Tight Integration of Cache, Path and Task-interference Modeling for the Analysis of Hard Real-time Systems

Dissertation

zur Erlangung des Grades eines

Doktors der Ingenieurwissenschaften

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

Jan C. Kleinsorge

Dortmund

2015

Ort:	TU Dortmund
Fakultät:	Informatik
Tag der mündlichen Prüfung:	28. Oktober 2015
Dekan:	Prof. Dr. Gernot Fink
Gutachter:	Prof. Dr. Peter Marwedel
	Prof. Dr. Björn Lisper

Abstract

Traditional timing analysis for hard real-time systems is a two-step approach consisting of isolated per-task timing analysis and subsequent scheduling analysis which is conceptually entirely separated and is based only on execution time bounds of whole tasks. Today this model is outdated as it relies on technical assumptions that are not feasible on modern processor architectures any longer. The key limiting factor in this traditional model is the interfacing from micro-architectural analysis of individual tasks to scheduling analysis — in particular path analysis as the binding step between the two is a major obstacle.

In this thesis, we contribute to traditional techniques that overcome this problem by means of bypassing path analysis entirely, and propose a general path analysis and several derivatives to support improved interfacing. Specifically, we discuss, on the basis of a precise cache analysis, how existing metrics to bound cache-related preemption delay (CRPD) can be derived from cache representation without separate analyses, and suggest optimizations to further reduce analysis complexity and to increase accuracy. In addition, we propose two new estimation methods for CRPD based on the explicit elimination of infeasible task interference scenarios. The first one is conventional in that path analysis is ignored, the second one specifically relies on it. We formally define a general path analysis framework in accordance to the principles of program analysis — as opposed to most existing approaches that differ conceptually and therefore either increase complexity or entail inherent loss of information — and propose solutions for several problems specific to timing analysis in this context. First, we suggest new and efficient methods for loop identification. Based on this, we show how path analysis itself is applied to the traditional problem of per-task worst-case execution time bounds, define its generalization to sub-tasks, discuss several optimizations and present an efficient reference algorithm. We further propose analyses to solve related problems in this domain, such as the estimation of bounds on best-case execution times, latest execution times, maximum blocking times and execution frequencies. Finally, we then demonstrate the utility of this additional information in scheduling analysis by proposing a new CRPD bound.

Zusammenfassung

Traditionelle Zeitanalyse von harten Echtzeitsystemen ist ein typischerweise zweischrittiges Verfahren bestehend aus der eigentlichen Analyse einzelner isolierter Tasks, sowie einer darauf folgenden Scheduling-Analyse, welche konzeptionell sehr verschieden ist und lediglich auf der Abschätzung der Gesamtlaufzeit einzelner Tasks beruht. Nach heutigen Maßstäben ist dieses Modell veraltet, da diesem Analyseprinzip längst überholte technische Annahmen zu Grunde liegen. Die zentrale Beschränkung hierbei bildet gerade die Schnittstelle zwischen Mikroarchitekturanalyse einzelner Tasks und der Scheduling-Analyse. Insbesondere die sogenannte Pfadanalyse, als Bindeglied beider Phasen, ist hierbei von zentraler Bedeutung.

Mit dieser Arbeit tragen wir zum einen dazu bei, Analyseverfahren, welche die Pfadanalyse vollständig umgehen, zu verbessern. Zum anderen stellen wir eine neue allgemeine Pfadanalyse, sowie mehrere Varianten vor, die dazu beitragen die Schnittstelle erheblich zu verbessern. Genauer diskutieren wir, wie auf Grundlage einer präzisen Cache-Analyse bereits existierende Metriken zur Abschätzung von Cache-related Preemption Delay (CRPD) ohne weitere separate Analysen abgeleitet werden können, und wir schlagen spezifische Optimierungen vor, die sowohl die Analysekomplexität senken, sowie die Genauigkeit erhöhen. Zusätzlich schlagen wir zwei neuartige Methoden zur Abschätzung von CRPD vor, basierend auf dem expliziten Ausschluss unmöglicher Präemptionsinterferenzen. Das Erste ist konventionell insofern, als dass die Pfadanalyse umgangen wird. Das Zweite hingegen hängt explizit von ihr ab. Wir definieren formal ein allgemeines Pfadanalyse-framework, das den Prinzipien der klassischen Programmanalyse entspricht. Existierende Lösungen weichen davon konzeptionell erheblich ab, zeigen geringe Performanz oder sind inhärent ungenau. In diesem Zusammenhang besprechen wir weiter wichtige Probleme speziell im Kontext der Zeitanalyse. Wir zeigen neue Möglichkeiten zur Identifikation von Kontrollflussschleifen auf, zeigen wie darauf aufbauend das klassische Problem der Worst-Case-Zeitanalyse ganzer Tasks, sowie von Teilmengen gelöst werden kann — zuzüglich zahlreicher Optimierungen und eines effizienten Referenzalgorithmus. Weiter zeigen wir wie Probleme wie Best-Case- und Latest-Ausführungszeit, sowie Maximale-Blockierzeit und Worst-Case-Ausführungsfrequenz gelöst werden können. Abschließend demonstrieren wir den Nutzen dieser Analyseergebnisse für die Scheduling-Analyse anhand einer neuen CRPD-Abschätzung.

Acknowledgements

This thesis would not have been possible without the support of many people and whom I owe a great deal of gratitude. First of all I would like to thank Prof. Dr. Peter Marwedel who gave me the opportunity to work in his group. Along with him, I would like to thank Prof. Dr. Heiko Falk. Both gave me the advice, the freedom and the time to pursue my research and have always been valuable sources of guidance and expertise.

I would never have found the stamina to finish this work if it had not been for my formidable colleagues who were always open for discussion and who have been a great source of inspiration over the years. I would like to specially thank Dr. Michael Engel for always having an open ear and amicable advice. I would also like to explicitly thank Dr. Timon Kelter with whom it was a pleasure to work with for all these years in our little two-men work group. It has been a constant in changing tides.

I will not provide a complete list of names at this point of all the people that shared this experience with me, be it colleagues or friends — often that is indistinguishable —, or my family. But be assured that if you are looking for your name right now, then you would be on it. Thank you for your time, your dedication, your patience, all the talk, all the fun, your friendship and your love.

Contents

1 Introduction **1**

1.1 Contribution . . . 3
1.2 Structure . . . 5
1.3 Contributing Publications . . . 5

2 Principles of Program Analysis **7**

2.1 Programs: Syntax, Semantics and Interpretation . . . 8
2.2 Trace Semantics . . . 8
2.3 Collecting Semantics . . . 10
2.4 Fixed Point Semantics . . . 12
2.5 Abstraction . . . 15
2.6 Convention and Practical Program Analysis . . . 18

3 Context **21**

3.1 Real-time Scheduling . . . 21
3.1.1 Basic Task Model . . . 22
3.1.2 Modes of Preemption . . . 25
3.1.3 Deadline Monotonic and Earliest Deadline First . . . 27
3.1.4 Schedulability . . . 28
3.1.5 Blocking and Synchronization . . . 30
3.2 Timing Analysis . . . 33
3.2.1 Practical Aspects . . . 34

4 Cache Analysis **39**

4.1 Computer Memories . . . 40
4.2 Processor Caches . . . 41
4.3 Cache Logic . . . 43
4.4 Static Cache Analysis . . . 44
4.4.1 LRU Cache Semantics . . . 45
4.4.2 Access Classification . . . 46
4.4.3 Abstraction . . . 47
4.5 Multitask Timing Analysis . . . 49
4.5.1 Costs of Preemption . . . 49
4.5.2 Cache-related Preemption Delay . . . 50
4.5.3 Bounding Cache-related Preemption Delay . . . 51
4.6 Synergetic Approach to CRPD Analysis . . . 58

4.6.1 Precise Cache Analysis 58
4.6.2 Computation of UCB, ECB and CBR 61
4.6.3 Restriction to Basic Block Boundaries 64
4.6.4 CRPD Bounds on Task Sets 66
4.7 Evaluation 72
4.8 Conclusion 77

5 Path Analysis 79

5.1 Fundamentals of Control Flow Analysis 81
5.1.1 Flows and Paths 81
5.1.2 Graph Structure 84
5.2 Path Problems in Timing Analysis 91
5.2.1 On Program Representation 91
5.2.2 On Control Flow Representation 93
5.2.3 On Path Analyses 98
5.3 A General Path Analysis 101
5.3.1 Motivation 101
5.3.2 Graph Structure and Loops 103
5.3.2.1 Related Work 104
5.3.2.2 Scopes 105
5.3.2.3 A General Algorithm for Precise Loop Detection 107
5.3.2.4 Handling Ambiguous Loop Nesting by Enumeration . . . 121
5.3.2.5 Handling Ambiguous Loop Nesting by Prenumbering . . 125
5.3.2.6 Conclusion 131
5.3.3 Computing Worst-Case Execution Time Bounds 131
5.3.3.1 Prerequisites 131
5.3.3.2 Computing WCET Bounds on a Single Scope 133
5.3.3.3 Computing WCET Bounds Globally 149
5.3.3.4 Computing WCET Bounds on Subgraphs 151
5.3.3.5 Practical Global Path Length Computation 153
5.3.3.6 Evaluation 159
5.3.3.7 Conclusion 163
5.3.4 Computing Best-case Execution Time Bounds 164
5.3.4.1 Prerequisites 164
5.3.4.2 Framework 165
5.3.4.3 Evaluation 169
5.3.4.4 Conclusion 171
5.3.5 Computing Latest Execution Time Bounds 171
5.3.5.1 Prerequisites 172
5.3.5.2 Framework 173
5.3.5.3 Evaluation 182
5.3.5.4 Conclusion 184

5.3.6 Computing Maximum Blocking Time Bounds . . . 184
5.3.6.1 Prerequisites . . . 186
5.3.6.2 Framework . . . 189
5.3.6.3 Evaluation . . . 202
5.3.6.4 Conclusion . . . 204
5.3.7 Computing Worst-Case Execution Frequencies . . . 204
5.3.7.1 Prerequisites . . . 205
5.3.7.2 Framework . . . 207
5.3.7.3 Evaluation . . . 213
5.3.7.4 Conclusion . . . 216
5.4 Remarks . . . 216
5.5 Conclusion . . . 218

6 Bounding Cache-related Preemption Delay 221
6.1 Improving Conventional CRPD Bounds . . . 221
6.1.1 Preliminaries . . . 222
6.1.2 A Review of Approaches . . . 223
6.1.3 A Refined Bound on CRPD . . . 230
6.2 Improving CRPD Estimation with Time Bounds . . . 237
6.3 Evaluation . . . 242
6.4 Conclusion . . . 245

7 Conclusion 247
7.1 Summary of Contributions . . . 247
7.2 Future Work . . . 249
7.3 Conclusion . . . 249

A Notations and Conventions 251
A.1 Mathematical Notation . . . 251
A.2 Pseudo-code Language . . . 253

B Reference Implementations of Basic Graph Algorithms 255
B.1 Breadth-first Search . . . 255
B.2 Maximum Flow . . . 256
B.3 Single-source Shortest Paths . . . 257
B.4 Topological Sort . . . 258
B.5 Single-source Shortest Paths on Directed Acyclic Graphs . . . 258
B.6 Depth-first Search . . . 258

C On Linear Programming 261

Bibliography 263

Index 279

Chapter 1

Introduction

Contents

1.1 Contribution **3**
1.2 Structure **5**
1.3 Contributing Publications **5**

In the history of digital computing systems, no generation of technology had an impact on the status quo of modern civilization as profoundly as *cyber-physical systems* [1]. Gradually and unobtrusively, *embedded computing systems* [2] with capabilities focused on interaction with the physical environment became ubiquitous in our everyday life. Today these systems can be found in just about anything from vehicles to medical implants. In a modern automobile for example, we are surrounded by systems controlling basic functions such as gasoline injection and transmission, safety-critical systems like airbags and driver assistance, and comfort features such as satellite navigation, air conditioning or entertainment systems. More than in any other digital computing domain are these systems in such direct interaction with, and their capabilities constrained by, their physical environment. Physical parameters such as *energy efficiency* — both in terms of operations per energy unit in mobile systems as well as in terms of heat emission in deeply embedded systems —, *resistance* to radiation or extreme temperatures or precise *timing* of operations are of significant concern. These *nonfunctional* properties add to the complexity of guaranteeing correctness of critical operations.

The legendary quote attributed to David Wheeler that "[a]ll problems in computer science can be solved by another level of indirection... Except for the problem of too many layers of indirection" is strikingly acute in the domain of embedded systems design where any layer that abstracts by means of indirection from the physical environment potentially jeopardizes provably correct operations. Every additional abstraction, be it in the form of hardware or software, potentially increases the complexity of causal chains, adding uncertainty about the behavior of functional as well as nonfunctional properties. This sets embedded computing apart from general purpose computing: holistic hardware/software co-design with a shallow abstraction hierarchy is the norm. In particular, multiple objectives such as efficiency or predictability must be balanced against throughput. Unfortunately, long gone are the times that computing performance

of simple system architectures could just be raised by increasing clock frequencies. First, memory technology fell short of keeping up to processor performance, followed by hitting physical boundaries to raising frequencies in general. Today, performance increments are achieved by features such as cache memory, pipelined, speculative or out-of-order execution, or the duplication of processing units. Some of these features are at odds with named nonfunctional properties, motivating their formal study to assess their applicability, devise formal analyses or give design recommendations according to the specific requirements of embedded systems. In this context it is important to recognize software as a level of indirection to cope with limited flexibility and high costs of hardware solutions which can not be avoided.

Due to their sensitivity to stimuli from physical environments, timeliness of operations in “real” physical time is an overarching concern in embedded systems. In particular in *hard real-time systems* the timing aspect blurs the border of nonfunctional to functional properties as correctness of computations becomes critically dependent on their time demand in addition to mere functional semantics [3]. Its formal study is referred to as *timing analysis*. Of particular relevance is *static timing analysis* which allows to derive provably correct best-case or worst-case timing estimates from mathematical models of hardware and software. The discovery and study of such models for existing and future systems is an established [4] but nonetheless active field of research [5].

A proven procedure for practical static timing analysis is the separation of analysis phases which roughly translate into discovery of potential execution paths (*control flow analysis*), component-wise timing analysis (*micro-architectural analysis*) and consolidation by selection of least or most time-demanding execution paths (*path analysis*). This is usually performed for each unit of functionality, referred to as *task*, in isolation. Since the components providing computing time as a resource to software tasks are usually shared, a *scheduling policy* defines global task orchestration. To determine global timing characteristics of a system, the phase of *scheduling analysis* consolidates per-task timings under such a given policy.

Not just the quality of analysis phases themselves but also their interfacing is a critical aspect of the overall process. Over time the interface between micro-architectural and scheduling analysis in particular became an increasingly severe bottleneck. Traditionally, all information obtained in phases preceding scheduling analysis is ultimately mapped onto a single scalar value to denote execution time per task as a parameter to this final phase. The assumption is that global timing characteristics solely depend on the shared resource of computation time, and scheduling analysis solely serves the purpose of estimating task interference on this resource. However, in modern architectures, task interference that affect global timing also — and not exclusively — occurs in cache memories and execution pipelines. The wider the gap between memory and processing performance, and the deeper and more sophisticated execution pipelines become, the more imprecise traditional scheduling analysis becomes. Today, we long passed the point where this simplification is tolerable.

1.1 Contribution

The problem of insufficient interfacing to scheduling analysis is known and understood [6]. In particular the inherent costs of task interference in cache memories have been subject to intense research. They are primary sources of imprecision and there exists a body of approaches that propose an additional interface by summarizing micro-architectural cache analysis results to tighten estimates of cache-related interference costs in scheduling analysis. Potential for optimization in this context can be exploited by either improvements in cache analysis itself, by improved interfacing or by improving cache-aware scheduling analysis. This thesis contributes to all three aspects as follows:

Cache Analysis and Cache-related Preemption Delay

On the basis of a precise cache analysis framework for set-associative caches, we show how to derive popular metrics for bounding cache interference — specifically cache-related preemption delay. We demonstrate that fast precise analysis for set-associative caches is indeed applicable nowadays, despite memory requirements, given a careful performance-conscious implementation. We show that the particular advantage is that from precise cache analysis results, metrics to bound interference can be directly and precisely derived without separate analyses. Further, we propose optimizations for instruction caches to significantly reduce overall memory requirements. In addition, we identify potential sources of pessimism for bounding task interference in set-associative caches and propose improvements.

Formal Discussion on Path Analysis

Path analysis is the central limiting factor in the interfacing of micro-architectural and scheduling analysis, as essentially all existing approaches in the context of timing analysis are limited to deriving simple per-task time bounds, which severely inhibits progress. As a historical consequence, in most approaches the mapping of cache interference to scheduling analysis completely bypasses path analysis, aggravating the problem of information loss. We formally discuss path analysis, give an overview of related issues, and show the conceptual and formal relation of existing approaches. In particular, we show the fundamental relations of approaches deemed conceptually different and we identify their specific limitations. Moreover, it becomes clear that existing approaches are not necessarily ideal fits to the general principles of program analysis in theoretical and technical terms.

Control Flow Reconstruction and Loop Identification

Path analysis is critically dependent on a precise representation of program structure. We contribute solutions to the problem of loop identification in particular by proposing a new general parametric and highly efficient algorithm that specifically addresses a key problem in the context of timing analysis: Semantics but not program structure is preserved from high-level to low-level representation of software during compilation, but path analysis critically depends on this information which must then be provided as parameters to the algorithm. Further, we accompany

the algorithm with two new methods to handle ambiguity in loop identification that cannot be tackled with the primary algorithm alone. We propose two efficient methods to either enumerate potential contexts to allow for safe but not necessarily precise path analysis, or to guide precise loop identification by annotation.

General Path Analysis

Our central contribution is the proposal of a new path analysis and a selection of several derivatives to approach various problems in timing analysis and scheduling analysis in general. We formally define a general and yet simple framework which perfectly fits principles of program analyses used in micro-architectural analysis, paving the way for more advanced and new applications. Initially, we motivate its construction along the use-case of traditional per-task worst-case execution time analysis. Beyond the base model, we propose several optimizations. We then further generalize beyond mere per-task analysis and show the application to arbitrary subgraphs and we show how to efficiently compute timing estimates from and to arbitrary program points. In addition, we also propose a highly efficient and carefully crafted reference algorithm. In all proposals, we carefully took the specific requirements of symbolic path analysis into account, formally as well as practically.

Derivatives of General Path Analysis

From the base model of path analysis which mainly serves the purpose of worst-case time estimates for traditional timing analysis, we derive several variants that either significantly improve upon existing approaches or provide entirely new solutions. We propose a framework for best-case analysis to compute lower time bounds. We propose a new notion of time bounds, to which we refer to as latest execution times, that specifically tightens estimates in fully preemptive schedules. We further propose an analysis for the efficient estimation of maximum blocking times which is the first proposal to allow for efficient preemption point placement, and we propose a framework to bound execution frequencies of individual program points independently of potential execution paths. All variants are well-defined, directly applicable to the proposed reference implementation, highly efficient and simple.

Improved Bounds on Cache-related Preemption Delay

We improve upon the state of the art by proposing two new methods of estimation. We identify common weaknesses in traditional approaches and specifically address them by proposing an improved bound that performs estimates conceptually orthogonal to existing approaches, avoiding their pessimism. We then further extend the interface of scheduling analysis by proposing an improved estimation method that exploits timing information from path analysis to exclude infeasible task interferences from consideration. This demonstrates that bypassing path analysis as in traditional approaches can lead to suboptimal results.

1.2 Structure

According to the order of contributions, this thesis is structured as follows. We introduce principles of program analysis and important practical aspects in Chapter 2. In Chapter 3 we set the topics subject to this thesis into perspective by providing a general overview on aspects of task scheduling and timing analysis. In Chapter 4 we give a thorough overview of cache analysis and discuss our own approach. Similarly, Chapter 5 starts with a thorough discussion of important aspects of path analysis followed by our approaches to loop identification and path analysis. In Chapter 6 we briefly review existing approaches to bound CRPD and then discuss our new bounds. We conclude the thesis in Chapter 7.

1.3 Contributing Publications

Contributions of this thesis have partially been published. The thesis bases on the following peer-reviewed publications:

- Jan Kleinsorge, Heiko Falk, and Peter Marwedel. A Synergetic Approach to Accurate Analysis of Cache-related Preemption Delay. In *Proceedings of the 7th International Conference on Embedded Software*, EMSOFT '11. ACM, October 2011
- Jan Kleinsorge, Heiko Falk, and Peter Marwedel. Simple Analysis of Partial Worst-case Execution Paths on General Control Flow Graphs. In *Proceedings of the 9th International Conference on Embedded Software*, EMSOFT '13. ACM, October 2013
- Jan Kleinsorge and Peter Marwedel. Computing Maximum Blocking Times with Explicit Path Analysis under Non-local Flow Bounds. In *Proceedings of the 10th International Conference on Embedded Software*, EMSOFT '14. IEEE, October 2014

The contributions to this thesis have been envisioned, specified, formalized and implemented by myself in their entirety in purely technical terms. Nevertheless inspiration, motivation, guidance and assistance is due to the co-authors of aforementioned papers by which they made invaluable contributions.

Chapter 2

Principles of Program Analysis

Contents

2.1 Programs: Syntax, Semantics and Interpretation **8**
2.2 Trace Semantics **8**
2.3 Collecting Semantics **10**
2.4 Fixed Point Semantics **12**
2.5 Abstraction **15**
2.6 Convention and Practical Program Analysis **18**

Program analysis is concerned with the discovery of facts about program execution such as the valuation of variables, the use of resources or the execution time. *Dynamic analysis* usually denotes the actual execution of a program under instrumentation to discover program facts directly. *Static analysis*, on the other hand, derives facts from a semantic model derived from programs, which does not necessarily encompass the entirety of program semantics and therefore allows for the restriction to specific facts of interest. In general, precise fact discovery is often undecidable. Most prominently, it is undecidable in general whether a program terminates — which implies execution time is undecidable either. Therefore, *abstraction* (approximation) of *concrete* semantics is necessary. Any such an abstraction has to be *sound*, which refers to the fact that only *true* facts about properties can be derived from it. Approximations are ideally *tight*, which means that facts derived from abstract semantics are not too imprecise to be useful. *Abstract Interpretation* [10, 11] denotes the general theory which provides the formal background to construct such program analyses. In this chapter we discuss its principles and the basic terminology.

In Section 2.1 we introduce some basic terminology. We then successively introduce different types of program semantics. In Section 2.2 we introduce trace semantics, followed by collecting (Section 2.3) and fixed point semantics (Section 2.4) as the foundation of the following chapters. We then focus on practical program analysis by first introducing the concept of abstraction (Section 2.5), followed by a brief primer on conventions we rely on later, and we give hints on practical implementations in Section 2.6.

2.1 Programs: Syntax, Semantics and Interpretation

A *program*[1] P is represented by sequence of *statements* $(l_1, \ldots, l_n)$ of a *language* L, representing its *syntax*, where each statement $l_i \in L$ is uniquely identified by its corresponding *program point* $q_i \in Q$. Every program P has a unique entry point $q_0 \in Q$ and a set of exit locations $Q_f \subseteq Q$. Statements l_i define the *semantics* of a program associated with program points q_i. Semantics define the transition of *program states* $S \subseteq \mathbb{D}$ in the given *state domain* $\mathbb{D}$ from one state to another. A state may, for example, represent the current program point or the valuation of variables. This transition of state is commonly referred to as *data flow* and transition of program location, specifically, is referred to as *control flow*. Semantics of a language define how statements affect state. In other words, it defines its *interpretation*. The successive interpretation of a given initial state is referred to as *evaluation*.

For the purpose of program analysis, we might not be interested in the complete state. For example, we might only be interested in whether program points are ever reached or just the sign of variable valuations. On the other hand, we might be interested in aspects beyond mere program semantics such as the state of hardware which is indirectly affected by execution but which is not subject to explicit language semantics. To this end, it is often a practical necessity to derive an *abstract interpretation*, which reduces or extends semantics according to our requirements.

In the following we discuss and formalize abstract interpretation to clarify important aspects of program analysis and to define a general framework for efficient practical program analyses.

2.2 Trace Semantics

We first define how semantics are applied to program state in general for our definition of programs. We assume that $q(s) \in Q$ denotes a program point contained in a program state $s \in \mathbb{D}$.

Definition 2.1 (Transfer Function) *For a program, let Q denote program points and let $\mathbb{D}$ denote its state domain. A* transfer function *(or* transformer*) denotes semantics at program points and is defined as:*

$$\mathrm{tf}\colon Q \mapsto (\mathbb{D} \mapsto \wp(\mathbb{D})) \tag{2.1}$$

Evaluation of a program then is denoted by the successive application of transformer tf to an initial state.

[1] We assume an *imperative* program model.

Definition 2.2 (Trace) *Let* $s_0 \in S_0$ *denote an initial state (input), then a* trace $\mathrm{tr}\colon \mathbb{D} \mapsto \mathbb{D}^*$ *is a sequence of states defined as:*

$$\mathrm{tr}(s_i) = \begin{cases} (s_i) \cdot \mathrm{tr}(s_{i+1})\colon s_{i+1} \in \mathrm{tf}(q(s_i))(s_i) & \text{if } q(s_i) \notin Q_f \\ \epsilon & \text{otherwise} \end{cases} \tag{2.2}$$

such that $\mathrm{tr}(s_0)$ *denotes all* reachable *states for the given input. A* trace path *(or execution path) is the sequence of program locations along a trace:* $(q(s_0), q(s_1), \dots)$*.*

A program is *deterministic* if and only if every evaluation from an initial state yields the same trace. Therefore, it must hold that $\forall s_i \in \mathbb{D}\colon |\mathrm{tf}(q(s_i))(s_i)| = 1$. A program *terminates* for an input $s_0 \in \mathbb{D}$ if and only if an evaluation has a finite number of steps: $|\mathrm{tr}(s_0)| \in \mathbb{N}$. Checking for termination is known as the *halting problem*: To decide whether a program terminates for a given input, we could construct a program that computes traces for a given program along with its input. This program itself might not terminate if traces cannot be computed in a finite number of steps. The halting problem is undecidable in the general case.

Definition 2.3 (Computation Tree) *Let* $s_0 \in S_0$ *denote program input and let* $\mathrm{tr}(s_0) = (s_0, \dots)$ *denote an execution trace. Then the digraph* $T = (S, R)$ *with* $S \in \mathrm{tr}(s_0)$ *and* $(s_i, s_{i+1}) \in R$ *is the corresponding* computation tree*.*

For a deterministic program and concrete program semantics, its computation tree is a simple path of potentially finite length. Abstracting semantics might cause non-determinism by losing information. For example, an abstract interpretation might only model the sign of variables instead of their concrete valuation: In case of program branch decisions depending on specific values, an analysis must take all potentially resulting execution paths into account. A class of program analyses known as *model checking* [12] tests for properties in computation trees specified in propositional logic.

Definition 2.4 (Trace Semantics) *The* trace semantics *of a program is the set of all traces:*

$$\{\mathrm{tr}(s_0)\colon s_0 \in S_0\} \tag{2.3}$$

In trace semantics we collect all reachable states of a program including their order. If a program does not terminate for a given input or interpretation, trace semantics is not computable. In any case, trace semantics might be too large to be practically computable.

By the definition above, we also recognize yet another problem for program analysis in general: The set of initial states $S_0 \subseteq \mathbb{D}$ does not necessarily reach all possible states. Hence, the computed semantics may be an under-approximation (unsound) of actual semantics if input is not exhaustive. For concrete semantics, the set of possible inputs might be too large to be practically applicable.

2.3 Collecting Semantics

A more efficient semantics is the *collecting semantics* (COL). It approximates trace semantics by losing information on the sequential order of program states. Instead of sets of traces, we only compute sets of states discriminated by program points. This semantics effectively separates control from data flow by requiring the existence of a control flow graph as an abstraction of execution paths.

Definition 2.5 (Control Flow Graph) *A* control flow graph *(CFG) $G = (V, E, s, t)$ is a digraph where vertices (nodes) V denote a set of program points, edges $E \subseteq V^2$ denotes their control flow relation, node $s \in V$ denotes the program entry and, without loss of generality, node $t \in V$ denotes the program exit. For convenience, we just write $G = (V, E)$ and assume nodes s and t to be implicitly be given.*

We assume CFG G to be connected. Reduction from multiple exits V_t to a single exit t is easily achieved by adding a node t and edges $V_t \times \{t\}$ to the graph. Deriving a CFG as a program abstraction is a difficult analysis problem on its own since sound and tight control flow transitions must be computed. In Chapter 5, we will address this issue in detail. For completeness, we provide some additional definitions in this context

Definition 2.6 (Predecessor, Successor, Degree) *Let $G = (V, E)$ be a CFG. Then for a node $u \in V$, the sets of predecessors and successors, respectively, are denoted by* $\text{pred}(u) := \{v \mid (v, u) \in E\}$ *and* $\text{succ}(u) := \{v \mid (u, v) \in E\}$. *The* indegree *of a node u is denoted by* $\deg_{in}(u) = |\text{pred}(u)|$, *its* outdegree *is denoted by* $\deg_{out}(u) = |\text{succ}(u)|$,

Definition 2.7 (Control Flow Path) *Let $G = (V, E)$ be a CFG. Then a* control flow path *is a sequence of nodes $\pi = (u_1, \ldots u_k) \in \Pi \subseteq V^k$ such that $(u_i, u_{i+1}) \in E$.*

It is easy to see that paths in a CFG potentially over-approximate execution paths since only a point-wise relation is maintained instead of the execution "history" of how a point is reached. Nevertheless, explicit separation of control and data flow now allows the analysis (and abstraction) of "data" independently. In the following we maintain this separation. Therefore, we redefine transfer functions for CFGs as:

$$\text{tf}\colon V \mapsto (\mathbb{D} \mapsto \mathbb{D}) \tag{2.4}$$

Definition 2.8 (Path Semantics) *Let $\pi = (u_1, \ldots, u_k)$ be a path. Then* path semantics *is defined as the composition of transfer functions* tf *along π such that:*

$$[\![\pi]\!](\text{tf}) = \begin{cases} \text{id} & \text{if } \pi = \epsilon \\ \text{tf}(u_i) \circ [\![(u_1, \ldots, u_{i-1})]\!](\text{tf}) & \text{otherwise} \end{cases} \tag{2.5}$$

Application of an initial state $s_0 \in \mathbb{D}$ yields the reachable state $[\![\pi]\!](s_0) = s$ along a path π. Hence, we can easily define the set of all reachable states in a program point.

Definition 2.9 (Collecting Semantics) *Let* Π *denote all paths in a CFG, let* $u_s \in V$ *be an entry node and let* $S_0 \in \mathbb{D}$ *denote initial state.* Collecting semantics *is then defined as:*

$$\mathrm{col}(u) = \bigcup_{s \in S_0} \bigcup \{ [\![\pi]\!](\mathrm{tf})(s) \mid \pi = (u_s, \ldots, u) \in \Pi \} \tag{2.6}$$

Then the set $\mathrm{col}(u)$ *denotes all reachable states.*

Collecting semantics as just defined is also called "state-based" (or first-order) since it denotes a mapping from program points to states. In other words, it denotes the set of "values" in this point. For example, it denotes the valuation of program variables. A "path-based" (second-order) collecting semantics on the other hand denotes the set of paths up to a program point.

Definition 2.10 (Path-based Collecting Semantics) *Let* Π *denote all paths in a CFG, let* $u_s \in V$ *be an entry node and let* v_t *an exit node.* Path-based collecting semantics $\mathrm{col}_\pi \colon V \mapsto \mathbb{D}$ *for a node* $u \in V$ *is then defined as follows. The set of paths ending in a point* u *is defined as:*

$$\mathrm{col}_\pi^{\rightarrow}(u) = \bigcup \{ \pi \mid \pi = (u_s, \ldots, u) \in \Pi \} \tag{2.7}$$

and the set of paths originating from a point u *is defined as:*

$$\mathrm{col}_\pi^{\leftarrow}(u) = \bigcup \{ \pi^{-1} \mid \pi = (u, \ldots, u_t) \in \Pi \} \tag{2.8}$$

such that $\mathrm{col}_\pi^{\rightarrow}(u)$ *denotes all paths reaching point* u *and* $\mathrm{col}_\pi^{\leftarrow}(u)$ *denotes all* reverse *paths originating from point* u*, respectively. The former is referred to as "forward semantics" the latter as "backward semantics".*

This semantics is useful for answering questions of general reachability such as whether a program point is reachable from the entry (forward) or, inversely, whether it can reach the exit (backward).

Second-order semantics can be expressed in terms of first-order semantics. It is easy to see that for $\mathbb{D} = \Pi$ and $\mathrm{tf}(u) = \lambda S \,.\, \{ \pi \cdot (u) \mid \pi \in S \}$, it holds that for an initial state $S_0 = \epsilon$, the set $\mathrm{col}(u) = \mathrm{col}_\pi^{\rightarrow}(u)$ denotes exactly forward second-order semantics. Backward semantics is defined symmetrically (for the respective redefinition of Equation 2.6).

Due to the abstraction of control flow, collecting semantics is not computable in the general case, as it requires the enumeration of paths, but length and number of paths is potentially unbounded due to cycles. To make analysis feasible in the general case, we therefore must avoid path enumeration for state collection.

2.4 Fixed Point Semantics

We now introduce a program analysis referred to as *fixed point analysis*, which is practically feasible even for unbounded paths at the potential loss of additional precision. In collecting semantics (COL), states are computed by means of a union over all paths up to a program point (Historically, this is referred to as *meet-over-all-paths* (MOP) although it is defined as a join over all paths.) With *fixed point semantics*, on the other hand, we can avoid computing paths in the first place by considering states from immediately preceding program points only.

Lattices

We first introduce the technical framework. Let $S \subset \mathbb{D}$ denote a set such that for a function $\mathrm{f}\colon \mathbb{D} \mapsto \mathbb{D}$ (e.g. a transfer function), it holds that $S \cup \{s \mid s \in \mathrm{f}(s)\} = S$. Then S denotes an upper bound with respect to subset inclusion. Subset inclusion $\subseteq$ denotes a *partial order* on $\mathbb{D}$.

Definition 2.11 (Partial Order, Partially Ordered Set) *A* partial order *is a binary relation $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$ over a set $S \subseteq \mathbb{D}$, which, for elements $x, y, z \in S$, is:*

- $x \sqsubseteq x$ *(reflexive)*
- $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$ *(anti-symmetric)*
- $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$ *(transitive)*

The tuple $(S, \sqsubseteq)$ as called a partially ordered set *or* poset*.*

An element $s \in S$ is an *upper bound* if $\forall s' \in S\colon s \sqsubseteq s'$ and a *least upper bound* if for all upper bounds $S_{ub} \subseteq S$, it holds that $\forall s' \in S_{lub}\colon s' \sqsubseteq s$. For a set $S \in \mathbb{D}$, $\bigsqcup S$ denotes its least upper bound. Symmetrically, this holds for *(greatest) lower bounds* denoted by $\bigsqcap S$.

Definition 2.12 (Complete Lattice) *A poset $(\mathbb{D}, \sqsubseteq)$ is a* complete lattice *if every subset $S \subseteq \mathbb{D}$ has a least upper bound and a greatest lower bound. Element $\top = \bigsqcup \mathbb{D}$ denotes the* top element *and $\bot = \bigsqcap \mathbb{D}$ denotes the* bottom element *of $\mathbb{L}$. A complete lattice is denoted by the tuple $\mathbb{L} = (\mathbb{D}, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$.*

Conventionally, operators $\sqcup$ and $\sqcap$ are referred to as "join" and "meet" respectively. An "incomplete" lattice $(\mathbb{D}, \top, \sqsubseteq, \sqcup)$ is therefore referred to as *join semi-lattice*. Analogously, the tuple $(\mathbb{D}, \bot, \sqsubseteq, \sqcap)$ is referred to as *meet semi-lattice*. Figure 2.1 illustrates a complete lattice for $\mathbb{D} = \wp(\{1, 2, 3\})$ and subset inclusion $\subseteq$ for $\sqsubseteq$. We recognize that every power set domain with subset inclusion $\subseteq$ as order relation forms a complete lattice.

In the context of program analysis, sets of states represent knowledge we collected about a program, partial order provides a means to tell whether knowledge increases by adding new information and a least upper bound denotes the smallest set containing maximal knowledge. Lattices define the relation of knowledge. We now define how knowledge is collected.

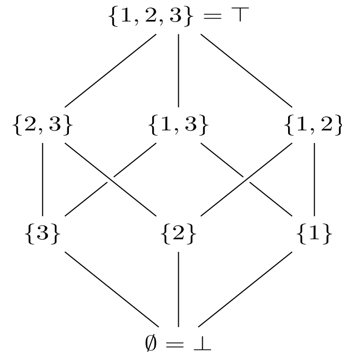


Figure 2.1: A complete lattice $\mathbb{L} = (\wp(\{1, 2, 3\}), \bot, \top, \subseteq, \cup, \cap)$.

Definition 2.13 (Monotonicity, Distributivity) *Let* $(\mathbb{D}_1, \sqsubseteq_1)$ *and* $(\mathbb{D}_2, \sqsubseteq_2)$ *denote posets. Then function* $\mathrm{f} \colon \mathbb{D}_1 \mapsto \mathbb{D}_2$ *is* monotone *(order-preserving) if and only if:*

$$\forall x, y \in \mathbb{D}_1 \colon x \sqsubseteq_1 y \Rightarrow f(x) \sqsubseteq_2 f(y) \tag{2.9}$$

It is distributive *(additive) if and only if:*

$$\forall x, y \in \mathbb{D}_1 \colon f(x \sqcup_1 y) \Rightarrow f(x) \sqcup_2 f(y) \tag{2.10}$$

Function $\mathrm{f} \colon \mathbb{D} \mapsto \mathbb{D}$ is *reductive* if $\forall x \in \mathbb{D} \colon f(x) \sqsubseteq x$ and *extensive* if $\forall x \in \mathbb{D} \colon x \sqsubseteq f(x)$.

Fixed Points

We can now define how upper and lower bounds on lattices can be computed, respectively, and how these bounds relate to collecting semantics.

Definition 2.14 (Fixed Point) *Let* $\mathbb{L}$ *be a lattice and let function* $\mathrm{f} \colon \mathbb{L} \mapsto \mathbb{L}$ *be monotone. Then* $x \in \mathbb{L}$ *with* $f(x) = x$ *denotes a* fixed point*.*

Theorem 2.15 (Tarski [13]) *Let* $\mathbb{L}$ *be a complete lattice and let function* $\mathrm{f} \colon \mathbb{L} \mapsto \mathbb{L}$ *be monotone, then the set of fixed points* $\mathrm{fix}\,\mathrm{f} = \{x \mid \mathrm{f}(x) = x\}$ *is a complete lattice.*

Since $\mathrm{fix}\,\mathrm{f}$ is a complete lattice, it has a least upper bound and a greatest lower bound. We denote the least upper bound of $\mathrm{fix}\,\mathrm{f}$ as the *least fixed point*

$$\mathrm{lfp(f)} = \bigsqcup \mathrm{fix}\,\mathrm{f} = \bigsqcup \{x \mid \mathrm{f}(x) \sqsubseteq x\} \tag{2.11}$$

and the greatest lower bound as the *greatest fixed point*

$$\mathrm{gfp(f)} = \bigsqcap \mathrm{fix}\,\mathrm{f} = \bigsqcap \{x \mid x \sqsubseteq \mathrm{f}(x)\} \tag{2.12}$$

Theorem 2.15 guarantees the existence of fixed points in complete lattices. But we have no notion of how it is computed yet.

Definition 2.16 (Chain) *For a poset* $(\mathbb{D}, \sqsubseteq)$*, a sequence* $S \subseteq \mathbb{D}^*$ *is a* chain *if all elements* $s \in S$ *are totally ordered. It is* ascending *if* $\forall x_i, x_{i+1} \in S \colon x_i \sqsubseteq x_{i+1}$ *and* descending *if* $\forall x_i, x_{i+1} \in S \colon x_{i+1} \sqsubseteq x_i$*.*

Definition 2.17 (Ascending Chain Condition) *A poset* $(\mathbb{D}, \sqsubseteq)$ *satisfies the* ascending chain condition *if all ascending chains eventually* stabilize*. A sequence of elements* $x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n$ *with* $x_i \in \mathbb{D}$ *stabilizes if there exists a* $k \in \mathbb{N}$ *such that* $x_k = x_{k+1}$*. Symmetrically, this holds for descending chains.*

Theorem 2.18 (Kleene [14]) *Let* $\mathbb{L}$ *be a lattice that satisfies the ascending chain condition and let* $\mathrm{f} \colon \mathbb{L} \mapsto \mathbb{L}$ *be a monotone function, then the least fixed point can be computed by the repeated application of* f *to* $\bot$*:*

$$\exists k \in \mathbb{N} \colon \mathrm{lfp}(\mathrm{f}) = \mathrm{f}^{\circ k}(\bot) \tag{2.13}$$

Analogously, if $\mathbb{L}$ *satisfied the descending chain condition, the greatest fixed point is computed by:*

$$\exists k \in \mathbb{N} \colon \mathrm{gfp}(\mathrm{f}) = \mathrm{f}^{\circ k}(\top) \tag{2.14}$$

By Theorem 2.18, we know the conditions under which the collection of analysis facts stabilizes and that the result will be a minimal (maximal) fixed point. We can now devise the algorithm to compute fixed point semantics.

Definition 2.19 (Minimal Fixed Point [15]) *Let* $G = (V, E, s, t)$ *denote a CFG, let* $\mathbb{D}$ *denote a complete lattice fulfilling the ascending chain condition and let* $\mathrm{tf} \colon V \mapsto (\mathbb{D} \mapsto \mathbb{D})$ *denote a (monotone) transfer function. Then the* minimal fixed point *(MFP) solution* $\mathrm{mfp} \colon V \mapsto (\mathbb{D} \mapsto \mathbb{D})$ *is the least fixed point* $\mathrm{lfp}(\mathrm{tf})$*:*

$$\mathrm{mfp}(u) = \begin{cases} \bigsqcup\{\mathrm{tf}(u)(\mathrm{mfp}(v)) \mid (v, u) \in E\} & \textit{if } u \neq s \\ \bot & \textit{otherwise} \end{cases} \tag{2.15}$$

Symmetrically, this holds for $\mathrm{gfp}(\mathrm{tf})$*. For semi-lattices, a suitable initial element but* $\bot$ *(*$\top$*) must be given.*

Note that not all ascending chains necessarily stabilize or do so only after an impractically large number of steps. To solve this, *widening* and *narrowing* [16] can be applied to deal with infinite chains in finite number of steps, at the expense of a loss of precision. To us, this will be of no relevance in the following.

Fixed point semantics is well suited for practical analysis as computations need not be carried out by the enumeration of CFG paths as in collecting semantics (which may not even be computable). Yet, we have not clarified the potential trade-off in precision.

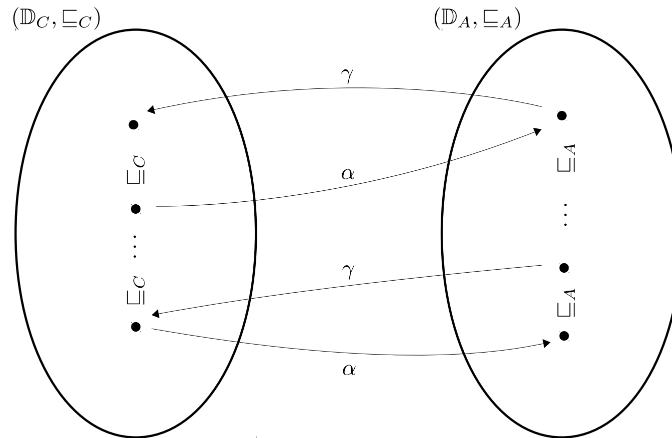


Figure 2.2: Illustration of the Galois Connection

Theorem 2.20 (Soundness of MFP [17, 18]) *Let $\mathbb{L}$ be a complete lattice fulfilling the ascending chain condition and let $\mathrm{f}\colon \mathbb{L} \mapsto \mathbb{L}$ be a monotone function. If all ascending chains stabilize in a finite number of steps, it holds that:*

$$\forall u \in V\colon \mathrm{col}(u) \sqsubseteq \mathrm{mfp}(u) \tag{2.16}$$

If in addition f is distributive, then it holds that:

$$\forall u \in V\colon \mathrm{col}(u) = \mathrm{mfp}(u) \tag{2.17}$$

Theorem 2.20 is also known as the *coincidence theorem.* For an analysis, if we can guarantee monotonicity of the corresponding transfer function for a value domain which is a lattice that fulfills the ascending chain condition, then MFP denotes a sound approximation of collecting semantics (COL). If, in addition, the transformer is also distributive, MFP equals COL.

2.5 Abstraction

Although the MFP solution can be computed independently of program paths, and thus circumvents the problem of infinite paths, some value domains might still yield impractically large solution sets. Similar to how we abstracted from concrete control flow in the step from trace to collecting semantics by introducing control flow graphs, abstraction can be applied to the effective analysis domain — again, at the potential additional loss of precision. In the following we discuss the foundation of abstraction.

Fundamentally, we are interested in the relation of a *concrete domain* $\mathbb{D}$ to its corresponding *abstract domain* $\hat{\mathbb{D}}$. If values computed in the abstract domain are always safe approximations of the concrete domain, we can directly apply fix point semantics. The relation of both domains is formally defined in terms of the *Galois connection.*

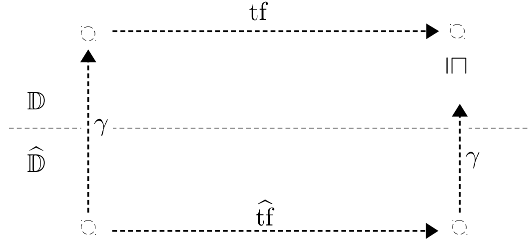


Figure 2.3: Illustration of local consistency

Definition 2.21 (Galois Connection) *A* Galois connection *is a tuple* $(\mathbb{D}, \alpha, \gamma, \widehat{\mathbb{D}})$ *where* $(\mathbb{D}, \sqsubseteq)$ *and* $(\widehat{\mathbb{D}}, \widehat{\sqsubseteq})$ *are partially ordered sets,* $\alpha\colon \mathbb{D} \mapsto \widehat{\mathbb{D}}$ *is an* abstraction function *and* $\gamma\colon \widehat{\mathbb{D}} \mapsto \mathbb{D}$ *is a* concretization function *such that* α *and* γ *are monotone and satisfy:*

$$\forall x \in \mathbb{D}\colon x \sqsubseteq (\gamma \circ \alpha)(x) \tag{2.18}$$

and

$$\forall x \in \widehat{\mathbb{D}}\colon (\alpha \circ \gamma)(x) \widehat{\sqsubseteq} x \tag{2.19}$$

A Galois Insertion*, in addition, satisfies:*

$$\forall x \in \widehat{\mathbb{D}}\colon x = (\alpha \circ \gamma)(x) \tag{2.20}$$

Monotonicity ensures that both mappings are order preserving. Property 2.18 guarantees that the abstraction is *sound*: The mapping into the abstract domain and back never loses information but potentially precision. Property 2.19 guarantees *precision* of the abstraction: The mapping into the concrete domain and back always yields a value as least as precise as the initial value. Equation 2.20 tightens these constraints: abstraction followed by concretization maps to the original abstract value. Figure 2.2 graphically illustrates the respective mappings between the two domains $\mathbb{D}$ and $\widehat{\mathbb{D}}$. The respective elements are ordered vertically according to the partial order relation of both posets.

To enable the applicability of the abstract domain to fixed point semantics, an *abstract transformer* must satisfy specific properties with regard to its concrete counterpart. Let $\mathrm{tf}\colon V \mapsto \mathbb{D} \mapsto \mathbb{D}$ be a concrete transformer and let $\widehat{\mathrm{tf}}\colon V \mapsto \widehat{\mathbb{D}} \mapsto \widehat{\mathbb{D}}$ be an abstract transformer.

Definition 2.22 (Local Consistency) *An abstract transformer* $\widehat{\mathrm{tf}}$ *is* locally consistent *with a concrete transformer* tf *if it holds that:*

$$\forall x \in \widehat{\mathbb{D}}\colon \forall u \in V\colon (\mathrm{tf}(u) \circ \gamma)(x) \sqsubseteq (\gamma \circ \widehat{\mathrm{tf}}(u))(x) \tag{2.21}$$

Local consistency guarantees that the abstract transfer function may be less precise than the concrete one, but it will never lose information and is therefore sound. Figure 2.3 illustrates this relation graphically. Concretization and a subsequent application of the concrete transformer is potentially more precise than the a application of the abstract transformer followed by the concretization. For a Galois insertion, $\gamma \circ \mathrm{tf} \circ \alpha$ is referred to as a *best abstract transformer*, which may be, however, infeasible if concretization is not realizable in practice.

