010: Sixth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems, Terrassa (Barcelona), Spain, July 2010.

PUBLICATIONS

- Florian Brandner and Alexander Jordan. "Refinement of Worst-Case Execution Time Bounds by Graph Pruning". *(under submission)*.
- Alexander Jordan. "Evaluating and Estimating the WCET Criticality Metric". In: *Proceedings of the 11th Workshop on Optimizations for DSP and Embedded Systems*. ODES '14. *(accepted for publication)*. ACM, 2014.
- Florian Brandner, Stefan Hepp, and Alexander Jordan. "Criticality: static profiling for real-time programs". In: *Real-Time Systems* (Oct. 2013). *(published online, pending print)*.
- Alexander Jordan, Florian Brandner, and Martin Schoeberl. "Static Analysis of Worst-case Stack Cache Behavior". In: *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. RTNS '13. ACM Press, 2013, pages 55–64.
- Alexander Jordan and Christophe Guillon. "Lightweight Scheduling in a Single-Pass JavaScript JIT Compiler". In: *International Workshop on Dynamic Compilation Everywhere, Berlin, January 2013*. 2013.
- Alexander Jordan, Nikolai Kim, and Andreas Krall. "IR-level versus machine-level if-conversion for predicated architectures". In: *Proceedings of the 10th Workshop on Optimizations for DSP and Embedded Systems - ODES '13*. ACM Press, 2013, page 3.
- Florian Brandner, Stefan Hepp, and Alexander Jordan. "Static profiling of the worst-case in real-time programs". In: *Proceedings of the International Conference on Real-Time and Network Systems*. RTNS '12. ACM Press, 2012, pages 101–110.