It remains to show that COL and MFP can equally well be applied in the abstract domain if local consistency of transformers is guaranteed. Analogously to Definition 2.8 and Definition 2.9 for concrete semantics, we define their abstract counterparts.

Definition 2.23 (Abstract Path Semantics) *Let $\pi = (u_1, \ldots, u_k)$ be a path. Then abstract* path semantics *is defined as the composition of abstract transfer functions $\widehat{\mathrm{tf}}$ along π such that:*

$$[\![\pi]\!](\widehat{\mathrm{tf}}) = \begin{cases} id & \text{if } \pi = \epsilon \\ \widehat{\mathrm{tf}}(u_i) \circ [\![(u_1, \ldots, u_{i-1})]\!](\widehat{\mathrm{tf}}) & \text{otherwise} \end{cases} \tag{2.22}$$

Then abstract collecting semantics *for a set of paths Π and initial states $\widehat{S}_0 \in \widehat{\mathbb{D}}$, is defined as:*

$$\widehat{\mathrm{col}}(u) = \bigsqcup_{s \in \widehat{S}_0} \bigsqcup \left\{ [\![\pi]\!](\widehat{\mathrm{tf}})(s) \mid \pi = (u_s, \ldots, u) \in \Pi \right\} \tag{2.23}$$

Note that in concrete collecting semantics, we assumed set union $\cup$ to collect all state. For abstract states, collection of information may be performed for some other definition of union (potentially $\sqcup \neq \cup$).

Lemma 2.24 (Correctness of Abstract Path Semantics) *Let $(\mathbb{D}, \alpha, \gamma, \widehat{\mathbb{D}})$ be a Galois connection, then it holds that*

$$\forall x \in \widehat{\mathbb{D}} \colon [\![\pi]\!](\mathrm{tf})(\gamma(x)) \subseteq \gamma([\![\pi]\!](\widehat{\mathrm{tf}})(x)) \tag{2.24}$$

if the corresponding concrete transformer tf *and abstract transformer* $\widehat{\mathrm{tf}}$ *are locally consistent.*

Proof. See [19]. □

Lemma 2.25 (Correctness of Abstract Collecting Semantics) *Let $(\mathbb{D}, \alpha, \gamma, \widehat{\mathbb{D}})$ be a Galois connection, let $\widehat{\mathrm{tf}}$ be an abstract transformer, $S_0 \in \mathbb{D}$ and $\widehat{S}_0 \in \widehat{\mathbb{D}}$ initial states such that $S_0 \sqsubseteq \gamma(\widehat{S}_0)$, then it holds that*

$$\forall u \in V \colon \, : \mathrm{col}(u) \sqsubseteq \gamma(\widehat{\mathrm{col}}(u)) \tag{2.25}$$

if for the concrete transformer tf*, the abstract transformer* $\widehat{\mathrm{tf}}$ *locally consistent.*

Proof. See [19]. □

Reduction to abstract path-based collecting semantics (Definition 2.10) is obvious. Also, the construction of the MFP solution (Definition 2.19) applies analogously to the abstract domain. Then Theorem 2.20 directly applies as well in the abstract domain.

To summarize, an abstraction of concrete semantics is sound if i) the abstract domain $\widehat{\mathbb{D}}$ is a poset ii) abstraction α and Concretization γ are sound iii) transformers $\widehat{\mathrm{tf}}$ and tf are locally consistent.

2.6 Convention and Practical Program Analysis

We will briefly address some practical aspects of program analysis. In particular, we define conventions and tools used throughout this thesis.

For convenience, we will usually define analysis problems in terms of collecting semantics. As we have seen, reduction from COL to MFP is straight forward. We denote forward semantics by $\mathrm{col}^{\rightarrow}$ and backward semantics by $\mathrm{col}^{\leftarrow}$ and path-based semantics by $\mathrm{col}_{\pi}^{\rightarrow/\leftarrow}$, respectively. Analogously, for the recursive equation for the MFP solution Equation 2.15, $\mathrm{mfp}^{\rightarrow/\leftarrow}$ denotes only predecessor/successor nodes in the corresponding CFG. In some cases, we restrict col and mfp to subgraphs with explicitly given source and sink nodes, or to a subset of edges. Equation 2.15 can be rewritten as:

$$\mathrm{in}(u) = \bigsqcup_{v \in \mathrm{pred}(u)} \mathrm{out}(v) \tag{2.26}$$

$$\mathrm{out}(u) = \mathrm{tf}(u)(\mathrm{in}) \tag{2.27}$$

which explicitly discriminates states prior and after application of a transformer, for a suitable initialization.

In practice, the least solution to the equation system above can be solved iteratively by successively updating program information in all program points by performing just one step at a time per node, cycling through all nodes (*round robin*).

Algorithm 2.1 Worklist algorithm for iterative data flow analysis

```
1   for u ∈ V do
2       in[u] ← out[u] ← S_0
3   W ← V
4   while w ≠ ∅ do
5       u ← pop W
6       in(u) ← ⊔_{v∈pred u} out(v)
7       out(u) ← tf(u)(in)
8       if out(u) changed
9           W ← W ∪ succ u
```

Algorithm 2.1 lists a practical implementation to compute a forward MFP solution [20]. In lines $1, 2$ the arrays in and out are assigned a suitable initial value, then a worklist

containing all nodes of the corresponding CFG is created (line 3). We recompute values in set *in* and set *out*, one node at a time, in some order by removing some node from W (line 5), then recompute the values (lines 6,7). If we have not reached a fixed point yet (line 8), we add the successors of the current node u (line 9) to the worklist.

If the underlying CFG is acyclic (DAG) and nodes are processed in topological order, a fixed point can be reached in time linear to the number of nodes. Consequently, if the worklist is ordered accordingly initially, a fixed point is likely to be reached quicker. In addition, the smallest semantic unit of a program is typically a statement (or an instruction) but not all statements affect control flow. Hence, it is customary to group statements in *basic blocks* [20], which denote maximal sequences of statements with control flow branching only at the first and the last statement, to denote a single CFG node. More details on the general subject of practical data flow analysis can be found in [21–23]. Note that *control flow analysis* [24–26] for the construction of CFG is an important topic on its own.

Chapter 3

Context

Contents

3.1 Real-time Scheduling **21**
3.1.1 Basic Task Model 22
3.1.2 Modes of Preemption 25
3.1.3 Deadline Monotonic and Earliest Deadline First 27
3.1.4 Schedulability 28
3.1.5 Blocking and Synchronization 30
3.2 Timing Analysis **33**
3.2.1 Practical Aspects 34

This chapter contains background information to set this thesis into context. Every topic discussed here is related, but not necessarily a requirement to understand the remainder of this thesis. In the following the topics of real-time scheduling, aspects of hardware in real-time systems and aspects of software for real-time systems, including timing analysis of software will be addressed.

Specifically, the chapter comprises of a discussion on basics of schedulability theory in Section 3.1 and fundamentals of timing analysis in Section 3.2.

3.1 Real-time Scheduling

We start with some basic terminology. The problem of allocating processing time for concurrently running software is known as the *scheduling problem*. In this, the basic unit of processing is that of a *task*. The assignment of tasks to processors is then usually performed under a given set of constraints, which is denoted as the *scheduling policy*. A *scheduling algorithm* (or scheduler), then, is the method of finding a *feasible schedule* such that all tasks can be completed according to a set of constraints. A set of tasks is *schedulable* if for a given algorithms all constraints hold.

Real-time scheduling can typically be found in embedded systems. As opposed to non-real-time systems, the emphasis is not on load-balancing or general responsiveness but on meeting timing guarantees such as timing deadlines. *Hard real-time* policies give

firm guarantees: The consequence of deadline misses in this case is typically a complete system failure. *Soft real-time* policies on the other hand allow deadline misses typically at the expense of a degraded quality of service.

A schedule is *preemptive* if it allows the temporary suspension of a task to assign another task to the processing unit. Otherwise, a schedule is said to be *non-preemptive*. If a scheduling policy allows instantaneous preemptions on demand, it is *fully preemptive*. The compromise between fully and non-preemptive schedules is called *deferred preemption* scheduling. Under such a policy, preemption is only allowed at specific points in time or at specific program points.

Typically, tasks are being assigned *priorities* such that the task of highest priority is assigned to a processor. A *static scheduling* assigns priorities according to tasks parameters known prior to actual system execution. A scheduling is said to be *dynamic* if priorities are assigned at run-time. Static schedules are typically more predictable at the cost of lower performance.

Besides mere timing, additional constraints can be imposed on tasks. *Precedence constraints* enforce a specific order on the execution of tasks. *Resource constraints* impose limits on the availability of resources other than processing time, such as the *mutual exclusion* of accesses to certain resources.

In particular hard real-time systems pose specific requirements on the predictability of system components. Therefore, not only has the hardware and the task software to be predictable, but the scheduling algorithms themselves should be predictable in the sense that i) a safe upper bound of their processing overhead can be determined and ii) safe upper bounds on the timing behavior of the final schedule can be obtained. In the following we formally introduce the basics of uni-processor, priority-based, hard real-time scheduling without explicit precedence constraints. A general overview of hard real-time scheduling can be found in [27].

In Section 3.1.1 we define the task model used throughout the thesis, in Section 3.1.2 we address important aspects of preemptive scheduling, then we introduce two widely employed scheduling policies in Section 3.1.3. For the latter, we briefly discuss schedulability tests in Section 3.1.4. We further characterize issues related to task blocking and synchronization in Section 3.1.5.

3.1.1 Basic Task Model

We assume the problem of scheduling a *task set* $\mathcal{T} = \{\tau_1, \tau_2, \dots\} \subseteq \mathbb{N}_0$ on a single processor. A single execution (instance) of a task is called a *job* where τ_i^j denotes the j'th job of τ_i.

Time in this model is discrete and measured in clock ticks if no other unit is specified explicitly. A job can be in one of three states: *ready*, *run* and *wait*, as illustrated in the automaton in Figure 3.1. The edges are labeled with the events that can occur. A job is *released* once it is scheduled for execution, and placed into the *ready-queue* – a queue of tasks *ready* for *dispatch* if another task is currently *run* by the CPU. The release of

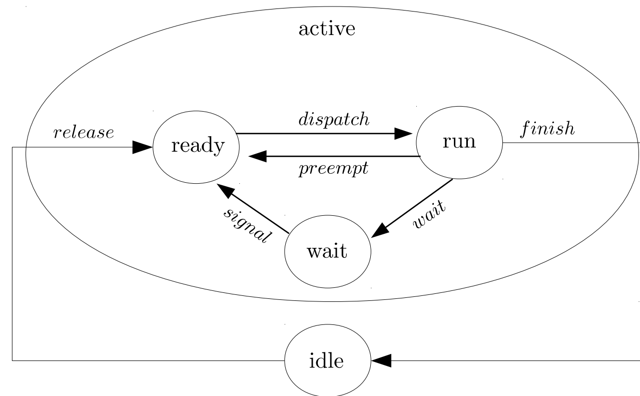


Figure 3.1: State-machine of a task with preemption and synchronization

$\mathcal{T} = \{\tau_1, \ldots, \tau_n\} \subseteq \mathbb{N}_0$	Task set
$H_{\mathcal{T}}$	Hyper period
pr_P	Priority under policy P
	Task τ_i
B_i	Blocking time
C_i	Computation time
D_i	Relative deadline
J_i	Release jitter
R_i	Response time
T_i	Period, Inter-arrival time
U_i	Utilization
	Job τ_i^j
a_i^j	Arrival time
d_i^j	Absolute deadline
f_i^j	Finishing time
l_i^j	Lateness
r_i^j	Response time
s_i^j	Starting time

Table 3.1: Task parameters

a higher priority task potentially *preempts* a running task. Alternatively, a job can be send into a *wait* state (specifically into a *wait-queue*) if, for example, it fails to acquire another resource by itself. All waiting jobs receiving a *signal* are put back into the ready state. A task is *active* if there's a job that is either ready, running or waiting. Otherwise, it is *idle*.

A task schedule is *aperiodic* if tasks are activated at arbitrary points in time. It is *periodic* if activation occurs in fixed intervals or *sporadic* if it is aperiodic with the constraint that there exists a *minimal inter-arrival time* between jobs. In the following we only consider periodic tasks.

A task τ_i is assigned a set of static parameters such that C_i denotes its *computation time*, D_i its *relative deadline* and T_i its *period* or, alternatively, its *inter-arrival time*. *Release jitter* is denoted by J_i and defines a possible imprecision in timing (e.g. time consumed by the scheduling decision and the context switch). Blocking time B_i denotes the time span a higher priority task is prevented from execution by lower priority tasks due to deferred preemption or exclusive resource access. A deadline is an *implicit deadline* if $D_i = P_i$. The *hyper period* of a periodic task set $\mathcal{T}$ is the least common multiple of its periods T_i:

$$H_{\mathcal{T}} = \mathrm{lcm}(T_1, \ldots, T_k) \tag{3.1}$$

The earliest release time of a job τ_i^j is called *arrival time* and is defined as:

$$a_i^j = a_i^{j-1} + T_i - J_i \tag{3.2}$$

The *absolute deadline* of a job is derived from the arrival time and its relative deadline:

$$d_i^j = a_i^j + D_i \tag{3.3}$$

We call the time instant at which a job executes for the first time after being released the *starting time* s_i^j and the time instant at which it completes the *finishing time* f_i^j. The *response time* of a job is the time span from activation to completion, defined as:

$$r_i^j = f_i^j - a_i^j \tag{3.4}$$

From individual jobs, we can derive the *response time* R_i, which is defined as:

$$R_i = \max_j \{r_i^j\} \tag{3.5}$$

A task set is *schedulable* if all of its jobs finish before their respective deadlines:

$$R_i \leq D_i - J_i \Leftrightarrow \tau_i \text{ schedulable} \tag{3.6}$$

The time difference of finishing time and its absolute deadline is a job's *lateness*:

$$l_i^j = f_i^j - d_i^j \tag{3.7}$$

A negative lateness denotes that a task finished before its deadline and is therefore a requirement in hard real-time systems.

Table 3.1 summarizes these and additional parameters discussed in the following. In Figure 3.2 key parameters for tasks and jobs are depicted graphically.

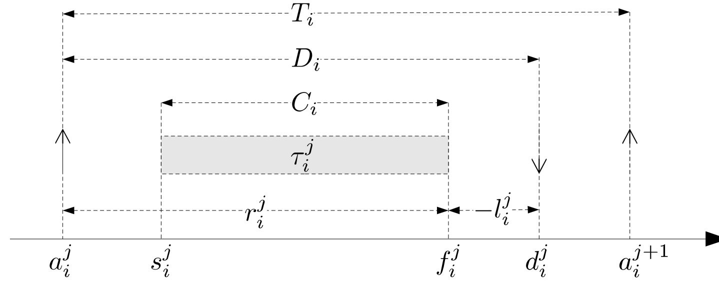
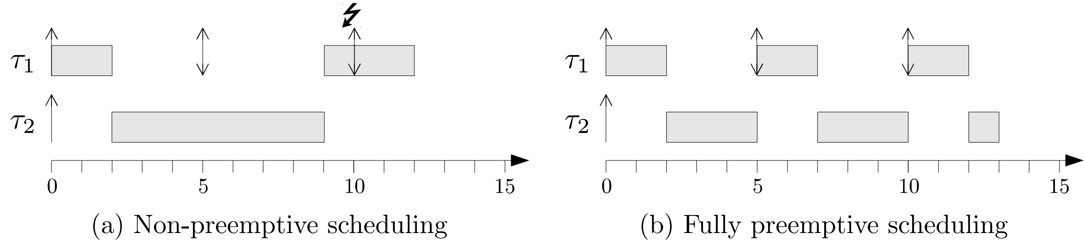


Figure 3.2: Illustration of task parameters



(a) Non-preemptive scheduling (b) Fully preemptive scheduling

Figure 3.3: Example of feasible schedule under fully preemptive scheduling

3.1.2 Modes of Preemption

Preemptions relax the scheduling problem by allowing subdivision of tasks and therefore potentially simplify the computation of a feasible schedule where non-preemptive scheduling might fail to do so due to excessive blocking.

Figure 3.3 illustrates how an infeasible schedule for two tasks can turn into a feasible one by allowing unrestricted preemptions. In both time lines, task τ_1 is given priority over task τ_2 when it is active. In Figure 3.3a a non-preemptive schedule is shown. Since τ_2 occupies the processor during the entire active period of τ_1, the latter is never run and misses its deadline. When scheduled fully preemptively, as shown in Figure 3.3b, τ_2 is interrupted instantly upon activation of τ_1, therefore allowing both tasks to meet their deadlines.

The downside of preemptions is that they cause a number of problems for hard real-time scheduling. The computation time C_i denotes the worst-case execution time of a task under the assumption of uninterrupted execution. Each suspension comes with associated context-switch costs and an unconsidered change of the system state that might affect the computation time long after having been resumed. In short, preemptions increase the level of unpredictability.

The trade-off between fully preemptive scheduling and non-preemptive scheduling can be mitigated by compromising such that preemption is *deferred*. Deferring preemption potentially enables schedulability of tasks at the cost of higher priority tasks potentially becoming unschedulable due to blocking. One way to defer preemptions is to allow preemptions just periodically in fixed time-intervals. This way, task subdivision is performed in the time domain. Since program points at which preemptions occur are not explicitly known, the program points between two preemptions is referred to as

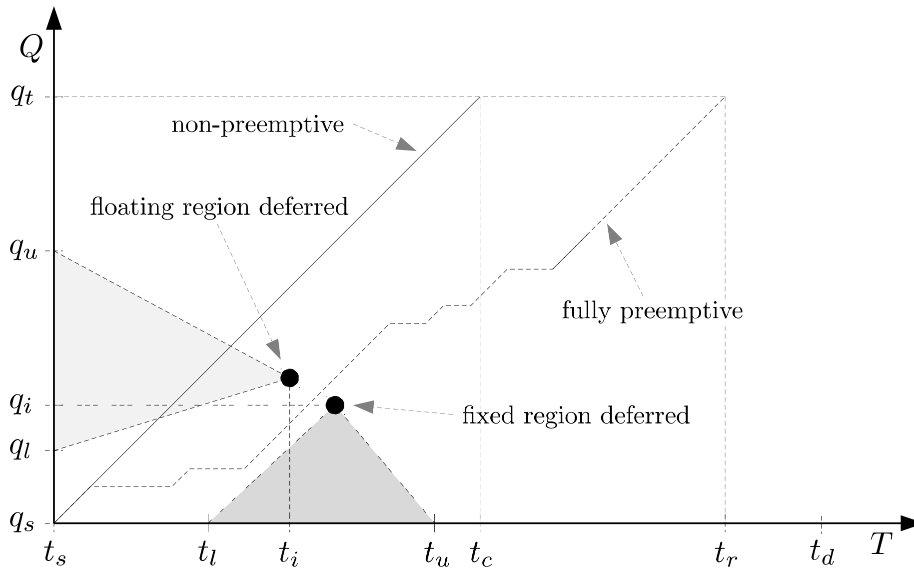


Figure 3.4: Progress and relation of space and time for non-preemptive, fully preemptive and deferred preemption scheduling with floating and fixed regions

a *floating region*. Another way is to subdivide a task in the space domain by placing explicit *preemption points* [28]. Then points in time are not explicitly known and the program locations not affected by preemption are referred to as a *fixed region*. Fixed region deferred preemption with static scheduling decisions is also known as *cooperative scheduling* [29, 30] to emphasize the fact that tasks explicitly have to yield the processor "voluntarily".

Figure 3.4 sketches the trade-off between the aforementioned classes of scheduling policies. The abscissa denotes time and the ordinate denotes program locations of a task. We assume that a non-preemptive execution of a job is a linear mapping from time to location. For a non-interrupted execution, the job with deadline t_d starts at time t_s and finishes at time t_c while traversing all program locations from source q_s to sink q_t. A fully preemptive execution is characterized by interruptions with no well-defined points in time or space and a response time of t_r, which potentially lies well beyond the isolated computation time for the uninterrupted execution t_c. For deferred preemption with floating regions, interruption is potentially less frequent. For some point in time t_i, it is, however, not necessarily possible to map to a corresponding program location but only to an interval $[q_u, q_l]$. Inversely, for deferred preemption with fixed regions, although the location of the interruption q_i is known, we might only map this point to an interval $[t_l, t_u]$.

Floating regions are modeled in two ways. In what the authors of [31] call the "floating model", preemptions are enabled or disabled by enabling or disabling interrupts explicitly by inserting the respective primitives into the software and the guarantee that none of the non-preemptive regions takes longer than a specific time bound to execute. It is called floating since the region bounds in time are not specified in the model. In regions with interrupts enabled, a standard scheduling policy applies. A variant of this

model is that of *final non-preemptive regions* [32] which reduces the problem to a single such region per task. The "activation-triggered model" [31, 33] defines that the arrival of a higher priority task causes all preemptions to be postponed for a defined period of time such that the initially postponed preemption occurs right after the period without being delayed any further by other tasks.

3.1.3 Deadline Monotonic and Earliest Deadline First

Two prominent scheduling policies for hard real-time scheduling on a single processor are *Deadline Monotonic Scheduling* (DM) and *Earliest Deadline First* (EDF). Both are priority-based, preemptive and popular choices for periodic scheduling due to their predictability and the existence of efficient schedulability tests. Both assign priorities to tasks such that the task of highest priority is run, and both have no notion of precedence or resource constraints.

Deadline Monotonic Scheduling [34] assigns priorities to tasks according their deadlines. Priority assignment is therefore *static* and the overhead at run-time is restricted to the selection of the highest priority task among all ready tasks. The task priority pr_{DM} is defined such that:

$$\mathrm{pr}_{DM}(\tau_i) < \mathrm{pr}_{DM}(\tau_j) \Leftrightarrow D_i \geq D_j \tag{3.8}$$

DM is optimal in the sense that no other static priority assignment schedule can exist if it cannot be scheduled with DM [35]. DM with implicit deadlines is known as *Rate Monotonic Scheduling* (RMS) [36].

For later reference, we define the following functions: For static and unique priorities pr_P for some scheduling policy P, we define the following task subsets:

$\mathrm{hp}(i) = \{j \mid \mathrm{pr}_P(i) < \mathrm{pr}_P(j)\}$	Higher priority tasks
$\mathrm{hep}(i) = \mathrm{hp}(i) \cup \{i\}$	Higher priority tasks and self
$\mathrm{lep}(i) = \mathcal{T} \setminus \mathrm{hp}(i)$	Lower priority tasks and self
$\mathrm{lp}(i) = \mathcal{T} \setminus \mathrm{hep}(i)$	Lower priority tasks
$\mathrm{aff}(i, j) = \mathrm{hep}(i) \cap \mathrm{lp}(j)$	Bounded set of higher priority tasks

Table 3.2: Definitions of priority-relative task sets

Earliest Deadline First [37] assigns priorities *dynamically*, which increases the run-time overhead of scheduling decisions at the advantage of dynamic adaption to changing runtime constraints and is therefore not restricted to periodic schedules. Priority pr_{EDF} is defined such that among all jobs ready to run, the job closest to its absolute deadline is given the highest priority. Let t denote the current absolute time and c_i^j the computation

time of a job τ_i^j (the effective time the job has run), then priority is defined such that:

$$\mathrm{pr}_{EDF}(t, \tau_i^j) < \mathrm{pr}_{EDF}(t, \tau_k^l) \Leftrightarrow (d_i^j - c_i^j) \geq (d_k^l - c_k^l) \tag{3.9}$$

EDF is optimal in the sense that no other dynamic priority assignment schedule can exist if it cannot be scheduled with EDF. EDF minimizes the maximum lateness [37]. Note that Table 3.2 is equally valid for EDF under the assumption that the sets denotes potential candidates throughout execution.

3.1.4 Schedulability

A schedulability test takes a task set and a scheduling policy as input and returns whether the test is passed or not. A schedulability test is *necessary* if it holds that i) if the test is positive, then there might exist a feasible schedule, but not necessarily ii) if the test is negative, there definitively does not exist a feasible schedule. A test is *sufficient* if the it holds that i) if the test is positive, there definitely exists a feasible schedule ii) if the test is negative, then there might exist a feasible schedule anyway. A test is *exact* if it is both necessary and sufficient.

Utilization

A measure to quantify the load of a processor is the *utilization factor*: the fraction of time a task uses from the available processing time, defined as::

$$U_i = C_i / T_i \tag{3.10}$$

The accumulated utilization for a task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, then, is defined as:

$$U_{\mathcal{T}} = \sum_{i=1}^{n} C_i / T_i \tag{3.11}$$

A (necessarily) exact test for EDF (with implicit deadlines) is simply:

$$U_{\mathcal{T}} \leq 1 \tag{3.12}$$

For EDF without implicit deadlines, a *demand bound function*[1] [33, 38] defines the time demand of all tasks in a given time interval L. If time demand does not exceed the length of any such L, the task set is schedulable:

$$\forall L > 0\colon \sum_{i=1}^{n} \left(\left\lceil \frac{L - D_i}{T_i} \right\rceil + 1 \right) C_i \leq L \tag{3.13}$$

[1] Also known as "processor demand criterion" [27]

In practice, not all possible values of L can be tested. The authors of [38] propose the *quick convergence processor demand algorithm* (QPA) for an efficient schedulability test of EDF.

The classical schedulability test for DM with implicit deadlines (e.g. RMS), is a sufficient test based on utilization [35], defined such that:

$$U_{\mathcal{T}} \leq n(2^{1/n} - 1) \Rightarrow \mathcal{T} \text{ schedulable} \tag{3.14}$$

where

$$\lim_{n \to \infty} n(2^{1/n} - 1) = ln(2) \approx 0.7 \tag{3.15}$$

The test suggests that with static priorities a processor is potentially not fully utilized if hard real-time guarantees must be given. A slightly more accurate test is the *hyperbolic bound* test [39].

Response time

An exact test for task schedulability for periodic fixed-priority schedules (such as DM) is the *response time analysis* (RTA) [40, 41]. Intuitively, an individual task must be schedulable if it meets its deadline under the worst-possible circumstances (cf. Equation 3.6). Such a worst-case scheduling scenario, referred to as the *critical instant*, occurs

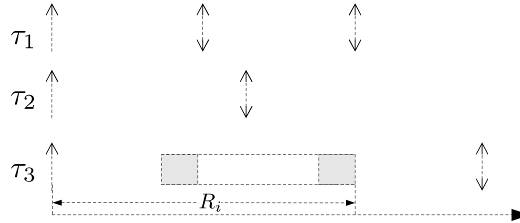


Figure 3.5: Critical instant

when the task and all its higher priority tasks are released simultaneously such that the response time is maximized . Figure 3.5 illustrates a schedule where all release times are left-aligned, which implies that the initial execution of the lowest priority task τ_3 is maximally postponed in addition to the delays caused by preemptions that inevitably occur. Response time R_i then comprises of the computation time C_i of τ_i and the computation times C_j of all higher priority tasks τ_j. Since the schedule is periodic, τ_i can be preempted at most $\lceil (R_i + J_i)/T_j \rceil$ times by τ_j during response time R_i. Let hp denote the set of all higher priority tasks. Then the *worst-case response time* (WCRT) is the least fixed point denoted by the following equation system:

$$R_i^{(0)} = C_i \tag{3.16}$$

$$R_i^{(k+1)} = C_i + \sum_{j \in \text{hp}(i)} \left\lceil \frac{R_i^{(k)} + J_j}{T_j} \right\rceil C_j \tag{3.17}$$

The value of R_i is monotonically increasing and eventually reaches a fixed point if $R_i \leq D_i$. A reasonable initial value is the computation time C_i. For brevity, we write:

$$R_i = C_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{3.18}$$

While WCRT is derived from the notion of critical instants, it is equally possible to derive the notion of the *best-case response time* (BCRT) [42] from the notion of *optimal instants*. An optimal instant is a scheduling scenario where a task τ_i instantly runs upon release and its completion coincides with the simultaneous release of all higher priority tasks. Figure 3.6 illustrates an optimal instant. Task τ_3 is the lowest priority task and runs right after release and all other tasks' next releases are right-aligned with its completion. The intuition behind BCRT is that all potentially preempting tasks will

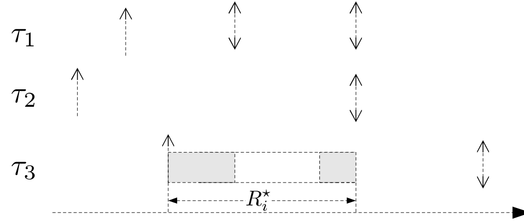


Figure 3.6: Optimal instant

already have run for their maximally possible amount of time and cause the least number of preemptions.

Accordingly, let $C_i^\star$ denote the *best-case execution time* (BCET) of task τ_i, then the BCRT (with an optimal blocking time equal to 0) is the greatest fixed point to the recursive equation:

$$R_i^{\star(0)} = R_i \tag{3.19}$$

$$R_i^{\star(k+1)} = C_i^\star + \sum_{j \in \mathrm{hp}(i)} \left(\left\lceil \frac{R_i^{\star(k)} - J_j}{T_j} \right\rceil - 1 \right) C_j^\star \tag{3.20}$$

$R_i^\star$ is monotonically decreasing for an initial value R_i (WCRT). For brevity, we write:

$$R_i^\star = C_i^\star + \sum_{j \in \mathrm{hp}(i)} \left(\left\lceil \frac{R_i^\star - J_j}{T_j} \right\rceil - 1 \right) C_j^\star \tag{3.21}$$

3.1.5 Blocking and Synchronization

Often tasks share common resources, beside the processor time, which must be protected from mutual access. DM and EDF are scheduling policies oblivious of resource constraints. In the following we discuss blocking and synchronization of concurrent resource accesses and we briefly sketch prominent resource constrained scheduling policies.

In fully preemptive scheduling without resource constraints, higher priority tasks are never blocked by lower priority ones. In non-preemptive scheduling, the blocking time

imposed on a task τ_i is simply the maximum computation time of all lower priority tasks.

$$B_i = \max_{j \in \mathrm{lp}(i)} C_j \tag{3.22}$$

In deferred preemptive scheduling, blocking depends on the type of strategy. For time-triggered floating regions, blocking time B_i is known by definition. For fixed region deferred preemption, though, it depends on the longest execution paths between preemption points. which will be subject to detailed discussion in Chapter 5. Once known, response time analysis can be extended by blocking time B_i such that:

$$R_i = C_i + B_i + \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{R_i + J_j}{T_j} \right\rceil C_j \tag{3.23}$$

Semaphores

Resources shared between tasks are typically protected by means of *mutual exclusion* to ensure consistency of data structures by *synchronization* of tasks. A code region executed under mutual exclusion is called a *critical section*. A common synchronization primitive is the *semaphore*. A task enters a critical section protected by a semaphore S_i by invoking an operation $wait(S_i)$, which either grants access — thus making the task the owner of the resource — or *blocks* the requesting task. When a task leaves its critical section, it invokes an operation $signal(S_i)$ to *free* the resource and which effectively unblocks all tasks waiting for that resource such that they can attempt again to enter their respective critical sections. In Figure 3.1, this is denoted by the state *wait*. A semaphore can allow for multiple tasks to access the resource. A semaphore that grants access to just a single task at a time is called a *mutex*.

Scheduling policies with limited preemption enforce implicit critical sections with mutual exclusion with the processor time as the shared resource. In non-preemptive scheduling, the critical section encompasses the entire task. In deferred preemption scheduling, tasks are partitioned into disjoint critical sections.

Priority Inversion Problem

Synchronization globally overrides task priorities to ensure semantic correctness. If the scheduling policy is unaware of resource constraints, the *priority inversion problem* can occur, as illustrated in Figure 3.7. In the figure, task priorities are static and decreasing from τ_1 on and only tasks τ_1, τ_3 share a protected resource. When τ_1 is released at time step 3, τ_3, which has already entered its critical section, is preempted according to the given priorities. When τ_1 attempts to enter its critical section at time step 4, it is blocked by τ_3, which in turn resumes execution. At time step 5, the priority inversion problem manifests: Although τ_1 is the highest priority active task, τ_2 is scheduled to run because τ_1 is blocked and τ_3 has lower priority than τ_2. Thus, τ_1 is not only delayed by the blocking due the critical section of τ_3 but also by the computation time of the unrelated

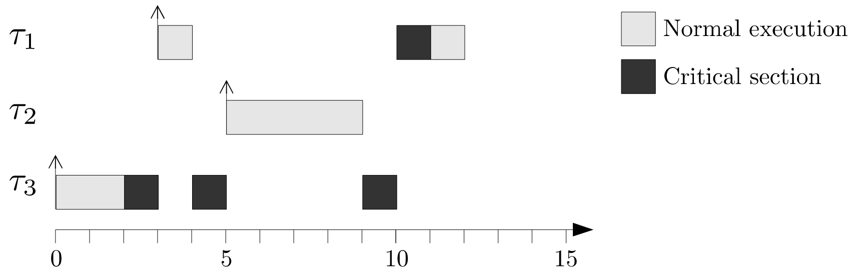


Figure 3.7: Example of a priority inversion problem

task of intermediate priority τ_2. Hence, for the time interval $[4, 10]$, priorities of τ_1, τ_3 are effectively inverting.

The priority inversion problem is critical since tasks of a nominally high priority are blocked for an unbounded amount of time. In the following we briefly sketch common strategies to counter the priority inversion problem by implementing explicit resource-awareness into the scheduling decisions. We restrict the discussion to static priorities.

Priority Inheritance Protocol

The *Priority Inheritance Protocol* (PIP) [43] directly addresses the priority inversion problem by temporarily assigning (inheriting) the highest priority of all tasks waiting for a resource to the task currently holding it. Figure 3.8 illustrates the scheduling scenario

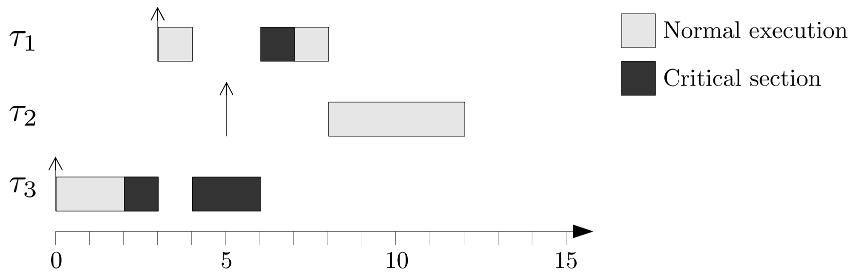


Figure 3.8: Example of the priority inheritance protocol

from Figure 3.7 under priority inheritance. When τ_1 attempts to enter its critical section in time step 4, τ_3 inherits the priority of τ_1, thus preventing τ_3 to be preempted by τ_2 in the following.

Generally, under PIP a task τ_i has a *nominal priority* $\mathrm{pr}(i)$ and an *active priority* $\mathrm{pr}^\star(i)$. If $\mathrm{pr}^\star(i)$ is the highest priority among all active tasks, τ_i is run. If τ_i attempts to enter a critical section and is blocked by a task τ_j, active priorities are assigned such that $\mathrm{pr}^\star(j) = \mathrm{pr}^\star(i)$. If τ_i leaves a critical section, $\mathrm{pr}^\star(i)$ is set to the maximal active priority among all tasks still blocked by τ_i (nested critical sections), or to $\mathrm{pr}(i)$ otherwise.

PIP prevents priority inversion and thus bounds the amount of blocking but still suffers from two problems:

- *chained blocking*: If a high priority task τ_i is bound to successively enter n critical sections protected by n different semaphores, in the worst-case it is blocked by n

critical sections of lower priority tasks. Thus blocking time, while being bounded, is potentially still significant.

- *deadlock*: If two tasks both enter critical sections and, in the following, attempt to acquire the semaphore held by the other additionally, a cyclic wait situation can occur.

Priority Ceiling Protocol

The *Priority Ceiling Protocol* (PCP) [43] is an extension to PIP and prevents priority inversion, chained blocking and deadlocks. The intuition is that a task is only allowed to enter a critical section if it can be guaranteed that it will not be blocked before leaving.

Let S_i denote a semaphore, then the (static) *priority ceiling* value $C(S_i)$ is the maximal priority among all tasks that potentially acquire S_i during execution. Further, let S_* dynamically denote the semaphore with the highest ceiling priority among all semaphores (at a specific point in time) and nominal respectively active priorities (cf. PIP).

The task τ_i of highest priority $\mathrm{pr}^\star(i)$ is run. Task τ_i is allowed to enter a critical section only if $\mathrm{pr}^\star(i) > C(S_*)$. Otherwise it is blocked and $\mathrm{pr}^\star(j) = \mathrm{pr}^\star(i)$ for the task τ_j holding S_*. If τ_i leaves a critical section, $\mathrm{pr}^\star(i)$ is set to the maximal active priority among all tasks still blocked by τ_i (nested critical sections), or to $\mathrm{pr}(i)$ otherwise.

PCP bounds the amount of blocking inflicted to a task τ_i to the longest critical section among all lower priority tasks τ_j that share semaphores S_k with τ_i and $C(S_k) > \mathrm{pr}(i)$.

3.2 Timing Analysis

Schedulability theory provides a means to guarantee feasibility of a system under given timing constraints. Failing to meet such constraints in hard real-time systems leads to system failure. To provide schedulability analysis with the necessary timing parameters, timing analysis per task is required. The computed values need not only be sound to ensure correctness of timing guarantees, but should also be tight to be of practical use. In the following will briefly define the objective of timing analysis. In Section 3.2.1, we briefly cover important aspects of practical analysis with a focus on hardware- and software-related issues, and we sketch a typical tool chain for timing analysis. Execution time ET of a program depends not only on its input $I_0 \subseteq I$ but also on the initial system state $S_0 \subseteq S$. Let $\pi_i \in \Pi$ denote the execution path for input $i \in I_0$. Then *best-case execution time* (BCET) is defined as:

$$\mathrm{BCET} = \min_{i \in I_0} \min_{s \in S_0} \mathrm{ET}(\pi_i, s) \tag{3.24}$$

And *worst-case execution time* (WCET) is defined as:

$$\mathrm{WCET} = \max_{i \in I_0} \max_{s \in S_0} \mathrm{ET}(\pi_i, s) \tag{3.25}$$

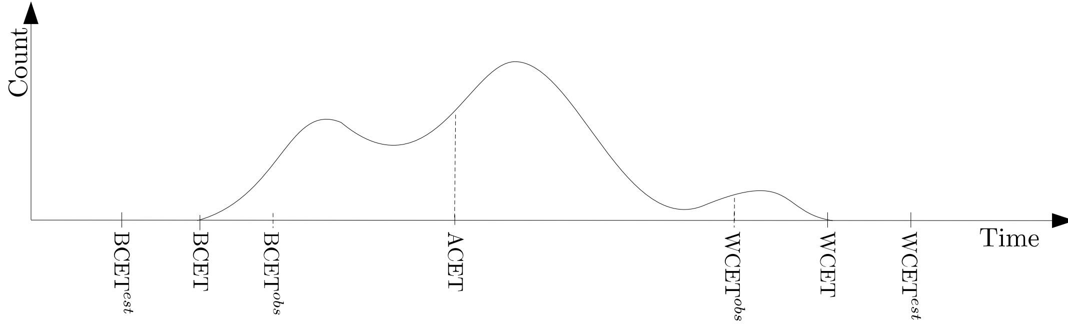


Figure 3.9: Worst-case and best-case execution times compared to observed and estimated bounds

where $\mathrm{ET}(\pi_i, s)$ denotes execution depending on control flow and initial system state. Consequently, given uniform distribution of input and initial state, the *average-case execution time* (ACET) is defined as:

$$\mathrm{ACET} = \left(\sum_{i \in I_0} \sum_{s \in S_0} \mathrm{ET}(\pi_i, s) \right) \times (|S_0| + |I_0|)^{-1} \tag{3.26}$$

In practice, it is infeasible to enumerate all executions for all feasible states and inputs to derive actual execution time bounds. Then two options are available. First, we can restrict the set initial states and inputs to a practically manageable size. Then the observed concrete execution time bounds, BCET_p^{obs} and WCET_p^{obs}, might neither be safe nor precise. Second, we can apply abstract interpretation. We construct abstract models of components affecting program timing, which typically yield small and well-defined initial best-case and worst-case states, and we abstract from control flow by means of CFGs to remove some of its input dependence. Then sound but potentially less precise bounds can practically be computed due to much reduced states spaces:

$$\forall i \in I_0 \colon \forall s \in S_0 \colon \mathrm{BCET}^{est} \leq \mathrm{ET}(\pi_i, s) \leq \mathrm{WCET}^{est} \tag{3.27}$$

Figure 3.9 illustrates the relation of the different timing values.

We will be only concerned with the computation of timing estimates by static timing analysis based on abstract interpretation. Note that for convenience, we will refer to WCET^{est} (BCET^{est}) by just WCET and BCET, respectively, unless stated explicitly otherwise.

3.2.1 Practical Aspects

In the following we shall briefly discuss various generally practical aspects of static timing analysis.

Hardware-related Aspects

Soundness and accuracy of timing analysis critically depends on the availability of appropriate models for the hardware under analysis. Leaving software aside, two system components are of particular interest: i) The CPU pipeline, which typically includes its bus systems. ii) The cache subsystem, where analysis concerned with the classification of accesses into hits and misses.

Pipeline analysis critically depends on the features of the CPU pipeline. Out-of-order execution and dynamic branch prediction (speculative prefetching) significantly lower the predictability of a system as execution time then increasingly depends on execution history. Highly predictable pipelines are therefore short and yield no dynamic speculative execution. For static analysis, this implies significantly simpler models and a more efficient analysis at a potentially reduced loss of information due to abstraction. The typical trade-off here is between predictability and average case performance [44–46]. In this thesis we are not concerned with aspects of pipeline-architectural analysis [47–49] and therefore omit its detailed discussion.

Cache analysis is concerned with deriving classifications of memory accesses into cache hits or misses. Semantics in this model is restricted to cache logic. Analysis results affect execution time estimation by denoting whether accesses cause additional reloads from subsequent levels in the memory hierarchy. Cache analysis is not only relevant for time estimation in uninterrupted execution, but is in particular central for bounding interference effects in preemptive scheduling scenarios. We will devote the following Chapter 4 to the details of static cache analysis. We refer to pipeline and cache analysis cumulatively as *micro-architecture analysis*, unless stated otherwise.

We shall clarify some terminology in this context: Particular hardware features yield *timing anomalies* [50] during execution. This refers to effects such as a local worst-case positively affecting the global worst-case; or vice versa. For example, in conjunction with speculative prefetching, a cache miss might cause a globally reduced WCET [51]. In addition, *domino effects* [50, 52] can occur. This refers to scenarios where the difference in execution time for an execution path starting in the same location but under different initial hardware states is not constant-bounded but is proportional to the path length. An example for such an effect is hardware states that do not converge within program loops. Cache replacement policies, such as FIFO and PLRU [53, 54], are prone to domino effects. Accordingly micro-architectures are classified by their respective predictability [46]: i) *Fully timing compositional* architectures neither yield timing anomalies nor domino effects. ii) *Constant-bounded compositional* architectures are susceptible to timing anomalies but do not exhibit domino effects. iii) *Non-compositional* architectures exhibit domino effects and timing anomalies. Compositionality refers to the ability to safely analyze system components separately.

Software-related Aspects

Sound and precise timing analysis is also dependent on information about the high-level semantics of a program and not necessarily on hardware-related information. An inherent problem is the availability of a sufficiently rich program representation. To perform timing analyses at micro-architectural level, typically the only option is the recovery of information from a program binary, which contains instructions, constant data and a description of the program memory layout. A key problem is the recovery of a sound and sufficiently precise CFG [25]. At this low level of representation, the CFG has to be recovered from a stream of binary data which involves decoding of instructions and the discovery of jump targets for basic block reconstruction. Depending on high-level language semantics, targets can be dynamic and CFG precision critically depends on precise control flow analysis [25, 26].

Most hard real-time software is highly static and therefore typically allows for relatively precise *control flow reconstruction* without explicit *value analysis* to determine jump targets. Nevertheless, value analysis provides valuable information about potential memory accesses or invariant CFG branch conditions to rule out infeasible paths. In addition, the availability of valuation of loop index variables also enable the automatic derivation of *loop bounds* for loop iterations and recursions [55–57]. Information on path infeasibility and iteration bounds is commonly referred to as *flow facts*. While micro-architectural analysis yields worst-case (best-case) time bounds for individual program points, computation of final timing estimates is performed by a step referred to as *path analysis*, which derives global timing estimates from local timings, the CFG and flow facts. We will devote Chapter 5 to this topic specifically.

A Typical Toolchain for Timing Analysis

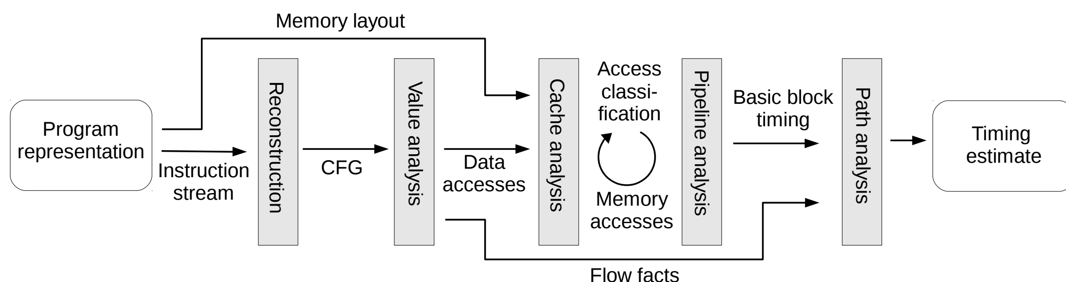


Figure 3.10: Typical tool chain for timing analysis

Figure 3.10 illustrates a work flow for static timing analysis similar to the commercial *aiT timing analyzer* [58]. While it is in principle possible to combine analysis stages, they are typically separated for efficiency reasons [47], although a combined analysis would potentially yield greater accuracy. For example, static analyses of values, caches and pipeline are oblivious of flow facts and therefore unnecessarily take infeasible paths into consideration. Only in path analysis, infeasible results might be pruned from the result set. For cache and pipeline analysis, loss of accuracy can be unacceptable with

an unidirectional information flow [59]: Pipeline timing potentially critically depends on cache analysis results and cache analysis depends on memory accesses issued by the pipeline. A survey of practices and tools and details of static timing analysis can be found in [4, 60]. Other timing analyzers [61–65] are similarly structured.

Chapter 4

Cache Analysis

Contents

4.1 Computer Memories **40**
4.2 Processor Caches **41**
4.3 Cache Logic **43**
4.4 Static Cache Analysis **44**
4.4.1 LRU Cache Semantics 45
4.4.2 Access Classification 46
4.4.3 Abstraction 47
4.5 Multitask Timing Analysis **49**
4.5.1 Costs of Preemption 49
4.5.2 Cache-related Preemption Delay 50
4.5.3 Bounding Cache-related Preemption Delay 51
4.6 Synergetic Approach to CRPD Analysis **58**
4.6.1 Precise Cache Analysis 58
4.6.2 Computation of UCB, ECB and CBR 61
4.6.3 Restriction to Basic Block Boundaries 64
4.6.4 CRPD Bounds on Task Sets 66
4.7 Evaluation **72**
4.8 Conclusion **77**

Of the timing analysis stages presented Chapter 3, cache analysis potentially has the greatest impact on overall program timing as access latencies vastly differ depending on the memory accesses, and worst-case timings in the memory subsystem greatly exceeds that of components not related to storage. Not only is cache analysis critical for the analysis of uninterrupted execution of single tasks (*task-level analysis*) but also plays a critical role for timing analysis in multitask scenarios to bound task interference — an issue we have not previously addressed.

This chapter first provides a study to static cache analysis in general and how it is applied to single-task and multitask timing estimation. Second, beyond the study

of existing static analysis and its application, we propose a highly accurate cache analysis framework which combines the most precise approaches to cache analysis and the estimation of cache-related preemption costs to date. The aim is to address and improve weaknesses inherent to some approaches and evaluate is applicability in an overall framework.

In the following we recap the basics of computer memories in general in Section 4.1, of caches in particular in Section 4.2 and the fundamentals of cache logic in Section 4.3. We then formalize static cache analysis in Section 4.4. In Section 4.5 we discuss various aspects of multitask timing analysis. We then propose our analysis framework for static cache analysis, along with extensions for bounding multitask scenarios in Section 4.6.

4.1 Computer Memories

The speed difference of CPU-bound and I/O-bound operations is known as the *memory gap*. It was historically steadily increasing due to the pace of CPU clock rate increments as compared to clock rates possible to random access memory technologies of that time. While the increase of CPU clock rates stalled due to technological limitations, today the rise of on-chip multi-processor systems even accelerates this trend. Memory access is the key performance bottleneck today [66]. We will now briefly provide technical background to memory systems.

Memory Technologies

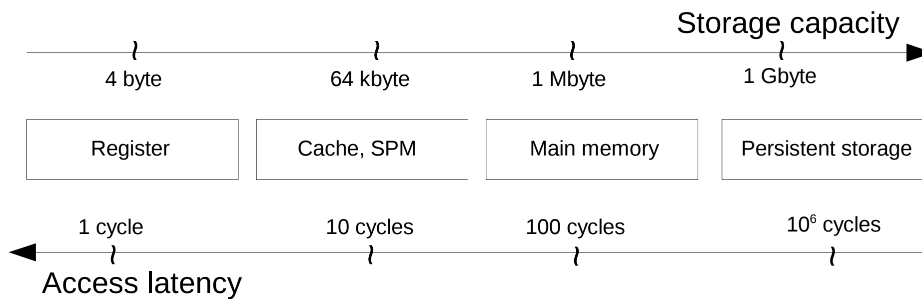


Figure 4.1: Illustration of accesses latencies and storage capacities of memories in a typical memory hierarchy

Different random access memory technologies available today exhibit trade-offs between performance — in terms of latency and bandwidth – and storage capacity. The fastest storage available for a CPU is its register file, which is clocked at the CPU frequency. The memory technology is *static RAM* (SRAM), a transistor-based storage which is fast but expensive both in terms of monetary costs and on-chip area. While registers are fixed storage locations, some embedded architectures feature *scratchpad memories* (SPM) — based on SRAM and in close proximity to the CPU — which are randomly addressable. High capacity random access memory is typically capacitor-based *dynamic RAM* (DRAM). As opposed to SRAM, DRAM cannot be clocked at rates

comparable to CPU clock rates and in addition require refresh cycles to maintain charges, severely impacting both performance as well as timing predictability [46]. SRAM and DRAM storage is fast but comparably small and non-persistent. Cheap, persistent but also slow local storage solutions are either *flash memory*, based on so-called floating-gate transistor technology, or traditional disk-based solutions. From registers to the slowest level of memories, performance and costs (by either metric, per bit of storage) drop exponentially, as illustrated in Figure 4.1. For a more thorough overview of memory technologies see [67, 68].

Memory Hierarchies

To counter the memory gap, different memory types are laid out in an hierarchy from memory close to the CPU (fast and expensive) to off-chip memories (large and cheap). The underlying principle of why performance is seemingly ever increasing despite slow memories is *locality of reference*. *Spatial locality* denotes the fact that memory locations in proximity to previously accessed locations are likely to be accessed as well. *Temporal locality* denotes that recently accessed locations are likely to be accessed again. Spatial locality occurs because most kinds of memory accesses are not truly random but often form linear accesses sequences, such as reading a sequence of CPU instructions, which motivates prefetching data from memory in larger blocks for increased efficiency. Temporal locality is caused by program loops and motivate the use of fast but small memories as caches, temporarily maintaining a *working set* of memory contents of slower but larger memories. Caches can be laid out in multiple levels of memories of increasing size (but decreasing speed) to hide individual latencies. This principle does not only apply to general-purpose memories but equally applies to more specialized components such as buffers of branch targets (branch-target buffers, BTB) or address translations for virtual memory mappings (translation look-aside buffers, TLB).

In the following we assume a memory hierarchy consisting of just a single cache-level for general memory accesses, which we refer to as *processor caches* or simply caches.

4.2 Processor Caches

We now provide technical background for cache memories specifically.

The kind of embedded systems we are concerned with in the following feature a single-level of caching of DRAM or flash memory, partitioned into separate data and instruction caches, which we refer to as *main memory* in the following. Scratchpad memories are not common. While both, SPM and caches, are based on SRAM, only SPM are randomly accessible. Cache memory is managed by a *cache logic*, that exclusively controls its contents depending on the accesses issued by the CPU. Since a cache maintains only a small image of a larger memory, the *replacement policy* of the cache logic determines what cache contents shall be maintained and for how long. All accesses pass through the cache and either cause a *cache hit*, if the requested data is available from the cache

or a *cache miss* otherwise. Upon a miss, data has to be loaded from the next lower memory level, causing a delay referred to as *cache miss penalty* (or *block reload time*). We distinguish three types of misses [67]: i) A *compulsory miss* is caused by a request to data has never been cached before and denotes the very first access. ii) A *capacity miss* occurs when the cache size is smaller than the current working set and is caused by insufficient hardware resources. iii) A *conflict miss* denotes a non-compulsory miss despite sufficient storage capacity in the cache and is caused by sub-optimal interaction of access requests, cache logic and memory layout of data.

For write operations from the CPU to main memory, two strategies are typically employed: i) *Write-back*: A datum is stored in the cache and is specially marked. Only once it is evicted from the cache, it is written to main memory. ii) *Write-through*: a datum is instantly written to the cache and main memory.

If a datum is not present in the cache when its address is written to (*write miss*), the datum can either be loaded into the cache, modified and written back (*write/allocate*), or the cache is not altered and changes are directly being written into main memory (*write/no-allocate*). Write-through improves timing predictability as main memory accesses can be accounted for instantly upon access as opposed to the analysis of lazy writes of write-back.

Geometry

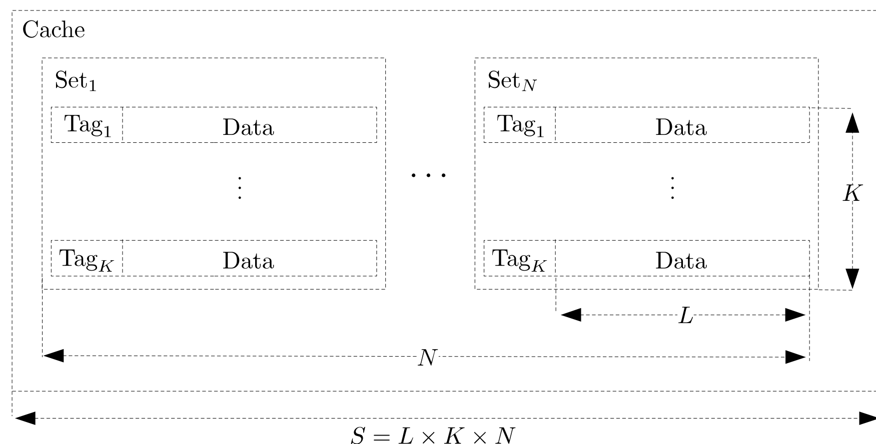


Figure 4.2: Conceptual layout of a cache memory and its geometry

A cache is organized in *cache lines* of size L which store the contents of a particular *memory block* of the cached storage. A *cache set* contains K cache lines, where K is referred to as the *associativity* of the cache. Each memory block maps to exactly one cache set. A cache itself is composed of N cache sets. Thus, it's storage capacity equals $S = L \times K \times N$. Figure 4.2 illustrates the geometry of such a *K-way set-associative* cache. Such a cache is called *direct-mapped* if $K = 1$ and *fully-associative* if $K = N$.

Arithmetic

Cache geometry parameters are given in powers of two. The bits of the address A_M of a memory block then determine the mapping into cache sets. A memory address is interpreted as the tuple

$$A_M = (T_M, I_M, O_M)$$

where *tag* T_M uniquely identifies a block within a set, *set index* I_M denotes which set a block is mapped to and *block offset* O_M determines which part of a block is actually being accessed [67]. For static cache analysis, only the cache set being accessed is of relevance. Thus, we can abstract from address arithmetic in the following and simply restrict the discussion to cache set accesses.

4.3 Cache Logic

In the following we will briefly address the most important cache replacement policies.

Since caches only map a subset of a larger memory and should maintain the current working set dynamically, eventually memory blocks are evicted from the cache when being updated. A *replacement policy* determines which blocks to evict from individual cache sets, which can be seen as queues ordered and updated in a particular fashion. While line size L caters spatial locality by causing updates of memory blocks — which are typically a lot larger than the actually requested data size — the replacement policy caters temporal locality by classifying blocks according to a notion of age.

The optimal replacement policy OPT [69] minimizes the number of cache misses by evicting only those blocks that will not be accessed for the longest time in the future and is necessarily an offline policy. Online policies differ in their heuristics to approximate OPT, differ in complexity of the hardware circuitry to implement them and differ in their predictability in terms of static timing analyses. We only informally describe the heuristics of the most prominent ones. For details and examples, refer to [19, 53].

- *Least Recently Used* (LRU) orders blocks in a set according to their age, which denotes the number of accesses to other blocks since the last access to this block. Upon conflict, the oldest block is evicted. Upon access, the age is reset. According to the principle of locality, a block that has not been accessed for a relatively extended period of time is less likely to be accessed again, while a block that has recently been accessed is likely to be accessed again.
- *Pseudo LRU* (PLRU) approximates LRU but is less complex in terms of hardware implementation for increasing associativities. Age is modeled in terms of a binary tree of depth $K - 1$, with leafs representing cache lines. A path in the tree is maintained which points to the block to be evicted next. At every junction, a 0 denotes the left subtree, a 1 denotes the right one. Upon access, all path bits are flipped away from the leaf to model age reset as in LRU.

	Cache geometry
L	Cache line size
K	Cache set associativity
N	Number of cache sets
$S = L \times K \times N$	Cache size
	Cache states
$\mathcal{M}$	Memory block
$\mathcal{M}_\perp = \mathcal{M} \cup \{\perp\}$	Cache block
$\mathcal{C} = (\mathcal{C}_s)^N$	Cache state
Age $= \{0, \ldots, K-1, \infty\}$	Block age

Table 4.1: Cache-related definitions

- *First-in first-out* (FIFO) models a queue such that upon a miss the newly loaded block is stored at the queue head and all existing elements age by one. A hit does not change the queue as opposed to LRU.
- *Most Recently Used* (MRU) counter-intuitively does *not* replace the most recently used element. Each element is attached a status bit which is set to 1 upon access to indicate a recent use. If all bits are set to 1, all bits but the most recent one are reset to 0. Upon a miss, a line with a 0 bit is replaced.

Of the listed policies, LRU is the most predictable one and is therefore of great interest for the construction of time critical systems. Key properties are, that the logic is not sensitive to whether a memory access is a hit or a miss, and, regardless of a specific initial state of a cache set, the state can be precisely determined after any sequence of K memory accesses [19]. In this thesis, we will focus on this policy. PLRU is hard to predict because an access does not only protect the element but also its neighbors from eviction. This may lead to the indefinite survival of an element. FIFO reduces predictability by treating hits and misses asymmetrically and therefore critically depends on the precision of access classification by static analysis. Similarly, MRU is hard to predict due to the dependence on precise information on status bits. PLRU, FIFO and MRU are prone to domino effects [54, 70]. In particular, they are sensitive to specific initial cache states and are therefore highly unpredictable in preemptive scenarios where unknown initial states do not only have to be taken into account at the start of a task but throughout its entire execution — after each potential preemption. For a detailed survey of policies, performance and predictability in task-level analysis see [19].

4.4 Static Cache Analysis

We now formally study static cache analysis under the LRU replacement policy which will serve as the formal basis to present our approach to estimate task timing under preemption. First, we formalize LRU cache semantics and discuss aspects of its static analyses for non-preemptive executions. Table 4.1 lists related definitions. In Section 4.4.1

we formalize basic semantics, in Section 4.4.2 we define how semantics are computed and it relates to timing analysis. In Section 4.4.3 we briefly introduce abstract cache semantics.

4.4.1 LRU Cache Semantics

We now formally define LRU cache semantics. Without loss of generality, we restrict ourselves to semantics of individual caches sets. Let $\mathcal{M}$ denote the set of memory blocks and let $\mathcal{M}_\perp = \mathcal{M} \cup \{\perp\}$ denote memory blocks mapped in a cache (cache blocks), where $\perp$ denotes an invalid cache line. A cache *set* state is a K-tuple

$$\mathcal{C}_s = (\mathcal{M}_\perp)^K \tag{4.1}$$

such that for a state $(b_0, \ldots, b_{k-1}) \in \mathcal{C}_s$, block b_0 denotes the youngest and b_{k-1} denotes the oldest element. Let set $\text{Age} = \{0, \ldots, K-1, \infty\}$, then we define:

$$\begin{aligned} \text{age}\colon \mathcal{M} \times \mathcal{C}_s &\mapsto \text{Age} \\ \text{age}(m, (b_0, \ldots, b_{k-1})) &= \begin{cases} i & \text{if } m = b_i \\ \infty & \text{otherwise} \end{cases} \end{aligned} \tag{4.2}$$

to denote the age of a block, where ∞ denotes that a block is not cached. Under LRU, if a block m is accessed, all blocks of lower age than m that are mapped to the same set age by one. Formally, we define the LRU update policy as:

$$\begin{aligned} \text{lru} : \mathcal{M} \times \mathcal{C}_s &\mapsto \mathcal{C}_s \\ \text{lru}(m, (b_0, \ldots, b_{k-1})) &= \begin{cases} (m, b_0, \ldots, b_{i-1}, b_{i+1}, \ldots b_{k-1}) & \text{if } m = b_i \\ (m, b_0, \ldots, b_{k-2}) & \text{otherwise} \end{cases} \end{aligned} \tag{4.3}$$

We now define the corresponding collecting semantics. Let $\iota\colon V \mapsto \mathcal{M}$ denote a memory access at a program point. Then we define a transfer function as:

$$\begin{aligned} \text{tf}^{lru}\colon V &\mapsto \mathcal{C}_s \mapsto \mathcal{C}_s \\ \text{tf}^{lru}(u) &= \lambda c \,.\, \text{lru}(\iota(u), c) \end{aligned} \tag{4.4}$$

such that semantics on a path $\pi = (u_1, \ldots, u_i)$ correspond to:

$$[\![\pi]\!](\text{tf}^{lru}) = \begin{cases} \text{id} & \text{if } \pi = \epsilon \\ \text{tf}^{lru}(u_i) \circ [\![(u_1, \ldots, u_{i-1})]\!](\text{tf}^{lru}) & \text{otherwise} \end{cases} \tag{4.5}$$

Accordingly, collecting cache semantics for a set of initial cache states $C_0 \subseteq \mathcal{C}_S$ and paths Π is defined as:

$$\text{col}^{lru}\colon V \mapsto \wp(\mathcal{C}_s)$$
$$\text{col}^{lru}(u_i) = \bigcup_{c_0 \in C_0} \left\{ [\![(u_i, \ldots, u_{j-1})]\!] (\text{tf}^{lru})(c_0) \;\middle|\; (u_i, \ldots, u_j) \in \Pi \right\} \tag{4.6}$$

For worst-case estimations, the set of initial states contains just the empty cache, which models the worst-case initial state for LRU[1] [19].

The computation of fixed point semantics relies on the transformer defined in Equation 4.4 and the join semi-lattice $(\wp(\mathbb{D}_\mathcal{C}), \top, \subseteq, \cup)$ where $\mathbb{D}_\mathcal{C} = \mathcal{C}$. Hence

$$\text{mfp}^{lru}_\mathcal{C}\colon V \mapsto \wp(\mathcal{C}) \tag{4.7}$$

denotes its fixed point solution. Note that this domain is very precise since although execution context is lost due to loss of concrete execution paths in the MFP solution, cache state itself retains limited execution history.

4.4.2 Access Classification

The original purpose of cache analysis is the classification of accesses ι into cache hits and misses such that the costs of memory accesses in a program point for micro-architectural analysis can be accounted for accordingly. For a given access $\iota(u)$, we distinguish three cases: i) *always hit* (ah) if the access guaranteed to be a cache hit in all reachable cache states ii) *always miss* (am) if the access is guaranteed to be miss in all reachable cache states iii) *not classified* (nc) otherwise. We formalize this notion according to the given semantics and define *access classification* acl as:

$$\text{acl}\colon \mathcal{M} \times V \mapsto \{am, ah, nc\}$$
$$\text{acl}(m, u) = \begin{cases} ah & \text{if } \forall c \in \text{col}^{lru}(u)\colon m \in c \\ am & \text{if } \forall c \in \text{col}^{lru}(u)\colon m \notin c \\ nc & \text{otherwise} \end{cases} \tag{4.8}$$

An access is classified *ah* if a block is guaranteed to be cached for all paths to point u, and *am* if is it guaranteed to be uncached. Otherwise, no classification can be given. This gives rise to the notion of *must* set, which denotes memory blocks that must be cached in a point, defined as:

$$\text{must}_s\colon V \mapsto \wp(\mathcal{M})$$
$$\text{must}_s(u) = \{m \mid \text{acl}(m, u) = ah\} \tag{4.9}$$

[1]This is not necessarily true for other replacement policies [19].

Symmetrically, a *may* set denotes all memory blocks that might be cached in some point (not always miss) and which is defined as:

$$\begin{aligned} \text{may}_s &: V \mapsto \wp(\mathcal{M}) \\ \text{may}_s(u) &= \{m \mid \text{acl}(m, u) \neq am\} \end{aligned} \tag{4.10}$$

Note that due to compulsory misses, accesses would never be classified *always hit* in a program point given a standard CFG, which does not distinguish execution context (the very purpose of CFGs). The problem can be mitigated by adding explicit context back into the CFG by *virtual function inlining, virtual loop unrolling* (VIVU) [26] or *persistence analysis* [71].

From access classification, bounds on the number of cache hits and misses along a path can be computed. Let $\pi = (u_1, \ldots, u_k)$ denote a path. Then an upper bound on the number of cache misses is given by counting all accesses classified *ah* such that:

$$\text{miss}_s : \Pi \times \mathcal{C}_s \mapsto \mathbb{N}_0$$
$$\text{miss}_s(\pi) = \begin{cases} \text{miss}_s(\pi \setminus (u_k)) + \begin{cases} 1 & \text{if } \text{acl}(\iota(u_k), u_k) \neq ah \\ 0 & \text{otherwise} \end{cases} & \text{if } \pi \neq \epsilon \\ 0 & \text{otherwise} \end{cases} \tag{4.11}$$

We denote miss counts for complete cache states by $\text{miss}: \Pi \times \mathcal{C} \mapsto \mathbb{N}_0$ in the following. Symmetrically, bounds for cache hits can be defined which will be of no interest to us.

4.4.3 Abstraction

An more efficient but less precise abstraction for LRU cache semantics has been proposed in [72]. The abstraction $\mathcal{A}: \mathcal{M} \mapsto \text{Age}$ from cache states $\mathcal{C}$ exploits that for may and must set computation we are effectively only interested in the maximal and minimal block ages among all states. For example, a block is contained in a may set if there exists at least one maximal age among all states less than associativity K. Since ages are only approximated, abstract may and must set computation requires two distinct analyses. Inversely, this holds for must sets. We shall formalize for reference.

Let $C \in \wp(\mathcal{C}_s)$ denote a set of cache states. For *may analysis*, abstraction α_{may} extracts the minimum age from C and is defined as:

$$\begin{aligned} \alpha_{may} &: \wp(C_s) \mapsto \mathcal{A} \\ \alpha_{may}(C) &= \lambda m \,.\, \min_{c \in C} \text{age}(m, c) \end{aligned} \tag{4.12}$$

Concretization γ_{may} returns a set of concrete states such that the age of each memory block m is at least $\alpha_{may}(C_s)$:

$$\gamma_{may}\colon \mathcal{A} \mapsto \wp(\mathcal{C}_s)$$
$$\gamma_{may}(\widehat{a}) = \{c \in \mathcal{C}_s \mid \forall m \in \mathcal{M}\colon \widehat{a}(m) \leq \text{age}(m, c)\} \tag{4.13}$$

Transformer tf^{lru}_{may} redefines the mapping from blocks to ages $\widehat{a} \in \mathcal{A}$ according to lru and is defined as:

$$\text{tf}^{lru}_{may}\colon V \mapsto \mathcal{A} \mapsto \mathcal{A}$$
$$\text{tf}^{lru}_{may}(u)(\widehat{a}) = \lambda m \, . \begin{cases} 0 & \text{if } m = \iota(u) \\ \widehat{a}(m) & \text{if } \widehat{a}(m) > \widehat{a}(m)(\iota(u)) \\ \widehat{a}(m) + 1 & \text{if } \widehat{a}(m) \leq \widehat{a}(m)(\iota(u)) \wedge \widehat{a}(m) < K - 1 \\ \infty & \text{otherwise} \end{cases} \tag{4.14}$$

Informally, the cases distinguished in the transformer are:

1. If a cached block is accessed, its age is reset.
2. All blocks older than the currently accessed block do not change their age.
3. All younger blocks age by one.
4. Blocks with ages exceeding the associativity do not need an exact age.

May analysis bases on the join semi-lattice $(\mathcal{A}, \top, \sqsubseteq, \sqcup)$ where $\sqsubseteq_{may}$ is defined as:

$$\widehat{a}_1 \sqsubseteq_{may} \widehat{a}_2 \Leftrightarrow \forall m \in \mathcal{M}\colon \widehat{a}_1(m) \leq \widehat{a}_2(m) \tag{4.15}$$

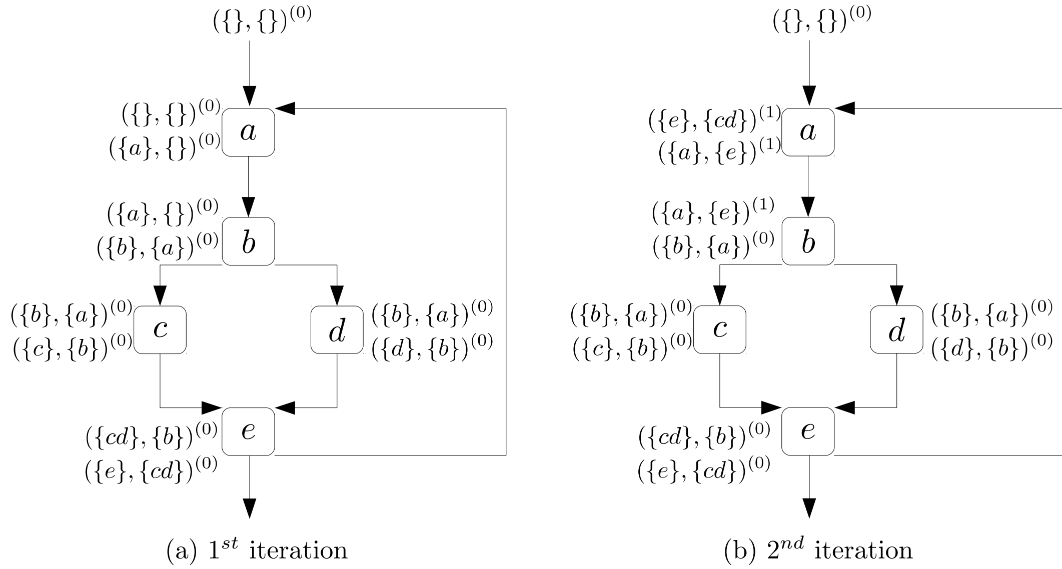
and $\sqcup_{may}$ is defined as:

$$\sqcup_{may}\colon \mathcal{A} \mapsto \mathcal{A} \mapsto \mathcal{A}$$
$$\widehat{a} \sqcup_{may} \widehat{a}' = \lambda m \, . \min(\widehat{a}(m), \widehat{a}'(m)) \tag{4.16}$$

Must analysis is symmetric in that we maintain only maximal ages, thus underestimating possible cache contents [72]. We do not formalize it here, as we will only be concerned with may analysis in the following.

This abstract domain is popular as it is efficient in terms of memory consumption as opposed to the power set domain above, which is critical for large memories and large software in particular. Note however that hard real-time systems and software are usually small and high accuracy of analysis is possibly more important than efficiency of analyses. Abstraction also comes with drawbacks. On the one hand, it inherently loses context and it is therefore not possible to distinguish mutually exclusive states as we will see later. Secondly, the domain is not distributive [19] (cf. Theorem 2.20). A trade-off between both domains has been proposed in [73].

Example *Figure 4.3 illustrates the fixed point solution under the abstract domain for*



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

Figure 4.3: Example of may set computation

a cache of associativity $K = 2$ *and number of sets* $N = 1$. *We represent cache states as tuples of sets, which depicts the mapping of memory blocks with ages* $\forall u \in V : \forall m \in \mathcal{M} \colon \widehat{a}(u)(m) < \infty$. *The nodes are labeled with memory blocks accessed at the respective program points such that* $\iota(a) = a$. *Figure 4.3a and Figure 4.3b depict the two iterations required to obtain the MFP solution. Superscripts at the cache states denote the number of updates of the respective states. The initial cache state is the empty cache such that* $\forall u \in V : \forall m \in \mathcal{M} \colon \widehat{a}(u)(m) = \infty$. *Then each access ages all elements by one according to Equation 4.14 and at each join the minimal ages of elements is maintained according to Equation 4.16.*

4.5 Multitask Timing Analysis

Task-level timing analysis as discussed in Section 3.2 computes worst-case (best-case time) bounds for a single uninterrupted execution of a task. Focusing on cache-related timing aspects in Section 4.4, we showed how cache hits and misses can be accounted for to contribute to the WCET and BCET of a task. In the presence of preemptions, additional *context-switch costs* must be taken into account.

In Section 4.5.1 we introduce basic terminology and context and in Section 4.5.2 we formally define the notion of preemption costs specifically. In Section 4.5.3 we introduce the basic building blocks to bound preemption costs.

4.5.1 Costs of Preemption

A preemption causes additional context-switch costs which must be accounted for in static timing analysis to obtain safe timing bounds in multitask scenarios. Different types of costs need to be distinguished: i) *Scheduling costs* denote software overhead of a preemption which include scheduling decisions and context-saving [74–77]. ii) *Pipeline*

costs denote hardware-related costs due to the interruption of pipelined execution. iii) *Cache costs* denote cache-related costs due to interfering cache usage of multiple tasks. A severe limitation of schedulability tests from Section 3.1.4 is the assumption that these costs are negligible. Indeed, scheduling and pipeline costs are comparably low and can therefore be bounded by a constant in practice without being overly pessimistic. As far as cache costs are concerned, historically, the memory gap was negligible when processor clock rates used to be low. Now, cache-related costs are significant and highly dynamically affect time bounds. Only scheduling and pipeline costs can directly be associated with a particular preemption. *Cache-related preemption delays* (CRPD) — also referred to as *extrinsic cache interference* [78] — on the other hand only occur once evicted blocks are accessed at some point after a preemption. In the following, we focus on worst-case CRPD only, since best-case CRPD it can always safely be bounded by assuming no interference at all.

4.5.2 Cache-related Preemption Delay

We now formally define preemptions and CRPD, and briefly address related work to prevent CRPD altogether.

Formal Basics

Let $\pi_i \in \Pi_i$ denote a path of an uninterrupted execution of a task $\tau_i \in \mathcal{T}$. Assume τ_j is a preempting task of τ_i, then a *preemption* is a pair (u, π_j) of preemption point $u \in \pi_i$ and execution path $\pi_j \in \Pi_j$ of the preempting task. The preemption partitions $\pi_i = (\dots u_i, u_{i+1}, \dots)$ such that $\widehat{\pi}_i = (\dots, u) \cdot \pi_j \cdot (u_{i+1}, \dots)$ denotes an execution path under preemption.

In general, for a task τ_i, let $\mathrm{pre}_i \colon \Pi_\mathcal{T} \mapsto \Pi_\mathcal{T}$ map from uninterrupted to preempted execution paths and let $\mathfrak{c} : V \times \mathcal{T}^2 \mapsto \{\top, \bot\}$ denote a scheduling decision for preemption in a point. Then preempted execution paths are defined recursively as follows: Let $\mathrm{sched}_i \colon \pi_i \mapsto \Pi_\mathcal{T}$ be a function that maps from program points in a task τ_i to (possibly themselves preempted) execution paths of tasks τ_j, scheduled to preempt in a point u and which is defined as:

$$\mathrm{sched}_i(u, \mathfrak{c}) = \begin{cases} \mathrm{pre}_j(\pi_j) \colon \pi_j \in \Pi_\mathcal{T} & \text{if } \mathfrak{c}(u, \tau_i, \tau_j) \\ \epsilon & \text{otherwise} \end{cases} \tag{4.17}$$

Let $\pi_i = (u_1, \dots)$ denote an execution path of task τ_i. Then we can define function pre_i as:

$$\mathrm{pre}_i(\pi_i) = \begin{cases} (u_1) \cdot \mathrm{sched}_i(u_1) \cdot \mathrm{pre}_i(u_2, \dots) & \text{if } \pi \neq \epsilon \\ \epsilon & \text{otherwise} \end{cases} \tag{4.18}$$

which interleaves a non-preempted path of task τ_i with preempting paths. Thus, $\widehat{\pi}_i = \text{pre}(\pi_i)$ denotes a preempted execution path.

Definition 4.1 (Cache-related Preemption Delay [6]) *Let* $\text{miss}\colon \Pi \times \mathcal{C}_s \mapsto \mathbb{N}_0$ *denote the number of cache misses on a path and let* BRT *denote block reload time. Then CRPD is defined as the difference in misses of uninterrupted and preempted execution of a task. Formally:*

$$\text{crpd}_{\pi}(\pi) = (\text{miss}(\widehat{\pi}) - \text{miss}(\pi)) \times \text{BRT} \tag{4.19}$$

Let $\text{ET}(\pi)$ denote execution time on a path π, then *only* in fully timing compositional or constant-bounded compositional architectures [6], the follow inequation holds:

$$\text{ET}(\widehat{\pi}) \leq \text{ET}(\pi) + \text{crpd}_{\pi}(\pi) \tag{4.20}$$

The sum might indeed be greater than the actual execution time since block reloads might occur in parallel with processes such as arithmetic computations in a pipeline, which is not reflected by this separation. Note that the architectural constraint above implies restriction to the LRU replacement policy as FIFO, PLRU and MRU exhibit domino effects [19] and therefore the number of additional misses due to preemption is not bounded. Consequently, various techniques have been proposed to circumvent CRPD altogether.

Avoidance of CRPD

Different measures can be taken to avoid CRPD altogether by temporal or spatial isolation. With *cache locking* [79–81], cache logic is disabled temporarily or for the entire execution of a task so that the underlying memory serves as a fast read-only buffer. Alternatively, scratchpad memories can be used for similar purposes [82–85]. The trade-off between the two is that a cache provides a transparent address translation, thus instructions can be allocated into the memory unmodified since all references remain intact. However, dynamic replacement at runtime is not easily achieved. On the other hand, an SPM is mapped into the address space and is therefore directly dynamically accessible. *Cache partitioning* [86, 87] is the problem of computing an optimal memory layout to avoid cache conflicts. Despite trade-offs in performance, these strategies increase predictability as they eliminate CRPD as an additional source of imprecision without disabling caching.

In general, these techniques rely on special hardware support. Hence, CRPD cannot be avoided in those cases. We will therefore subsequently be concerned with the computation of safe and ideally tight approximations.

4.5.3 Bounding Cache-related Preemption Delay

In the following we will be concerned with standard techniques to bound CRPD. We unify different techniques in one formal framework and discuss specific inaccuracies in

their original definitions. We do not address practical analysis yet but constrain ourselves to concepts.

Useful Cache Blocks

To estimate CRPD, we are ultimately interested in the number of additional cache misses due to preemption. *Useful cache blocks* (UCB) [88] bound this number by computing sets of memory blocks potentially being accessed and reused during uninterrupted execution of a task. Intuitively, a preemption can not inflict more misses than cached blocks potentially being re-accessed.

Definition 4.2 (Useful Cache Block) *A memory block $m \in \mathcal{M}$ is is a* useful cache block *in a point $u_i \in V$ if and only if it is cached in u_i and reused in a point u_j such that $u_i \rightsquigarrow u_j$ and m is not evicted before u_j is reached.*

A safe but pessimistic bound on CRPD is given by the assumption that preemptions evict all UCB.

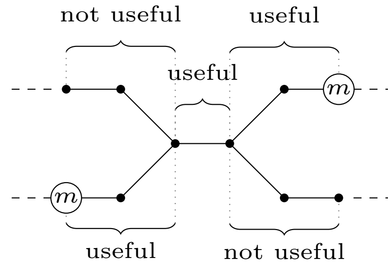


Figure 4.4: Useful Cache Blocks in a control flow graph

A set of UCB contains all memory blocks which *may* be cached at some point and *may* be reused at a later point in time. Figure 4.4 illustrates this notion. The white circles represent accesses to m along a control flow path. Block m is only classified "useful" at those program points from where an access to m is reachable in forward *and* backward direction without m being evicted along the path. We refer to the number of memory accesses distinct from m as *distance* between accesses to m.

We shall now formalize the notion of UCB. For an initial state $c_\emptyset \in \mathcal{C}_s$, a memory block $m \in \mathcal{M}$ is cached in a program point $u_i \in V$ if for a path $\pi = (u_0, \ldots, u_i, \ldots, u_n)$, it holds that:

$$\text{age}\Big(m, [\![(u_0, \ldots, u_i)]\!](\text{tf}^{lru})(c_\emptyset)\Big) \leq K - 1 \tag{4.21}$$

where age denotes the age of a block in a cache set state (Equation 4.2) and $[\![\pi]\!](\text{tf}^{lru})$ denotes the cache set semantics (Equation 4.4) on π. We define the set of all cached

blocks, given a path π as:

$$\begin{aligned} \mathrm{cb}_\pi &\colon \Pi \mapsto \wp(\mathcal{M}) \\ \mathrm{cb}_\pi(\pi) &= \left\{ m \,\middle|\, \forall m \in \mathcal{M} \colon \mathrm{age}(m, [\![\pi]\!](\mathrm{tf}^{lru})(c_\emptyset)) \leq K - 1 \right\} \end{aligned} \tag{4.22}$$

We define UCB via path-based collecting semantics. Recall that path-based semantics $\mathrm{col}_\pi^{\rightarrow/\leftarrow}$ (Definition 2.10 on page 11) denote sets of paths from CFG source to a point u or from CFG sink t u, respectively. Let UCB $\subseteq V \mapsto \wp(\mathcal{M})$ denote the function that map from points to sets of UCB, then we define:

$$\begin{aligned} \mathrm{ucb} &\colon \mathrm{UCB} \\ \mathrm{ucb}(u) &= \left\{ m \middle| \exists \pi \in \mathrm{col}_\pi^{\rightarrow}(u) \wedge \exists \pi' \in \mathrm{col}_\pi^{\leftarrow}(u) \colon m \in (\mathrm{cb}_\pi(\pi) \cap \mathrm{cb}_\pi(\pi')) \right\} \end{aligned} \tag{4.23}$$

For reference, let set: $m \mapsto [1, N]$ denote the cache set a memory block is mapped to, then we define the set of ucb in cache set s as:

$$\begin{aligned} \mathrm{ucb}_s &\colon V \mapsto \wp(\mathcal{M}) \\ \mathrm{ucb}_s(u) &= \{ m \mid m \in \mathrm{ucb}(u), \mathrm{set}(m) = s \} \end{aligned} \tag{4.24}$$

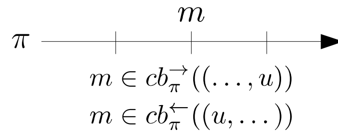


Figure 4.5: UCB overestimation due to a single access

Note that Equation 4.23 overestimates UCB as illustrated in Figure 4.5: Let memory block m be accessed in program point u, then m will be classified useful regardless of an actual reuse in another program point.

CRPD due to a preemption in a point u is then denoted by:

$$\begin{aligned} \mathrm{crpd}_u^{\mathrm{ucb}} &\colon V \times \mathrm{UCB} \mapsto \mathbb{N}_0 \\ \mathrm{crpd}_u^{\mathrm{ucb}}(u, \mathrm{ucb}) &= |\mathrm{ucb}(u)| \times \mathrm{BRT} \end{aligned} \tag{4.25}$$

In general, it is statically not possible to precisely determine the preemption point. Hence, a safe bound is denoted by [88]:

$$\begin{aligned} \mathrm{crpd}^{\mathrm{ucb}} &\colon \mathrm{UCB} \mapsto \mathbb{N}_0 \\ \mathrm{crpd}^{\mathrm{ucb}}(\mathrm{ucb}) &= \max_{u \in V} \mathrm{crpd}_u^{\mathrm{ucb}}(u, \mathrm{ucb}) \end{aligned} \tag{4.26}$$

Let # denote a bound on the number of preemptions, then a bound for the preempted execution of a task is given by:

$$\mathrm{WCET}^{est} + \mathrm{crpd}^{\mathrm{ucb}}(\mathrm{ucb}) \times \# \tag{4.27}$$

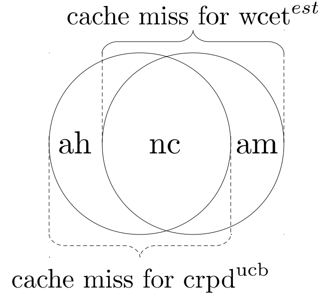


Figure 4.6: Relation of access classification (cf. Section 4.4.2) for CRPD and WCET

Recall from Section 4.4.2 that for safe WCET computation all memory accesses classified “always miss” and “not classified” (not cache on all paths) must be accounted for as cache misses in uninterrupted execution. Similarly, function cb_π denotes all blocks classified “always hit” or “not classified” (cached on at least one path). Consequently, function WCET^{est} and function crpd account for all accesses “not classified” twice, as illustrated in Figure 4.6. This gives rise to the notion of *definitely cached useful cache blocks* (DCUCB) [89] to mitigate redundancy in analysis, and which is defined as the constrained set of “useful” memory blocks classified as:

$$\begin{aligned} \mathrm{dcucb}\colon V &\mapsto \wp(\mathcal{M}) \\ \mathrm{dcucb}(u) &= \{m \in \mathrm{ucb}(u) \wedge \mathrm{acl}(m,u) = ah\} \end{aligned} \tag{4.28}$$

Note that Equation 4.27 is only safe if WCET and CRPD bounds are derived from the same cache analysis, which might not be possible in practice due to inaccessibility of proprietary WCET analyses frameworks.

Evicting Cache Blocks

Bound crpd^{ucb} denotes an approximation on the number additional cache misses due to preemption by any task. It does not take into account the accesses that are actually performed by individual preempters, leading to unnecessary pessimism. In [78, 90], the consideration of *evicting cache blocks* (ECB) is proposed.

Definition 4.3 (Evicting Cache Block) *A memory block $m \in \mathcal{M}$ is an* evicting cache block *if it is ever accessed on execution path π.*

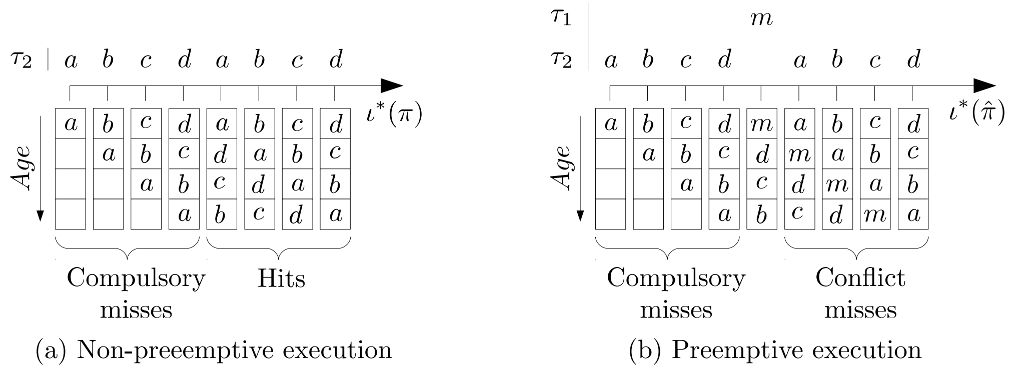
Consequently, let $\mathrm{ECB} \subseteq \wp(\mathcal{M})$ denote sets of memory blocks. Then the set of ECB is simply defined as the same of cached blocks along a path:

$$\begin{aligned} \mathrm{ecb}\colon &\mathrm{ECB} \\ \mathrm{ecb} &= \{m \in \mathrm{cb}_\pi(\pi) \mid \pi \in \Pi\} \end{aligned} \tag{4.29}$$

For reference, we define the set of ECB per cache set s as:

$$\begin{aligned} \text{ecb}_s &: \wp(\mathcal{M}) \\ \text{ecb}_s &= \{m \mid m \in \text{ecb}, \text{set}(m) = s\} \end{aligned} \tag{4.30}$$

Symmetrically to function ucb, ecb is an upper bound on the number of additional misses for *any* preempted task. It is important to recognize that under LRU, a single eviction can cause up to associativity K additional cache misses in a preemptee [91]. This needs to be taken into account when deriving CRPD bounds.



(a) Non-preeemptive execution

(b) Preemptive execution

Figure 4.7: Example of miss behavior under LRU and preemption

Example *Figure 4.7a illustrates the memory access sequence $\iota^*(\pi)$ of a task τ_2 and the contents of a cache set for $K = 4$ under LRU into which the memory blocks are mapped. All initial accesses necessarily lead to compulsory cache misses. After that, all accesses are hits. As opposed to this, Figure 4.7b illustrates the effect of a single access by a preempting task τ_1. All hits become misses since the accessed block has just been evicted before its access. Hence, the number of ECB is not a safe upper bound for additional cache misses.*

Consequently, a safe bound on CRPD is given by the number of invalidated *cache sets* times associativity K:

$$\begin{aligned} \text{crpd}^{\text{ecb}} &: \text{ECB} \mapsto \mathbb{N}_0 \\ \text{crpd}^{\text{ecb}}(\text{ecb}) &= |\{s \mid s \in N, \text{ecb}_s \neq \emptyset\}| \times K \times \text{BRT} \end{aligned} \tag{4.31}$$

Originally [90], Equation 4.31 has been proposed for direct mapped caches only. Its extension [92] to non-direct mapped caches, however, has been shown to be unsound [91]. The bound given above has not been proposed before to the best of the author's knowledge.

Cache Block Resilience

Equation 4.31 is based on a safe but not very precise bound on the number of additional misses as it does not take UCB into account. From the example illustrated in Figure 4.7 we conclude that a possible improvement by taking UCB and ECB into account can

be achieved by excluding cache sets with empty intersection of UCB and ECB [91]. Recall that otherwise we have to assume complete set invalidation to be safe. However, in [93], the authors recognize that by maintaining age information of UCBs, *cache block resilience* (CBR) with respect to preemption eviction can be computed to reduce over-approximation.

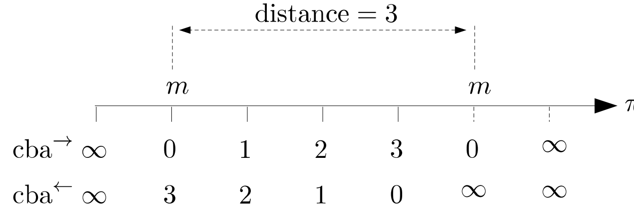


Figure 4.8: Example of block aging and distance metric of accesses

For an intuition, consider Figure 4.8 which illustrates two memory accesses to a block m on a path π. We ignore $\mathrm{cba}^{\rightarrow/\leftarrow}$ for the moment. The distance between the two accesses equals 3. If we assume an associativity $K = 4$, then the resilience of m between the two accesses is $(K - 1) - 3 = 0$ in all potential preemption points. Consequently, any additional access due to preemption causes a cache miss upon the next access to m. For any shorter distance between two accesses, a block's resilience increases: we can guarantee m is still cached after an access to some other block that maps into the same cache set if the distance is less than 3.

Definition 4.4 (Cache Block Resilience [93]) *The maximal amount of* additional accesses *distinct from $m \in \mathcal{M}$ by preempting tasks* without *causing a cache miss upon the next access to m is referred to as* cache block resilience *(CBR).*

Let us formalize this notion. To compute CBR, information on block ages need to be available in all program points. UCB (cf. Equation 4.23) do not include this information anymore. Let $\mathrm{CBR} \subseteq V \mapsto \mathrm{Age}$ denote the set of functions from program points to block ages. Then ages of memory blocks for a given execution path π are denoted by:

$$\mathrm{cba}_\pi \colon \Pi \mapsto \mathrm{CBR}$$
$$\mathrm{cba}_\pi(\pi) = \lambda m \,.\, \mathrm{age}(m, [\![\pi]\!](\mathrm{tf}^{lru})(c_\emptyset)) \tag{4.32}$$

Recall that uncached memory blocks yield an age of ∞ and that a memory block m may only be cached in point u if $m \in \mathrm{ucb}(u)$. The *maximal age* (distance in terms of memory accesses) of a useful block m in point u for all forward paths leading to that point is then defined as:

$$\mathrm{cba}^{\rightarrow} \colon V \mapsto \mathrm{CBR}$$
$$\mathrm{cba}^{\rightarrow}(u) = \lambda m \,.\, \begin{cases} \max\,\{\mathrm{cba}_\pi(\pi)(m) \mid \pi \in \mathrm{col}_\pi^{\rightarrow}(u)\} & \text{if } m \in \mathrm{ucb}(u) \\ \infty & \text{otherwise} \end{cases} \tag{4.33}$$

Symmetrically, the maximal aging from point u to its next access is defined as:

$$\text{cba}^{\leftarrow}\colon V \mapsto \text{CBR}$$
$$\text{cba}^{\leftarrow}(u) = \lambda m \,.\, \begin{cases} \max\left\{\text{cba}_\pi(\pi)(m) \mid \pi \cdot (u) \in \text{col}_\pi^{\leftarrow}(u)\right\} & \text{if } m \in \text{ucb}(u) \\ \infty & \text{otherwise} \end{cases} \tag{4.34}$$

We deviate from the original definition [93] in that we are collecting aging up to but not including respective program points in $\text{cba}^{\leftarrow}$ — as opposed to $\text{cba}^{\rightarrow}$ — to avoid overestimation in the original proposal. Figure 4.8 illustrates the respective valuation of $\text{cba}^{\rightarrow/\leftarrow}$.

CBR is the difference between the associativity (its maximal resilience) and the maximal distance between two accesses. We thus define a bound CBR_u^m for memory block m and program point u as:

$$\text{cbr}\colon V \mapsto \text{CBR}$$
$$\text{cbr}(u) = \lambda m \,.\, (K-1) - \min(K-1, \text{cba}^{\rightarrow}(u)(m) + \text{cba}^{\leftarrow}(u)(m)) \tag{4.35}$$

The sum of block ages denotes the distance, which is bounded by $K-1$ such that distances equal or greater than $K-1$ yield a CBR of 0.

CRPD in a program point u for a single preemption is then bounded by those whose UCB whose CBR is greater than the number of ECB. Formally, we define:

$$\text{crpd}_u^{\text{ucb,cbr,ecb}}\colon V \times \text{UCB} \times \text{ECB} \times \text{CBR} \mapsto \mathbb{N}_0$$
$$\text{crpd}_u^{\text{ucb,cbr,ecb}}(u, \text{ucb}, \text{cbr}, \text{ecb}) = \sum_s |\text{ucb}_s(u) \setminus \{m \mid \text{cbr}(u)(m) \geq \text{ecb}_s(u)|\}| \tag{4.36}$$

A globally safe bound for a preemption is then denoted by the maximal interference over all program points:

$$\text{crpd}^{\text{ucb,cbr,ecb}}\colon UCB \times ECB \times CBR \mapsto \mathbb{N}_0$$
$$\text{crpd}^{\text{ucb,cbr,ecb}}(\text{ucb}, \text{cbr}, \text{ecb}) = \max_{u \in V} \text{crpd}_u^{\text{cbr}}(u, \text{ucb}, \text{cbr}, \text{ecb}) \tag{4.37}$$

Summary

In this section we have provided an overview of the basic building blocks to bound CRPD and an initial notion of how these can be used to compute CRPD estimates. Note however, that the bounds so far only account preemptions by a single preempting task and merely serve the purpose of clarifying their conceptual use. For multiple preempters, in particular for bounds based on explicit interference of UCB and ECB, care has to be taken not to underestimate CRPD. We will address this subject in detail below. For a more detailed and general overview of various proposals, see [94]. In the following we leave the conceptual level and propose in detail a specific analysis framework for very precise CRPD estimation, addressing various shortcomings of existing approaches.

4.6 Synergetic Approach to CRPD Analysis

In this section we present our approach to bound CRPD in K-way set-associative caches under the LRU replacement policy for fixed-priority periodic schedules by proposing an analysis framework from the ground up. We propose a cache analysis which trades efficiency for precision, specifically with the intention of evaluating its practical applicability. We argue that this trade-off is justified since we are interested in cache analysis result of caches close to the CPU — which are typically small and yield low associativity. Although we do not take multi-level cache analyses into consideration, imprecision at this level can have significant impact on other stages of analyses as uncertainty is introduced early. Ultimately, we are interested in precise bounds on CRPD, and imprecision potentially degrades the overall result quality. For direct mapped caches, the authors of [73, 95] proposed similar analysis domains. We propose the generalization for K-way set associative caches. We further show how UCB, ECB and CBR can be computed cumulatively in a single analysis pass for this framework, instead of requiring several distinct analyses as previously proposed. We also propose optimizations that can be applied in the context of instruction caches specifically. We then show how results can be used to compute improved bounds on CRPD. Throughout the discussion, we address several weaknesses of existing approaches and propose improvements accordingly.

In Section 4.6.1 we formally define our cache analysis framework. We show in Section 4.6.2 how UCB, ECB and CBR can be derived from analysis results and how our state representation simplifies analysis. In Section 4.6.3 we propose an instruction cache optimization for the reduction of the analysis state space. In Section 4.6.4 we show how these results can be applied to compute CRPD estimates and propose improved bounds for set-associative caches.

4.6.1 Precise Cache Analysis

In this section we introduce basics of our cache analysis framework [7]. Primarily two techniques [88, 95] have been proposed to solve the problem of cache analysis, which differ in precision and complexity. In the literature, they are known as the "set-based" and "state-based" approaches [73]. In the following, we sketch the construction of our approach to state-based analysis of K-way set-associative caches, which we will use as the basis for our following proposals. We seek to exploit its unique properties. State-based analysis has only been proposed for direct mapped caches yet. We extend the basic idea for higher associativities.

First we briefly review imprecision of abstract cache analysis, then we define precise cache analysis.

Imprecision of Cache Abstraction

We briefly discuss the inherent imprecision introduced by abstract cache analysis from Section 4.4.3. Recall that abstract may analysis is based on the join semi-lattice $(\mathcal{A}, \top, \sqsubseteq$

$,\sqcup)$ where $\mathcal{A}\colon \mathcal{M} \mapsto \text{Age}$ is the set of functions mapping from memory blocks to block ages. Information is lost in abstraction as well as joining of states, as we will show with two examples.

Example *Let $c_s = \{(a,b,c,d),(b,a,c,e)\}$ denote two states of set-associative cache sets of associativity $K = 4$. Abstracting (cf. Equation 4.12) from c_s yields:*

$$\widehat{c_s} = \alpha_{may}(\{(a,b,c,d),(b,a,c,e)\}) = (\{ab\},\{\},\{c\},\{d,e\}) \tag{4.38}$$

Concretization (cf. Equation 4.13) of $\widehat{c_s}$ then yields:

$$c_s \sqsubseteq \alpha_{may}(\widehat{c_s}) = \begin{Bmatrix} (a,b,c,d),(b,a,c,d),(a,b,c,e),(b,a,c,e), \\ (a,b,c,\perp),(b,a,c,\perp),(a,b,\perp,\perp),(b,a,\perp,\perp), \\ (a,\perp,\perp,\perp),(b,\perp,\perp,\perp),(\perp,\perp,\perp,\perp) \end{Bmatrix} \tag{4.39}$$

Abstraction yields a loss of information such that its concretization is a large overestimation of the original states. Similarly, joining abstract states leads to loss of information such that for an abstract states $\widehat{s}$ and $\widehat{t}$ it holds that $\gamma_{\text{may}}(\widehat{s} \sqcup^{lru}_{may} \widehat{t}) \nsubseteq \gamma_{\text{may}}(\widehat{s}) \cup \gamma_{\text{may}}(\widehat{t})$.

Example *Consider abstract cache set states $\widehat{s} = (\{a\},\{\},\{\},\{\})$ and $\widehat{t} = (\{b\},\{\},\{\},\{\})$. Joining (cf. Equation 4.16) $\widehat{s}$ and $\widehat{t}$ yields:*

$$\widehat{st} = \widehat{s} \sqcup^{lru}_{may} \widehat{t} = (\{a,b\},\{\},\{\},\{\}) \tag{4.40}$$

Its concretization then yields:

$$\gamma_{\text{may}}(\widehat{st}) = \begin{Bmatrix} (a,b,\perp,\perp),(b,a,\perp,\perp), \\ (a,\perp,\perp,\perp),(b,\perp,\perp,\perp), \\ (\perp,\perp,\perp,\perp) \end{Bmatrix} \tag{4.41}$$

In particular, we lose information on mutual exclusion of cache states: an inherent loss of information on execution history on every join. This means for CRPD computation, we are considering non-existing cache contents. Since UCB, ECB and CBR are already approximations themselves, imprecision accumulates and CRPD is even further overestimated.

State-based Analysis for Set-associative Caches

Recall that $\mathcal{M}_\perp = \mathcal{M} \cup \{\perp\}$ denotes states of a cache block, $\mathcal{C}_s = \mathcal{M}_\perp^K$ denotes state of a cache set and, consequently, $\mathcal{C} = \mathcal{C}_s^N$ denotes state of a cache. Then a precise analysis domain is denoted by $\mathbb{D}_\mathcal{C} = \wp(\mathcal{C})$ and the corresponding join semi-lattice is defined as:

$$(\mathbb{D}_\mathcal{C}, \wp(\mathbb{D}_\mathcal{C}), \subseteq, \cup) \tag{4.42}$$

We refer to forward cache semantics as *reaching cache state* (RCS) and distinguish RCS just before ($\text{cs}^{\rightarrow\bullet}$) and after a program point ($\text{cs}^{\bullet\rightarrow}$), respectively. Let $\text{lru} : \mathcal{M} \times \mathcal{C}_s \mapsto \mathcal{C}_s$ model LRU cache set semantics. Then we define the corresponding cache set transformer $\text{tf}_s \colon \mathcal{M} \times \mathcal{C}_s \mapsto \mathcal{C}_s$ as:

$$\text{tf}_s(m, s) = \begin{cases} \text{lru}(m, s) & \text{if } m \neq \bot \\ s & \text{otherwise} \end{cases} \tag{4.43}$$

Let $\iota \colon V \mapsto \mathcal{M}$ denote memory accesses in a point and let $\text{set} \colon m \mapsto [1, N]$ denote the cache set blocks are mapped to. We define a function gen which models accesses issued in a program point mapped to a specific cache sets as:

$$\begin{aligned} &\text{gen} \colon V \times [1, N] \mapsto \mathcal{M} \\ \text{gen}(u, i) = &\begin{cases} m & \text{if } \iota(u) = m \wedge \text{set}(m) = i \\ \bot & \text{otherwise} \end{cases} \end{aligned} \tag{4.44}$$

Then $\text{tf}_c \colon \mathcal{C} \mapsto \mathcal{C}$ denotes a cache state transformer, defined as:

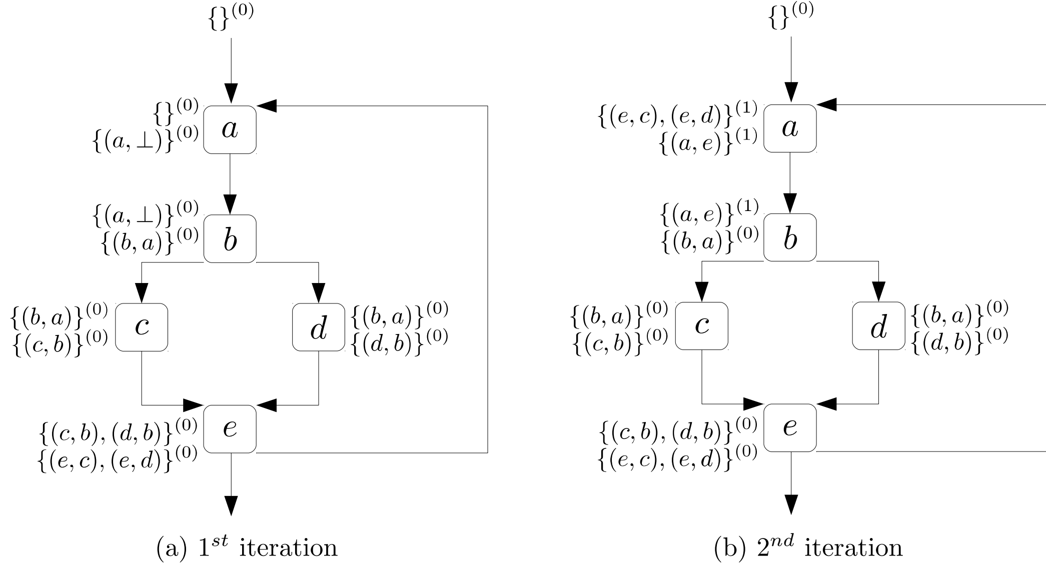
$$\text{tf}_c((s_1, \ldots, s_N)) = (\text{tf}_s(\text{gen}(u_i, 1), s_1), \ldots, \text{tf}_s(\text{gen}(p_i, N), s_N)) \tag{4.45}$$

Precise cache semantics is denoted by the least RCS solution to the following equation system:

$$\begin{aligned} \text{cs}^{\rightarrow\bullet/\bullet\rightarrow} &\colon V \mapsto \mathbb{D}_{\mathcal{C}} \\ \text{cs}^{\rightarrow\bullet}(u) &= \bigcup_{v \in \text{pred}(u)} \text{cs}^{\bullet\rightarrow}(v) \\ \text{cs}^{\bullet\rightarrow}(u) &= \{\text{tf}_c(r) \mid r \in \text{cs}^{\rightarrow\bullet}(u)\} \end{aligned} \tag{4.46}$$

Example *Figure 4.9 illustrates the fixed point computation for RCS for a cache of associativity $K = 2$, number of sets $N = 1$ and memory accesses denoted by node labels, similar to Figure 4.3, such that $\iota(a) = a$. We represent cache states as sets of tuples, where $\bot$ denotes empty cache lines. Figure 4.9a and Figure 4.9a depict the two iterations required to obtain the MFP solution. Superscripts at the cache states denote the number of updates of the respective states. The initial cache state is the empty cache. Then each access ages all elements by one according to Equation 4.43 and at each join the union of cache states in computed. As opposed to Figure 4.3, where the state after evaluation of node e equals $(\{e\}, \{c, d\})$, in Figure 4.9 the state $\{(e, c), (e, d)\}$ retains information of mutual exclusion of memory blocks c and d.*

We refer to backward cache semantics as *live cache states* (LCS). While RCS denote memory blocks accessed during execution for particular program points, LCS represent memory blocks at particular program points which may be accessed again in the future without being evicted. Computation is symmetric to RCS: LCS is the least solution to



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

Figure 4.9: Example of RCS computation

the equations defined by:

$$
\begin{aligned}
\mathrm{cs}^{\leftarrow\bullet/\bullet\leftarrow}&\colon V \mapsto \mathbb{D}_{\mathcal{C}} \\
\mathrm{cs}^{\bullet\leftarrow}(u) &= \bigcup_{v\in \mathrm{succ}(u)} \mathrm{cs}^{\leftarrow\bullet}(v) \\
\mathrm{cs}^{\leftarrow\bullet}(u) &= \{t_c(r) \mid r \in \mathrm{cs}^{\bullet\leftarrow}(u)\}
\end{aligned}
\tag{4.47}
$$

4.6.2 Computation of UCB, ECB and CBR

We proceed to define how UCB, ECB and CBR are derived from RCS and LCS in $\mathbb{D}_{\mathcal{C}}$, respectively. We maintain contextual discrimination for maximal precision by keeping information encoded in $\mathbb{D}_{\mathcal{C}}$ as opposed to the original definitions above.

Useful Cache Blocks

We define the set of UCB as the intersection of RCS and LCS:

$$
\begin{aligned}
\mathrm{ucb}_{\mathcal{C}}&\colon V \mapsto \mathbb{D}_{\mathcal{C}} \\
\mathrm{ucb}_{\mathcal{C}}(u) &= \{r \cap_c l \mid r \in \mathrm{cs}^{\bullet\rightarrow}(u), l \in \mathrm{cs}^{\leftarrow\bullet}(u)\}
\end{aligned}
\tag{4.48}
$$

where operator $\cap_c\colon \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$ evaluates to cache set-wise intersection, defined as:

$$
c \cap_c c' = (s_1 \cap_s s'_1, s_2 \cap_s s'_2, \ldots, s_N \cap_s s'_N) \tag{4.49}
$$

where operator $\cap_s\colon \mathcal{C}_s \times \mathcal{C}_s \mapsto \mathcal{C}_s$ denotes block-wise intersection, defined as:

$$s_i \cap_s s'_i = (b_1, \dots, b_k)\colon \forall b_i\colon \begin{cases} b_i = m & \text{if } m \in s_i \wedge m \in s'_i \\ b_i = \bot & \text{otherwise} \end{cases} \tag{4.50}$$

The set of UCB per cache set is per definition bounded by associativity K and loses all information on block ages.

Example *Let $s = (a, b, c, d)$ and $s' = (b, a, d, \bot)$ denote cache sets states for $K = 4$, then $s \cap_s s' = (a, b, \bot, d)$. The resulting tuple yields no particular order.*

Note that $\mathrm{ucb}_\mathcal{C}$ maintains the product of RCS and LCS and therefore represents only actually feasible combinations. For convenience, we define a predicate that denotes usefulness of a memory block:

$$\begin{aligned} \mathrm{pucb}_\mathcal{C}\colon V \mapsto \mathcal{M} \mapsto \{\top, \bot\} \\ \mathrm{pucb}_\mathcal{C}(u) = \lambda m \,.\, \exists c \in \mathrm{ucb}_\mathcal{C}(u)\colon \exists s \in c\colon m \in s \end{aligned} \tag{4.51}$$

A memory block is useful if it is member of one element in $\mathrm{ucb}_\mathcal{C}$.

Evicting Cache Blocks

We directly derive ECB from the result. Let $t \in V$ denote the CFG sink node. Then $\mathrm{ecb}_\mathcal{C}$ denotes the set of evicting cache states, defined as:

$$\begin{aligned} \mathrm{ecb}_\mathcal{C}\colon \mathbb{D}_\mathcal{C} \\ \mathrm{ecb}_\mathcal{C} = \{c \in \mathrm{cs}^{\bullet\to}(t)\} \end{aligned} \tag{4.52}$$

Note that RCS does not necessarily map all blocks that cause evictions as some of them may have been evicted within the same task themselves in turn. Rather, they indicate the pattern of cache usage of the preempting task and denote which cache sets have been used and to what extent. For RCS specifically, this is more accurate than in other approaches [73], as RCS only holds those states that are actually reachable along all paths leading to the terminal program points.

Example *In Figure 4.9 reachable cache states (for geometry $K = 2$, $N = 1$) and terminal node* e*, states* $(\{(e, c), (e, d)\})$ *are being explicitly discriminated, reflecting mutually exclusive paths leading to* e*, potentially eliminating imprecision of ECB-based CRPD estimations.*

Cache Block Resiliencies

From RCS and LCS we can also directly derive CBR. Under abstraction, this is not possible and would necessitate a separate analysis [93], potentially losing additional information. Analysis based on $\mathbb{D}_\mathcal{C}$ maintains precise ages and resiliences.

To this end, we define a helper function that returns a default block age of 0 if age (Equation 4.2) returns ∞:

$$\mathrm{age}_0\colon \mathcal{M} \times \mathcal{C} \mapsto \mathit{Age}$$
$$\mathrm{age}_0(m,s) = \begin{cases} \mathrm{age}(m,s) & \text{if } \mathrm{age}(m,s) \neq \infty \\ 0 & \text{otherwise} \end{cases} \tag{4.53}$$

Then we can define the maximal age of a memory block m with regard to CBR right before a program point u in forward direction as:

$$\mathrm{age}^{\rightarrow\bullet}\colon V \mapsto \mathcal{M} \mapsto \mathit{Age}$$
$$\mathrm{age}^{\rightarrow\bullet}(u) = \lambda m \,.\begin{cases} \max\left\{ a \in \mathrm{age}_0(m,s) \;\middle|\; \begin{array}{l} s \in c, \\ c \in \mathrm{cs}^{\bullet\rightarrow}(v), \\ v \in \mathrm{pred}(u) \end{array} \right\} & \text{if } \mathrm{pucb}(u,m) \\ \infty & \text{otherwise} \end{cases} \tag{4.54}$$

The maximal age of a block m before a point u is the maximal age of all states in preceding points, where potentially uncached blocks yield ages equal to 0, but only if m is useful. The latter constraint guarantees the existence of at least one state containing m. If a memory block is not useful, we can ignore it (∞). Analogously, we define the maximal age of a memory block m with regard to CBR right after a program point u in forward direction as:

$$\mathrm{age}^{\bullet\rightarrow}\colon V \mapsto \mathcal{M} \mapsto \mathit{Age}$$
$$\mathrm{age}^{\bullet\rightarrow}(u) = \lambda m \,.\begin{cases} \max\left\{ a \in \mathrm{age}_0(m,s) \;\middle|\; \begin{array}{l} s \in c, \\ c \in \mathrm{cs}^{\bullet\rightarrow}(u) \end{array} \right\} & \text{if } \mathrm{pucb}(u,m) \\ \infty & \text{otherwise} \end{cases} \tag{4.55}$$

Symmetrically, we define $\mathrm{age}^{\bullet\leftarrow}$ and $\mathrm{age}^{\leftarrow\bullet}$ for backward semantics.

Under the assumption that program points are atomic in the sense that preemptions occur only after their complete execution, we define CBR as:

$$\mathrm{cbr}\colon V \mapsto \mathcal{M} \mapsto \mathit{Age}$$
$$\mathrm{cbr}(u) = \lambda m \,.\, (K-1) - \min(K-1, \mathrm{age}^{\bullet\rightarrow}(u)(m) + \mathrm{age}^{\bullet\leftarrow}(u)(m)) \tag{4.56}$$

This definition yields more precise results than in the original proposal [93], which treats aging symmetrically, similar to Equation 4.35.

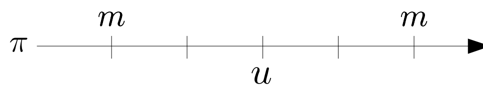


Figure 4.10: CBR overestimation due to symmetric aging ($K > 4$)

Example *In Figure 4.10 memory block m is accessed twice and is therefore considered useful in program point u. For Equation 4.35, it holds that* $\mathrm{age}^{\bullet\rightarrow}(u)(m)+\mathrm{age}^{\leftarrow\bullet}(u)(m) = 2+2 = 4$*, whereas for Equation 4.56 it holds that* $\mathrm{age}^{\bullet\rightarrow}(u)(m)+\mathrm{age}^{\bullet\leftarrow}(u)(m) = 2+1 = 3$.

Set associativities are usually small and BRT can be significant, so reduction of overestimation is important. For non-atomic program points with multiple accesses and multiple potential preemptions, such as basic blocks, care has to be taken not to underestimate aging though. We address this in the following.

4.6.3 Restriction to Basic Block Boundaries

We assumed that program points are atomic in that preemptions are only possible after their completion and in that their execution only issues just a single memory access. For example, program points correspond to single CPU instructions. Domain $\mathbb{D}_{\mathcal{C}}$ is inherently expensive in terms of memory consumption. Therefore, we now show how to restrict computations to basic blocks such that it is sufficient to perform computations only on their boundaries instead of all interior points. Recall that a basic block is a sequence of instructions without interior jump targets. Nevertheless, preemptions can occur within this sequence. Therefore, computing UCB or CBR only on basic block boundaries requires to compute safe and ideally tight approximations. In [88], a rough idea of reducing UCB analysis to basic block boundaries has been given informally and for direct mapped caches only.

In the following we formalize strategies for UCB and CBR on basic block boundaries. Note that these techniques are only applicable to instruction caches.

UCB on Basic Block Boundaries

Let $\pi = \pi' \cdot \pi^{BB}$ be an execution path terminating at the end of a basic block, with π^{BB} denoting the path through a basic block, then all paths by definition share the same suffix such that:

$$\forall \pi_i, \pi_j \in \Pi \colon \pi_i^{BB} = \pi_j^{BB} \tag{4.57}$$

In a basic block, an instruction memory block m can only be accessed once, since interior program points do not repeat: $\forall (u_1, \ldots, u_n) \subseteq \pi_{BB} \colon |(u_1, \ldots, u_n)| = |\{u_1, \ldots, u_n\}|$. Also multiple interior points accessing the same memory block must be consecutive: Let $I = \{i \mid u_i \in \pi^{BB} \wedge \iota(u) = m\}$ be the index set of program points within a basic block that access the same memory block, then these indices must be consecutive: $\nexists j \in [\min I, \max I] \colon \iota(u_j) \neq m$.

Let program points now denote complete basic blocks and let $\iota^* \colon V \mapsto \mathcal{M}^*$ denote the sequence of accesses (into the same cache set) within basic blocks. It holds, by definition of LRU, that cache state right after each basic block only depends on the last K accesses:

$$m \in \mathrm{cs}^{\bullet\rightarrow}(u) \Leftrightarrow m \in (m_{j-K}, \ldots, m_j) \subseteq (m_i, \ldots, m_j) = \iota^*(u) \tag{4.58}$$

Consequently, all referenced memory blocks $m_{i\leq l<j-K}$ need not be considered for CRPD, as they are evicted from the cache without preemption already. The same holds true for $\text{cs}^{\leftarrow\bullet}$. Consequently, Equation 4.48 remains to denote a safe bound on UCB: A memory block referenced within a basic block is only useful if it does not cause a guaranteed miss without preemption.

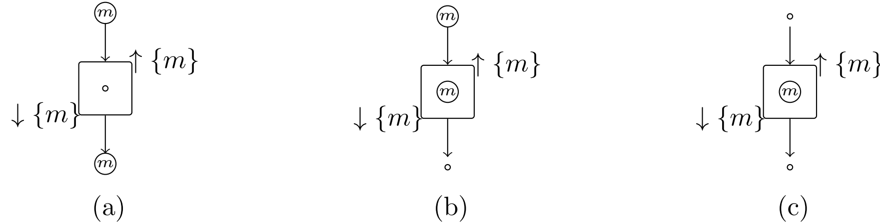


Figure 4.11: Scenarios for UCB on basic block boundaries

Example *Figure 4.11 illustrates the possible scenarios, where* $\downarrow \{\}$ *denotes* $\text{cs}^{\bullet\rightarrow}$ *and* $\uparrow \{\}$ *denotes* $\text{cs}^{\leftarrow\bullet}$*, respectively. In Figure 4.11a both references to block* m *are external to the basic block and* m *is classified useful throughout the entire basic block. In Figure 4.11b block* m *remains useful nonetheless if* m *only occurs in the last* k *accesses. Figure 4.11c depicts the result, given that* m *is only referenced once within the basic block. In this case an overestimation occurs. In all cases,* m *remains useful and would therefore safely overestimate the CRPD regardless of where a preemption actually occurs within the basic block.*

To summarize, the K last accesses to a cache set fully determine its state right after a basic block and a memory block is classified useful for an entire basic block if it is not guaranteed to cause a cache miss in some interior point without preemption.

CBR on Basic Block Boundaries

We will now address CBR on basic block boundaries. To this end, we show how to compute safe approximations for block ages for all interior points within basic blocks.

Recall that we only allow for consecutive access to a memory block within a basic block. This effectively splits a basic block in two halves: i) Interior points from the top of the basic block to the access. ii) Interior points from the access to the bottom of a the basic block. We define the distance between the top and an access as:

$$^\bullet\text{dist}\colon V \mapsto \mathcal{M} \mapsto Age$$

$$^\bullet\text{dist}(u) = \lambda m \,.\, \begin{cases} \min(K-1, \text{age}^{\rightarrow\bullet}(u)(m) + \text{age}^{\bullet\leftarrow}(u)(m)) & \text{if } \text{pucb}(u,m) \\ \infty & \text{otherwise} \end{cases} \tag{4.59}$$

	$\text{age}^{\rightarrow}$	$\text{age}^{\leftarrow}$	dist	max
top	0	1	1	4
bot.	1	3	4	
top	1	3	4	4
bot.	3	1	4	
top	3	1	4	4
bot.	1	0	1	

Figure 4.12: Computing safe block-wise ages

Analogously, for the bottom half of a basic block, we define:

$$\text{dist}^{\bullet} \colon V \mapsto \mathcal{M} \mapsto \mathit{Age}$$

$$\text{dist}^{\bullet}(u) = \lambda m \,.\, \begin{cases} \min(K-1, \text{age}^{\bullet\rightarrow}(u)(m) + \text{age}^{\bullet\leftarrow}(u)(m)) & \text{if } \text{pucb}(u,m) \\ \infty & \text{otherwise} \end{cases} \tag{4.60}$$

Let us assume sequences of memory accesses as paths π^{ι}. Then a sequence of accesses for an execution path is defined as:

$$\pi^{\iota} = \pi_1^{\iota} \cdot \pi_2^{\iota} \cdot \pi_3^{\iota} = (\ldots, u_i^{\iota}) \cdot (v_1^{\iota}, \ldots, v_m^{\iota}, \ldots, v_j^{\iota}) \cdot (w_k^{\iota}, \ldots)$$

where π_2^{ι} denotes a basic block and v_m^{ι} denotes a memory access to m. Then a safe block age for an entire basic block is the maximal age of paths from the last access to the current access $(u_{f \leq i}^{\iota}, \ldots, v_m^{\iota})$ and the maximal age from this access to the next access $(v_{m+1}^{\iota}, \ldots, w_{l \geq k}^{\iota})$. Thus, we redefine CBR for basic blocks as:

$$\text{cbr}^{BB} \colon V \mapsto \mathcal{M} \mapsto \mathit{Age}$$

$$\text{cbr}^{BB}(u) = \lambda m \,.\, (K-1) - \max({}^{\bullet}\text{dist}(u)(m), \text{dist}^{\bullet}(u)(m)) \tag{4.61}$$

Example *Consider Figure 4.12. To the left, memory accesses and basic blocks are depicted. We consider accesses to* m *specifically. We maintain ages of* m *at the* top *and the* bot *(bottom) of the respective basic blocks.* $\text{age}^{\rightarrow}$ *denotes* $\text{age}^{\rightarrow\bullet}$ *and* $\text{age}^{\bullet\rightarrow}$*, respectively.* $\text{age}^{\leftarrow}$ *denotes* $\text{age}^{\bullet\leftarrow}$ *and* $\text{age}^{\leftarrow\bullet}$*, respectively. Function* dist *denotes the respective distances according to Equation 4.59 and Equation 4.60, respectively. Function* max *denotes the maximal ages for both paths. Recall that memory blocks potentially not cached but classified useful, yield a default of age* 0*.*

4.6.4 CRPD Bounds on Task Sets

We now show how CRPD is computed for our cache analysis and we improve upon existing analyses by showing how to reduce pessimism in case of multiple preempting

tasks. We will restrict the discussion to our approach of bounding CRPD for this specific. A thorough and general overview and a discussion of alternative bounds follows in Chapter 6 below.

First, we show how CRPD is computed in general for our framework, then we address specific issues related to multiple preempters and propose an optimized variant.

CRPD for Precise Cache Analysis

Recall from Section 4.5.3 how CRPD is bounded by taking interference denoted by UCB, ECB and CBR into account. Application to domain $\mathbb{D}_{\mathcal{C}}$ is achieved as follows.

We extend the equation system for WCRT computation (cf. Section 3.1.4) by an additional parameter $\gamma_{i,j}$, which denotes CRPD imposed by a task τ_j upon a task τ_j. Further, we denote the number of preemptions of τ_j during the response time of τ_i by:

$$\#(i,j) := \left\lceil \frac{R_i + J_i}{T_j} \right\rceil \tag{4.62}$$

Then WCRT including CRPD can be defined as [78]:

$$R_i = C_i + \sum_{j \in \text{hp}(i)} \#(i,j)(C_j + \gamma_{i,j}) \tag{4.63}$$

We define a helper function to count the number of mapped memory blocks in a cache set:

$$\begin{aligned} 1_c &\colon \mathcal{C}_s \mapsto \mathbb{N}_0 \\ 1_c(c) &= |\{m \in c \mid m \neq \bot\}| \end{aligned} \tag{4.64}$$

Then we can define the interference χ of cache states $\mathcal{C}$ taking UCB, CBR and ECB into account as:

$$\begin{aligned} &\chi^{\text{ucb/cbr/ecb}}(C^{\text{ucb}}, R, C^{\text{ecb}}) \\ &\quad := \sum_{i \in [1,N]} \left\{ c_i \setminus \{m \mid R(m) \geq 1_c(c_i')\} \mid c_i \in C^{\text{ucb}} \wedge c_i' \in C^{\text{ecb}} \right\} \end{aligned} \tag{4.65}$$

where $C^{\text{ucb}} \in \mathcal{C}$ denotes a cache state representing UCB, $R \subseteq \mathcal{M} \mapsto \text{Age}$ denotes CBR and $C^{\text{ecb}} \in \mathcal{C}$ denotes a cache state representing ECB. Similar to Equation 4.36 on page 57, we exclude from the set of UCB those memory blocks whose resilience is equal or greater then the number of evictions imposed by ECB, and we accumulate over all cache sets.

Finally, CRPD $\gamma_{i,j}$ *for a single preempting task* is then denoted by maximum among all possible interferences and is defined as.

$$\gamma_{i,j} := \text{crpd}_{\mathcal{C}}^{\text{ucb/cbr/ecb},1}\left(\text{ucb}_{\mathcal{C}}, \text{cbr}_{\mathcal{C}}, \text{ecb}_{\mathcal{C}}\right)$$
$$= \max \left\{ \chi^{\text{ucb/cbr/ecb}}(C, R, C') \;\middle|\; \begin{array}{l} C \in \text{ucb}_{\mathcal{C}}(u), \\ R \in \text{cbr}_{\mathcal{C}}(u), \\ u \in V, \\ C' \in \text{ecb}_{\mathcal{C}} \end{array} \right\} \times \text{BRT} \tag{4.66}$$

Below, we relax the restriction to multiple preempting tasks.

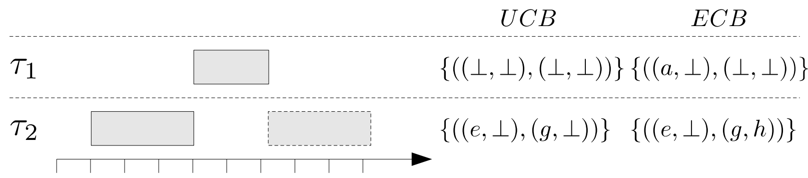


Figure 4.13: Example with UCB and ECB for direct preemption ($K = 2, N = 2$)

Example *Figure 4.13 illustrates the preemption of a task* τ_2 *by a task* τ_1 *for a single cache state for UCB and ECB, respectively (*$BRT = 1$*). For both UCB* e *and* g *of task* τ_2*, resilience equals* 1*. Then Equation 4.66 yields:*

$$\begin{aligned}
&\text{crpd}_{\mathcal{C}}^{\text{ucb/cbr/ecb},1}\left(\text{ucb}, \text{cbr}, \text{ecb}\right) \\
&= \max \left\{ \chi^{\text{ucb/cbr/ecb}}(C, R, C') \mid C \in \text{ucb}(u), R \in \text{cbr}(u), C' \in \text{ecb} \right\} \\
&= \max \left\{ \chi^{\text{ucb/cbr/ecb}}\left(((e,\bot),(g,\bot)), \{e \to 1, g \to 1\}, ((a,\bot),(\bot,\bot))\right) \right\} \\
&= \max\{\{|\{\}|\}\} = 0
\end{aligned}$$

Bounding with Multiple Preempters

So far, we have limited the discussion to CRPD for a single preempter. Multiple preemptions by the same task are safely bounded since the same set of ECB is applied for each such preemption. CRPD with multiple preempters, however, can not be safely bounded by considering tasks in isolation as they potentially *interact* [93], causing additional evictions. We distinguish two types of interaction: i) *nested interaction*: preempters are potentially preempted themselves ii) *successive interaction* multiple preemptions occur in succession between memory accesses in the preempting task In both cases, preempters combined cause greater block aging than preempters considered in isolation. In these cases, Equation 4.66 is unsafe. We illustrate this with two examples.

Example (Nested Interaction) *Consider Figure 4.14, which depicts nested interaction for the preemption of task* τ_3 *by tasks* τ_1, τ_2*. The sets of UCB and ECB are given for a direct mapped cache and we assume* $\text{BRT} = 1$*. Black marks indicate actual CRPD. The worst-case CRPD of* τ_2 *preempting* τ_3 *equals* 2 *since* τ_2 *is itself preempted by* τ_1*.*

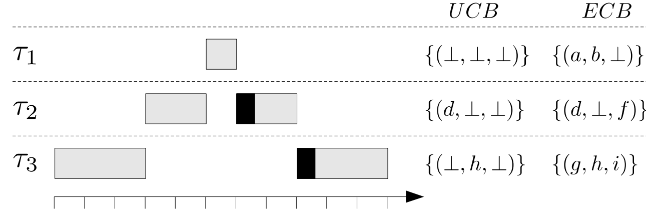


Figure 4.14: Interaction by nested preemption($K = 1, N = 3$)

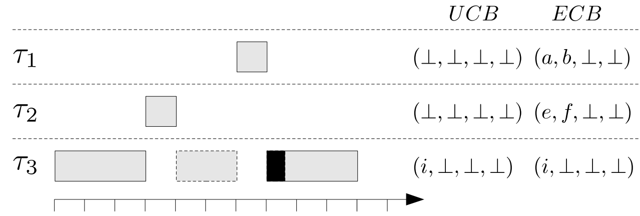


Figure 4.15: Interaction by successive preemption ($K = 4, N = 1$)

Example (Successive Interaction) *Figure 4.15 illustrates the preemption of task τ_3 by tasks τ_1 and $\tau 2$ without nesting. We assume a associativity $K = 4$ and* BRT $= 1$*. The black mark indicates actual CRPD. Each preemption in isolation does not increase the aging of the useful cache block i such that it is being evicted. However, both preemptions in succession evict block i. Note that this problem occurs only for non-direct mapped caches.*

We first address nested preemption. A simple way to bound evictions due to nested preemptions is to accumulate costs of all indirect preemptions. Let τ_i be a preemptee and τ_j a preempter. Recall that all tasks of higher priority than τ_i but lower or equal priority of τ_j is denoted by $\text{aff}(i, j) = \text{hp}(i) \cap \text{lep}(j)$ (cf. Table 3.2 on page 27). Then we bound CRPD for a single preemption by:

$$\gamma_{i,j} := \sum_{k \in \text{aff}(i,j)} \text{crpd}_{\mathcal{C}}^{\text{ucb/cbr/ecb,1}} \left(\text{ucb}_{\mathcal{C}}^{\tau_k}, \text{cbr}_{\mathcal{C}}^{\tau_k}, \text{ecb}_{\mathcal{C}}^{\tau_j} \right) \tag{4.67}$$

This correctly accounts for a CRPD equal to 2 in Figure 4.14. Note that various different bounds for CRPD have been proposed in the literature. We dedicate Chapter 6 to their discussion. Here, this simple bound shall suffice.

It remains to tackle the underestimation in successive interaction, as illustrated in Figure 4.15. A possible solution to this problem is to assume that each preemption by one task is an immediate succession of preemptions by all possible preempters instead, which has been proposed in [93]. For nested interaction, we assume that a preempting task has itself already been preempted. We adapt their proposal to our framework and propose an improvement.

We first define a join operator over $\mathbb{D}_{\mathcal{C}}$ as a cache set-wise join which collects up to K memory blocks per cache set. Function lru (Equation 4.3 on page 45) already provides the desired semantics. Let $\text{lru}^* \colon \mathcal{M}^* \times \mathcal{C} \mapsto \mathcal{C}$ denote a successive application of memory

accesses to a cache state such that:

$$\mathrm{lru}^*((m_1,\ldots,m_n),C) = \mathrm{lru}(m_n, \mathrm{lru}(m_{n-1},\ldots \mathrm{lru}(m_1,C)\ldots)) \tag{4.68}$$

Then joining two cache states is defined as the set-wise application of lru^*:

$$\begin{aligned} \cup^{\mathcal{C}} &: \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C} \\ C \cup^{\mathcal{C}} C' &= (s_1,\ldots,s_N) \cup^{\mathcal{C}} (s'_1,\ldots,s'_N) = (\mathrm{lru}^*(s'_1,s_1),\ldots,\mathrm{lru}^*(s'_N,s_N)) \end{aligned} \tag{4.69}$$

We lift this to sets of cache states by defining:

$$\begin{aligned} \cup^{\mathbb{D}_{\mathcal{C}}} &: \mathbb{D}_{\mathcal{C}} \times \mathbb{D}_{\mathcal{C}} \mapsto \mathbb{D}_{\mathcal{C}} \\ S \cup^{\mathbb{D}_{\mathcal{C}}} T &= \left\{ C \cup^{\mathcal{C}} C' \;\middle|\; C \in S, C' \in T \right\} \end{aligned} \tag{4.70}$$

CRPD given interaction for associativities $K > 1$ is then bounded by the combination of all ECB of potential preempters:

$$\gamma_{i,j} := \sum_{k \in \mathrm{aff}(i,j)} \mathrm{crpd}_{\mathcal{C}}^{\mathrm{ucb/cbr/ecb}} \left(\mathrm{ucb}_{\mathcal{C}}^{\tau_k}, \mathrm{cbr}_{\mathcal{C}}^{\tau_k}, \bigcup\nolimits_{l \in \mathrm{hep}(j)}^{\mathbb{D}_{\mathcal{C}}} \mathrm{ecb}_{\mathcal{C}}^{\tau_l} \right) \times \mathrm{BRT} \tag{4.71}$$

This bound is safe in either case but not very precise. In the following we seek to reduce overestimation in bounding successive interaction specifically. We will be concerned with various kinds of pessimism in a more general discussion in Chapter 6.

Pessimism in Interaction

Equation 4.71 can be improved since currently we assume that all tasks interact unconditionally. Recall that cache interference is considered cache set-wise (Equation 4.65). Also recall that interaction denotes the fact that two preemptions cause greater block aging than preemptions considered in isolation. We can avoid accounting for some presumed interaction by recognizing that a preempting task whose ECB cache-set is empty can never interact with other preempters.

Figure 4.16: Example scenarios of successive interaction

Example *Figure 4.16 illustrates two successive interaction scenarios for preempting tasks τ_1, τ_2, given accesses to a memory block m. We only consider a single cache*

set of associativity $K = 4$. Thus, $|ecb_n|$ denotes the utilization of the set belonging to τ_n. Arrows denote preemptions and circles denote presence in the cache. Figure 4.16a depicts successive interaction of two preemptions, with a ECB set utilization of 3 and 2, respectively. One preemption alone does not evict m from the cache. Preemptions in succession, however, cause a cache miss in the third memory access. Equation 4.71 correctly models this scenario. Figure 4.16b, on the other hand, illustrates pessimism of this approach. Task τ_0 does not contribute to block aging. Only the ECB of τ_1 cause eviction all by itself. Nonetheless, Equation 4.71 causes consideration of interaction for τ_0, in addition to considering the evictions caused by τ_2 alone. Hence accounting for the same scenario twice.

Similarly, this holds true for nested interaction. We conclude from this example that we can safely omit joining ECB cache sets that are empty in the lowest priority task, because possible cache misses due to interaction are independent of the latter.

We redefine Equation 4.69 and Equation 4.70 to address this finding. Let task τ_j be the preempting task. Recall that 1_c (Equation 4.64) denotes the number of non-empty cache lines in a set. Then we define the constrained join of cache states as:

$$\cup^{\mathcal{C},\star} \colon \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$$

$$C \cup^{\mathcal{C},\star} C' = (s_1, \ldots, s_N) \cup^{\mathcal{C},\star} (s'_1, \ldots, s'_N)$$
$$= \left(\begin{Bmatrix} \text{lru}^*(s'_1, s_1) & \text{if } 1_c(s_1) > 0 \\ s_1 & \text{otherwise} \end{Bmatrix}, \ldots, \begin{Bmatrix} \text{lru}^*(s'_N, s_N) & \text{if } 1_c(s_N) > 0 \\ s_K & \text{otherwise} \end{Bmatrix} \right) \tag{4.72}$$

The operator is not commutative. Either the left-hand-side operand is non-empty, then we apply lru* analogous to Equation 4.69, or it is empty. Then we do not apply any changes and return the empty set. We lift this to sets of cache states, as in Equation 4.70, by defining:

$$\cup^{\mathbb{D}_\mathcal{C},\star} \colon \mathbb{D}_\mathcal{C} \times \mathbb{D}_\mathcal{C} \mapsto \mathbb{D}_\mathcal{C}$$
$$S \cup^{\mathbb{D}_\mathcal{C},\star} T = \{ C \cup^{\mathcal{C},\star} C' \mid C \in S, C' \in T \} \tag{4.73}$$

Joining of cache states must now be performed in ascending order of priority, so that non-interacting task can be omitted accordingly. Hence, Equation 4.71 becomes:

$$\gamma_{i,j} := \sum_{k \in \text{aff}(i,j)} \text{crpd}_\mathcal{C}^{\text{ucb/cbr/ecb}} \left(\text{ucb}_\mathcal{C}^{\tau_k}, \text{cbr}_\mathcal{C}^{\tau_k}, \bigcup\nolimits_{l \in \text{hep}(j)}^{\mathbb{D}_\mathcal{C},\star} \text{ecb}_\mathcal{C}^{\tau_l} \right) \times \text{BRT} \tag{4.74}$$

Example *Figure 4.17 illustrates a successive preemption of a task τ_3 by two other tasks. UCB and ECB denote a single state of a cache with associativity $K = 4$ and number of sets $N = 2$. Task τ_2 alone does not cause any evictions since it only ages memory block j by 1. As opposed to this, task τ_1 alone evicts memory block i and ages block j by one. Together, they invalidate all UCB. A CRPD by Equation 4.74 yields a bound of 1 for $\gamma_{3,1}$, since the aging of memory block i exceeds its resilience. For $\gamma_{2,1}$, we compute a*

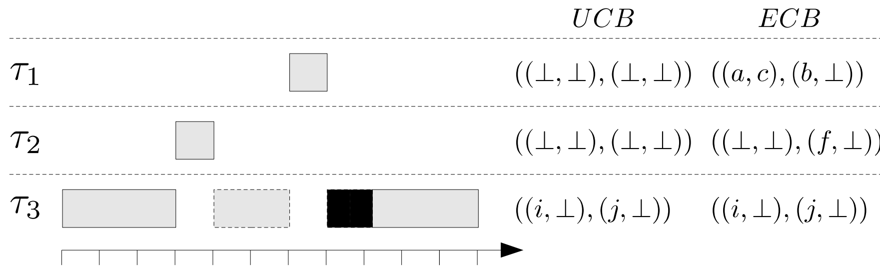


Figure 4.17: Example scenario for successive interaction ($K = 2, N = 2$)

bound of 2*, since ECB combined evict all UCB. This overestimates the actual CRPD since aging caused by* $\tau 1$ *alone has incorrectly been attributed to* τ_2 *as well. Equation 4.74 correctly yields a bound of* 2*, since due to the empty cache set of* τ_2*, aging due to higher priority tasks is ignored for this set.*

4.7 Evaluation

In this section we evaluate various aspects of the proposed cache analysis, including basic CRPD computations. We conducted the experiments with our static analysis framework, in which this cache analysis is one component. As compiler we used the WCET-aware C Compiler (WCC) [96]. Cache analysis itself uses a built-in static pipeline analysis for the *Tricore 1.3* architecture (TC1796b clocked at 150 MHz) to obtain precise static memory access information. Analysis passes themselves are constrained to basic block boundaries.

Evaluation has been carried out on an Intel E5630 (2.53 GHz) CPU with no parallel computations. We made use of the Mälardalen WCET Benchmarks (MRTC) [97] which comprises of typical real-time applications. MRTC benchmarks do not form a task set. As such, static scheduling parameters are unknown. To evaluate a realistic task set, we make use of the PapaBench [98] benchmark suite which models tasks of an Unmanned Aerial Vehicle (UAV) along with static scheduling parameters. We evaluated the benchmarks with floating point operations carried out on the FPU.

To adapt to problem sizes of the benchmarks, we did not maintain the original memory specification of the TC1796 and chose a 2-way set-associative cache of 4 kB total size with 32 B line size and LRU replacement policy and assume a BRT of 1 cycle.

MRTC

For MRTC, we modeled *two* preemptions of a benchmark by a single other benchmark to amplify CRPD results for improved visuals. We analyze preemptions with different benchmarks as preempters. And we perform three evaluations per scenario:

- *UCB*: CRPD from only the preemptee's UCB as in Equation 4.26.
- *ECB*: CRPD from full cache sets that interfere with ECB as specified in Equation 4.31.
- *CBR*: CRPD from Equation 4.37.

Name	Size (B)	UCB	ECB
adpcm_encoder	2 844	59	81
binarysearch	134	5	6
countnegative	278	7	11
crc	976	17	18
edn	3 054	43	98
fdct	2 192	43	70
fft1	4 866	56	58
fibcall	56	2	3
jfdctint	2 740	52	87
lcdnum	1 184	8	10
lms	1 834	31	43
matmult	520	9	15
ndes	2 586	54	61
qurt	1 772	20	21
sqrt	236	7	9
st	1 410	20	24

Table 4.2: Properties of MRTC benchmarks

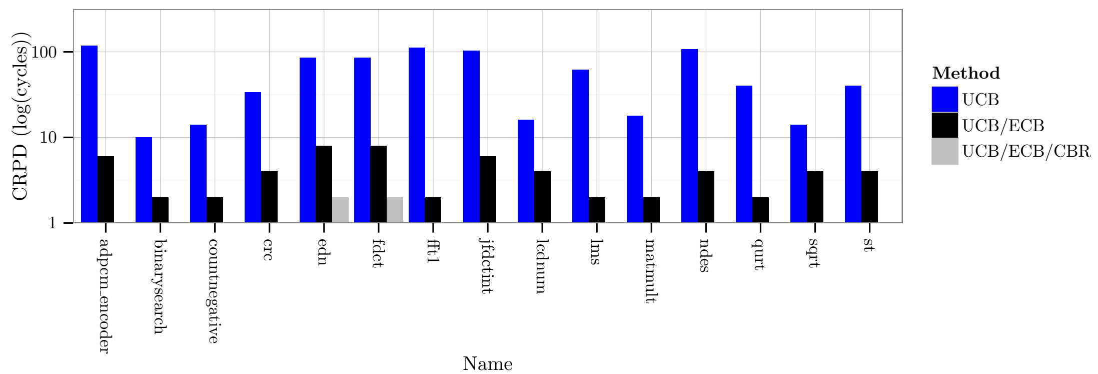


Figure 4.18: MRTC: Preemption by task FIBCALL

We selected a subset of benchmarks for graphical representation which are listed in Table 4.2. The size of benchmarks is given in bytes of the program binaries. UCB and ECB are the maximal values among all cache states for all program points for the former and those of terminal states for the latter. Note that all diagrams depicting CRPD are of logarithmic scale.

In Figure 4.18 CRPD for preemptions by FIBCALL is depicted, which only shows a minimal impact on the preempted tasks due to its small number of ECB. For purely UCB-based computations, CRPD ranges from 16 misses for LCDNUM to 118 misses for ADPCM_ENCODER. Considering the ECB of the preempter already causes a reduction of 75 % (16 to 4 misses) for the first and 95 % (118 to 6 misses) for the latter benchmark. For both benchmarks, CBR-based analysis reduces the estimated CRPD to 0. Only for EDN and FDCT, a CRPD of 2 is estimated as opposed to 86 for the purely UCB based computation and 8 for ECB, in each case.

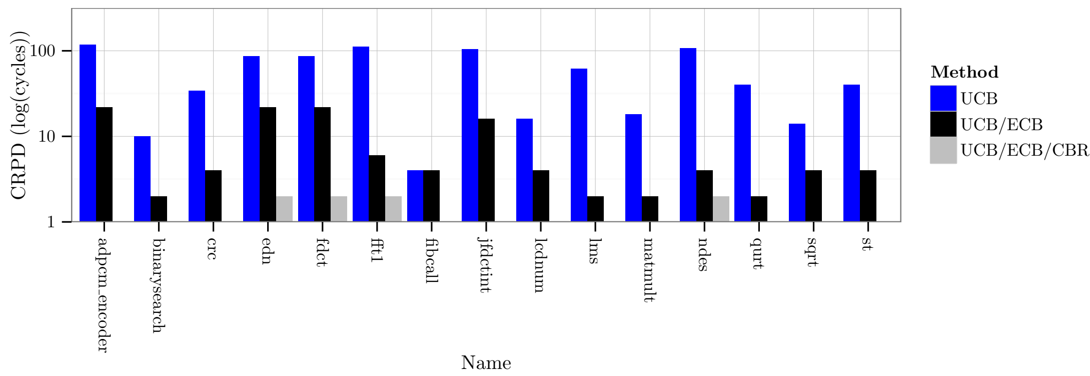


Figure 4.19: MRTC: Preemption by task COUNTNEGATIVE

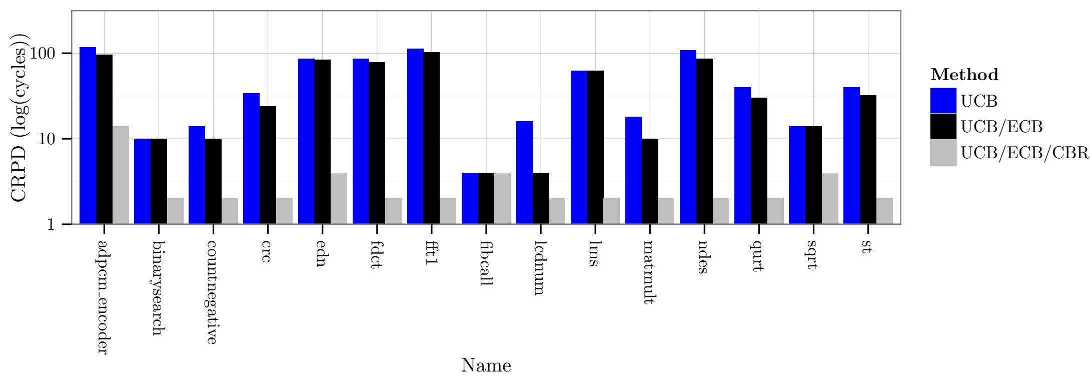


Figure 4.20: MRTC: Preemption by task JFDCTINT

Figure 4.19 illustrates the evaluation for an preempter with a non-extreme number of ECB. The preempting task is COUNTNEGATIVE. The UCB-based CRPD ranges from 4 misses for FIBCALL to 112 misses for FFT1. Taking ECB into consideration does not reduce the number of misses for the first benchmarks, but reduces the amount of additional misses by 95 % (112 to 6 misses). This further reduces to 2 misses under CBR.

Fig. 4.20 depicts the results for preemption with JFDCTINT, which yields a comparably large number of ECB. As can be seen, even though ECB are taken into account, CRPD estimation is almost identical to the estimation with UCB alone. This behavior is typical for these estimations when comparably high cache-usage occurs. Particularly in such cases, CBR-based estimation is superior to the other approaches. For ADPCM_ENCODER for example, a 88 % (118 to 14 misses) tighter bound is computed, whereas ECB-based estimation is just 8 % (118 to 96 misses) tighter than the plain UCB-based computation. The preempter's high cache usage leads to constantly 2 misses in all preemptees.

Figure 4.21 illustrates the saving of program points by the reduction to basic block boundaries, as discussed in Section 4.6.3. The bars show the ratio from all program points to the number of program points required under reduction. Reductions between 96 % for JFDCTINT and 34 % for LCDNUM can be observed. On average, 66 % less program points were required for the computations when limited to basic block bounds (73 on

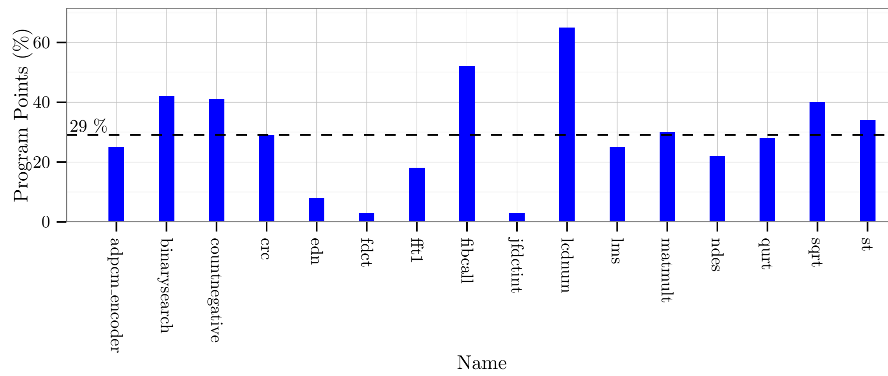


Figure 4.21: MRTC reduction of sample points

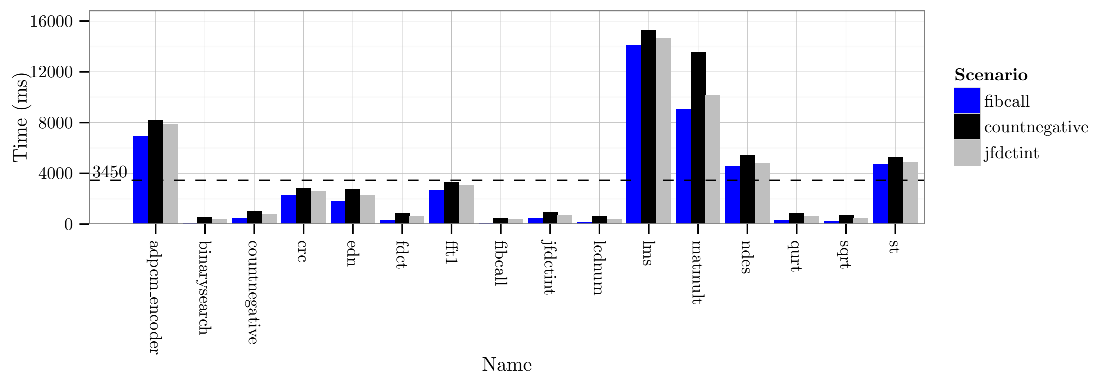


Figure 4.22: MRTC analysis duration

Name	Size (b)	UCB	ECB	Period	Priority	Preemptions
T5	166	4	7	375	5	23
T6	752	5	26	375	6	26
T7	164	5	7	75	2	4
T10	462	9	17	375	7	28
T12	700	5	26	75	3	6
I4	338	3	10	150	4	8
I5	260	3	5	75	0	0
I6	108	3	13	75	1	2

Table 4.3: Properties of PapaBench tasks

average, instead of 328 on average at instruction-level granularity).

The analysis duration for the different preemption scenarios with varying preempters, as just discussed, is shown in Figure 4.22. Duration ranges from 300 ms to 14 s, and takes 3.5 s on average per benchmark and scenario. Despite the presumed inefficiency of the precise domain, this illustrates that with a careful implementation, reasonable analysis performance can be achieved.

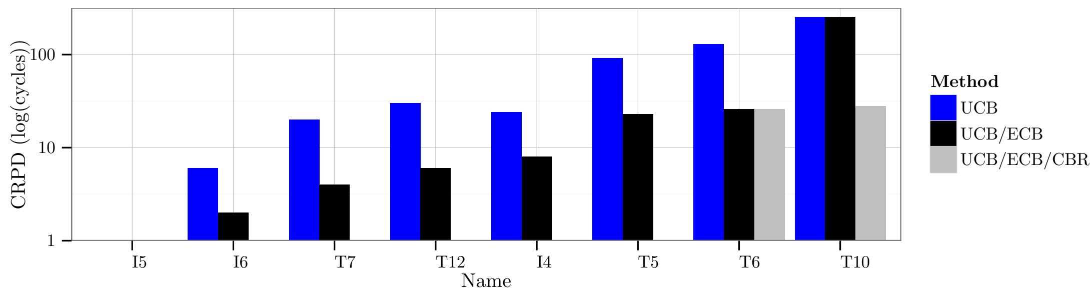


Figure 4.23: PapaBench: tasks preempted by all higher priority tasks

PapaBench

PapaBench composes an entire multitask system for an autonomous aircraft. Table 4.3 lists the considered tasks' properties. The size is given in bytes, the period is given as a cycle-factor[2]. As a side note, the only task we left out is task T9 because its disproportional size, in combination with its predefined high priority, are not beneficial to a meaningful evaluation, since it would result in an actual eviction of the entire cache (regardless of what a CRPD analysis would estimate). Execution modes are not distinguished. To obtain deterministic results from identical periods, we manually set fixed priorities by setting periods off by 1 cycle. The priority equals 0 for the highest priority task. The last column denotes an upper bound on number of total preemptions of the task. The total analysis time is 7 s with only using 2 844 computations on basic block bounds as opposed to 18 462 computations which would be necessary at instruction-level granularity (85 % less).

Figure 4.23 shows the results of CRPD computations for all tasks, where each one can be preempted by all higher priority tasks. As can be seen, the UCB-based analysis is significantly overestimating all CRPD except for I5 which is not preempted. Except for T10, considering the ECB already tightens CRPD estimation significantly (33 % for I6). In all cases (except T6, T10), the CBR-based estimation yields a CRPD of 0. For T6, 80 % (130 to 26 misses, ECB and CBR) and for T10 88 % (252 to 28 misses, CBR) tighter estimations are computed. In all cases, the CBR-based estimations outperform the UCB-only approach by 89 % to 100 %.

Figure 4.24 depicts the ratio for reduced overestimation in successive interaction due to Equation 4.74 and Equation 4.74. For an increasing number of preempters, ECB and therefore CRPD is largely overestimated. The results for I5 and I6, as the two highest priority tasks, are obviously 0 or match the ECB of the one evicting task. For T5, 75 % (54 to 13 ECB) tighter bounds are computed with reduced sets of ECB. On average we computed 58 % tighter bounds.

[2]Period is *value* $\times 10^8$ cycles (at 150 MHz)

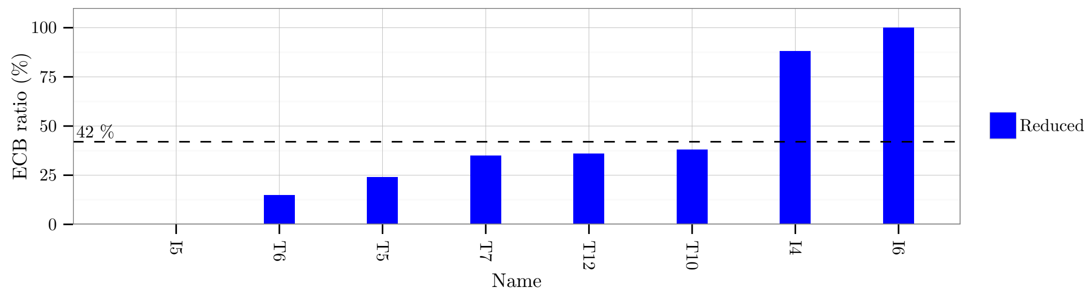


Figure 4.24: PapaBench: Effect of tight ECB composition for interaction

4.8 Conclusion

In this chapter we have discussed static cache analysis in general and we have proposed the application of a precise state domain to improve estimates for set-associative LRU caches by deliberately sacrificing performance for accuracy. We also showed how to construct UCB, ECB and CBR analyses in this framework and discussed its respective application to bound CRPD. In this context we identified imprecision in existing approaches and proposed improvements such as the reduction of sample points for instruction caches and pessimism in preemption interaction. We specifically focused on improved estimates for single preemptions and only addressed CRPD for multiple preempters as far as necessary. A broader and more general discussion focusing on CRPD specifically will follow in Chapter 6.

Chapter 5

Path Analysis

Contents

5.1 Fundamentals of Control Flow Analysis **81**

5.1.1 Flows and Paths 81

5.1.2 Graph Structure 84

5.2 Path Problems in Timing Analysis **91**

5.2.1 On Program Representation 91

5.2.2 On Control Flow Representation 93

5.2.3 On Path Analyses 98

5.3 A General Path Analysis **101**

5.3.1 Motivation 101

5.3.2 Graph Structure and Loops 103

5.3.2.1 Related Work 104

5.3.2.2 Scopes 105

5.3.2.3 A General Algorithm for Precise Loop Detection 107

5.3.2.4 Handling Ambiguous Loop Nesting by Enumeration 121

5.3.2.5 Handling Ambiguous Loop Nesting by Prenumbering 125

5.3.2.6 Conclusion 131

5.3.3 Computing Worst-Case Execution Time Bounds 131

5.3.3.1 Prerequisites 131

5.3.3.2 Computing WCET Bounds on a Single Scope 133

5.3.3.3 Computing WCET Bounds Globally 149

5.3.3.4 Computing WCET Bounds on Subgraphs 151

5.3.3.5 Practical Global Path Length Computation 153

5.3.3.6 Evaluation 159

5.3.3.7 Conclusion 163

5.3.4 Computing Best-case Execution Time Bounds 164

5.3.4.1 Prerequisites 164

5.3.4.2 Framework 165

5.3.4.3 Evaluation 169

5.3.4.4 Conclusion 171

5.3.5 Computing Latest Execution Time Bounds 171

5.3.5.1 Prerequisites 172

5.3.5.2 Framework 173

5.3.5.3 Evaluation 182

5.3.5.4 Conclusion 184

5.3.6 Computing Maximum Blocking Time Bounds 184

5.3.6.1 Prerequisites 186

5.3.6.2 Framework 189

5.3.6.3 Evaluation 202

5.3.6.4 Conclusion 204

5.3.7 Computing Worst-Case Execution Frequencies 204

5.3.7.1 Prerequisites 205

5.3.7.2 Framework 207

5.3.7.3 Evaluation 213

5.3.7.4 Conclusion 216

5.4 Remarks **216**

5.5 Conclusion **218**

In this chapter we are concerned with diverse aspects of path analysis. Recall from Section 3.2.1 that path analysis in the context of timing analysis refers to the consolidation of worst-case timings of program points to compute bounds in the WCET of entire tasks. In general, we regard path analysis as a class of diverse analyses which are concerned with different problems related to timing analysis of which per-task WCET estimation is just a specific case. Doubtlessly, path analysis — synonymously for worst-case path length analysis in this context — as the terminal stage of traditional timing analysis (cf. Figure 3.10 on page 36) is predominant. This, however, leads to an unfortunate focus on just this single problem. The consequence of this development is twofold. First, the objective of per-task timing analysis is just to project all (program) state onto a single scalar value: a bound on WCET. In the face of multitasking, the interfacing between such timing analysis and schedulability analysis is extremely low on information. Historically, this might be explained by the fact that scheduling theory existed before static timing analysis such that the latter only adopted to the requirement. In the face of ever increasing (hardware) complexity (cf. Section 3.2.1) the strict separation of timing analysis from scheduling by such principle is unfortunate. All information on program state is entirely lost in path analysis. CRPD analysis is an exception in that it (cf. Section 4.5.1) defines one additional interface by maintaining summaries of cache usage patterns between timing and scheduling analysis. Providing a better framework with increased expressiveness will allow improved interfacing beyond this. The second and tightly related consequence is that by focusing only on per-task WCET bounds, there

exists a significant lack of general tools for other important problems that frequently occur in the domain of (multitask) timing analysis, such as bounding maximum blocking times or execution frequencies. Another crippling aspect is the predominant use of linear programming (cf. Appendix C) for path analysis, which inherently prevents advances due its lack of expressiveness.

In the following we take a step back from the state of the art and its current focus on per-task timing analysis and propose a fully general and highly efficient framework to address various problems in the domain of path analysis. We partition the discussion into two parts which ultimately leads to the specification of a single unified framework. First, we will address the problem of control flow reconstruction with a focus on loop detection. We propose a new, highly efficient loop detection which is designed to serve the specific problems encountered in timing analysis as opposed to traditional heuristics. Second, we base a general path analysis on the respective program representation. We first show how traditional per-task WCET estimates are obtained, then we discuss several variations. Along with a formal model, we also address practical analyses by proposing an efficient reference implementation and by proposing numerous optimizations. In all cases, our focus is on generality of the formal model and performance of its practical implementation.

Specifically, in Section 5.1 we discuss basic formal principles of control flow analysis. In Section 5.2 we provide an overview of issues related to timing analysis in particular. In Section 5.3 we will then discuss our proposal for a path analysis framework. In Section 5.4 we discuss our proposals in a broader context. In Section 5.5 we conclude the chapter.

5.1 Fundamentals of Control Flow Analysis

In this section we provide a formal introduction to the fundamental concepts of flows and paths on graphs (Section 5.1.1), and important concepts and techniques for graph structure and transformation (Section 5.1.2). The reader may choose to only skim through this section as a refresh of established notions and the relation of various concepts, or to use it as a reference for the upcoming discussion. Note that we also put classical graph algorithms into their respective context, of which we provide reference implementations in Appendix B. Later on we will propose derivatives for some of them. So we assume familiarity with their base versions.

5.1.1 Flows and Paths

In the following we elaborate on the concepts of flows and paths in digraphs, explain their relation and put classical graph algorithms into perspective accordingly.

A *flow network* $G = (V, E, s, t)$ is a digraph with source node s such that $\deg_{in}(s) = 0$, sink node t such that $\deg_{out}(t) = 0$ with the property $\forall u \in V \setminus \{s, t\} \colon s \rightsquigarrow u \rightsquigarrow t$, and an

additional *flow capacity* $c\colon E \mapsto \mathbb{N}_0$. A *flow* $f\colon E \mapsto \mathbb{Z}$, for some value $q \in \mathbb{N}_0$, satisfies:

$$\forall u, v \in V\colon f(u,v) - f(v,u) = \begin{cases} q & \text{if } u = s \\ -q & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \tag{5.1}$$

$$\forall (u,v) \in E\colon f(u,v) \leq c(u,v) \tag{5.2}$$

Sensibly, it also holds that $\forall (u,v) \notin E\colon \Rightarrow c(u,v) = 0$. The valuation of f is called *net flow*. The *value* of a flow is defined as the total flow out of the source: $\sum_{(s,v)\in E} f(s,v)$. It is important to note that, except for s and t, the positive net flow into a node equals the net flow out of it:

$$\forall v \in V \setminus \{s,t\}\colon \sum_{u \in V\colon f(u,v)>0} f(u,v) = \sum_{w \in V\colon f(v,w)>0} f(v,w) \tag{5.3}$$

The *maximum flow problem* (MAXFLOW [99]) is to find a flow of maximum value from source to sink. Formally, we define the problem as:

$$\begin{aligned} \max \quad & \sum_{u \in V} f(s,u) \\ \text{s.t.} \quad & \forall v \in V \setminus \{s,t\}\colon \sum_{u \in V} f(u,v) - \sum_{u \in V} f(v,u) = 0 \\ & \forall u, v \in V\colon f(u,v) \leq c(u,v) \end{aligned} \tag{5.4}$$

Given two nodes $(u,v) \in E$, the additional net flow that can be "pushed" from u to v before exceeding the capacity constraint c is the *residual capacity* $r\colon E \mapsto \mathbb{Z}$ such that:

$$r(u,v) = c(u,v) - f(u,v) \tag{5.5}$$

The graph $G^R = (V, \{(u,v) \in E(G)\colon r(u,v) > 0\})$ is the *residual flow network*, containing only those edges of the original network which have not been *saturated*.

A path $\pi_{s,t}$ from s to t is an *increasing path*, if there exists a corresponding path in G^R. Along such a path, $f_{\min} = \min\{r(u,v) \mid (u,v) \in \pi_{s,t}\}$ additional flow can be pushed. If no such paths exists, the flow is maximal. Thus, successively increasing the flow by finding increasing paths solves MAXFLOW [100][1]. This computation principle is known as the *Ford-Fulkerson method* (cf. Algorithm B.12 on page 256).

An extension to the flow network is the introduction of a lower flow bound $l\colon E \mapsto \mathbb{Z}$ such that for the net flow it must also hold that:

$$\forall (u,v) \in E\colon l(u,v) \leq f(u,v) \leq c(u,v) \tag{5.6}$$

Given such a lower bound, the *minimum flow problem* (MINFLOW [102]) is to find a flow

[1]Unsurprisingly, since 1956 more efficient algorithms have been devised [101].

of minimum value from source to sink. Thus, the objective is to verify feasibility of flow rather than optimization.

To find a path in a flow network, *breadth-first search* (BFS [103], cf. Algorithm B.11 on page 256) can be used. A unit flow represents exactly one path leaving the source. Thus, finding some path from source s to sink t ($\pi_{s,t}$) guarantees at least unit flow. The cardinality $|\pi_{s,t}|$ denotes path length for an implicit weight function $\omega\colon E \mapsto \mathbb{N}_0$ such that $\forall e \in E\colon \omega(e) = 1$. In this restricted case BFS computes a solution to the *single-source shortest paths* problem (SSSP [104]). For generalized weights and an explicit weight function such that $\forall e \in E\colon \omega(e) \geq 0$, path length — in terms of set cardinality — and weight must be discriminated explicitly.

The problem of sending a maximum amount of flow with minimal costs is known as the *minimum cost flow problem* (MINCOST FLOW [101]) and is formalized as:

$$
\begin{aligned}
\min \quad & \sum_{(u,v)\in E} f(u,v)\omega(u,v) \\
\text{s.t.} \quad & \forall u \in V\colon \sum_{w\in V} f(u,w) - \sum_{v\in V} f(v,u) = b_u \\
& b_u = \begin{cases} q & \text{if } u = s \\ -q & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\
& \forall (u,v) \in E\colon l(u,v) \leq f(u,v) \leq c(u,v)
\end{aligned}
\tag{5.7}
$$

The variable b_u is also called *supply* if $b_u > 0$, and *demand* of $b_u < 0$. If we interpret each unit flow as a single walk from s to t, then constraining $q \leq 1$ yields the solution to a problem we refer to as the *minimum path length* problem (MINLEN) in the following: A *single* shortest path through the flow network.

We can extend this idea to define the SSSP. The intuition is that there must be $|V| - 1$ paths leaving source node s and in each node $u \in V \setminus \{s\}$, a shortest path must terminate, respectively. Thus, we formally capture this notion and define:

$$
\begin{aligned}
\min \quad & \sum_{(u,v)\in E} f(u,v)\omega(u,v) \\
\text{s.t.} \quad & \sum_{u\in V} f(u,s) - \sum_{u\in V} f(s,u) = |V| - 1 \\
& \forall v \in V \setminus \{s\}\colon \sum_{u\in V} f(u,v) - \sum_{w\in V} f(v,w) = 1 \\
& \forall (u,v) \in E\colon l(u,v) \leq f(u,v) \leq c(u,v)
\end{aligned}
\tag{5.8}
$$

Algorithm B.13 on page 257 specifies a generalization of BFS[2] to solve SSSP. Note that in Algorithm B.13, upper flow bounds are not taken into account, and for lower flow bounds it is assumed that $\forall (u,v) \in E\colon l(u,v) = 0$. Note that for the minimization problems so

[2]Here, it is a variant of the venerable algorithm proposed by E. Dijkstra [104]

far only non-negative weight graph cycles can be encountered since flow f (Equation 5.4) and weight ω must both be non-negative.

A path $(u_1, \ldots, u_n)$ is cyclic if $\exists i, j \colon i \neq j, u_i = u_j$ and edges (u_{j-1}, u_j) are referred to as *back edges*. Removing all back edges from G forms a *directed acyclic graph* (DAG), in which edges denote the *topological order* of nodes. Given a DAG $\vec{G}$, Algorithm B.14 computes the corresponding sequence of nodes. Shortest paths, given that sequence, can then be computed by means of Algorithm B.15.

Inversely, for the computation of *longest paths*, cycles cannot be ignored any longer and, hence, upper flow bounds must explicitly be taken into account. In a flow network with unit flow, path lengths are bounded by capacity constraints. Each traversal of a node (in a cycle) increases its specific net flow but not the flow (value). Hence, path length is bounded since net flow is bounded, and for a single path, the flow (value) is constrained to equal 1. Thus, the *maximum path length* problem (MAXLEN) for a weighted flow network is formally defined as:

$$\begin{aligned} \max \quad & \sum_{(u,v) \in E} f(u,v)\omega(u,v) \\ \text{s.t.} \quad & \forall u \in V \colon \sum_{w \in V} f(u,w) - \sum_{v \in V} f(v,u) = b_u \\ & b_u = \begin{cases} 1 & \text{if } u = s \\ -1 & \text{if } u = t \\ = 0 & \text{otherwise} \end{cases} \\ & \forall (u,v) \in E \colon l(u,v) \leq f(u,v) \leq c(u,v) \end{aligned} \tag{5.9}$$

Analogously, *single-source longest paths* (SSLP) could be defined. MAXLEN is of particular interest in timing analysis. We will devote Section 5.2 specifically to this topic.

In case of node weights and node capacities, they can easily be mapped to edges by assigning them to either in- or out-edges [105] such that, for example, for a node capacity $\eta \colon V \mapsto \mathbb{N}_0$ we define $\forall (u,v) \in E \colon c(u,v) = \eta(u)$. Inversely, edge capacities and weights are mapped to nodes by expanding the underlying graph to contain additional nodes for each edge such that, for example, for the expanded edge set E' it holds that $\forall (u,w) \in E \colon \exists (u,v), (v,w) \in E' \colon \eta(v) = c(u,w)$.

5.1.2 Graph Structure

In this section we give an overview of the most important concepts and techniques related to graph structure. We first define basic terminology, then discuss Depth-First Search and its properties, and address graph reducibility, loops and graph grammars.

Basic Terms

We assume a connected digraph $G = (V, E, s, t)$ with source node s and sink node t.

Definition 5.1 (Subgraph) *For G and a set of nodes $W \subseteq V$, the* subgraph *$G|W$ induced by W is defined as $G|W = (W, (W \times W) \cap E)$ such that:*

$$G|W = (W, H) \subseteq (V, E) := W \subseteq V \wedge H \subseteq E \tag{5.10}$$

Definition 5.2 (Reduced Graph) *In a* reduced graph*, subgraphs $G|W = (W, H)$ are replaced by representative nodes. Formally, we define $G \setminus_w G|W = (V', E')$, for a representative $w \in W$, where*

$$V' = (V \setminus W) \cup \{w\} \tag{5.11}$$

$$E' = \{(u, w) \mid (u, v) \in E, v \in W\} \cup \{(w, v) \mid (u, v) \in E, u \in W\} \tag{5.12}$$

Definition 5.3 (Transitive Graph Closure [106]) *The* transitive closure *of G is $G^+ = (V, \{(u, v) \in E\colon u \rightsquigarrow v\})$, which extends the set of edges such that all reachable nodes become adjacent.*

Definition 5.4 (Dominance Relation) *Node $u \in V$* dominates *node $v \in V$, if all paths from s to v pass through u:*

$$u \operatorname{dom} v := \forall \pi_{s,v} \in \Pi_G\colon u \in \pi_{s,v} \tag{5.13}$$

By definition every node dominates itself, but it does not strictly *dominate itself. The* immediate dominator *of a node $u \in V$ is a strictly dominating node $v \in V$, which does not strictly dominate any other node.*

Definition 5.5 (Strongly Connected Component) *Two nodes $u, v \in V$ are* strongly connected *if they can reach each other in G, denoted by the relation $\overset{sc}{\sim}$ such that:*

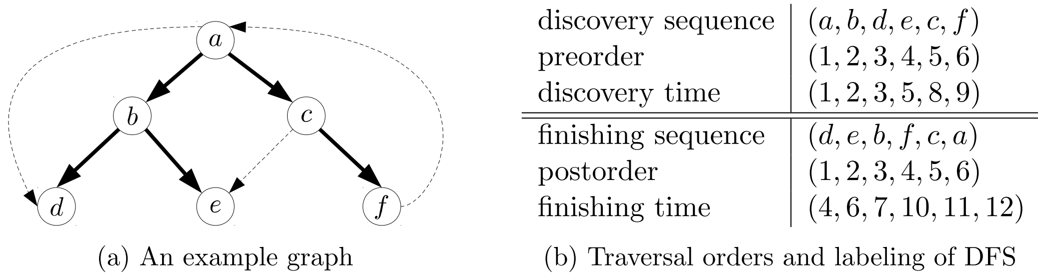
$$u \overset{sc}{\sim} v := \{(u, v), (v, u)\} \subseteq E(G^+) \tag{5.14}$$

This relation induces equivalence classes of nodes, the strongly connected components *(SCC) $V/\overset{sc}{\sim}$, which form maximal subgraphs maintaining strong connectedness. By definition, strongly connected components are disjoint.*

Definition 5.6 (Condensation Graph) *Let $S = \{S_1, S_2, \dots\}$ denote the set of SCCs. Then the graph $(((G \setminus G|S_1) \setminus G|S_2) \dots)$ is referred to as the* condensation graph *of G, which is by definition acyclic.*

Depth-first Search

A *depth-first search* (DFS) [107] reveals the structure of a connected digraph $G = (V, E)$ by partitioning its set of edges into *tree edges* T, *back edges* B, *forward edges* F and *cross edges* C such that $T \cup B \cup F \cup C = E$, and it labels nodes with time stamps reflecting discovery and finished processing.



discovery sequence	(a, b, d, e, c, f)
preorder	$(1, 2, 3, 4, 5, 6)$
discovery time	$(1, 2, 3, 5, 8, 9)$
finishing sequence	(d, e, b, f, c, a)
postorder	$(1, 2, 3, 4, 5, 6)$
finishing time	$(4, 6, 7, 10, 11, 12)$

(a) An example graph (b) Traversal orders and labeling of DFS

Figure 5.1: Example of a DFS application to a given graph

Definition 5.7 (Depth-first Spanning Tree) *The set of tree edges T induces a* depth-first spanning tree *(DFST) $D = (V, T)$ with $V(D) = \{u \mid (u, v) \in T \vee (v, u) \in T\}$.*

Example *In Figure 5.1a thick edges denote tree edges, (a, d) is a forward edge, (c, e) is a cross edge and (f, a) is a back edge.*

Algorithm B.16 on page 259 specifies a non-recursive implementation of DFS which, besides edge classification, returns a pair of time stamps that denote *discovery time* and *finishing time* of nodes for the traversal of the corresponding DFST. In the following we denote the discovery time with $d \colon V \mapsto \mathbb{N}_0$ and the finishing time with $f \colon V \mapsto \mathbb{N}_0$, respectively. "Preorder" and "postorder" sequence numbers [108] are "dense" representations of d and f, respectively. The difference is the use of a shared counter for both labels. As an example, Figure 5.1b lists the various sequence labels. We now formally elaborate on their relation.

Lemma 5.8 *Let ${}^\bullet t \colon V \mapsto \mathbb{N}_0$ and $t^\bullet \colon V \mapsto \mathbb{N}_0$ denote preorder and postorder numbers of the DFST, then it holds:*

$$d(u) < d(v) \Leftrightarrow {}^\bullet t(u) < {}^\bullet t(v) \tag{5.15}$$

$$f(u) < f(v) \Leftrightarrow t^\bullet(u) < t^\bullet(v) \tag{5.16}$$

In particular, $\forall u \in V : d(u) \leq f(u)$.

Proof. In DFS, node u is pushed onto the stack before it is popped off. □

Lemma 5.9 *For nodes $u, v \in V$, it holds that: $d(u) \neq d(v) \neq f(u) \neq f(v)$*

Proof. It holds that $d(u) < f(u)$ and each newly visited node increases the sequence counter. Hence, $d(u) < d(v) \vee d(u) > d(v)$. □

Lemma 5.10 *For nodes $u, v \in V$, it holds that if $d(u) < d(v)$ then $f(u) > f(v)$.*

Proof. Unfinished nodes are queued in LIFO order. □

Theorem 5.11 (Parenthesis) *For a graph $G = (V, E)$, and $u, v \in V$ such that $d[u] < d[v]$, it holds that the intervals $[d(u), f(u)]$ and $[d(v), f(v)]$ are either nested or disjoint.*

Proof. If $d(u) < d(v) < f(v)$, then node u is on the stack when node v is reached first, and u is still on the stack, when v is finished. Otherwise, u is not on the stack when v is visited first. □

Theorem 5.12 (DFST Reachability) *A node u is a predecessor of a node v if and only if it holds that:*

$$^{\bullet}t(u) < {}^{\bullet}t(v) \wedge t^{\bullet}(u) > t^{\bullet}(v) \tag{5.17}$$

Proof. Follows directly from Lemma 5.8 and Theorem 5.11. □

Note that *reverse postorder* $t^{\bullet^{-1}} : V \mapsto \mathbb{N}_0$ denotes topological order imposed by the DFST. Note also that BFS order is not necessarily a topological order. In Figure 5.3b node a could be visited before b' in BFS order — as opposed to topological order.

Graph Reducibility

An important class of digraphs for program analysis is the reducible graph. In the following we will characterize its properties.

We use traditional notation. Let S be a set and let $\Longrightarrow$ be a relation on S such that $a_1 \Longrightarrow a_2 \Longrightarrow \dots a_n$, with $a_i \in S$ is a chain of length n. We write $a_i \overset{*}{\Longrightarrow} a_j$ to denote the existence of a chain from a_i to a_j. The relation is finite, if it holds that for all $a_i \in S$, with all a_i distinct, there exists an $n \in \mathbb{N}_0$ such that for a chain $a_1 \Longrightarrow a_2 \Longrightarrow \dots a_k$, it holds that $k \leq n$.

Definition 5.13 (Church-Rosser Transformation [109]) *A relation $\Longrightarrow$, for a set S, is a finite* Church-Rosser transformation *(or* locally confluent*), if and only if the relation is finite and for an element $a \in S$, it holds that $\forall b, c \in S\colon a \Longrightarrow b \wedge a \Longrightarrow c$ implies $\exists t \in S\colon b \overset{*}{\Longrightarrow} d$ and $c \overset{*}{\Longrightarrow} d$.*

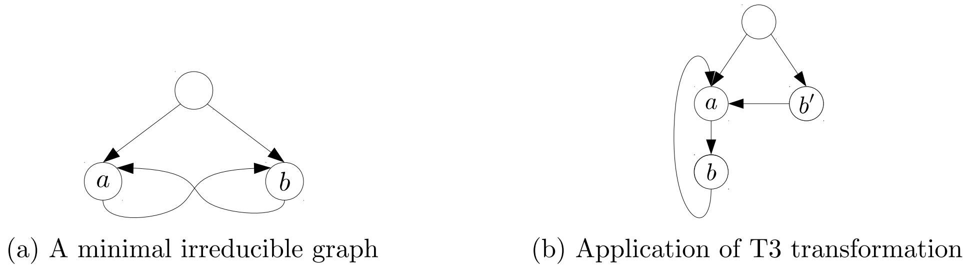
Figure 5.2: Grammar of T1-T2

We define a finite Church-Rosser transformation by means of the grammar specified in Figure 5.2, which is known as *T1-T2* reduction [110], to parse a graph by iteratively applying the following steps according to the given production rules:

T1) Remove any edge $(u, u) \in E$ that connects a node $u \in V$ to itself (self-loop).
T2) For any node u that has exactly one predecessor v, reduce $G^{(i)}$ such that $G^{(i+1)} = G^{(i)} \setminus G^{(i)}|\{u, v\}$.

Definition 5.14 (Reducibility [110, 111]) *A graph $G = (V, E)$ is* reducible *if and only if T1-T2 transformation results in a trivial graph $G^{(k)} = (V, E)$ with $|V| = 1$.*

A consequence of Theorem 5.13 is that graph reduction is independent of the order in which it is applied and therefore is a function of the graph alone rather than also depending on a specific traversal.



(a) A minimal irreducible graph (b) Application of T3 transformation

Figure 5.3: Example of an irreducible graph and its corresponding node splitting

Example *Figure 5.3a illustrates a minimal graph for which further reduction by T1-T2 is not possible.*

An irreducible graph can be transformed into a reducible one by *node splitting* [112–114]. T1-T2 transformation, as specified above, is extended by a third step:

T3) For any node u with at least two predecessors, duplicate u and reconnect an edge from one of the predecessors to the duplicate.

Example *Figure 5.3b depicts the graph from Figure 5.3a with node splitting applied.*

All reducible graphs share the following properties.

Theorem 5.15 *If a graph $G = (V, E)$ is reducible, then it holds that:*

- *The set of back edges B is the same for all DFST of G.*
- *For all back edges $(u, v) \in B$ it holds that $v \operatorname{dom} v$.*
- *All* loops *have a single entry.*

Proof. See [110, 115]. □

We still have yet to characterize loops specifically.

Loops

We refer to the set $B_n \subseteq B$ such that $\forall (u, v) \in B_n \colon v \operatorname{dom} u$ as *natural* back edges. This notion is more strict than general back edges as defined above.

Definition 5.16 (Natural Loop) *For a maximal set of back edges $B_L \subseteq B$ such that $(b_i, h) \in B_L$, a* natural loop *is a maximal set $L \subseteq V$ such that $\forall u \in L \colon h \operatorname{dom} u$ and all $u \in L$ can reach any b_i without passing through h. We refer to h as the* loop head *and to b_i as a* loop bottom*. A* loop body *is the acyclic subgraph $G_L = (V(G|L), E((G|L) \setminus B)$.*

Note that our definition is more strict than in the literature [20, 116], in that we strictly but without loss of generality interpret a set of back edges sharing the same head as just a single natural loop.

Lemma 5.17 *A natural loop L_h is uniquely identified by its head $h \in V$.*

Natural loops can be identified by DFS deterministically (Theorem 5.15). The implication of this is that in reducible graphs, loop identification is a function independent of the specific DFST traversal order and therefore solely depends on the graph structure.

As a direct consequence of the parenthesis theorem (Theorem 5.11), we conclude:

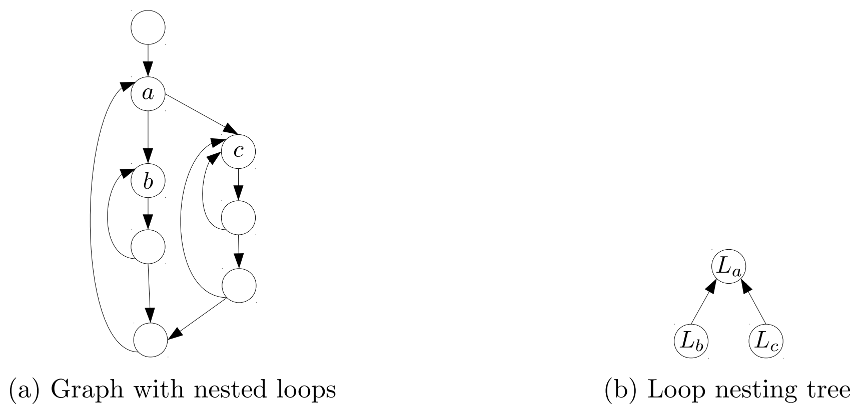
Theorem 5.18 *For two natural loops L and L', it holds that either $L \subset L'$ or $L \cap L' = \emptyset$.*

Proof. A loop head uniquely identifies a loop and Theorem 5.11 applies. □

Definition 5.19 (Entry, Exit) *Let $G = (V, E, s, t)$. Given a subgraph $G|S$ with $S \subseteq V$, a node $v \in S$ is an* entry *if and only if either $v = s$ or $\exists(u, v) \in E\colon u \neq S$. Node v is an* exit *if and only if either $v = t$ or $\exists(v, w) \in E\colon w \neq S$.*

In irreducible graphs, not all paths pass through potential loop headers — their sole entry. Specifically, all and only irreducible graphs yield loops with multiple entries [110].

Definition 5.20 (Loop Nesting Forest [117]) *A natural loop L with header h is* nested *within a loop $L' \supset L$ if $h \in L'$. Let $\mathbb{L}$ denote a set of loops and $L_h \in \mathbb{L}$ denote a loop with head h, then $N = (\{h \mid L_h \in \mathbb{L}\}, \{(h, h')|L_h \supset L_{h'}\})$ is a loop nesting forest. Its edges define its* nesting relation. *The* loop depth *is the distance from the root in a loop nesting tree.*



(a) Graph with nested loops

(b) Loop nesting tree

Figure 5.4: Example of a reducible digraph and its corresponding loop nesting tree

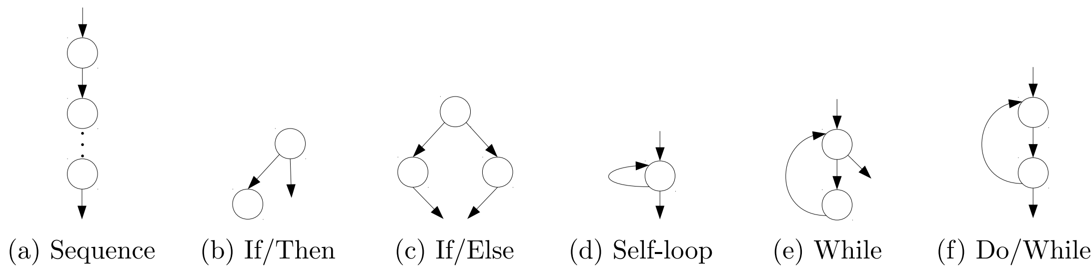
Example *Figure 5.4a illustrates a digraph of three loops with loop headers a, b and c, respectively. Figure 5.4b depicts the corresponding loop nesting forest, consisting of just a single tree.*

Definition 5.21 (Iteration) *An iteration of a loop is a path that starts in an entry and ends in either an exit or a bottom without passing through any back edge of this loop. It may pass through back edges of nested loops though.*

Definition 5.22 (Kernel, Entry Path, Exit Path) *Given a loop L with entries $I(L)$, exits $O(L)$, head $h(L)$ and bottoms $B(L)$. A* kernel *is a path $(h(L), \ldots, b)\colon b \in B(L)$ from the loop head to a bottom. An* entry path *is a path $(i, \ldots, b)\colon i \in I(L), b \in B(L)$ from an entry to a bottom. An* exit path *$(h(L), \ldots, o)\colon o \in O(L)$ from the head to an exit.*

Graph Grammars

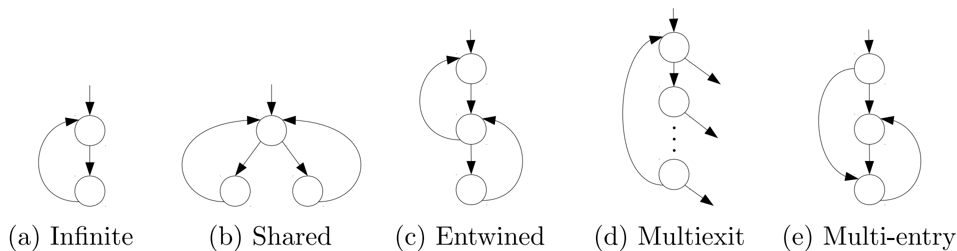
The construction of a loop nesting forest corresponds to the parsing of a graph with an appropriate grammar. T1-T2 as specified in Figure 5.2 is only one possibility. We can think of a loop nesting forest as just a set of *abstract syntax trees* (AST) [116] as a model of a certain language. As a case in point, T1-T2 is a very simple grammar of a language which allows for the discovery of loop structure only. In practice, different source (programming) languages typically share common constructs and it might be of interest, for program analysis in particular, to expose structure with a more detailed grammar. For example, the *semi-structured flow graph grammar* (SSFG) [118] provides production rules for such typical constructs (*structural analysis* [119]) and allows for the identification of specific control flow constructs such as sequences, loops and different kinds of branching constructs. Figure 5.5 illustrates examples for typical control flow constructs subject



(a) Sequence (b) If/Then (c) If/Else (d) Self-loop (e) While (f) Do/While

Figure 5.5: Typical grammatical constructs

to reduction. Complementing the list with node splitting as in Figure 5.3a allows the reduction of irreducible regions. Notably, Figure 5.5d, Figure 5.5e and Figure 5.5f define different constructs which are all natural loops according to Definition 5.16. In our



(a) Infinite (b) Shared (c) Entwined (d) Multiexit (e) Multi-entry

Figure 5.6: Typical loop variants

context, identification of such specific loop types is of particular interest as they yield information on loop iteration semantics. Figure 5.6 lists loop types — in addition to the three loop types in Figure 5.5 — that are typically encountered in practice. Note that

the constructs listed may be compositions of the aforementioned ones. Also note that Figure 5.6e is equivalent to Figure 5.3a, and, according to Definition 5.16, Figure 5.6b forms a single loop.

Numerous parsing algorithms have been proposed in the literature [21, 111, 120, 121] that are significantly more efficient than T1-T2 and which are specifically concerned with the construction of loop nesting forests. They typically depend on information about node dominance and edge classification — hence, they ultimately depend on reducibility — to form larger regions for reduction to speed up processing [24]. Irreducibility can also be resolved more efficiently than with T3 transformation as defined above [122, 123]. Other algorithms handle irreducible graphs directly [120, 124, 125]. Famously, the author of [126] noted that most control flow graphs are reducible due to constraints imposed by the source (programming) language.

5.2 Path Problems in Timing Analysis

In timing analysis, we are concerned with two kinds of path-related problems. First, static analyses such as those for values, caches and CPU pipelines depend on the availability of a control flow graph — depending on the representation of the source program under analysis its construction is not necessarily straight forward [26]. Second, after domain-specific analyses of timing behavior of program points, timing information needs to be consolidated to compute the worst-case path length in the CFG, given time bounds as weights of nodes on this path. In Figure 3.10 on page 36 these problems are illustrated as reconstruction and path analysis phase, respectively.

In the following we briefly address the relation of program representation and timing analysis in Section 5.2.1. Then we discuss control flow representation and the role of flow facts in this context specifically in Section 5.2.2 and define their relation to path analysis along with related work on path analysis in general in Section 5.2.3.

5.2.1 On Program Representation

The reconstruction of control flow refers to the process of extracting a control flow graph from a given program representation. Here, we shall summarize different existing approaches. Note that we assume only statically typed and statically bound source programs in the following (think ANSI C).

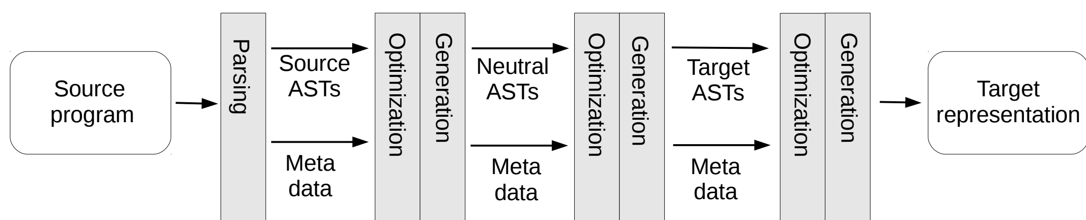


Figure 5.7: Typical compilation pipeline

Figure 5.7 illustrates the typical stages of a compilation pipeline. A program of a given source language is parsed to construct syntax trees and meta data. The latter typically includes reverse program location mappings for debug purposes but might also include compiler-specific language extensions to instruct subsequent stages. Representation at this level preserves structural information of a given source program, such as information subject to graph structure (cf. Section 5.1.2). Optimization at this level involves expression simplifications but also, and in particular, optimization of loop structure. Only at this level the originally "intended" loop structure is available, without having to deal with loop detection and resolving ambiguity, such as depicted in Figure 5.6. For meta data — such as reverse mappings — loop transformations have to be taken into account. For multi-source-multi-target compilers, it is then convenient to generate a neutral, more efficient program representation for optimizations that are not source-level dependent. At this stage, meta data is the only link to the source representation. In particular, transformations are usually not structure preserving anymore, merely semantic preserving. For efficient analysis and optimization, irreducible programs would typically be transformed into their reducible semantic equivalents [113, 122, 123] — for example by node splitting — and transformed into SSA form [116]. It is also here that reverse mappings in meta data potentially becomes imprecise. The final representation is target-specific but structure and semantic preserving. Target-specific problems such as instruction scheduling and register allocation is addressed here. The target representation then includes a binary stream of CPU instructions, static data values, memory maps, symbol tables and meta data. For details on the specific techniques, see [20, 116].

Static timing analysis can in principle be performed at any representation level in this pipeline, as long as its safety can be guaranteed.

Analysis at a high representation level has the advantage that structure is preserved: high-level constructs are immediately available and consequently no ambiguities arise. Moreover, expressions are still being preserved in their symbolic form. Loop index variables and termination conditions are easier to determine than in any lowered representation. For timing analysis, bounds on loop iterations are of particular interest to bound path lengths [56, 127, 128]. The drawback of such an approach is that nothing can be said about target architecture semantics. Neither CPU instructions nor memory layout are known at this stage. A possible approach then is to perform low-level timing analysis and to rely on meta data to *back-annotate* high-level constructs with timing data. Reverse mapping becomes more imprecise by transformations [129] between representations and optimization. Hence, soundness is hard to guarantee [130].

In turn, analysis at a low representation level allows for sound and (comparably) precise micro-architectural analysis at the expense of original program structure potentially being lost. Recovery of this information can be a problem, as structural ambiguities can arise, in particular given irreducible program structure, but also regarding loop nesting. This also relates to the problem of meta data co-transformation: Loop structure and loop bounds obtained from higher levels must continue to be sound even after program

representation has changed [131]. At the lowest possible level, not even meta data might be available. The problem then is twofold. First, a control flow graph must be reconstructed from a binary stream representing instructions (cf. Section 5.2.1). Second, sound loop bounds must be obtained automatically [56, 127, 128, 132], and, in case this fails, must be provided by means of manual annotations.

Compiler-support for low-level timing analysis is an active research topic [44]. The *WCET-aware C compiler* (WCC) [96], for example, supports co-transformation of high-level loop bounds that are either manually specified as a language extension or automatically derived [55]. It makes use of *aiT* [58] for static timing analysis on binary code. A similar framework is the *Open Timing Analysis Platform* (OTAP) [133], which is based on LLVM [134] as an architecture-neutral program representation and is able to employ the *SWEedish Execution time Tool* (SWEET) [62] for flow fact analysis, based on a similar intermediate format, called ALF [135]. OTAP can also make use of aiT.

5.2.2 On Control Flow Representation

We will now discuss important basics of control flow representation, characterize information loss in CFGs and introduce path expressions as a convenient formal representation of path related problems. In this context, we also put flow facts into perspective. The section forms the basis of the subsequent discussion of practical path analyses.

Control Flow Abstraction

In Section 2.3 we introduced control flow graphs as an abstraction of execution paths. To obtain a precise notion control flow reconstruction from some representation and the role of flow facts in path analysis, we now formally define CFG abstraction. Let $\widehat{\mathbb{D}}_\Pi$ denote a set of CFGs such that $G = (V, E, s, t) \in \widehat{\mathbb{D}}_\Pi$ is a (sound) abstraction for a set of (concrete) paths $\Pi \in \mathbb{D}_\Pi$. Without loss of generality, we assume $\forall (u_s, \ldots, u_t), (v_s, \ldots, v_t) \in \Pi \colon u_s = v_s \wedge u_t = v_t$.

We first define abstraction. Let P denote a set of paths, then let f_v denote the set of all elements in P such that:

$$f_v := \lambda P \,.\, \{u \mid u \in \pi, \pi \in P\} \tag{5.18}$$

and let f_e denote the set of consecutive pairs in all sequences such that:

$$f_e := \lambda S \,.\, \{(u_i, u_{i+1}) \mid (u_i, u_{i+1}) \in P\} \tag{5.19}$$

Then abstraction α_Π is defined as:

$$\begin{aligned} \alpha_\Pi \colon \mathbb{D}_\Pi &\mapsto \widehat{\mathbb{D}}_\Pi \\ \alpha_\Pi(\Pi) &= (f_v(\Pi), f_e(\Pi), u_s, u_t) \end{aligned} \tag{5.20}$$

We now define concretization which computes from a CFG a set of paths. To this end, we define a helper function which recursively extends all paths in a given set according control flow relation R for a source node u and a terminal node w as:

$$p \colon V \times V \times V^2 \mapsto \wp(V^*)$$
$$p(u,w,R) = \begin{cases} \{\pi \cdot (w) \mid \pi \in p(u,v,R), (v,w) \in R\} & \text{if } u \neq w \\ \{(u)\} & \text{otherwise} \end{cases} \tag{5.21}$$

Then concretization γ_Π is defined as:

$$\gamma_\Pi \colon \widehat{\mathbb{D}}_\Pi \mapsto \mathbb{D}_\Pi$$
$$\gamma_\Pi(G) = p(s,t,E) \tag{5.22}$$

We also define the corresponding transformers for both domains. For the concrete domain $\mathbb{D}_\pi$, transformer tf_Π is just path-based forward collecting semantics, defined as:

$$\mathrm{tf}_\Pi \colon V \mapsto \mathbb{D}_\Pi \mapsto \mathbb{D}_\Pi$$
$$\mathrm{tf}_\Pi(u) = \lambda P \,.\, \{\pi \cdot u \mid \pi \in P\} \tag{5.23}$$

Let $\mathrm{succ}_\Pi(u) = \lambda P.\, \{v \mid (u,v) \in P\}$ denote successors, then the abstract transformer $\widehat{\mathrm{tf}_\Pi}$ is defined as:

$$\widehat{\mathrm{tf}_\Pi} \colon V \mapsto \widehat{\mathbb{D}}_\Pi \mapsto \widehat{\mathbb{D}}_\Pi$$
$$\widehat{\mathrm{tf}_\Pi}(u) = \lambda G \,.\, ((V \cup \{u\}, E \cup \{(v,u)\}) \mid v \in V \wedge u \in \mathrm{succ}_\Pi(v)) \tag{5.24}$$

Correctness of the abstraction is easy to see. To be sound, the abstract domain must be a poset, the abstraction must be sound and the transformers must be locally consistent.

1. *$(\widehat{\mathbb{D}}_\Pi, \sqsubseteq_G)$ is a poset*:
   The order relation $\sqsubseteq_G$ is defined as set inclusion: $G \sqsubseteq G' \Leftrightarrow V(G) \subseteq V(G') \wedge E(G) \subseteq E(G')$. $\widehat{\mathbb{D}}_\Pi$ is a complete lattice with $\top = (V, V^2)$ and $\bot = (V, \emptyset)$ with $G \bigsqcup G' = (V(G) \cup V(G'), E(G) \cup E(G'))$.
2. *Abstraction α_Π and concretization γ_Π form a sound abstraction:*
   The abstract transformer is monotone by definition (set union) such that $\forall G, G' \in \widehat{\mathbb{D}}_\Pi \colon G \sqsubseteq G' \Leftrightarrow \widehat{\mathrm{tf}_\Pi}(u)(G) \sqsubseteq \widehat{\mathrm{tf}_\Pi}(u)(G')$. By construction, a CFG overapproximates sets of paths $\forall P \in \mathbb{D}_\Pi \colon P \subseteq \gamma_\Pi(\alpha_\Pi(P))$.
3. *The transfer functions* tf_Π *and* $\widehat{\mathrm{tf}_\Pi}$ *are locally consistent:*
   $\forall G \in \widehat{\mathbb{D}}_\Pi \colon \forall u \in V(G) \colon (\mathrm{tf}_\Pi(u) \circ \gamma_\Pi)(G) \sqsubseteq (\gamma_\Pi \circ \widehat{\mathrm{tf}_\Pi}(u))(G)$.

Note also that abstraction and concretization form the Galois Insertion $(\mathbb{D}_\Pi, \alpha, \gamma, \widehat{\mathbb{D}}_\Pi)$.

The computation of all possible paths is infeasible in general and a CFG must be constructed by means of sound *heuristics*. The problem of control flow reconstruction is to find a suitable approximation $\widehat{\mathrm{succ}}_\Pi \colon V \mapsto \wp(V)$ such that $\forall u \in V \colon \mathrm{succ}_\Pi(u) \subseteq \widehat{\mathrm{succ}}_\Pi(u)$, which enables the construction of a sound CFG by Equation 5.24.

Figure 5.8: Two example graphs to demonstrate unboundedness and infeasibility

Unbounded Concretization, Imprecision and Flow Facts

Let us characterize the loss of information due to abstraction of paths by means of CFGs.

First, it is easy to see that without further constraints, cycles yield unbounded sets of paths

Lemma 5.23 *For a cyclic CFG G, its concretization $\gamma_\Pi(G)$ is unbounded.*

Example *Consider the CFG illustrated in Figure 5.8a. According to Equation 5.22, concretization yields an unbounded set of unbounded paths:*

$$
\begin{aligned}
\gamma_\Pi(G) &= p(s,t,E) \\
&= \{p(s,u,E)\cdot(t)\} \\
&= \{p(s,s,E)\cdot(u,t), p(s,u,E)\cdot(u,t)\} \\
&= \{(s,u,t), p(s,u,E)\cdot(u,t), p(s,u,E)\cdot(u,u,t),\dots\} \\
&= \{(s,u,t), p(s,s,E)\cdot(u,u,t), p(s,s,E)\cdot(u,u,u,t),\dots\} \\
&= \dots \\
&= \{(s,u,t),(s,u,u,t),(s,u,u,u,t),\dots\}
\end{aligned}
$$

Second, another source of imprecision is infeasible (mutually exclusive) paths. Since a CFG does not encode execution history per se, its concretization contains all structurally possible but — under execution — potentially infeasible paths.

Lemma 5.24 *Even for a set of acyclic paths $\Pi_{s,t} = \{(s,\dots,t)\}$, it holds that $\Pi_{s,t} \subseteq \gamma_\Pi(\alpha_\Pi(\Pi_{s,t}))$.*

Example *Consider Figure 5.8b, which depicts the CFG $\alpha_\Pi(P)$ corresponding to the set of paths $P = \{(s,a,b,c,t),(s,a,x,b,c,t),(s,a,b,y,c,t)\}$. Apparently, nodes x and y are mutually exclusive in concrete semantics but concretization yields the set $\gamma_\Pi(G) = \Pi \cup \{(s,a,x,b,y,c,t)\}$, which includes all structurally possible paths.*

Additional information needed to obtain sound and tight concretization is referred to as *flow facts* [4, 136–138]. These include — but are not necessarily restricted to — constraints on the repetition of nodes on paths to obtain bounded path sets or possibly denote mutual exclusion. We cumulatively refer to this subset of flow facts as *flow constraints.*

Path Expressions

We shall formalize the notion of flow constraints to establish a well-defined connection between program structure, flow constraints and path problems, which subsequently directly leads us to matters of practical path analysis.

Without loss of generality, we assume reducibility (see [9] for irreducible graphs). For practical reasons, we use original notation in the following.

Every path in a CFG $G = (V, E, s, t)$ can be interpreted as a string over its edges[3] E. For nodes $u, v \in V$, a *path expression* [139] is a regular expression [17] P *of type* (u, v) (written as P(u,v)) such that every string π in the language $L(P) \subseteq V^*$ is a path from u to v. Note that $L(P)$ equals concretization $\gamma_\Pi(G)$ (Equation 5.22).

Let $P(u, v)$ be a path expression of type (u, v). Then subexpressions P_1 and P_2 of P are also path expressions whose type is recursively defined by the productions:

$$P(u,v) := P_1(u,v) \cup P_2(u,v) \tag{5.25}$$

$$P(u,w) := P_1(u,v) \cdot P_2(v,w) \tag{5.26}$$

$$P(u,u) := P^*(u,u) \tag{5.27}$$

These rules define alternative paths (5.25), concatenation (5.26) and repetition (5.27), respectively. *Complete* path expressions describe all structurally possible (but yet unconstrained) paths of a CFG.

The underlying algebraic structure of path expressions is a Kleene algebra [140] (idempotent semi-ring with additional “Kleene closure” operator) $(E, \cup, \emptyset, \cdot, \epsilon, ^*)$, where $\cup$ is addition with neutral element $\emptyset$, $\cdot$ is multiplication with neutral element ϵ (the empty string) and the additional operator *, which denotes repetition (Kleene closure). The order of operator precedence is $^* > \cdot > \cup$. For convenience, we omit $\cdot$ for multiplication and parenthesis if possible.

The construction of path expressions corresponds to structural analysis of loops since we are not just recovering paths as in concretization $\gamma_\Pi(G)$ (Equation 5.22) but also represent repetitions more efficiently.

For the CFG with edges classified by DFS, let $E^F = E \setminus B = T \cup F \cup C$ refer to the set of non-back edges and let $H \subseteq V$ denote loop heads. Then path expression P for a reducible[4] CFG from source s to sink t is recursively defined as:

$$P(s,t) = \begin{cases} \bigcup_{(u,t) \in E^F} P(s,u)(u,t) & \text{if } t \notin H \wedge s \neq t \\ (\bigcup_{(u,t) \in E^F} P(s,u)(u,t))P(t,t) & \text{if } t \in H \wedge s \neq t \\ \epsilon & \text{if } s = t \end{cases} \tag{5.28}$$

$$P(h,h) = (\bigcup_{(b,h) \in E \setminus E^F} P(h,b)(b,h))^* \tag{5.29}$$

[3] Path expression are defined over edges but reduction to nodes is straight forward [105].

[4] In [9], we define the construction for irreducible graphs.

A path expression $P(s,t)$ in the acyclic case (Equation 5.28) is the union of all paths leading to the predecessors u of node t and the edges leading to t. In the cyclic case (Equation 5.29), if some node t is a loop head, then $P(s,t)$ is a prefix of all paths in the loop body ($P(h,h)$) from the head to its bottoms, back to its head. The expression $P(h,b)$ denotes the kernels of the loop and every exit path is represented by the expression $P(h,t)$ for a head h and some exit node t.

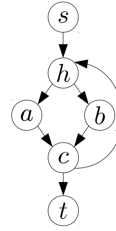


Figure 5.9: Example graph for flow bounds

Example *Consider the graph illustrated in Figure 5.9. The minimal path expression for this graph is* $P = (s,h)((h,a)(a,c) \cup (h,b)(b,c))^*(c,t)$. *Note that Equation 5.28 yields a much larger but equivalent expression. In particular, common prefixes are not factored out and the final loop iteration is represented explicitly although, in this graph, it equals the kernel expression.*

The path language $L(P)$ is unbounded for cyclic graphs and flow constraints must be introduced to obtain feasible solutions. Let the indicator function 1_e be defined as $1_e(u) = 1$ if $(u, _) \in E$, then the *multiplicity* (or frequency) m_π of a node $u \in V$ on a path π is defined as:

$$m_\pi(u) = \sum_{(u,_)\in\pi} 1_{(u,_)}(u) \tag{5.30}$$

Given a set of flow constraints $\mathcal{C}$, the subset of structurally possible paths $L(P,\mathcal{C}) \subseteq L(P)$ that satisfy the constraints then is denoted by:

$$L(P,\mathcal{C}) = \{\pi \in L(P) \mid \forall C \in \mathcal{C} : C(m_\pi)\} \tag{5.31}$$

A constraint set $\mathcal{C}'$ is an approximation if $L(P,\mathcal{C}) \subseteq L(P,\mathcal{C}')$. A typical approximation of flow bounds constraining frequencies of individual nodes are loop bounds, which only constrain frequencies of loop heads, consequently lifting the specification of constraints to entire loops.

Example *Reconsider Figure 5.9. Given constraints* $\mathcal{C} = \{0 \leq m_\pi(a) \leq 2, 1 \leq m_\pi(b) \leq 5\}$. *A sound approximation is* $\mathcal{C}_L = \{\min(0,1) \leq m_\pi(h) \leq \max(2,5)\}$ *such that* $L(P, mathcalC) \subseteq L(P,\mathcal{C}_L)$.

Node infeasibility for a node u is obviously expressed as constraint $\{m_\Pi(u) = 0\}$. Path infeasibility can be expressed as mutual exclusion of nodes.

Example *For Figure 5.9, let* $\mathcal{C}_X = \{\neg(m_\pi(a) > 0 \wedge m_\pi(b) > 0)\}$*, then it holds that* $L(P, \mathcal{C} \cup \mathcal{C}_X) \subseteq L(P, \mathcal{C}) \subseteq L(P, \mathcal{C}_L)$*.*

Depending on the constraint model, different degrees of tightness can be achieved.

In [137], path expressions are used to define a formal framework for parametric WCET formulae. Besides flow bounds and exclusion constraints, in [141] a model with conditional constraints is proposed that enables the modeling of flows depending on loop iteration numbers, which allows the partitioning of loop iterations. Value constraints are proposed in [138]. This further increases the level of accuracy for value-dependent (dynamic) control flow. In [142], a flow bound model is extended by predicate logic to express the effect software configurations on path feasibility.

5.2.3 On Path Analyses

The majority of approaches to path analysis in the field of timing analysis is concerned with the computation of WCET: the length of a longest path, given a control flow graph, time bounds of program points as node weights and a flow constraint model. Despite their common theoretical basis, existing approaches are quite heterogeneous in nature. We first show how path lengths can be computed from path expressions, then we show how existing proposals on path analysis relate to this representation. Further, we provide a brief overview of the most prominent approaches in general.

From Path Expressions to Path Lengths

There exists a simple homomorphism between path expressions and the problem of computing length bounds of paths. Recall the underlying algebraic structure of path expressions. If we ignore mutual exclusion, then *bounded path expressions* are defined over the algebra $(E, \cup, \emptyset, \cdot, \epsilon, ^{[l,h]})$ such that for a given set of constraints, l and h denote lower and upper flow bounds. The expression $P^{[l,h]}$ denotes the finite expansion to $\cup_{l..h} P$ with $P^* = P^{[0,\infty]}$. Let $\omega(u, v)$ denote the cost of an edge[5]. Then function W denotes the costs of the longest path recursively by:

$$W(P) = \begin{cases} -\infty & \text{if } P = \emptyset \\ 0 & \text{if } P = \epsilon \\ \omega(u,v) & \text{if } P = e = (u,v) \\ \max(W(P_1), W(P_2)) & \text{if } P = P_1 \cup P_2 \\ W(P_1) + W(P_2) & \text{if } P = P_1 \cdot P_2 \\ \sum_{l..h} W(P) & \text{if } P = P^{[l,h]} \end{cases} \tag{5.32}$$

In other words, the underlying algebraic structure of bounded path expressions can simply be replaced by $(\mathbb{N}_0 \cup \{-\infty\}, \max, -\infty, +, 0)$ to obtain maximal path lengths

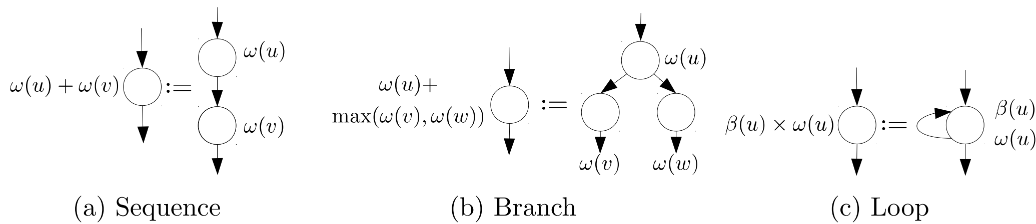
[5]Reduction to nodes is achieved by attributing costs to either the source or target node of the edge.

($W(P) = -\infty$ denotes infeasibility). Symmetrically, lengths of shortest paths can be computed by the algebra $(\mathbb{N}_0 \cup \{-\infty\}, \min, \infty, +, 0)$.

Path expressions form a general formal basis for path problems encountered in timing analysis. In particular, they define the connection between symbolic representation and evaluation.

Approaches to Path Analysis

Path expressions are based on a (graph) grammar that supports sequences, branches and cycles. The cost model defined in Equation 5.32 specifies semantic rules that can be applied during reduction already; hence, costs can be computed without an explicit representation of program structure by means of path expressions in the first place.



(a) Sequence (b) Branch (c) Loop

Figure 5.10: Attributed grammar for longest paths

Example *Consider Figure 5.10 which illustrates schematically a grammar along with semantic rules for cost computation of longest paths. Let ω denote time bounds for the execution of elements and let β denote loop bounds. Then the computed cost for a fully reduced graph yields a global time bound.*

A path expression is just an AST of a corresponding grammar in canonical form. Approaches to path analysis that compute costs from AST are commonly referred to as (*syntax*) *tree-based* [4] approaches. Usually, their grammar directly reflects more complex high-level constructs, such as those listed in Figure 5.5 and Figure 5.6.

Computing costs by reduction has some drawbacks. Reduced regions are small and potentially yield multiple exits. Semantics to derive time bounds is therefore significantly locally constrained and potentially less precise. Moreover, upon reduction, an approximation has to be computed which is globally sound. Hence each reduction further decreases precision. For example in Figure 5.10b, total cost of the branch region will always be the longest path regardless the cost of the two individual paths. Another significant problem of both theoretic and practical nature is that parsing and reduction is not easily unified with concepts of general program analysis as reasoning about paths is limited to regions denoted by grammar “non-terminals”. Safe cost estimates are therefore typically overly pessimistic. Initially, the authors of [143] proposed this approach for high-level timing analysis (cf. Section 5.2.1). Different variants in different scenarios of timing analysis sharing the same principle have been further proposed in [144–148]. Generation of symbolic expressions is explicitly addressed in [148, 149].

Path-based [4] analysis generally refers to approaches that perform partial cost computation during reduction. Intuitively, it is exploited that in reducible graphs path lengths are cheaply computed on the DAGs that form the respective loop bodies (cf. Algorithm B.15). Hence, explicit reduction is limited to loops only. All of these approaches share a common outline, which is sketched as follows:

1) For each loop L with head h in postorder of loop nesting tree:
   1.1) Compute longest path π in loop body
   1.2) Reduce graph $G^{(i+1)} \leftarrow G^{(i)} \setminus_h (G^{(i)}|L)$
   1.3) Reassign weight $\omega(h) \leftarrow \beta(h) \times |\pi|$
2) Compute longest path on condensation graph $G^{(k)}$

As long as there are loops, compute the weight of innermost loops first. Replace the loops by representatives, multiply a loop bound with a given weight and repeat. The final graph will be a DAG for which weight is computed once again.

The advantage over tree-based approaches is that semantics can be computed on usually much larger regions — the loop bodies — which potentially results in tighter time bounds. Otherwise, it shares the general limitations of reduction approaches: Upon reduction an approximation has to be computed to obtain a cost metric which is sound for all region exits. For example, in Figure 5.6d, the final iteration of this loop must be considered a kernel to yield a safe bound on path length for all exits. Second, program semantics are not easily taken into account since upon reduction of some inner loop, context in terms of program state "up to" the loop region is unavailable. Also reduction makes it difficult to discriminate loops with shared heads without explicit prior graph transformation. Proposed path-based analyses vary in their objectives and their approaches towards the general limitations. A simple and formally clean approach to path-based analysis is proposed in [150] for the recursive generation of path expressions, similar to our notion of path expressions. Other, comparably complex approaches [128, 151, 152] focus on interleaving path analysis with micro-architectural analysis with different approaches to limit imprecision due to reduction. In [137], the problem is approached by means of path expressions as defined above. It shall be noted that that some approaches [153] provide countermeasures to mitigate imprecision upon reduction.

A body of approaches is based on *integer linear programming* (ILP) (cf. Appendix C). We have already formalized path problems in Section 5.1.1, all of which have linear constraint models and can be directly solved in ILP. Most approaches are based on Equation 5.9 to solve MAXLEN, but provide diverse extensions for greater semantic richness or to address its specific shortcomings. Collectively, these approaches are commonly referred to as being based on the *implicit path enumeration technique* (IPET) [154]. Solutions to flow problems as discussed in Section 5.1.1 are always integer [99] due to the totally unimodularity property (cf. Appendix C). Hence, they are efficiently solvable as non-integer linear programs, although in general ILP is NP-complete. The complexity of variations of MAXLEN is potentially worse. In particular, adding additional constraints to flow problems, such as mutual infeasibility, make them NP complete [137, 155, 156].

A general framework for mutual exclusion constraints for IPET is discussed in [157]. Various flow bound extension are proposed in [153]. IPET gains much of its popularity due to the simplicity of the base model and the fact that global constraints do not require extra effort. On the other hand, the consideration of additional semantics is difficult due to the limited expressiveness of the linear equation model. Numerous proposals [45, 136, 142, 158–161] address the problem of adding program or architecture semantics for improved time bounds. Besides IPET, in [82, 162] graph reduction is modeled in ILP. Experiments with parametric ILP [163] have been conducted in [164, 165].

Apart from these isolated approaches to path analysis, holistic approaches based on model checking combining various stages of timing analysis have been proposed [166–168].

5.3 A General Path Analysis

In this section we propose a general path analysis which integrates loop structure detection and path analysis itself in a single self-sufficient framework. We propose new and efficient techniques for the various problems that are encountered. We discuss loop detection in the first half. In the second half, we then discuss a general path analysis framework with the initial objective to solve the problem of computing whole-task WCET bounds, for which we then propose several optimizations and from which we derive solutions for specific subproblems. From this base framework, we also derive various solutions for problems frequently encountered in timing and scheduling analysis such as, among others, maximum blocking time or worst-case execution frequencies.

In the following Section 5.3.1 we motivate our approach by discussing existing alternatives and their specific limitations. In Section 5.3.2 we discuss specific problems related to loop detection during control flow reconstruction and we propose new approaches to this problem. Specifically, we propose a loop representation and an efficient general algorithm for loop detection. We then propose two variants to support graph irreducibility. Based on these proposals, we then discuss our proposal for a general framework for path analysis itself. We initially propose the framework according to the specific use-case of per-task WCET in Section 5.3.3 along with numerous optimizations, we show how it is applied to compute WCET bounds from and to individual program points, and we propose a highly efficient reference algorithm. Based on this basic framework, we discuss several variants that solve typical problems in timing analysis. We propose a framework for BCET estimates in Section 5.3.4, for "latest execution times" — a much better metric than WCET for preemptive tasks — in Section 5.3.5, for maximum blocking times in Section 5.3.6 and for worst-case execution frequencies in Section 5.3.7. We conclude the discussion in Section 5.5.

5.3.1 Motivation

Despite the body of existing approaches to path analysis, each comes with its own set of limitations. Partly due to theoretical or technical constraints, partly due to a limited

scope which results in specializations leaving little room for improvement. Another important issue is complexity, both in terms of computation and implementation, which quite practically limits adoption and leads to heterogeneity.

Although the base model of IPET is simple, its applicability to problems beyond vanilla MAXLEN is limited. Figure 3.10 on page 36 outlines the tool chain of the aiT WCET analyzer, whose purpose is the estimation of WCET of a single, non-concurrent, uninterrupted task execution. Separation of phases — such as cache, pipeline or path analysis — is a deliberate engineering decision [47], ensuring modularity and separation of concerns. For complex pipeline architectures, the strict separation of cache and micro-architectural analysis leads to intolerable imprecision, which motivates interaction of both stages in aiT [59]. IPET prevents an even higher degree of integration, although other stages would profit from contextual information provided by path analysis. Heart of the problem is the inherent incompatibility of the linear equation model and traditional program analyses, forcing the encoding of analysis semantics as ILP, which is hard and leaves little room for improvements — as the numerous approaches to higher levels of integration, as discussed above, suggest. Consequently, encoding program state subject to analysis in linear equations comes along with an inherent loss of information due to limited expressiveness of the language. Worse yet, all program state is ultimately narrowed down to a single scalar objective value — the WCET — which serves as input to schedulability tests. The interface of single task analysis and task set analysis is therefore necessarily primitive. Symbolic problem representations [163] for parametric WCET analysis have proven impractical due to complexity constraints. IPET suggests itself as a terminal stage of a traditional single-task WCET analysis but is otherwise limited in scope.

A key limitation of non-ILP approaches is the heterogeneity of approaches and a lack of simple, general, primitive building blocks that enable practical adoption, extension and formalization. One of the most advanced non-ILP path analyses [153] features extensive constraint capabilities for single task WCET computation, but it is at the same time highly complex such that simple adoption (re-implementation) or extension to other problems is practically infeasible. On the other hand, simple approaches, such as [149, 150], are easy to adopt but limited by implicit assumptions about their scope of application or are not efficient, and are therefore impractical in more general settings. For symbolic program representations in particular, tighter integration of analysis stages and provision of complex program state to subsequent analyses beyond single-task WCET, explicit path models are the only feasible option.

A common limitation is the assumption of reducible graph structure. While it is true that most CFGs are reducible, it is not necessarily the case. And while it is true that there exist approaches to transform CFGs into reducible graphs, the problem of co-transforming flow facts to adopt to the new structure is inherent. Recovery of control flow from a binary representation is purposely performed to match actual semantics as close as possible. Further transformation runs counter to this intent. Another, purely practical

consideration is the problem to adapt to existing tool chains: no assumption about the program representation should be made. In particular, intermediate formats used in analysis frameworks are not necessarily designed for further structural transformation. Hence, immutability of input should be assumed. Root of the liberal assumption of mutability is the reuse of traditional algorithms from the field of compilation, where mutation is necessary for optimization, while maintaining program semantics, ignoring timing semantics.

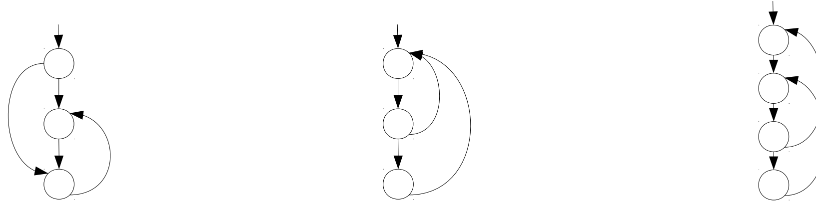
From a purely practical perspective, the state as of this writing is a focus on IPET in most if not all WCET analysis frameworks. Flow fact models tend to take on the form of linear equations for easy adaption to IPET, which limits universality. Due to the increased interest in symbolic representations for parametric analyses [137, 149, 150, 165], this is a critical development and can only be countered by developing simple, general alternatives to IPET.

Our approach to path analysis is a framework approach from scratch with specific design goals in mind. Primary objectives are generality and simplicity. Generality, in the sense that we do not provide a solution for just WCET computations but the provision of a framework for a larger set of problems related to timing analysis. Simplicity, in the sense that a framework and its constituents enable a formal specification and analysis, and directly suggest a specific implementation such that computational complexities are already exposed in its specification. Further, it should be consistent with the concepts of traditional program analysis. The predominant scenario of path analysis is single-task WCET computation from source to sink. To maximize flexibility, it should be possible to carry out analyses on arbitrary subgraphs and without specific assumptions about graph structure. All input is assumed to be immutable. The framework should enable to carry out computations directly on the input without unnecessary redundancy. Hence, transformation including reduction is to be avoided. A careful design of augmenting data structures should enable concurrency of computations with low contention. Hence, mutable data should remain locally constrained, while all globally accessed data structures should remain read-only.

In the remainder of this chapter, we propose a general framework for path analysis that meets these requirements. Further we propose solutions to the single-task WCET path analysis problem as well to a number of variants for which no solutions have been proposed yet at all.

5.3.2 Graph Structure and Loops

In this section we are concerned with the recovery of loop structure to guide path analysis itself. Explicit separation of loop structure from a CFG allows for a much larger degree of flexibility than it is the case for approaches that perform specific graph transformations, such as to establish reducibility: The intent of such transformation is to ensure a deterministic partitioning of edge types for DFS for the purpose of loop detection. It is important to recognize that this does not solve the problem of ambiguous



(a) Ambiguous loop head (b) Ambiguous number of loops (c) Ambiguous loop nesting

Figure 5.11: Sources of ambiguity for loop detection via DFS

graph structure *at all*, as it merely shifts non-determinism from DFS to preceding graph transformations. It is also important to recognize that irreducibility is only one of several sources of ambiguity in the context of path analysis. Figure 5.11 illustrates three possible source of ambiguities that arise for loop detection by DFS. Figure 5.11a depicts irreducibility as discussed earlier. Figure 5.11b illustrates a scenario in which it is not clear from the structure alone, whether a single or two nested loops are modeled. In Definition 5.16, we quite arbitrarily defined this to denote a single loop, which is not necessarily the case in practice. In Figure 5.11c even the nesting relation might be ambiguous: depending on program semantics either loop could be nested if, for example, potential loop counters are reset upon entry into one or the other cycle. All these cases can be resolved by transformations that preserve program semantics but at the same time obfuscate the mapping of flow facts to low-level semantics by not being structure preserving, as discussed in Section 5.2.1. A common weakness of existing approaches is the dependence on such transformations, which result in much harder problems to solve, unless crippling assumptions about program structure are imposed or inaccuracy is accepted. *The key observation is this*: The algorithms used are based on standard compiler techniques where significant structural transformation is not only acceptable but a means to optimization. We claim that for the purpose of timing analysis this is not necessarily the best option.

In the following we propose a set of algorithms to support loop detection without transformation. We first address related work in Section 5.3.2.1. Then we formalize scopes as an alternative, more general notion of loops suitable for irreducible graphs in Section 5.3.2.2. In Section 5.3.2.3 we then propose an efficient, precise and general algorithm for loop detection. We then propose two methods to handle structural ambiguity of irreducible CFGs for this algorithm: In Section 5.3.2.4 we propose an algorithm for the enumeration of context-sensitive loops and in Section 5.3.2.5 a method for disambiguation based on additional annotation is proposed. The proposed algorithms are preprocessing steps for the following discussion of path analysis itself.

5.3.2.1 Related Work

Reduction by T1-T2 [110] as discussed above is one among a comparably small set of approaches to loop detection and is also relatively inefficient due to the small step size of reductions. A more efficient reduction — also for the sake of proving reducibility —

by specifying larger regions (intervals) is proposed in [24]. In [117], an algorithm for the construction of loop nesting forests for reducible graphs is proposed. Subsequently, different approaches have been proposed to detect loops and to handle irreducible CFGs [120, 124, 125], all of which are based on graph reduction of some form which inherently introduces imprecision or complexity in case of irreducibility. The authors of [111] show that all of these algorithms yield quadratic worst-case complexity and propose variants to these algorithms to achieve almost linear time complexity. Summaries and quantitative comparisons of these approaches are published in [111, 169]. Curiously, there is no consensus on the definition of loop nesting relations among those approaches: For irreducible graphs, they all yield different structural descriptions [111, 121]. To the best of our knowledge, the proposal in [121] is the only one for loop identification without explicit graph reduction.

5.3.2.2 Scopes

$\mathring{G} = (\mathring{V}, \mathring{E})$	Scope tree
$\mathring{s} \in \mathring{V}$	Scope
$\mathring{\gamma} \colon V \mapsto \mathring{V}$	Scope label
$\text{entry} \colon \mathring{V} \mapsto \wp(V)$	Entry nodes
$\text{exit} \colon \mathring{V} \mapsto \wp(V)$	Exit nodes
$\text{top} \colon \mathring{V} \mapsto \wp(V)$	Top nodes
$\text{bottom} \colon \mathring{V} \mapsto \wp(V)$	Bottom nodes

Table 5.1: Scope-related definitions

We first provide additional definitions that will be used throughout the remainder of this chapter. The traditional definition of (natural) loops (Definition 5.16 on page 88) is based on dominance relation and is therefore too strict in the context of irreducibility. For general graphs, we provide a generalized definition which relies on node reachability only. Let $G = (V, E, s, t)$ denote a CFG, let B denote its back edges and let $\vec{G} = (V, E \setminus B)$ such that $\vec{\rightsquigarrow}$ denotes reachability in $\vec{G}$.

Definition 5.25 (Loop) *The (non-empty) set $B_L \subseteq B$ such that $(b_i, h) \in B_L$, induces a* loop*, which is a maximal set of nodes $L \subseteq V$ such that for the DAG $\vec{G} = (V, E \setminus B)$, it holds that $\forall u \in L \colon h \vec{\rightsquigarrow} u \vec{\rightsquigarrow} b_i$.*

Nesting relations of such general loops are not a function of mere graph structure but in general must be established through additional annotations. A data structure similar to a loop nesting forest (Definition 5.20) specifies such (nesting) relations. Unlike with natural loops, structural containment is not a requirement but it may serve as a reasonable heuristic in many cases.

Definition 5.26 (Scope) *A* scope[6] *$\mathring{s} \in \mathring{S}$ is a symbolic representation of a loop. Scope membership is denoted by the labeling function $\mathring{\gamma} \colon V \mapsto \mathring{S}$.*

[6]This notion of scope is unrelated to the definitions given in [148] or [153].

In the following we use the terms scope and loop synonymously unless stated otherwise.

Definition 5.27 (Scope Tree) *A* scope tree $\mathring{G} = (\mathring{V}, \mathring{E})$ *with* $\mathring{E} = \mathring{V} \times \mathring{V}$ *is a generalization of a loop nesting forest (Definition 5.20) such that* $\mathring{V} \subseteq \mathring{S}$ *denotes its nodes and* $(\mathring{s}, \mathring{t}) \in \mathring{E}$ *denotes the nesting of scope* $\mathring{s}$ *within scope* $\mathring{t}$*. The label* $\mathring{\gamma}$ *denotes scope membership such that* $\mathring{\gamma}(u)$ *maps to the innermost scope a node* $u \in V$ *belongs to. For clarity, let* $\operatorname{par}(\mathring{s}) = \operatorname{succ}(\mathring{s})$ *denote a parent scope and let* $\operatorname{dsc}(\mathring{s}) = \operatorname{pred}(\mathring{s})$ *denotes descendants.*

Our objective will be to compute a suitable scope tree $\mathring{G}$ and a suitable mapping $\mathring{\gamma}$ such that a subsequent path analysis based entirely on this structural information is safe and precise.

Definition 5.19 for entries and exits continues to hold, but we strengthen and extend the definition for scopes.

Definition 5.28 (Scope Entry, Scope Exit) *Let* $(u, v) \in G$ *such that* $\mathring{\gamma}(u) \neq \mathring{\gamma}(v)$*. Node* v *is a* scope entry *if and only if* $\mathring{\gamma}(v) \rightsquigarrow \mathring{\gamma}(u)$*. Node* v *is a* far entry *(multi-level entry) if it is an entry and* $(\mathring{\gamma}(v), \mathring{\gamma}(u)) \notin \mathring{E}$*. Symmetrically, node* u *is a* scope exit *if and only if* $\mathring{\gamma}(u) \rightsquigarrow \mathring{\gamma}(v)$*. Node* u *is a* far exit *if it is an exit and* $(\mathring{\gamma}(u), \mathring{\gamma}(v)) \notin \mathring{E}$*. We denote entries by* $\operatorname{entry}\colon \mathring{V} \mapsto \wp(V)$ *and exits by* $\operatorname{exit}\colon \mathring{V} \mapsto \wp(V)$*, respectively.*

Definition 5.29 (Scope Top, Scope Bottom) *Node* $u \in V$ *is a* scope top *if and only if* $\forall v \in V\colon \mathring{\gamma}(u) = \mathring{\gamma}(v) \Rightarrow u \overset{\rightarrow}{\rightsquigarrow} v$*. Node* u *is a* scope bottom *if and only if* $\nexists v \in V\colon \mathring{\gamma}(u) = \mathring{\gamma}(v) \Rightarrow u \overset{\rightarrow}{\rightsquigarrow} v$*. We denote the singleton set of tops by* $\operatorname{top}\colon \mathring{V} \mapsto \wp(V)$ *and the set of bottoms by* $\operatorname{bottom}\colon \mathring{V} \mapsto \wp(V)$*, respectively.*

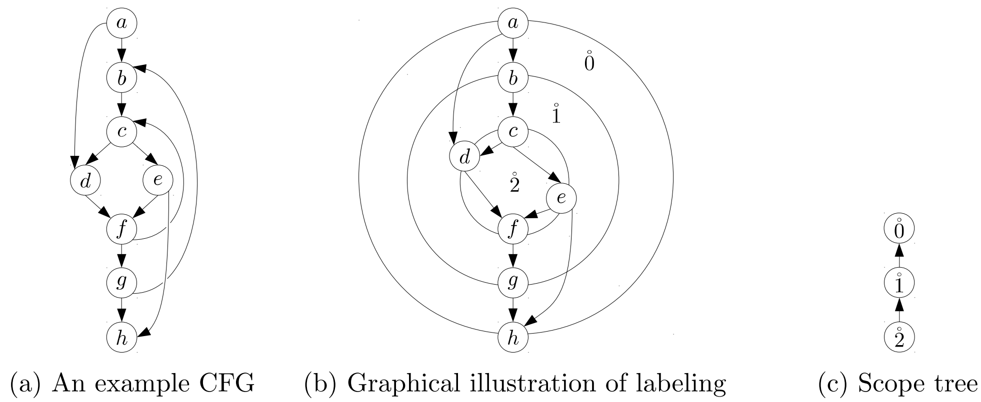
By definition, a scope has only a single top but potentially multiple bottoms, entries and exits. Table 5.1 summarizes scope properties so far. As previously, we define some extra terminology.

Definition 5.30 (Scope Iteration) *An* iteration *is a (not necessarily acyclic) path* $\pi = (s, \ldots, t)$ *such that* $s \in \operatorname{entry}(\mathring{s})$ *and* $\forall u \in \pi \setminus \{s\}\colon \mathring{\gamma}(u) \in \{\mathring{s}\} \cup \operatorname{dsc}(\mathring{s})$*.*

Note that in general, we do not impose any restriction on the terminal node. For now, can simply assume node t to denote a scope bottom or exit of the same scope as the entry. Later, we will relax this notion. Similar to loop iterations, we distinguish iterations types.

Definition 5.31 (Iteration Type) *An iteration* $\pi = (s, \ldots t)$ *is an* entry *if* $s \in \operatorname{entry}(\mathring{s}) \wedge t \in \operatorname{bottom}(\mathring{s})$*, it is an* exit *if* $s \in \operatorname{top}(\mathring{s}) \wedge t \in \operatorname{exit}(\mathring{s})$*, and it is a* kernel *if* $s \in \operatorname{top}(\mathring{s}) \wedge t \in \operatorname{bottom}(\mathring{s})$*.*

Example *Consider Figure 5.12, which illustrates a CFG and its relation to scopes. The CFG in Figure 5.12a features two loops identified by back edges* (g, b) *and* (f, c)*. The loop heads and bottoms are also entries and exits respectively. Edge* (a, d) *is a far entry and edge* (d, h) *is a far exit. For convenience, we use the syntax illustrated in Figure 5.12b,*



(a) An example CFG (b) Graphical illustration of labeling (c) Scope tree

Figure 5.12: An example of scope tree representation

to represent labeling according to Table 5.1. Here, circles denote scope membership and edges denote reachability of entries and exits. Interior nodes that are neither top, bottom, entry or exit are not shown in general. Figure 5.12c depicts the corresponding scope tree. Note that all scope members are reachable from the top node, all member nodes reach at least one bottom but not all entries necessarily reach all exits.

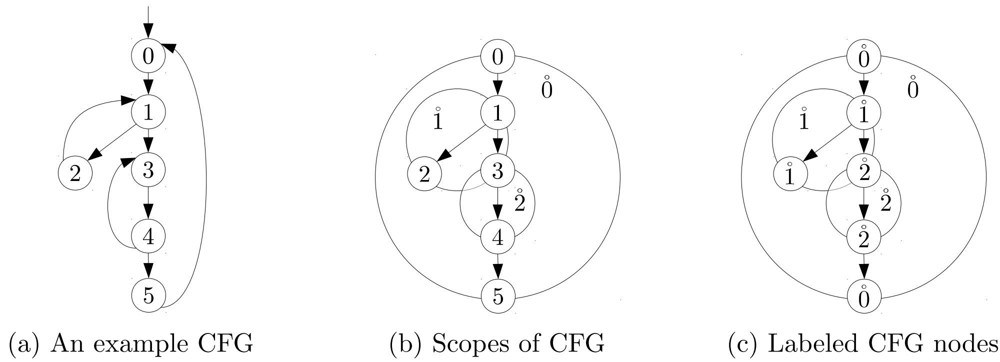
5.3.2.3 A General Algorithm for Precise Loop Detection

In this section we propose a new algorithm for loop detection that does not make any assumptions about graph structure, does not require complex auxiliary data structures, is not dependent on the availability of additional analysis passes and operates on immutable inputs. It is formally simple and easy to implement. The proposal in [121] and our approach share the same intuition, which has been discovered independently, but is effectively a generalization of the former approach with a simpler formal definition, stricter specification and focused on our problem set.

The approach is based on a single DFS pass in which structural information is collected and applied to construct a scope tree for a given CFG. Reducibility is not a requirement, but as Theorem 5.15 on page 88 suggests, the discovered loop nestings are potentially non-deterministic as this depends on the traversal sequence of DFS. Our strategy to approach irreducibility is to provide an algorithm which is oblivious of such considerations and hence does not preemptively introduce imprecision as in other approaches. To achieve safety in addition to precision, we later propose methods to steer DFS itself. This maximizes flexibility and precision while keeping implementations simple.

Example *Figure 5.13 illustrates a motivating example. In Figure 5.13a a CFG labeled by preorder values is depicted. Structurally, the CFG decomposes into three scopes as shown in Figure 5.13b with their respective nesting relations. Figure 5.13c shows the respective scopes labeling of CFG nodes.*

In the following we give a formal specification and propose an efficient algorithm. First, we discuss technical prerequisites. We then formally construct an analysis for



(a) An example CFG (b) Scopes of CFG (c) Labeled CFG nodes

Figure 5.13: Properties of scope tree construction

which we propose an equivalent efficient algorithm for loop detection, followed by a brief discussion on complexity bounds and a thorough evaluation.

Prerequisites

During scope tree construction we have to take ambiguities as illustrated in Figure 5.11 into account. We chose to either allow for annotations to support construction or to revert to heuristics otherwise. In case of irreducibility (Figure 5.11a), back edges are ambiguous and so is the identification of scopes. In the following we assume the construction of scope trees for any traversal order. So in case of irreducibility, the result is non-deterministic initially. Later we provide a solution to this issue. Shared heads (Figure 5.11b) prevent the discrimination of individual loops by, for example, their preorder label ${}^\bullet t$ (cf. Section 5.1.2) alone. We introduce annotations to allow for explicit steering of loop detection to address ambiguous loop counts, such as illustrated in Figure 5.11b and ambiguous nesting relations, such as illustrated in Figure 5.11c. These annotations are not inherently safe, but neither is scope tree construction by mere heuristic. Our overall intent is to provide these annotations alongside flow constraints to allow for the construction of feasible loop models that match flow annotations.

Informally, our strategy will be this: Scope trees are constructed by a single DFS pass over a CFG. Decisions are carried out only once nodes finish. Initially, we assume that every single (scope) bottom denotes a separate scope. Hence, once a bottom node is finished, we assume a new scope. We cannot know the nesting relation, its corresponding CFG nodes or whether the bottom denotes only one of a set of bottoms constituting a loop comprised of multiple back edges at this point in time and therefore refer to it as a *pending* scope; for which information is still incomplete. A scope which is not pending is either *complete* or yet unknown. A pending scope is complete once all information has been gathered, which is the case when its top is finished. Multiple completing scopes potentially merge or form nestings, depending on the given annotations.

We now discuss how scope nesting relation is established and maintained within the algorithm. A scope is uniquely identified by the pair of top and bottom nodes. The traditional default heuristic in (natural) loop detection is to identify loops only by their

head node (cf. Definition 5.20). We model this behavior by default but allow for explicit modification. Let $\mathring{\sigma}_\top : \mathring{S} \mapsto \mathbb{N}_0$ denote a symbolic label for top nodes and let $\mathring{\sigma}_\bot : \mathring{S} \mapsto \mathbb{N}_0$ denote a symbolic label for bottom nodes. We can define a partial order of scopes in the scope tree by the relation:

$$\mathring{<} : \mathring{S} \times \mathring{S}$$
$$\mathring{s} \mathring{<} \mathring{t} \;\Leftrightarrow\; \mathring{\sigma}_\top(\mathring{s}) < \mathring{\sigma}_\top(\mathring{t}) \vee (\mathring{\sigma}_\top(\mathring{s}) = \mathring{\sigma}_\top(\mathring{t}) \wedge \mathring{\sigma}_\bot(\mathring{s}) < \mathring{\sigma}_\bot(\mathring{t})) \tag{5.33}$$

To model the traditional heuristic, we define:

$$\mathring{\sigma}_\top(\mathring{s}) = {}^\bullet t(\mathrm{top}(\mathring{s})) \tag{5.34}$$
$$\mathring{\sigma}_\bot(\mathring{s}) = 0 \tag{5.35}$$

By default, scopes are never discriminated by their bottoms and all scopes sharing the same top node — identified by their preorder label — are considered equal. We establish a corresponding equivalence relation, defined as:

$$\mathring{\sim} : \mathring{S} \times \mathring{S}$$
$$\mathring{s} \mathring{\sim} \mathring{t} \;:= \neg(\mathring{s} \mathring{<} \mathring{t} \vee \mathring{t} \mathring{<} \mathring{s}) \tag{5.36}$$

The scope tree only consists of representatives of the respective equivalence classes.

Axiom 5.32 *For any scope tree $\mathring{G} = (\mathring{V}, \mathring{E})$, it holds that $\mathring{V} \subseteq \{\mathring{s} \mid [\mathring{s}] \in \mathring{S}/\mathring{\sim}\}$.*

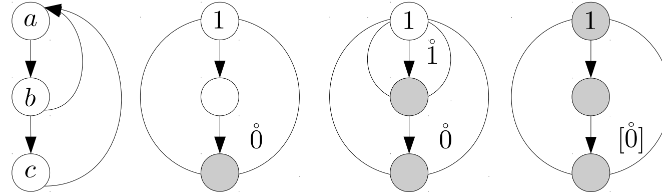


Figure 5.14: Creating scopes without explicit discrimination

Example *Consider Figure 5.14, which illustrates the instantiation of scopes by the default heuristic, while finishing nodes in DFS postorder. Once potential scope bottoms are reached, a new scope is created. Given default annotations, both potential scopes become elements of the same equivalence class $[\mathring{0}]$, eventually. In contrast, Figure 5.15*

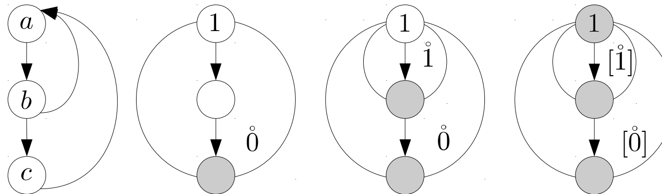


Figure 5.15: Creating scopes given explicit discrimination

illustrates scope creation for $\mathring{\sigma}_\bot(\mathring{0}) \neq \mathring{\sigma}_\bot(\mathring{1})$. Thus, scopes remain explicitly discriminated.

Edges of scope trees denote their respective parent relations.

Definition 5.33 (Parent Scope) *Given scopes $\mathring{s}, \mathring{t} \in \mathring{V}$. Let $t_{\mathring{s}} \in$ top$(\mathring{s})$ and let $B_{\mathring{t}} \subseteq$ bottom$(\mathring{t})$. Then scope $\mathring{t}$ is a* parent *of scope $\mathring{s}$ if and only if $\mathring{t} \mathring{<} \mathring{s}$ and $\exists b_{\mathring{t}} \in B_{\mathring{t}}\colon t_{\mathring{s}} \overset{\rightarrow}{\rightsquigarrow} b_{\mathring{t}}$.*

Intuitively, assuming default heuristics, this corresponds to natural loop nesting relations of Definition 5.20 and our relaxed definition of loops of Definition 5.25 ($u \overset{\rightarrow}{\rightsquigarrow} v \Rightarrow {}^{\bullet}t(u) < {}^{\bullet}t(v)$). For example, in Figure 5.13c, it holds that $\neg(\mathring{1} \mathring{<} \mathring{2})$ but $\mathring{0} \mathring{<} \mathring{2}$.

Definition 5.34 (Immediate Parent Scope) *Scope $\mathring{t}$ is an* immediate *parent of $\mathring{s}$ if and only if it is parent and it holds that $\forall \mathring{p} \in \{\mathring{r} \in \mathring{S} \colon \mathring{r} \mathring{<} \mathring{s}\} \setminus \{\mathring{t}\} \colon \mathring{p} \mathring{<} \mathring{t}$.*

Note that there may be multiple feasible immediate parents $\mathring{t}$, but only one equivalence class of immediate parents $[\mathring{t}]$ (Axiom 5.32).

Lemma 5.35 *Set $[\mathring{t}]$ is an immediate parent of $[\mathring{s}]$ if and only if Definition 5.34 holds for all elements in $[\mathring{s}]$ and $[\mathring{t}]$.*

We have not precisely defined yet when scopes are considered to be complete.

Definition 5.36 *During DFS, once a node $u \in V$ finishes, all pending scopes $\mathring{s} \in \mathring{P}$ with ${}^{\bullet}t(u) \leq {}^{\bullet}t(\text{top}(\mathring{s}))$ are completed* in the order *denoted by $\mathring{<}$.*

With default heuristics, all scopes $\mathring{s}$ with $\mathring{\sigma}_{\top}(\mathring{s}) = {}^{\bullet}t(\text{top}(\mathring{s}))$ complete upon finishing its top node. Consider Figure 5.16, which depicts ambiguous nesting. Using preorder

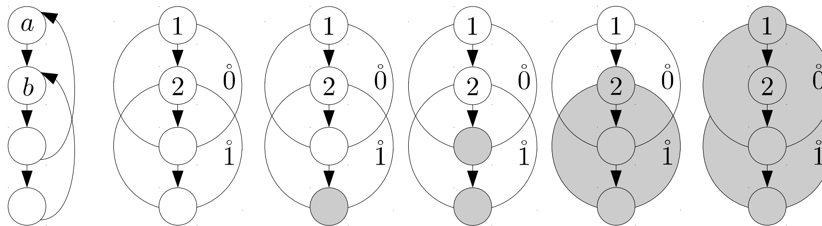


Figure 5.16: Completing scopes in preorder

labels as given by DFS such that $\mathring{\sigma}_{\top} = \{\mathring{0} \rightarrow 1, \mathring{1} \rightarrow 2\}$, a corresponding sequence of completing scopes is that scope $\mathring{1}$ completes before scope $\mathring{0}$. We can modify $\mathring{\sigma}_{\top}$ to

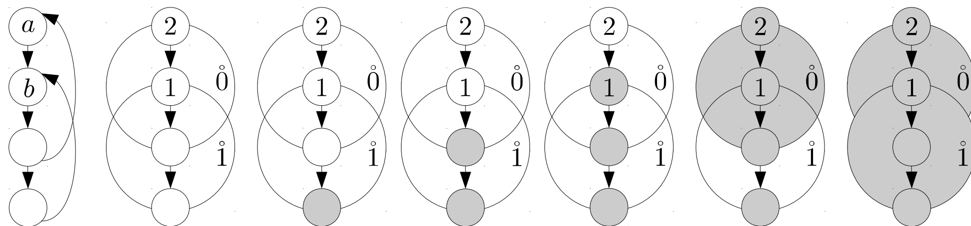


Figure 5.17: Completing scopes in modified order

enforce alternative nestings. Given $\mathring{\sigma}_{\top} = \{\mathring{0} \rightarrow 2, \mathring{1} \rightarrow 1\}$, the corresponding sequence of completion is illustrated in Figure 5.17. Scope $\mathring{1}$, which is now the immediate parent of $\mathring{0}$, is not completed until $\mathring{0}$ is completed. Such postponement is a technical requirement for simple scope tree construction as is reflected by:

Lemma 5.37 *Once a scope completes, all potential parent scopes are still pending.*

Proof. The tops of all potential parent scopes are still on the DFS stack. □

In particular, this holds for immediate parents. Hence, only upon completion, we establish scope relation between a scope and an immediate parent by searching the set of pending scopes.

Theorem 5.38 *During DFS, let the poset* $\mathring{P}$ *denote pending scopes and let* $[\mathring{s}] \in \mathring{P}$ *be a set of finishing scopes. Then* $[\mathring{t}]$ *with* $\mathring{t} = \max_{\mathring{<}} \mathring{P} \setminus [\mathring{s}]$ *is an immediate parent of* $[\mathring{s}]$.

Proof. Follows directly from Lemma 5.35 and Lemma 5.37. □

It remains to define how scope labels $\mathring{\gamma}$ are determined.

Theorem 5.39 *Let poset* $\mathring{P}$ *denote pending scopes. For any finishing node* $u \in V$*, it holds that* $\mathring{\gamma}(u) = \max_{\mathring{<}} \mathring{P}$.

Proof. Follows directly from the definition of loops (Definition 5.25) and the fact that the currently maximal element in the set of pending scopes is not finished (Definition 5.36). □

Scope Tree Construction

We can now formalize the construction of scope trees, which is a simple program analysis based on backward path-based semantics on a CFG. For the specification, we initially assume reducibility. We will later relax this constraint. The scope tree is built bottom up — leaves first — with all pending scope representatives in partial order $\mathring{<}$.

Let $\mathring{S}$ denote the set of scopes and let $[\mathring{S}] = \{\mathring{s} \mid [\mathring{s}] \in \mathring{S}/\mathring{\sim}\}$ denote the set of representatives of equivalence classes in $\mathring{S}$. We define the state domain as:

$$\mathring{\mathbb{D}} = (\mathring{S}, \mathring{S} \times \mathring{S}, \mathring{S}, V \mapsto \mathring{S}) \tag{5.37}$$

which models nodes and edges of a scope tree, a set of pending scopes and scope labels, respectively.

We define two transfer functions. Function gen introduces new pending scopes, handles re-entries into complete scopes and otherwise propagates state. Let $B = \{(u, v), \dots\})$ denote a set of back edges and let the tuple $(\mathring{V}, \mathring{E}, \mathring{P}, \mathring{\gamma})$ denote a scope tree with nodes $\mathring{V}$, edges $\mathring{E}$, a poset of pending scopes $\mathring{P}$ and scope labels $\mathring{\gamma}$. Also, let $s^\star \in \mathring{S}$ denote a new unique scope label. Then gen is defined as:

$$\mathrm{gen}_{\mathring{G}} \colon \wp(B) \times \mathring{\mathbb{D}} \mapsto \mathring{\mathbb{D}}$$

$$\begin{aligned} &\mathrm{gen}_{\mathring{G}}(B, (\mathring{V}, \mathring{E}, \mathring{P}, \mathring{\gamma})) \\ &\quad = \begin{cases} \mathrm{gen}_{\mathring{G}}\left(B \setminus \{(u,v)\}, \left(\mathring{V}, \mathring{E}, P \cup \{\mathring{s}^\star\}, \mathring{\gamma}[u \mapsto \max_{\mathring{<}}([\mathring{P} \cup \{\mathring{s}^\star\}])]\right)\right) & \text{if } B \neq \emptyset \\ (\mathring{V}, \mathring{E}, \mathring{P}, \mathring{\gamma}[u \mapsto \max_{\mathring{<}}[\mathring{P}]]) & \text{otherwise} \end{cases} \end{aligned} \tag{5.38}$$

In the first case ($B \neq \emptyset$), scopes $\mathring{s}^\star$ are created successively for each back edge $(u,v) \in B$. Each new scope $\mathring{s}^\star$ is added to the poset of pending scopes $\mathring{P}$ and scope label $\mathring{\gamma}$ is reset to reflect the innermost enclosing scope according to Lemma 5.39. In the second case, scope labels $\mathring{\gamma}$ are just updated to denote the topmost pending scope for node u.

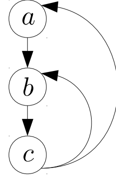


Figure 5.18: Example graph to demonstrate $\mathrm{gen}_{\mathring{G}}$

Example *Consider the CFG illustrated in Figure 5.18. We assume default heuristics. Upon finishing node c,* $\mathrm{gen}_{\mathring{G}}$ *yields:*

$$\begin{aligned}
&\mathrm{gen}_{\mathring{G}}(\{c \to a, c \to b\}, (\emptyset, \emptyset, \emptyset, \emptyset)) \\
&\quad = \mathrm{gen}_{\mathring{G}}(\mathrm{gen}_{\mathring{G}}(\{c \to b\}, (\emptyset, \emptyset, \{\mathring{0}\}, \{c \to \mathring{0}\}))) \\
&\quad = \mathrm{gen}_{\mathring{G}}(\mathrm{gen}_{\mathring{G}}(\mathrm{gen}_{\mathring{G}}(\emptyset, (\emptyset, \emptyset, \{\mathring{0}, \mathring{1}\}, \{c \to \mathring{1}\})))) \\
&\quad = (\emptyset, \emptyset, \{\mathring{0}, \mathring{1}\}, \{c \to \mathring{1}\})
\end{aligned}$$

The scope tree is still empty, two scopes $\mathring{0}, \mathring{1}$ *are pending and the current node c is mapped to the innermost scope* 1.

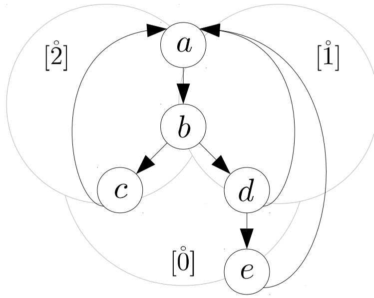
While function gen creates pending scopes, function kill constructs the scope tree by completing scopes. Let $u \in V$ denote the currently finishing node and as before let tuple $(\mathring{V}, \mathring{E}, \mathring{P}, \mathring{\gamma})$ denote a scope tree. Let $\mathring{s}_{\max} := \max_{\mathring{<}}[\mathring{P}]$ denote the topmost pending scope. Then kill is defined as:

$$\mathrm{kill}_{\mathring{G}} \colon V \times \mathring{\mathbb{D}} \mapsto \mathring{\mathbb{D}}$$

$$\begin{aligned}
&\mathrm{kill}_{\mathring{G}}(u, (\mathring{V}, \mathring{E}, \mathring{P}, \mathring{\gamma})) \\
&= \begin{cases}
\mathrm{kill}_{\mathring{G}}\left(u, \begin{pmatrix} \mathring{V} \cup \{\mathring{s}_{\max}\}, \\ \mathring{E} \cup \{(\mathring{s}_{\max}, \max_{\mathring{<}}([\mathring{P}] \setminus \{\mathring{s}_{\max}\}))\}, \\ \mathring{P} \setminus \mathring{s}_{\max}, \\ \mathring{\gamma} \end{pmatrix}\right) & \text{if } {}^\bullet t(u) \leq {}^\bullet t(\mathrm{top}(\mathring{s}_{\max})) \\
(\mathring{V}, \mathring{E}, \mathring{P}, \mathring{\gamma}) & \text{otherwise}
\end{cases}
\end{aligned} \tag{5.39}$$

In the first case, $\mathrm{kill}_{\mathring{G}}$ is invoked recursively for each maximal element in the poset $\mathring{P}$ whose corresponding scope top node has (already) been finished. In each invocation, scopes are completed in the order of $\mathring{P}$ (descendant scopes first according to completion order given by Definition 5.36). A scope is completed by assigning an additional scope tree node to $\mathring{V}$, by defining an edge from the currently completing scope to the (yet pending) immediate parent scope (Definition 5.34) and by removing the respective scopes

(all members of the same partition) from the set of pending scopes $\mathring{P}$. In the second case, the scope tree is returned unmodified.



(a) Example CFG

discovery sequence	(a, b, d, e, c)
preorder	$(1, 2, 3, 4, 5)$
scopes	$\{\mathring{0}, \mathring{1}, \mathring{2}\}$
top	$\{\mathring{0} \to a, \mathring{1} \to a, \mathring{2} \to a\}$
bottom	$\{\mathring{0} \to e, \mathring{1} \to d, \mathring{2} \to c\}$
$\mathring{\sigma}_\top$	$\{\mathring{0} \to 1, \mathring{1} \to 1, \mathring{2} \to 1\}$
$\mathring{\sigma}_\bot$	$\{\mathring{0} \to 1, \mathring{1} \to 2, \mathring{2} \to 1\}$
equivalences	$[\mathring{0}] = \{\mathring{0}, \mathring{2}\}, [\mathring{1}] = \{\mathring{1}\}$

(b) Annotations and properties

Figure 5.19: Example graph to demonstrate $\text{kill}_{\mathring{G}}$

Example *Figure 5.19a illustrates an example CFG of three scopes. Figure 5.19b lists the properties of this graph. Notably,* $\mathring{\sigma}_\top$ *corresponds to preorder and* $\mathring{\sigma}_\bot$ *models equivalences of scopes* $\mathring{0}$ *and* $\mathring{1}$ *such that for the scope relation it holds that* $[\mathring{0}] \mathrel{\mathring{<}} [\mathring{1}]$*. Once node* a *is finished, scope tree construction by* $\text{kill}_{\mathring{G}}$ *yields:*

$$
\begin{aligned}
&\text{kill}_{\mathring{G}}(a, (\emptyset, \emptyset, \{\mathring{0}, \mathring{2}, \mathring{1}\}, \{a \to \mathring{1}, b \to \mathring{1}, c \to \mathring{0}, d \to \mathring{1}, e \to \mathring{0}\})) \\
&= \text{kill}_{\mathring{G}}(a, (\text{kill}_{\mathring{G}}(a, \{\mathring{1}\}, \{\mathring{1} \to \mathring{0}\}, \{\mathring{0}, \mathring{2}\}, \dots))) \\
&= \text{kill}_{\mathring{G}}(a, (\text{kill}_{\mathring{G}}(a, \text{kill}_{\mathring{G}}(a, \{\mathring{1}, \mathring{0}\}, \{\mathring{1} \to \mathring{0}, \mathring{0} \to \bot\}, \emptyset, \dots)))) \\
&= (\{\mathring{1}, \mathring{0}\}, \{\mathring{1} \to \mathring{0}, \mathring{0} \to \bot\}, \emptyset, \dots)
\end{aligned}
$$

Note that scope labels $\mathring{\gamma}$ *correspond to the representative of each equivalence class such that* $c \to \mathring{0}$*.*

Finally, a scope tree for a CFG $G = (V, E, s, t)$ with back edges B, non-back edges $E^F = E \setminus B$ and $\text{top}(\mathring{s}_0) = \{s\}$ is constructed by $\text{stree}(s)$ given:

$$
\begin{aligned}
&\text{stree} \colon V \mapsto \mathring{\mathbb{D}} \\
&\text{stree}(u) = \begin{cases} \text{kill}_{\mathring{G}}\left(u, \text{gen}(B, \bigcup_{(u,v) \in E^F} \text{stree}(v))\right) & \text{if } u \neq t \\ (\emptyset, \emptyset, \{\mathring{s}_0\}, \emptyset) & \text{otherwise} \end{cases}
\end{aligned}
\tag{5.40}
$$

In the first case, if the current node u is not the sink node t of G, obtain a scope tree by construction in postorder. In the second case if $u = t$, an initial scope is marked pending.

Example *Figure 5.20 illustrates scope tree construction for the initial example in Figure 5.13 such that when node* a *is finished, the scope tree* $\mathring{G} = (\{\mathring{0}, \mathring{1}, \mathring{2}\}, \{\mathring{1} \to \mathring{0}, \mathring{2} \to \mathring{0}, \mathring{0} \to \bot\})$ *is fully specified.*

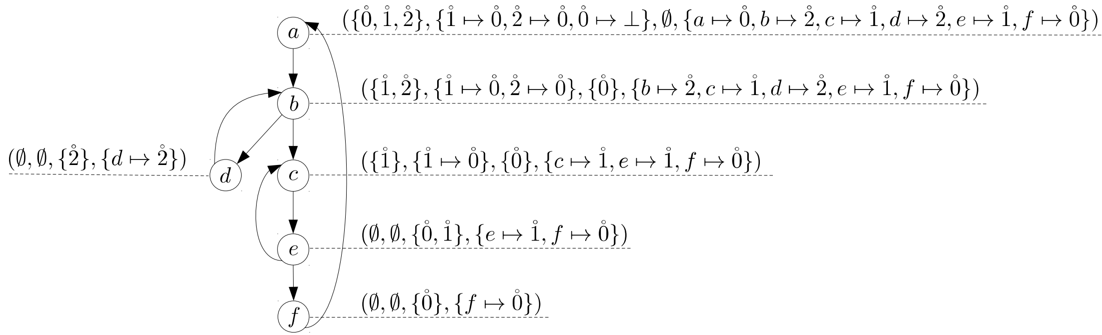


Figure 5.20: Complete example of scope tree construction

An Efficient Algorithm

The suggested construction of scope trees as proposed above served only as a means to specify its principle ideas. We now propose an equivalent, efficient and non-recursive algorithm.

For an efficient representation, we recognize that it is not necessary to propagate information along the DFST, which unnecessarily increases overhead.

Lemma 5.40 *During scope tree construction (Equation 5.40), the subtrees* $(\mathring{V}, \mathring{E})$ *that eventually form the final scope tree, the sets of pending scopes* $\mathring{P}$ *and the scope labels* $\mathring{\gamma}$ *are all disjoint for neighboring nodes in the corresponding DFST.*

Proof. Follows directly from the parenthesis theorem (Theorem 5.11 on page 86). □

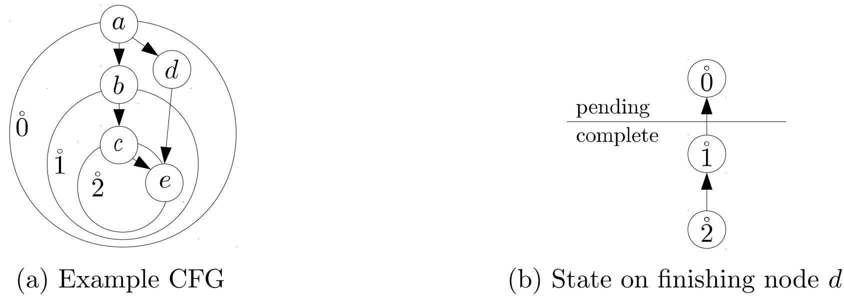
Consequently, it is not necessary to propagate data along the graph: "global" data structures suffice to maintain sets. For our proposed algorithm, arrays are sufficient to encode all data. We assume the existence of a DFS such as proposed in Algorithm B.16. For simplicity, we assume default heuristics in the following code specification but already provide provisions such that extension for a fully parameterized version is obvious. We explicitly allow for irreducibility. Back edge discovery still depends on the traversal sequence order of DFS but will be addressed separately later.

In the proposed Algorithm 5.2, three arrays maintain state. Scopes are identified by integer values. Array S (line 1) encodes pending states as well as the scope tree. A scope is encoded as a tuple consisting of back edge head node *Top*, an ordinal *Ord*, which corresponds to labeling $\mathring{\sigma}_\top$, a "tag" value *Tag*, which corresponds to labeling $\mathring{\sigma}_\bot$, a reference *Class* to maintain equivalence classes and a reference *Next* to other scopes in S, which is used to either model lists of pending scopes or parents in the scope tree which is to be constructed. Array H (line 2) maintains the heads of sorted lists of pending scopes. H consistently references the maximal elements of pending scopes. Array $\mathring{\gamma}$ (line 3) models scope labels.

We define two types of relations to sort lists of pending scopes. Relation $\mathring{R}$ (line 4) denotes membership in equivalence classes by restricting comparison to *Ord* and *Tag* only. Relation $\mathring{R}^*$ is an extension of $\mathring{R}$, which strictly discriminates individual scopes, not just their equivalence classes.

Algorithm 5.2 Generalized Loop Detection

```
 1  S : ℕ₀ ∪ {⊥} ↦ (Top, Ord, Tag, Class, Next)
 2  H : V(G) ↦ ℕ₀
 3  γ̊ : V(G) ↦ V̊(G̊)
 4  let R̊ s t = Ord S(s) < Ord S(t) ∨ (Ord S(s) = Ord S(t) ∧ Tag S(s) < Tag S(t))
 5  let R̊* s t = R̊ s t ∨ (Ord S(s) = Ord S(t) ∧ Tag S(s) = Tag S(t) ∧ s < t)
 6
 7  let init G =                                          ◁ G = (V, E, s, t)
 8      H(t(G)) ← |S|                                     ◁ t(G) sink of CFG
 9      S(|S|) ← (s(G), •t s(G), ⊥, |S|, ⊥)
10
11  let union s t =                 ◁ union of sorted lists (descending order)
12      r ← ⊥
13      while s ≠ ⊥ ∧ t ≠ ⊥ do
14          if R̊* s t then
15              Next S(r) ← s
16              s ← Next S(s)
17          else if R̊* t s then
18              Next S(r) ← t
19              t ← Next S(t)
20          else
21              Next S(r) ← t                              ◁ either s or t
22              break
23          r ← Next S(r)
24      Next S(r) ← if s ≠ ⊥ then s else t
25      return Next S(⊥)
26
27  let finish u =
28      for v ∈ succ u do
29          if (u, v) ∈ B(G) then
30              S(|S|) ← (v, •t v, ⊥, |S|, ⊥)
31              H(u) ← union H(u) S(|S| − 1)
32          else let f s =
33              if Ord S(s) = ⊥ then f Next S(s) else s in
34                  H(u) ← union H(u) f(s)
35      s ← γ̊(u) ← H(u)
36      while •t u ≤ •t Top S(s) do
37          t ← s
38          s ← Next S(s)
39          if s ≠ ⊥ then
40              if R̊ s t then
41                  (Next S(Class S(t)), Ord S(Class S(t))) ← (s, ⊥)
42              else
43                  (Class S(s), Ord S(s)) ← (Class S(t), ⊥)
44          else
45              for s ∈ γ̊ do γ̊(s) ← Class S(s)
46              break
47      H(u) ← s
```



(a) Example CFG (b) State on finishing node d

Figure 5.21: Example to demonstrate handling of re-entry into scopes

Function *init* (line 7) is invoked for initialization. Head of the list of pending scopes is the initial scope identified by $|S| = 0$, which is defined in line 9. We define top as CFG source, *Ord* as ${}^{\bullet}t$, no specific *Tag*, an equivalence class $|S| = 0$, consisting only of this very scope, and no adjacent scope in the list of scopes.

Function *union* (line 11) simply defines the union of two sorted, singly-linked scope lists and is only given for completeness. Its implementation is not relevant in the following. Note that lists are ordered by the stricter relation $\mathring{R}^*$.

Function *finish* (line 27) is invoked for every finishing node during DFS. It consists of two parts, resembling $\text{gen}_{\mathring{G}}$ and $\text{kill}_{\mathring{G}}$ as defined earlier. For all succeeding CFG nodes (line 28), either create a new scope (line 30) and insert it into the list of pending scopes (line 31), or join lists of pending scopes (lines 32-34).

At this point, irreducibility is handled. To distinguish pending and completed scopes in S, we reset *Ord* to $\perp$ upon completion. When propagating the head of a pending scope list from a succeeding node, but the presumed head already belongs to a completed scope, we traverse the scope tree until the first pending scope is encountered.

Example *Figure 5.21 illustrates an example. In Figure 5.21a, once node d completes, $H(e)$ references a complete scope. To infer a correct scope label for node d, the scope tree as depicted in Figure 5.21b is traversed upwards until a pending scope is reached.*

This is encoded in lines 33-34 by means of function f.

In line 35 scope label $\mathring{\gamma}$ is set as the maximum element of pending scopes and an index s is set. From line 36 on, scopes are being completed. While there exist pending scopes whose top nodes have already been finished, or are to be finished, they are grouped according to *Class* to form equivalence classes. In lines 37, 38 the current top element t of the list of pending scopes is saved and the next element s is obtained. If $s \neq \perp$, the topmost scope has not been reached yet. By means of the weaker scope relation $\mathring{R}$ (line 40), if s is an element of another equivalence class, the current equivalence class, denoted by Class $(S(t))$, is set to reference the next one in the order of $\mathring{R}$ (Next S(Class $(S(t))) \leftarrow s$) and *Ord* is "cleared" to denote the completion of this class (Ord S(Class $(S(t))) \leftarrow \perp$), in line 41. Otherwise, if the next element in the list of pending scopes does belong to the same equivalence class (line 42), then the scope s inherits the class identifier from scope t (Class $S(s) \leftarrow$ Class $S(t)$) and and *Ord* is "cleared" to denote the completion of this scope (Ord $S(s) \leftarrow \perp$), in line 43.

Once we reach the topmost scope (line 44), then all references in the scope tree are reset to only refer to equivalence class representatives. In line 47 the list of pending scopes is adjusted to the maximal non-completed scope.

Upon finishing source node s, array S contains the scope tree, where $\mathring{V} = \{\mathring{s} \in S \colon \text{Class}(s) = s\}$, which excludes all scopes not being representatives of equivalence classes.

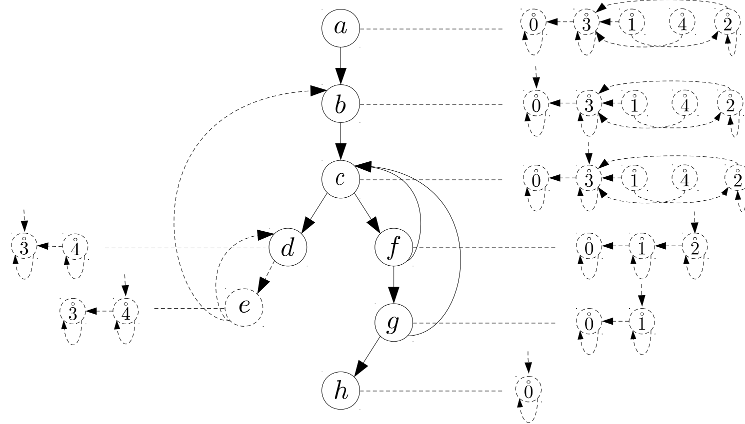


Figure 5.22: Example of scope tree construction by Algorithm 5.2

Example *Figure 5.22 illustrates an example of a scope tree construction with Algorithm 5.2. We assume default heuristics. Hence loops with shared heads yield single scopes. The figure shows a CFG annotated with the contents of array S, along with references* Next*, denoted by solid arrows and* Class*, denoted by dashed arrows. The vertical dashed arrow indicates the head of the pending scopes list. Initially all scopes form singleton equivalence classes and the scope tree is constructed as usual. Upon finishing node c, the lists of pending scopes are joined such that order of $\mathring{0}$ and $\mathring{3}$ is maintained for future completion. Scopes $\mathring{1}$ and $\mathring{2}$ complete and form $[\mathring{2}]$. Otherwise, the algorithm proceeds as usual. Upon finishing node a, S encodes the scope tree ($\mathring{1}$ is not considered as it is a member of $[\mathring{2}]$) Note that still $\mathring{\gamma}(g) = \mathring{1}$, which is eventually adjusted such that $\mathring{\gamma}(g) \leftarrow \mathring{2} = Class(\mathring{1})$.*

Complexity

The algorithm requires a single DFS traversal ($O(|V| + |E|)$, [103]). At each node, lists of pending scopes need to be joined. Joining sorted lists yields a complexity bound of $O(d)$, where d denotes the maximal depth of the scope tree. Hence, complexity is asymptotically bounded by $O(d|V| + E)$. Complexity can potentially be lowered to $O(\log_2(d)|V| + |E|)$ by maintaining pending scopes in search trees. From a practical perspective, introducing more sophisticated data structures for improved theoretical performance is likely to lower average performance. Empirical studies [121, 126] suggest low loop nesting depths in general. Also maintaining all state within an array leads to optimal cache utilization and a potentially large constant overhead is incurred due

NUM	Soft bound on number of CFG nodes
SEQ	Length bound sequence of constructs
DEPTH	Nesting bound
LOOP DEPTH	Loop nesting bound
EXIT SPAN	Bound of nesting levels an exit may leap
ENTRY SPAN	Bound of nesting levels an entry may leap
P(BLOCK)	Probability of basic blocks
P(IF)	Probability of "if" constructs
P(IFELSE)	Probability of "ifelse" constructs
P(WHILE)	Probability of "while" constructs
P(DOWHILE)	Probability of "dowhile" constructs
P(SEQ)	Probability of sequences
P(EXIT)	Probability of exits
P(ENTRY)	Probability of entries

Table 5.2: Parameters of CFG generator

to increased operation complexity. It should be noted, however, that loop depths can be significantly larger for interprocedural CFGs — commonly used in timing analysis. However, scopes are typically inserted near the end of the pending list since most scope entries or exits are not far (cf. Definition 5.28) and, thus, do not span across functions, unless recursion is modeled.

Evaluation

We briefly evaluate our approach to loop detection in the following and compare our proposed Algorithm 5.2 (ARRAY) with an implementation directly derived from its formalization in Equation 5.40 on page 113 (SET) which propagates data along the graph edges and the traditional loop detection based on reduction [117] as proposed in [111] (REDUC). We base all three on the same DFS implementation (cf. Section B.6). The union-find data structure for REDUC is implemented with path compression and union-by-rank [103]. Care has been taken to minimize dynamic memory allocation in all three cases.

All approaches are very fast: Real-world benchmarks are not suitable to explore corner-cases of these algorithms as the processing time is typically so low that other operations on the benchmarking system could potentially affect results significantly. We therefore choose a sampling resolution of at least 1 ms to mitigate these effects. To amplify the impact of the core operations of these algorithms on the runtime, we chose to generate large control flow graphs from randomly (parameterized) chosen AST. This allows the investigation of unusual cases such as extreme loop depths, which is highly uncommon in practice. Parameters specify probabilities of high-level constructs, "gotos" and nesting bounds (see Table 5.2). We only consider reducible graphs as REDUC cannot handle other cases.

Experiments are carried out on a single core of an *Intel Xeon E5630* (2.53 GHz, 4 cores, 128 kB/1 024 kB/12 MB (L1/L2/L3) cache) CPU. We measure the accumulated

CPU time of each run in which the same CFG is applied to all three approaches. Graphs are scaled from 10 000 to 600 000 nodes (soft bound) in step sizes of 5 000 and two samples per size. Average values are computed by the arithmetic mean.

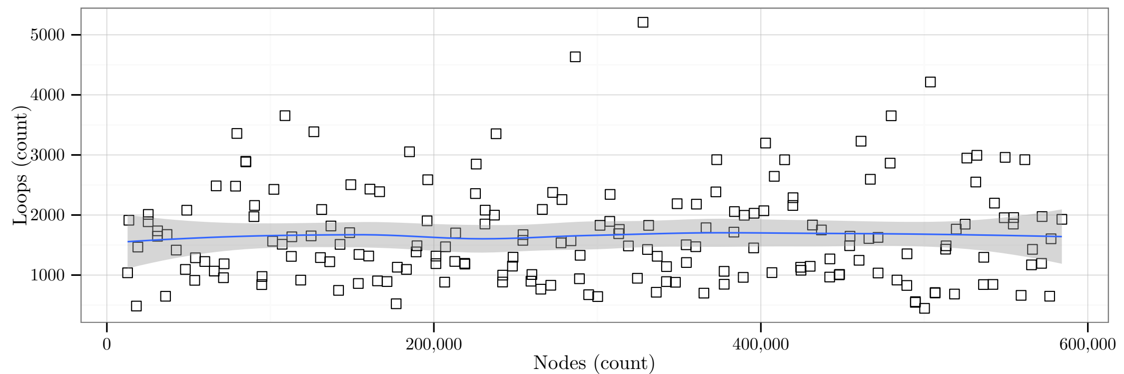


Figure 5.23: Distribution of loop sizes

Figure 5.23 illustrates the distribution of loop sizes of the AST generator. The line of best fit indicates a uniform distribution for all graph sizes.

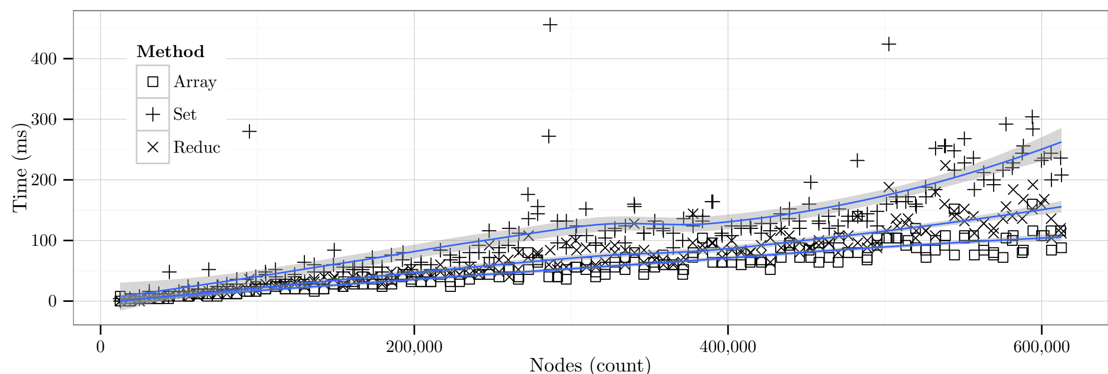


Figure 5.24: Runtime for non-degenerated CFGs (DEPTH $= 4$, LOOP DEPTH $= 3$, P(IF) $= 0.1$, IFELSE $= 0.2$, P(WHILE) $= 0.3$, P(DOWHILE) $= 0.4$, P(EXIT) $= 0.02$)

In Figure 5.24 we relate graph sizes with execution times in ms, given a "typical" distribution of constructs. From the lines of best fit, we can see that ARRAY dominates the other two approaches on average. It processes 600 000 nodes in around 100 ms, while REDUC require 50 % and SET 250 % more processing time on average. Notably, SET is prone to excessive runtimes due to its non-linear memory demand.

Figure 5.25 depicts the deviation (in %) in runtimes of SET and REDUC from ARRAY. For all graph sizes, REDUC performs ~20 % worse and SET performs ~50 % worse on average.

To illustrate the mere contribution of graph traversal, Figure 5.26 illustrates deviations as above for acyclic graphs. It suggests that performance is only insignificantly affected by the number of loops for "typical" loop counts, as ratios are comparable to Figure 5.25.

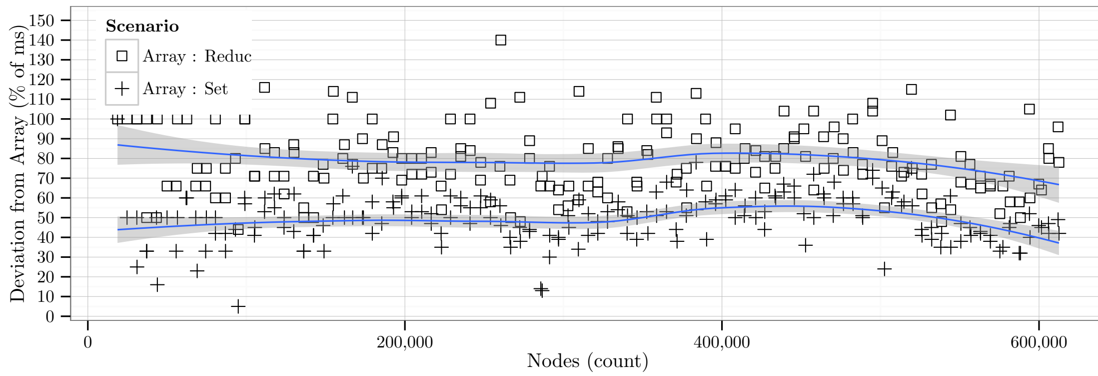


Figure 5.25: Correlation of runtimes for non-degenerated CFGs (DEPTH $= 4$, LOOP DEPTH $= 3$, P(IF) $= 0.1$, IFELSE $= 0.2$, P(WHILE) $= 0.3$, P(DOWHILE) $= 0.4$, P(EXIT) $= 0.02$)

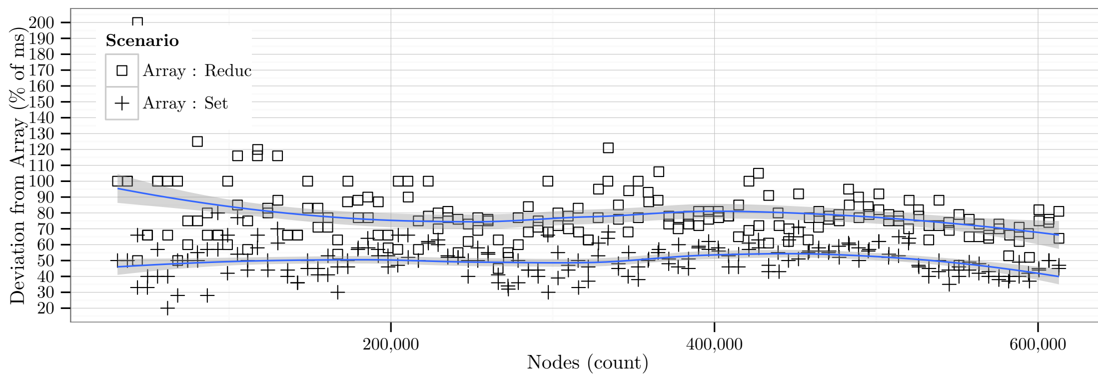


Figure 5.26: Correlation of runtimes for acyclic CFGs (DEPTH $= 4$, LOOP DEPTH $= 0$, P(IF) $= 0.4$, IFELSE $= 0.6$, P(WHILE) $= 0.0$, P(DOWHILE) $= 0.0$, P(EXIT) $= 0.0$)

To expose the impact of loops, we generated graphs with high probabilities for loops and with high loop nesting depth bounds, whose results are illustrated in Figure 5.27. For all graph sizes, REDUC performs ∼35 % worse and SET performs ∼55 % worse on average. REDUC is more affected by such deep nestings due to the overhead imposed by the reduction via union-find.

Conclusion

In this section we proposed an efficient and general algorithm for loop detection. As opposed to existing approaches which use heuristics that are solely guided by graph structure alone, we allow for explicit annotations to establish a simple interface to guide detection. Incorrect detection is a problem since flow facts — for example loop bounds — typically depend on the correct reconstruction of the original program structure. Our motivation is the provision of additional structural constraints along with other flow facts. In particular, the proposed algorithm is shown to outperform the traditional approach for

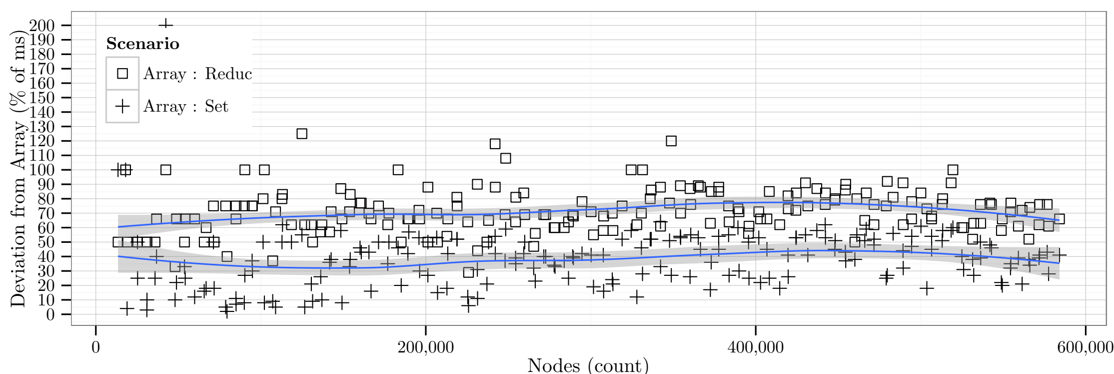


Figure 5.27: Correlation of runtimes for deeply nested CFGs ($\text{DEPTH} = 32, \text{LOOP DEPTH} = 30, \mathrm{P}(\text{IF}) = 0.1, \text{IFELSE} = 0.1, \mathrm{P}(\text{WHILE}) = 0.4, \mathrm{P}(\text{DOWHILE}) = 0.4, \mathrm{P}(\text{EXIT}) = 0.0, \text{EXIT SPAN} = 3$)

loop detection by at least 20 %. We have yet to address proper handling of irreducibility. We do this by proposing variants to traditional DFS in the following two sections.

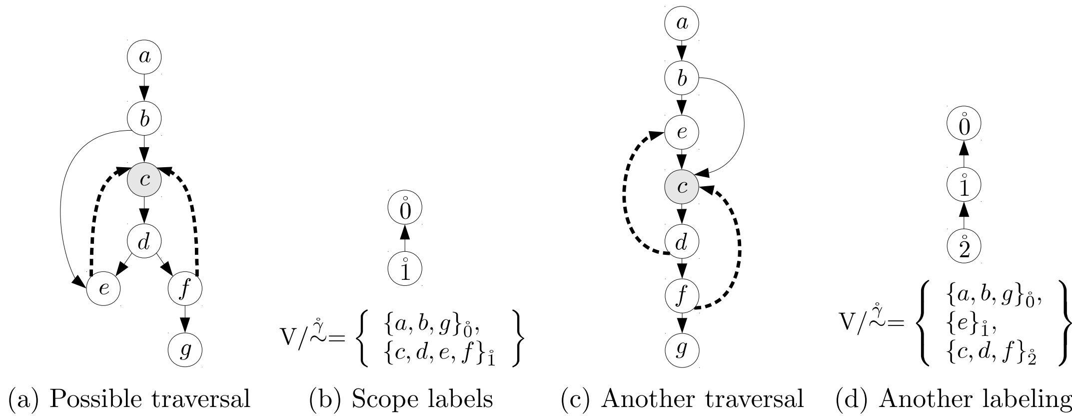
5.3.2.4 Handling Ambiguous Loop Nesting by Enumeration

Loop detection as proposed in the previous section provides a solution to the problems of ambiguous loop counts in the case of shared loop heads by multiple back edges, and a means to deal with ambiguous loop nesting in the case of entwined back edges (cf. Figure 5.11 on page 104). We have not, however, taken care of the issue of ambiguous loop heads, such as illustrated in Figure 5.11a, as this depends on the traversal sequence order of the underlying DFS. In this section we propose a solution to this problem by constructing a context-sensitive scope tree which models all feasible combinations of loop nestings, regardless of the DFS sequence. By this, a sound — but possibly not tight — path analysis can be performed despite irreducibility and lack of sufficient structural information to construct an unambiguous loop nesting model.

We first motivate the proposal, followed by a discussion on technical prerequisite and a proposal for a corresponding algorithm.

Motivation

Figure 5.28 illustrates an example of this issue. In Figure 5.28a, a CFG is illustrated in a layout that reflects the original program structure: a single loop with two back edges (dashed) and a side entry. We assume that a loop bound is provided for its head node c. If the DFS which is backing loop detection visits node c first, the resulting scope tree is depicted in Figure 5.28b, with the corresponding scope labeling $\mathring{\gamma}$ shown below. If instead node e is visited before node c, the assumed graph layout is that of Figure 5.28c. This also implies that a loop bound is missing for one of the two detected loops. The corresponding scope tree along with its labels is given in Figure 5.28d. Although by



(a) Possible traversal (b) Scope labels (c) Another traversal (d) Another labeling

Figure 5.28: Example of back edge ambiguity for loop detection

means of parameters for loop detection as proposed above, we could force loops to be recognized as a single scope, loop bounds remain to be incompletely specified.

A method to handle this problem is to enumerate all feasible combinations of nestings. Consequently, a path analysis for some context might be infeasible due to invalid or missing constraints but the maximal solution for all combinations is guaranteed to be sound.

Prerequisites

Each re-entry into an irreducible region yields a different context in which different back edge classifications are possible. Consequently CFG nodes are mapped to multiple and unrelated scopes within their respective contexts.

Without loss of generality, a scope is complete once its top node is finished. At this point in time, its relation to its parent scope can be established, as discussed previously.

Lemma 5.41 *A re-entry edge into a completed scope is a DFS forward or cross edge.*

Proof. Forward and cross edges lead to already finished nodes and a scope is complete if and only if all its CFG nodes are finished. □

By definition of scope entries and exits (Definition 5.28 on page 106), an entry edge is a CFG edge from a parent scope into a descendant one. (cf. Figure 5.28).

Lemma 5.42 *A re-entry edge leads from a pending scope to a complete scope.*

Proof. Scopes are complete if their top nodes are finished, which is also the point in time the parent is known; which itself cannot be complete before its descendants. The lemma then instantly follows from the definition of scope exits (Definition 5.28). □

Consequently, all other edges must be entirely contained within the same scope or lead to a parent scope (exit edge).

An entry edge from a pending to a complete scope yields a new context in which edges are to be reclassified by DFS and in which scopes are identified accordingly. This reclassification is bounded as there must exist exit edges back into parent scopes.

Theorem 5.43 *Let $\mathring{s}$ be a (pending) parent scope. Then for all possible descendant scopes $\mathring{t}$, it holds that an edge from $\mathring{t}$ to $\mathring{s}$ is an exit edge in all contexts.*

Proof. By Definition 5.28 of scope exits. □

Example *In Figure 5.28a and Figure 5.28c edge (f, g) remains the exit for any combination of descendant scopes to the very same parent scope. Although exits specific to newly identified descendant scopes might be discovered for new contexts, the exit edges to the originating parent scope remain. As only edges within already complete descendants are to be reclassified, a DFS traversal is bounded by these specific exit edges.*

To encode this enumeration of scopes, scope labels merely need to be extended by context. We define

$$\mathring{\gamma}^{\star} : V \mapsto \mathring{V}^{*} \tag{5.41}$$

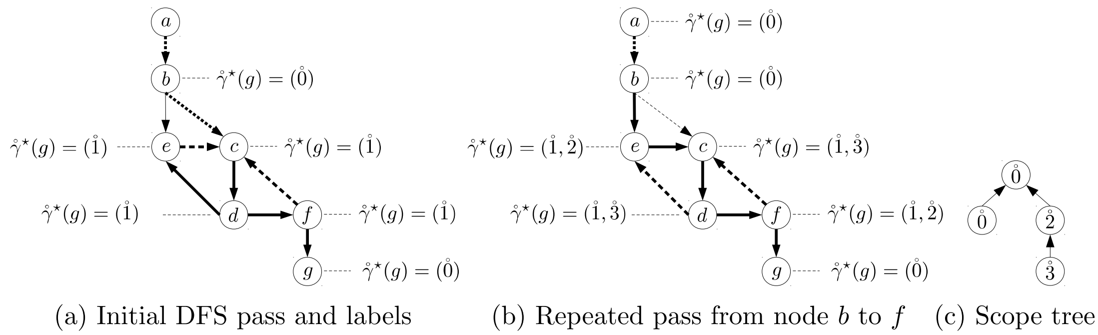
such that $\mathring{\gamma}^{\star}(u)$ denotes a "stack" of scopes node u is member of. The definition of scope tree $\mathring{G} = (\mathring{V}, \mathring{E})$ remains unchanged as it only defines parent relations of scopes and scopes of different contexts yield unrelated neighboring subtrees.

Algorithm

Algorithm 5.3 Scope-enumerating DFS

```
 1  let enumdfs s G = do                                   ◁ G = (V, E)
 2      initialize
 3      enqueue s in Q
 4      while nodes in queue Q do
 5          dequeue u from Q
 6          for each adjacent node v do
 7              if v finished then                         ◁ cross/forward edge
 8                  if scope "top of γ̊⋆(v)" not complete then
 9                      continue with next adjacent node
10                  else
11                      mark v as unvisited
12              if v unvisited then                        ◁ tree edge
13                  enqueue u in Q
14                  u ← v
15              else if v unfinished                       ◁ back edge
16                  mark (u, v) as back edge
17          mark u finished
```

Algorithm 5.3 specifies pseudo-code for the enumerating scope tree construction by a modification of non-recursive DFS as listed in full detail in Algorithm B.16. The function *enumdfs* (line 1) is invoked for a root node s and a CFG. After initialization (lines 2,3), nodes u in worklist Q are visited in order of their discovery (lines 4,5). For all adjacent nodes (line 6), we change the standard DFS semantics to meet our requirements. Recall that nodes are either *unvisited*, if they are discovered for the first time, *unfinished*, if there



(a) Initial DFS pass and labels (b) Repeated pass from node b to f (c) Scope tree

Figure 5.29: Example of enumerating loop detection

are still unvisited descendants, or otherwise *finished*. If an adjacent node v is already *finished* (line 7), we distinguish two cases (line 8): Either the most recently assigned scope label denotes membership to a pending scope (line 9), then edge (u, v) is a cross or forward edge, either within the same scope or as an exit, and, as usual, we just ignore it. Otherwise the most recently assigned scope label to node u denotes a completed scope (line 11), then we mark u as being unvisited. Subsequently, if the adjacent node v is marked unvisited, its predecessor u is queued (line 13) and v is visited (again). All (retreating) edges to unfinished nodes are back edges (line 16). Once all adjacent nodes have been finished, finish the current node u (line 17).

Algorithm 5.4 Scope-enumerating loop detection

```
1  let finish u = do
2      propagate scope lists H from successors of u
3      push list head H(u) onto stack γ̊*(u)
4      complete scopes as necessary
```

Algorithm 5.3 interacts with the loop detection proposed earlier. Algorithm 5.4 lists pseudo-code of the original Algorithm 5.2, stripped of details not relevant in this context. As usual, scope tree construction is performed once a node u is finished (line 1) in DFS. The algorithm decomposes into three sections: propagation of pending scopes (line 2), assignment of scope labels (line 3), and scope completion (line 4). We modify scope label assignment such that every time u finishes, an additional scope label is assigned.

Example *Figure 5.29 illustrates an execution of the aforementioned algorithm on the very same graph as depicted in Figure 5.28. For the CFG, thick solid edges denote tree edges and thick dashed edges denote back edges, respectively. In Figure 5.29a, the traversal of the graph which corresponds to Figure 5.28a is shown. Nodes are finished in post order and scope labels are assigned as usual. Upon revisiting node b, the forward edge (b, e) into the completed scope $\mathring{1}$ is encountered. Accordingly, node e and all nodes reachable from e that belong to complete scopes will be revisited. Consequently, the extended DFS proceeds as illustrated in Figure 5.29b, reclassifying edges along the way. Node f is the last such node. Hence, nodes finish in the order of the new tree edges, instantiating two additional*

scopes just as depicted in Figure 5.28c. Assuming default heuristics, the resulting scope tree corresponds to Figure 5.29c.

5.3.2.5 Handling Ambiguous Loop Nesting by Prenumbering

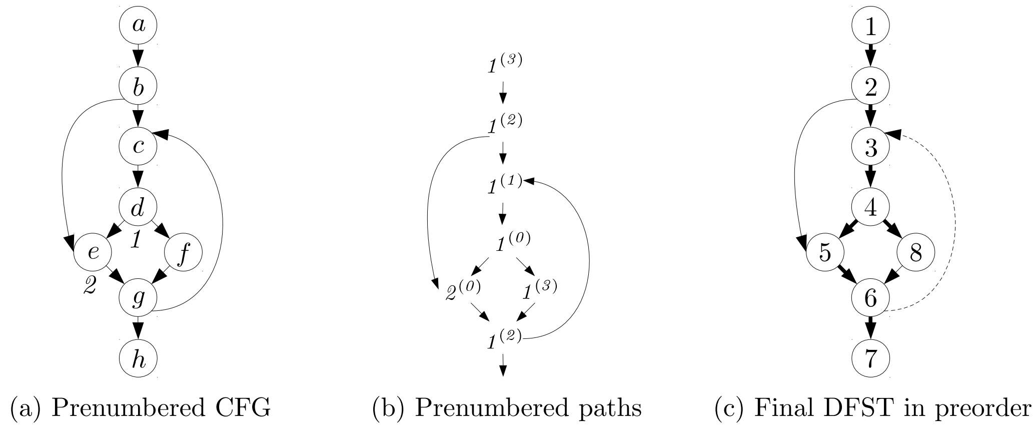
Another method to handle ambiguity is to allow for additional annotation to prevent ambiguity altogether. This eventually allows for most precise analysis results. In the following we propose a structural annotation we refer to as *prenumbering*, which allows for removal of ambiguity of DFS edge classification. The motivation to this approach is identical to the enumeration approach proposed above: flow constraints must match the presumed graph structure. If CFG reconstruction and flow constraints are independent, this easily leads to unsound results.

Studies [121, 126] suggest that reducible CFGs are the norm but a single irreducible subgraph deems an entire CFG irreducible. Many loop analyses [111] take the approach to isolate such subgraphs within their respective loop nesting representation which are then subject to transformation such as node splitting. Our hypothesis is that these regions are nevertheless still typically the result of "structured" programming, in the sense that irreducible control flow is still simple and logically structured – often a result of human programming — typically modeling exception handling or automata, and that such subgraphs are sparsely distributed over entire CFGs. As already elaborated earlier, heart of the problem is ambiguous DFS traversal. We propose prenumbering as a sparsely applied annotation with the intent to remove DFS ambiguity only in those locations required, and stick to standard DFS semantics in all remaining reducible regions otherwise. Prenumbering is optional, as we can always revert to — or combine this with — enumeration.

Intuition

Recall that in reducible graphs, although there might exist multiple feasible DFST, the set of back edges is unique (cf. Definition 5.7 on page 86). For irreducible graphs on the other hand, this is not the case. Prenumbering models constraints on visitation order of DFS to exclude the infeasible DFST explicitly.

Consider Figure 5.30, which illustrates the intuition of prenumbering. In Figure 5.30a an irreducible CFG is depicted, which yields three possible DFST, of which only the two in which edge (g, c) is a back edge are considered feasible. The italic labels next to nodes d and e denote prenumbers, which denote the constraint that any in feasible DFST, node d must have a smaller *preorder* label than node e after DFS traversal. We resolve prenumbers by labeling the entire CFG as shown in Figure 5.30b. These labels model feasible paths for a following DFS traversal to detect loops. Specifically, the labels model shortest paths from every program point to the smallest reachable prenumber. Distance is denoted by the number of intermediate nodes to a labeled target node (illustrated as superscripts in Figure 5.30b). A modified DFS then follows a shortest path, towards the



(a) Prenumbered CFG (b) Prenumbered paths (c) Final DFST in preorder

Figure 5.30: Example of prenumbering

smallest prenumbered node that has not been visited yet. A possibly resulting preorder labeling for one of the two feasible DFST under prenumbering is shown in Figure 5.30c.

Algorithm

We solve the problem of path precomputation by computing a fixed point of node labeling (cf. Figure 5.30b). The value domain is defined as the set of all functions $P \mapsto D$:

$$\mathbb{D}_{pre} = P \mapsto D \tag{5.42}$$

with "prenumbers" $P \subseteq \mathbb{N}_0$ that will guide a modified DFS and distance values $D \subseteq \mathbb{N}_0$ that enable DFS to follow the shortest path to the next prenumber labeling. The problem of shortest paths to prenumbered nodes has the structure of the semi-lattice:

$$(\mathbb{D}_{pre}, \top, \sqsubseteq, \sqcup) \tag{5.43}$$

where $\sqsubseteq$ denotes additional mappings or lower prenumbering defined as:

$$f \sqsubseteq g \Leftrightarrow \text{def}(f) \subseteq \text{def}(g) \wedge \forall p \in \text{def}(f) \cap \text{def}(g)\colon f(p) \leq g(p) \tag{5.44}$$

and $\sqcup$ denotes unification of mappings defined as:

$$f \sqcup g := \left\{ p \to \begin{cases} \min(f(p), g(p)) & \text{if } p \in (\text{def}(f) \cap \text{def}(g)) \\ f(p) & \text{if } p \notin \text{def}(g) \\ g(p) & \text{if } p \notin \text{def}(f) \end{cases} \right\} \tag{5.45}$$

For a transformer defined as:

$$\text{tf}_{pre} \colon V \mapsto (\mathbb{D}_{pre} \mapsto \mathbb{D}_{pre})$$
$$\text{tf}_{pre}(u) = \lambda f \,.\begin{cases} \{\text{pre}(u) \to 0\} & \text{if } \text{pre}(u) \neq \bot \\ \{p \to d+1 \mid (p \to d) \in f\} & \text{otherwise} \end{cases} \tag{5.46}$$

A least fixed point then denotes, for a node u, the set of shortest path distances to the reachable prenumbered nodes from u such that no path passes through another prenumbered node. This corresponds to the least solution of equation:

$$l_{pre} \colon V \mapsto \mathbb{D}_{pre} \mapsto \mathbb{D}_{pre}$$
$$l_{pre}(u) = \bigsqcup \{\text{tf}_{pre}(u)(l_{pre}(v)) \mid (u,v) \in E\} \tag{5.47}$$

Informally, we collect all prenumbers by backward propagation, “annotating” all program points with the reachable labels and the minimal distance to reach them.

Since we can expect most subgraphs to be reducible, quick convergence can be expected by processing nodes in reverse postorder [17] of *some* DFS, which is optimal in reducible subgraphs only.

We use the computed labeling l_{pre} to guide DFS to compute feasible DFST only. The intuition of the following algorithm is to discover prenumbered nodes in the specified order without contradicting discovery time. The strategy is to guide DFS to follow the shortest path to the lowest undiscovered prenumbered node, respectively.

Definition 5.44 (Feasible DFST) *Let $D(u) = (V_D, E_D)$ with $V_D \subseteq V$ and $E_D \subseteq T$ denote a DFST rooted in node u, and let d denote discovery time stamps of DFS. Then D is* feasible *if and only if $\forall u_i \in V_D$ with $\text{pre}(u_i) \neq \bot$ such that $\text{pre}(u_1) < \text{pre}(u_2) < \cdots < \text{pre}(u_k)$ it holds that $d(u_1) < d(u_2) < \cdots < d(u_k)$.*

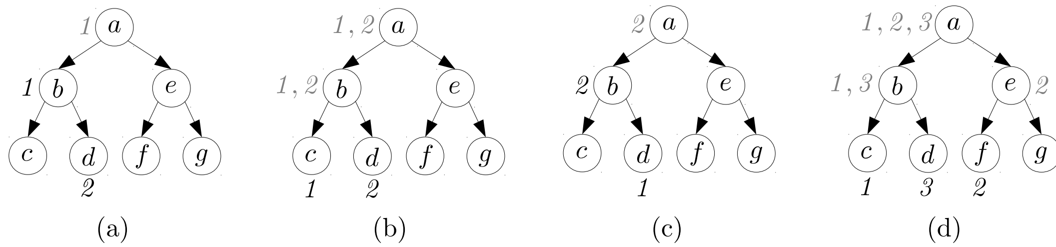


Figure 5.31: Examples of feasible (a,b) and infeasible DFST (c,d)

Example *Figure 5.31 illustrates examples of feasible and infeasible DFST with respect to prenumbering. Prenumbering is denoted by black italic labels next to graph nodes. Gray labels denote synthesized labels from l_{pre}. In each case, our guided DFS starts at node a. In Figure 5.31a node b is discovered first, according to our traversal strategy. The next greater prenumbered node d is contained in the subgraph rooted in b, which yields $d(b) < d(d)$. Since no more prenumbered nodes are to be visited, DFS proceeds as usual.*

In Figure 5.31b, prenumbered nodes are contained in neighboring subgraphs but it holds that $d(c) < d(d)$. Figure 5.31c and Figure 5.31d illustrate infeasible cases, respectively. In the first case, the lowest prenumbered node d is not reachable without passing through node b first, hence $d(b) \not< d(d)$. The second case violates the order of discovery since $d(c) < d(d) \not< d(f)$.

Algorithm 5.5 Prenumbered DFS

```
1  let prenumdfs s G = do                                          ◁ G = (V, E)
2      initialize
3      Q ← {(s, succ(s))}
4      W ← {p | p ∈ img(pre)}
5      if min def(l_pre(s)) > min W then
6          infeasible
7      while Q ≠ ∅ do
8          (u, S) ← pop Q
9          while S ≠ ∅ do
10             C ← {w ∈ S: l_pre(w)(min W) ≥ min W)}
11             if C ≠ ∅ then
12                 if min ⋃_{w∈C} def(l_pre(w)) > min W then
13                     infeasible
14                 v ← w: w ∈ C ∧ ∀w′ ∈ C: l_pre(w)(min W) ≤ l_pre(w′)(min W)
15             else
16                 v ← any S
17             S ← S \ {v}
18             if v unvisited then                                  ◁ tree edge
19                 Q ← Q · ((u, S))
20                 if pre(v) = min W then
21                     W ← W \ min W
22                 (u, S) ← (v, succ(v))
23             else if v unfinished then                            ◁ back edge
24                 mark (u, v) as back edge
25             else
26                 skip                                     ◁ cross/forward edge
27         mark u finished
```

Let $(p \to d) = (p, d)$. In the following we assume that it holds that $\min((p, d), (p, d')) = (p, \min(d, d'))$ that $\{(p, d)\} \cup \{(p, d')\} = \{(p, \min(d, d')\}$ and that $x \neq \mathrm{def}(f) \Leftrightarrow f(x) = \bot$ and $\forall y \in \mathrm{img}(f)\colon \bot < y$.

The extension to DFS is listed in Algorithm 5.5 (cf. Algorithm B.16 for details on DFS). In addition to the usual stack for DFS Q (line 3), we maintain a worklist W of all prenumber labels in ascending order during regular DFS (line 4). If the node of lowest label is not reachable, the algorithm fails (lines 5,6). If there exist unvisited adjacent nodes (line 9), set C denotes the set of nodes such that each such potential branch target must be on a path to a prenumbered node with a label at least as high as the next prenumber in the worklist (line 10). This does not differentiate yet whether those candidates are feasible. However, it excludes already visited and unlabeled nodes. If there are no such reachable nodes (line 11), continue with DFS as usual (line 16). Otherwise, if the lowest labeled reachable node is not the lowest label in W, prenumbering is infeasible

(lines 12,13). Otherwise, the lowest reachable label equals the lowest label in W and we select the candidate of minimal distance to the target (line 14). If a node v with an explicit prenumber is being discovered, it is removed from worklist W (lines 20,21). Otherwise, DFS proceeds as usual.

Correctness

Equation 5.47 computes shortest paths to reachable annotated nodes. To reach a fixed point, the transformer must be monotone.

Lemma 5.45 *Transformer* tf_{pre} *(Equation 5.46) is monotone.*

Proof. Given a node u, let $t = \mathrm{tf}_{pre}(u)$ and functions $f, g \colon P \mapsto D$ such that $f \sqsubseteq g$.
If $\mathrm{pre}(u) \neq \bot$:

$$
\begin{aligned}
& t(f) \sqsubseteq t(g) \\
& \Leftrightarrow \{\ (\mathrm{pre}(u), 0)\} \sqsubseteq \{(\mathrm{pre}(u), 0)\} \\
& \Leftrightarrow \{\ \mathrm{pre}(u)\} \subseteq \{\mathrm{pre}(u)\} \wedge 0 \leq 0
\end{aligned}
$$

If $\mathrm{pre}(u) = \bot$:

$$
\begin{aligned}
& t(f) \sqsubseteq t(g) \\
& \Leftrightarrow t(\{(p_0, d_0), \dots\}) \sqsubseteq t(\{(p_0', d_0'), \dots\}) \\
& \Leftrightarrow \{(p_0, d_0 + 1), \dots\} \sqsubseteq \{(p_0', d_0 + 1'), \dots\} \\
& \Leftrightarrow \{p_0, \dots\} \subseteq \{p_0' \dots\} \wedge \forall p_i \in \{p_0, \dots\} \cap \{p_0', \dots\} \colon d_i + 1 < d_i' + 1
\end{aligned}
$$

□

Lemma 5.46 *For node u and a prenumbered node v, l_{pre} denotes the shortest path length from u to v without passing through another prenumbered node:*

$$
l_{pre}(u)(\mathrm{pre}(v)) = \min\{|\pi| \colon u \overset{\pi}{\rightsquigarrow} v \wedge \nexists w \in \pi \setminus \{u, v\} \colon \mathrm{pre}(w) \neq \bot\} \tag{5.48}
$$

Proof. Property holds by definition of $\sqcup$ (Equation 5.45) and $\mathrm{tf}_{\mathrm{pre}}$ (Equation 5.46). □

Why are we considering only shortest paths?

Corollary 5.47 *Shortest paths are acyclic due to positive weight cycles [103]. Hence, prenumbered nodes are discovered via tree edges only and explicit guiding can be ceased early.*

Why are labeled intermediate nodes not allowed on paths?

Corollary 5.48 *Let $u \overset{\pi}{\rightsquigarrow} v \overset{\pi'}{\rightsquigarrow} w$ such that $\pi \cdot \pi'$ denotes an unconstrained path from u to w and let v, w be prenumbered. Either pre(v) $>$ $\mathrm{pre}(w)$, then π' is infeasible and therefore w is not reachable from u via v. Or pre(v) $<$ $\mathrm{pre}(w)$, then we reach w by reaching v first, from where information on w is available.*

By exploiting transitivity, sets in tf_{pre} remain small.

Intuitively, DFS_{pre} (Algorithm 5.5) follows a (shortest) path to the node with the lowest label in the worklist without passing through another labeled node. Once the specific target node has been reached, a path to the next greater label is searched. Unlabeled paths are traversed non-deterministically as usual. In either case, nodes are visited in preorder or the prenumbering is infeasible.

Lemma 5.49 *During* DFS_{pre}*, the minimal element of worklist W denotes the next node to discover.*

Proof. DFST feasibility (Definition 5.44) is guaranteed to hold. □

Lemma 5.50 *During* DFS_{pre}*, worklist W denotes only undiscovered nodes.*

Proof. Prenumbering is unique and Lemma 5.49 holds. The first next prenumbered node will be removed from W once discovered. □

Theorem 5.51 (Candidates) *During* DFS_{pre}*, for a currently visited node u, the set of possible branch candidates $C \subseteq S$ denotes only undiscovered nodes.*

Proof. By Lemma 5.50, discovered nodes are discarded. □

Theorem 5.52 (Next Target) *Let d denote discovery time and let f denote finishing time. For nodes u, v such that* $\text{pre}(u) < \text{pre}(v)$*, it holds that either $d(u) < f(u) < d(v) < f(u)$ or $d(u) < d(v) < f(u) < f(u)$.*

Proof. By parenthesis theorem (Theorem 5.11). □

Consequently, the candidate set C may be empty if the next target for DFS_{pre} lies in a neighboring subtree but we eventually retreat to a point where v is reachable again.

Theorem 5.53 (Infeasible Targets) *Let $D(u) = (V_D, E_D)$ denote a DFST rooted in a node u. Let $P(u)$ denote the set of reachable prenumbers in $D(u)$ and let W denote the worklist in* DFS_{pre}*. Prenumbering is infeasible if and only if* $\min P(u) > \min W$.

Proof. Recall that $P(u)$ only contains the directly reachable labels. Given node v such that $\text{pre}(v) = \min P(u)$ and node w such that $\text{pre}(w) = \min W$. If $v \in V_D(D(u))$ and $w \in V_D(D(v))$ then $d(v) < d(w)$ which contradicts $\text{pre}(v) > \text{pre}(w)$ (cf. Figure 5.31c). If $v \in V_D(D(u))$ and $w \notin V_D(D(u))$ then $d(v) < d(w)$ which contradicts $\text{pre}(v) > \text{pre}(w)$. (cf. Figure 5.31d). Otherwise, w can be reached from u directly without going through v, which guarantees $d(w) < d(v)$. □

In Algorithm 5.5, we check for this property in lines 5 and 12, where $\text{def}(l_{pre}(u)) = P(u)$ denotes the set of reachable labels.

Theorem 5.54 (Feasible Target) *If prenumbering is feasible, for every most recently discovered node u, it holds that* $\min P(u) = \min W$.

Proof. By Theorem 5.53 it holds that $\min P(u) \leq \min W$, and $\min P(u) < \min W$ contradicts the fact that DFS_{pre} follows a path to $\min W$ first. Therefore, for $\text{pre}(w) = \min W$ and $\forall v \in V_D(D(u))$, $d(w) < d(v)$. □

Consequently, in Algorithm 5.5 line 14, it is sufficient to select the prenumbered node of smallest distance as the next branch target.

5.3.2.6 Conclusion

We have been concerned with aspects of control flow reconstruction. Specifically, we addressed the problem of mismatching of flow constraints and reconstructed control flow structure due to ambiguity in loop detection. To this end, we introduced scopes and scope trees as the fundamental data structures to guide the following path analysis and we proposed different techniques for their construction. We proposed a general, configurable and efficient loop detection. In addition, we proposed enumeration and prenumbering as techniques to guide loop detection in case of irreducible control flow graphs. Overall, we proposed a set of structural annotations and the corresponding technical framework for their application.

5.3.3 Computing Worst-Case Execution Time Bounds

In this section we are concerned with the computation of upper bounds on the WCET of a task. Effectively, the underlying framework represents a general, efficient and flexible way to express various path-related problems in timing analysis. Computation of WCET bounds is just one specific use case. Later, we will discuss different variations of this framework for other problems specifically. All fundamental principles will only be discussed in this section and will not be repeated later.

In Section 5.3.3.1 we provide formal basic and technical prerequisites. We first propose how to efficiently compute a WCET bound for a single scope in Section 5.3.3.2 to introduce the basic constraint model and we propose several optimizations, followed by an extension to complete tasks in Section 5.3.3.3. In Section 5.3.3.4 we show how to compute WCET bounds within subgraphs: from and to arbitrary program points. From the formal model, we derive an efficient algorithm in Section 5.3.3.5. We evaluate our proposal in Section 5.3.3.6 and conclude the discussion in Section 5.3.3.7.

5.3.3.1 Prerequisites

We first provide fundamental definitions that will be used throughout the remainder of this chapter.

We first briefly introduce the most important definitions given in Table 5.3 without elaborating on their specific purpose yet. Let $G = (V, E, s, t)$ denote a CFG with entry s and exit t, let $\vec{G} = (V, E^F, s, t)$ denote its corresponding DAG, let $\mathring{G} = (\mathring{V}, \mathring{E})$ denote its corresponding scope tree. Let A_π denote a set of *symbolic annotations* such that

$G = (V, E, s, t)$	Control flow graph
$\vec{G} = (V, E^F = E \setminus B, s, t)$	DAG of G
$\mathring{G} = (\mathring{V}, \mathring{E})$	Scope tree
$\mathring{\gamma}\colon V \mapsto \mathring{V}$	Scope labeling
A_π	Annotation labels
$\alpha_\pi\colon V \mapsto {A_\pi}^*$	Annotation labeling
$\alpha_\pi^{-1}\colon A_\pi \mapsto V$	Annotation location
S_π	Path states
$s_\pi = (\delta_\pi, \sigma_\pi, o_\pi) \in S_\pi$	Path state
$\delta_\pi\colon S_\pi \mapsto \mathbb{N}_0^{\infty,\perp}$	Path length
$\sigma_\pi\colon S_\pi \mapsto {A_\pi}^*$	Path signature
$o_\pi\colon S_\pi \mapsto V$	Path origin
$\omega\colon V \mapsto \mathbb{N}_0$	Node weight
$\beta_\pi\colon A_\pi \mapsto \mathbb{N}_0$	Flow bound

Table 5.3: Definitions for path analysis

$\alpha_\pi\colon V \mapsto {A_\pi}^*$ denotes *annotation labeling* at a program point and $\alpha_\pi^{-1}\colon A_\pi \mapsto V$ denotes *annotation locations* in the CFG.

We represent paths by means of *path states* $(\delta_\pi, \sigma_\pi, o_\pi) = s_\pi \in S_\pi$, which encode path *length* $\delta_\pi\colon S_\pi \mapsto \mathbb{N}_0^{\infty,\perp}$ where $\mathbb{N}_0^{\infty,\perp} = \mathbb{N}_0 \cup \{\infty, \perp\}$, *signature* $\sigma_\pi\colon S_\pi \mapsto {A_\pi}^*$, which denotes a sequence of annotations along paths, and *origin* of paths $o_\pi\colon S_\pi \mapsto V$.

The only annotations we will use here are flow bounds $\beta_\pi\colon A_\pi \mapsto \mathbb{N}_0$ as capacity bounds and $\omega\colon V \mapsto \mathbb{N}_0$ to denote node weights in terms of upper bounds on the WCET per program point.

We now define the purpose of signatures and the underlying arithmetic, and we introduce additional terminology.

Signatures induce equivalence classes of path states of identical signature and origin by the relation

$$\begin{gathered} \overset{A_\pi}{\sim}\colon S_\pi \times S_\pi \\ s_\pi \overset{A_\pi}{\sim} {s_\pi}' \Leftrightarrow \sigma_\pi(s_\pi) = \sigma_\pi({s_\pi}') \wedge o_\pi(s_\pi) = o_\pi({s_\pi}') \end{gathered} \tag{5.49}$$

such that for a set of path states S, $S/\overset{A_\pi}{\sim}$ denotes the set of all equivalence classes. Intuitively, paths are *comparable* by length if and only if their signature and origin matches. The rationale is that if two paths are associated with identical sets of constraints, then there is no reason to represent both explicitly as they belong to a set of paths of which only the longest is of relevance. The underlying algebraic structure is the commutative semi-ring

$$(\mathbb{N}_0^{\infty,\perp}, \max, \perp, +, 0) \tag{5.50}$$

where $\mathbb{N}_0^{\infty,\perp} = \mathbb{N}_0 \cup \{\infty, \perp\}$ with max for addition and $+$ for multiplication, with neutral elements $\perp$ and 0, respectively such that:

$$\max(a,b) := \begin{cases} \max_{\mathbb{N}_0^\infty}(a,b) & \text{if } a \neq \perp \wedge b \neq \perp \\ a & \text{if } a \neq \perp \\ b & \text{otherwise} \end{cases} \tag{5.51}$$

$$a + b := \begin{cases} a +_{\mathbb{N}_0^\infty} b & \text{if } a \neq \perp \wedge b \neq \perp \\ \perp & \text{otherwise} \end{cases} \tag{5.52}$$

where $\max_{\mathbb{N}_0^\infty}$ denotes maximum and $+_{\mathbb{N}_0}$ denotes addition in $\mathbb{N}_0^\infty$. We lift (without explicit definition) max to path states of the *same* equivalence class and define:

$$\max_\pi \colon \wp(S_\pi) \mapsto S_\pi$$

$$\max_\pi(S) \in \{s \in S \mid \forall s' \in S \colon \delta_\pi(s) = \max(\delta_\pi(s), \delta_\pi(s'))\} \tag{5.53}$$

For an equivalence class $S = [s]$, state $\max_\pi([s])$ denotes a path of maximal length.

In the context of network flows, we provide additional definitions related to scopes (cf. Section 5.3.2.2 on page 105). Let $f \colon V \mapsto \mathbb{N}_0^\infty$ denote net flow (out of) a node.

Definition 5.55 (Feasible Iteration) *An iteration π is* feasible *if and only if* $\forall u \in \pi \colon f(u) > 0$.

Definition 5.56 (Unroll) *An* unroll $\pi_\mu = (s, \ldots, t)$ *of a scope $\mathring{s}$ is a (not necessarily cyclic) concatenation of iterations such that π_μ is connected:*

$$\pi_\mu = (s, \ldots, t) \wedge \forall (u_i, u_{i+1}) \subseteq \pi_\mu \colon (u_i, u_{i+1}) \in E(V) \tag{5.54}$$

Recall that by definition of scope iterations (Definition 5.30 on page 106), unrolls start in scope entries and terminate in nodes which are mapped to scope $\mathring{s}$ or one of its descendants. We can assume terminal node t to denote an exit of the same scope for now. We will later relax this notion. Similarly to iterations, unrolls are subject to flow constraints.

Definition 5.57 (Feasible Unroll, Bounded Unroll) *An unroll π_μ is* feasible *if all its constituting iterations are feasible. It is* bounded *if and only if* $\forall u \in \pi_\mu \colon f(u) \leq \infty$.

The algebra in Equation 5.50 denotes infeasibility by $\perp$ and unboundedness by ∞.

5.3.3.2 Computing WCET Bounds on a Single Scope

We first consider the computation of maximal path lengths for a single scope without parents or descendants. We assume annotations to denote only flow bounds in the following: $\forall A \in \text{img}(\alpha_\pi) : \forall a \in A \colon \beta_\pi(a) \neq \perp$. Intuitively, we solve a variant of MAXLEN (cf. Equation 5.9) with node weights and node constraints. Although a CFG could be

interpreted as a flow network directly to tackle this problem, our hypothesis is that constraints are often sparsely distributed — for example, loop bounds denote just a single constraint per loop. We therefore compute a path-compressed network representation of the CFG, which only denotes annotations and their relations. This effectively decouples path search from constraint solving.

In the following we first discuss the computation of iterations, then we discuss unrolling in principle by reduction to flow networks, followed by a proposal for direct unrolling without explicit reduction. We initially show how to test feasibility of unrolls, followed by the corresponding maximization of path length. We then focus on optimization, proposing various techniques to reduce overhead to achieve high performance.

Computation of Iterations

We first compute *possible* iterations of a scope. We use relation $\overset{A_\pi}{\sim}$ to compute partitions of simple paths, constrained by identical annotations and whose representative denotes the longest path in each partition, respectively. This problem has the structure of the semi-lattice:

$$(\mathbb{D}_{wcet}, \top, \sqsubseteq, \sqcup) \tag{5.55}$$

where $\mathbb{D}_{wcet} = \wp(S_\pi)$ is a set of path states, where $S_\pi \sqsubseteq {S_\pi}' \Leftrightarrow S_\pi \subseteq {S_\pi}'$ and where $S_\pi \sqcup {S_\pi}' := \{\max_\pi[s] \mid [s] \in (S_\pi \cup {S_\pi}')/\overset{A_\pi}{\sim}\}$ denotes the set of path states of different signature, origin and maximal length. We define the corresponding transformer as:

$$\begin{aligned} \mathrm{tf}_{wcet} &\colon V \mapsto (\mathbb{D}_{wcet} \mapsto \mathbb{D}_{wcet}) \\ \mathrm{tf}_{wcet}(u) &= \lambda S \,.\, \{(\delta_\pi(s) + \omega(u), \sigma_\pi(s) \cdot \alpha_\pi(u), o_\pi(s)) \mid s \in S\} \end{aligned} \tag{5.56}$$

Function tf_{wcet} increases distance δ_π by the node weight ω, adds annotations α_π to σ_π and maintains the path origin.

Let $\mathrm{pred}^{\rightarrow}$ denote predecessors on forward edges E^F. For an initial state $(0, a, s)$, where 0 denotes the initial distance, a denotes an initial annotation label and s denotes the origin of the path, the set of reachable path states from entry node s is denoted by:

$$\begin{aligned} S_{wcet} &\colon V^2 \times A_\pi \mapsto S_\pi \\ S_{wcet}(s, v, a) &= \begin{cases} \mathrm{tf}_{wcet}(u)(\{(0, a, s)\}) & \text{if } v = s \\ \bigsqcup \{\mathrm{tf}_{wcet}(u)(S_{wcet}(s, u, a)) \mid u \in \mathrm{pred}^{\rightarrow}(v)\} & \text{otherwise} \end{cases} \end{aligned} \tag{5.57}$$

which computes a least fixed point denoting longest paths in a program point. Consequently, $\max_\pi S_{wcet}(s, t, \epsilon)$ denotes the maximal simple path length in scope $\mathring{s}$ from entry s to node t. Path states $S_{wcet}(s, t, \epsilon)$ denote possible but not necessarily feasible iterations of maximal length.

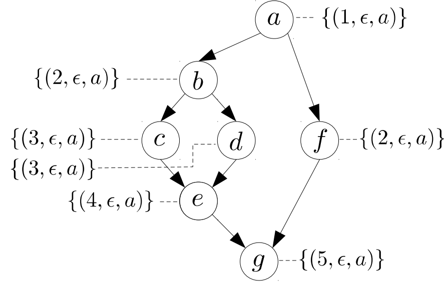


Figure 5.32: Example of longest path without annotations

Example *Let $\alpha_\pi = \emptyset$ and $\forall u \in V : \omega(u) = 1$, then Figure 5.32 illustrates the corresponding states per node $u \in V$ for $S_{wcet}(a, u, \epsilon)$, where $\max_\pi S_{wcet}(a, u, \epsilon)$ denotes the longest path to u.*

We lift S_{wcet} to distinguish iteration types (cf. Definition 5.31 on page 106). Let $a_0^u \in A_\pi$ denote a *unique* default annotation such that $\beta_\pi(a_0^i) = \infty$ for every entry $u \in \text{entry}(\mathring{s})$. This is a technical provision for otherwise unbounded paths. Then entry paths, for an entry node s, are denoted by:

$$S^I_{wcet} \colon \mathring{V} \times V \mapsto \wp(S_\pi)$$
$$S^I_{wcet}(\mathring{s}, s) = \bigcup_{u \in \text{bottom}(\mathring{s})} S_{wcet}(s, u, a_0^s) \tag{5.58}$$

Exit paths, for an exit node t, are denoted by:

$$S^O_{wcet} \colon \mathring{V} \times V \mapsto \wp(S_\pi)$$
$$S^O_{wcet}(\mathring{s}, t) = \bigcup_{u \in \text{top}(\mathring{s})} S_{wcet}(u, t, a_0^u) \tag{5.59}$$

Kernel paths are denoted by:

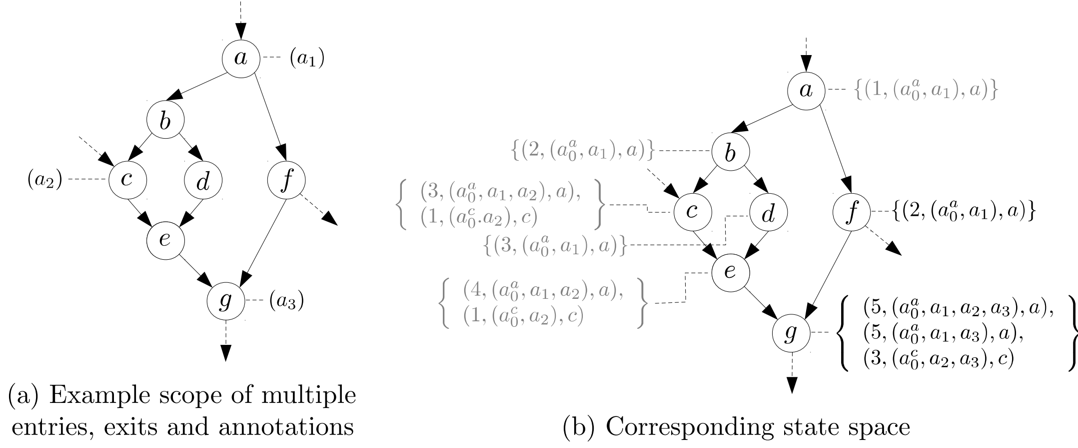
$$S^K_{wcet} \colon \mathring{V} \mapsto \wp(S_\pi)$$
$$S^K_{wcet}(\mathring{s}) = \bigcup_{t \in \text{top}(\mathring{s})} \bigcup_{b \in \text{bottom}(\mathring{s})} S_{wcet}(t, b, a_0^t) \tag{5.60}$$

In addition, we define *direct* paths from an entry s to an exit t as:

$$S^D_{wcet} \colon \mathring{V} \times V \times V \mapsto \wp(S_\pi)$$
$$S^D_{wcet}(\mathring{s}, s, t) = S_{wcet}(s, t, a_0^s) \tag{5.61}$$

The unused parameter $\mathring{s}$ to S^D_{wcet} is a technical provision and can be ignored for now.

Example *Figure 5.33 provides an example of an annotated scope $\mathring{s}$ with top a, bottom g, entries a, c and exits f, g. Figure 5.33a illustrates only non-empty annotations and Figure 5.33b illustrates the corresponding state space induced by S_{wcet} which summarizes maximal paths partitioned by signature and origin. States at scope bottoms and exits*



(a) Example scope of multiple entries, exits and annotations

(b) Corresponding state space

Figure 5.33: Example of path states of general scopes

summarize the respective longest path partitions. From these, we will construct our network. All other states are grayed out. For entry c and exit f, discrimination by iteration type yields:

$$
\begin{aligned}
S^I_{wcet}(\mathring{s}, c) &= \{(3, (a^c_0, a_2, a_3), c)\} && \text{(entries)} \\
S^O_{wcet}(\mathring{s}, f) &= \{(2, (a^a_0, a_1), a)\} && \text{(exits)} \\
S^K_{wcet}(\mathring{s}) &= \{(5, (a^a_0, a_1, a_2, a_3), a), (5, (a^a_0, a_1, a_3), a)\} && \text{(kernels)} \\
S^D_{wcet}(\mathring{s}, c, f) &= \emptyset && \text{(direct)}
\end{aligned}
$$

Flow Networks for Unrolling

The signature of a path state is a path-compressed representation of annotations along a CFG path. The length value of a path state denotes the maximal length among all CFG paths sharing the same set of annotations. This effectively separates the problem of finding longest possible iterations from the problem of deciding unroll feasibility or the computation unrolls of maximal length.

It is straight-forward to reduce signatures to flow networks which we define as:

$$\mathring{N} = (V, E, s, t, \omega, \ell, \beta) \in \mathring{\mathbb{N}} \tag{5.62}$$

where ℓ denotes lower and β denotes upper flow bounds, respectively. We explicitly have to distinguish two types of networks: Network $\mathring{N}_{iok}$ models annotations in order to compute unrolls by composition of entry, exit and kernel paths. Network $\mathring{N}_d$ models annotations in order to find direct paths from an entry to an exit. Every signature by construction represents exactly one path in these networks, which is why we can avoid their explicit construction for all problems that can be tackled by the Ford-Fulkerson method [100, 101] (cf. Section B.2).

We define a helper function which, for a set of path states, collects all annotations, representing network vertices:

$$n := \lambda S \,.\, \{a \mid a \in \sigma_\pi(s), s \in S\} \tag{5.63}$$

Further, we define a helper function that turns signatures π into sets of network edges, and which defines additional edges to connect node s to the head of π and node t to its tail, respectively:

$$l := \lambda(s, t, \pi) \,.\, \begin{cases} \{(s,t)\} & \text{if } \pi = \epsilon \\ \{(s, \pi_1), (\pi_{|\pi|}, t)\} \cup \{(u,v) \subseteq \pi\} & \text{otherwise} \end{cases} \tag{5.64}$$

Besides nodes for annotations, the networks contain additional nodes where s, t denote network source and sink, and i, o model paths into and out of the scope, respectively.

We first define network $\mathring{N}_{iok} \in \mathring{\mathbb{N}}$. Let $s_i \in S^I_{wcet}(\mathring{s}, u)$ and $s_o \in S^O_{wcet}(\mathring{s}, v)$ denote entry and exit states. Then the network is defined by:

$$\begin{aligned} \mathring{n}_{iok} &\colon S_\pi \times S_\pi \mapsto \mathring{\mathbb{N}} \\ \mathring{n}_{iok}(s_i, s_o) &= (V_{iok}, E_{iok}, s, t, \ell_{iok}, \beta_{iok}) \end{aligned} \tag{5.65}$$

where

$$\begin{aligned}
V_{iok} &:= n(\{s_i, s_o\} \cup S^K_{wcet}(\mathring{s})) \cup \{s, t, i, o\} \\
E_{iok} &:= \{(s, i), (o, t)\} \\
&\quad \cup l(i, t, \sigma_\pi(s_i)) && \text{(entry)} \\
&\quad \cup l(s, o, \sigma_\pi(s_o)) && \text{(exit)} \\
&\quad \cup \bigcup_{s_k \in S^K_{wcet}(\mathring{s})} l(s, t, \sigma_\pi(s_k)) && \text{(kernels)} \\
\ell_{iok} &:= \lambda u \,.\, \begin{cases} 1 & \text{if } u \in \{i, o\} \\ 0 & \text{otherwise} \end{cases} && \text{(lower bounds)} \\
\beta_{iok} &:= \lambda u \,.\, \begin{cases} 1 & \text{if } u \in \{i, o\} \\ \infty & \text{if } u \in \{s, t\} \\ \beta_\pi & \text{otherwise} \end{cases} && \text{(upper bounds)}
\end{aligned}$$

This construction is easy to understand by example.

Example *Figure 5.34 illustrates the network for the CFG of Figure 5.33 for the entry/exit pair (c, f) and respective entry path state $(3, (a_1, a_2), c)$ and exit path state $(2, (a_0), a)$. Upper flow bounds β_π are extended in β_{iok} to restrict entry and exit paths. On the other hand, lower flow bounds ℓ_{iok} for nodes i and o guarantee the existence of feasible entry and exit paths to and from the scope.*

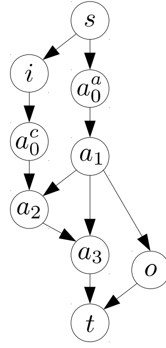


Figure 5.34: Example network $\mathring{n}_{iok}$ for Figure 5.33 for entry c and exit f

Similarly, we define the network for direct paths $\mathring{N}_d$ as:

$$\begin{aligned} \mathring{n}_d &\colon S_\pi \mapsto \mathring{\mathbb{N}} \\ \mathring{n}_d(s_d) &= (V_d, E_d, s, t, \ell_d, \beta_d) \end{aligned} \tag{5.66}$$

where

$$\begin{aligned} V_d &:= n(\{s_d\}) \cup \{s, t, i, o\} \\ E_d &:= \{(s,i), (o,t)\} \\ &\quad \cup l(i, o, \sigma_\pi(s_d)) && \text{(entry)} \\ \ell_d &:= \lambda u \,.\begin{cases} 1 & \text{if } u \in \{i, o\} \\ 0 & \text{otherwise} \end{cases} && \text{(lower bounds)} \\ \beta_d &:= \lambda u \,.\begin{cases} 1 & \text{if } u \in \{i, o\} \\ \infty & \text{if } u \in \{s, t\} \\ \beta_\pi & \text{otherwise} \end{cases} && \text{(upper bounds)} \end{aligned}$$

Again, the construction is easy to understand by example.

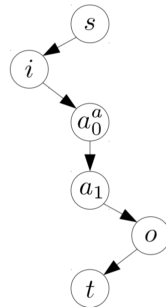


Figure 5.35: Example network $\mathring{n}_d$ for Figure 5.33 for entry a and exit f

Example *Figure 5.35 illustrates the network for the CFG of Figure 5.33 for the entry/exit pair* (a, f) *and respective direct path state* $(2, (a_0), a)$*. As before, additional lower and upper flow bounds further constrain paths.*

The networks we just constructed are a path-compressed representation of control flow annotations and their relations. An implication of the separation of network flows from control flow paths is that complexities of the subsequent network flow problems do not scale with the CFG size but rather with the number of annotations. In addition, this allows for scalability since flow bounds can be approximated which in turn increases efficiency of analysis accordingly.

Computing Feasible Unrolls

We will first be concerned with the computation of feasible unrolls to introduce the general approach. Recall the classical Ford-Fulkerson method for MAXFLOW [100, 103] (cf. Section 5.1.1 on page 81). The strategy is to search (flow) increasing paths from source to sink successively and to push the maximal admissible flow along such a path until no more paths can be found. In our case, we already computed all possible paths prior to network construction. We now show that we do not need to explicitly generate the flow networks as devised above.

Recall from Section 5.1.1, that $f_{in}^{\Sigma}(u)$ and $f_{out}^{\Sigma}(u)$ denote net flow of a node u. Formally, the problem of unroll feasibility for either network $\mathring{N}_{iok}$ or $\mathring{N}_d$ is defined as:

$$\begin{aligned} \max \quad & \sum_{u \in V} f_{out}^{\Sigma}(u) \\ \text{s.t.} \quad & \forall u \in V: f_{in}^{\Sigma}(u) - f_{out}^{\Sigma}(u) = b_u \\ & b_u = \begin{cases} q & \text{if } u = s \\ -q & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\ & \forall u \in V: \ell(u) \leq f(u) \leq \beta(u) \end{aligned} \tag{5.67}$$

Lemma 5.58 *An unroll is feasible if and only if Equation 5.67 has a feasible solution.*

Proof. Due to capacity constraints, it must hold that $f_{out}^{\Sigma}(i) = f_{out}^{\Sigma}(o) = 1$. □

Lemma 5.59 *Let $\mathring{n}_{iok}(s_i, s_o)$ denote a network for entry state s_i and exit state s_o. Let $a_0^{o_\pi(s_i)}$ denote the default initial annotation as defined in Equation 5.58 and let $a_0^{o_\pi(s_o)}$ denote the default initial annotation as defined in Equation 5.59. Then, by construction, it holds that:*

$$f_{out}^{\Sigma}(i) > 0 \Leftrightarrow f_{out}^{\Sigma}(a_0^{o_\pi(s_i)}) > 0 \tag{5.68}$$

$$f_{out}^{\Sigma}(o) > 0 \Leftrightarrow f_{out}^{\Sigma}(a_0^{o_\pi(s_o)}) > 0 \tag{5.69}$$

Analogously, this holds for $\mathring{n}_d$.

Since network nodes s, t do not constrain flow otherwise, we can simply ignore nodes s, t, i, o on increasing paths as long as we can "simulate" satisfaction of lower capacity

bounds of nodes i, o. In other words, explicit reduction to networks is not necessary beyond formal problem definition.

We modify the traditional Ford-Fulkerson method accordingly to correctly solve MAXFLOW directly on path states. Let $f : V \mapsto \mathbb{N}_0^\infty$ denote net flow (out of a node). As usual, for a path π, given (upper) node capacity constraints c, the maximal admissible flow along π is denoted by:

$$\text{test}_f := \lambda(c, f, \pi) \,.\, \min\{c(u) - f(u) \mid u \in \pi\} \tag{5.70}$$

Pushing additional admissible flow n over such a path is defined as:

$$\text{set}_f := \lambda(f, n, \pi) \,.\, f[u \to f(u) + n \mid u \in \pi] \tag{5.71}$$

We define a function $f^1_\top$, which pushes *unit* admissible flow over a single path $\sigma_\pi(s)$, given path state s and a tuple (r, f, c) where $r \in \{\top, \bot\}$, where f denotes existing net flow and c denotes capacity constraints, and which also returns such a tuple.

$$f^1_\top := \lambda s \,.\, \lambda(r, f, c) \,.\, \begin{cases} (\top, \text{set}_f(f, 1, \sigma_\pi(s)), c) & \text{if } r \neq \bot \wedge \text{test}_f(c, f, \sigma_\pi(s)) > 0 \\ (\bot, \emptyset) & \text{otherwise} \end{cases} \tag{5.72}$$

The intuition is to chain $f^1_\top$ to successively test for feasibility of paths.

Theorem 5.60 (Feasible Unroll) *Let $\mathring{s} \in \mathring{V}$ be a scope, let $u \in \text{entry}(\mathring{s})$ and $v \in \text{exit}(\mathring{s})$. Let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then an unroll from u to v is feasible if and only if:*

$$\begin{aligned} &\forall s_i \in S^I_{wcet}(\mathring{s}, u) \colon \forall s_o \in S^O_{wcet}(\mathring{s}, v) \colon \exists f^{io}_\top = f^1_\top(s_o) \circ f^1_\top(s_i) \colon f^{io}_\top(\top, f_0, \beta_\pi) = (\top, f, \beta_\pi) \\ &\quad \vee \forall s_d \in S^D_{wcet}(\mathring{s}, u, v) \colon \exists f^d_\top = f^1_\top(s_d) \colon f^d_\top(\top, f_0, \beta_\pi) = (\top, f, \beta_\pi) \end{aligned} \tag{5.73}$$

Proof. Each invocation of $f^1_\top$ sends unit flow along an iteration denoted by the respective state if and only if $r = \top$ and the admissible flow is greater 1. Hence, it only returns $\top$ if a previous iteration (if any) and the current iteration is feasible. By Definition 5.57, the unroll is feasible if and only if all successive applications of $f^1_\top$ are feasible. □

Note that kernel states are not relevant to unroll feasibility. Consequently, we did not test for unboundedness either.

Example *We reconsider Figure 5.33 and test whether there exists a feasible unroll from entry c to exit f given the following flow bounds:*

$$\beta_\pi = \{a^c_0 \to \infty, a^a_0 \to \infty, a_1 \to 1, a_2 \to 1, a_3 \to 1\}$$

Both paths are represented by exactly one state, respectively:

$$s_i = (3, (a_0^c, a_2, a_3), c)$$
$$s_o = (2, (a_0^a, a_1), a)$$

Then the unroll feasibility test yields:

$$\begin{aligned}
&(f_\top^1(s_o) \circ f_\top^1(s_i))(\top, f_0, \beta_\pi) \\
&= f_\top^1(s_o)(f_\top^1(s_i)(\top, f_0, \beta_\pi)) \\
&= f_\top^1(s_o)((\top, \mathrm{set}_f(f, 1, \sigma_\pi(s_i)), \beta_\pi)) && (r \neq \bot \wedge \mathrm{test}_f(c, f_0, \sigma_\pi(s_i)) > 0) \\
&= f_\top^1(s_o)((\top, \mathrm{set}_f(f, 1, (a_0^c, a_2, a_3)), \beta_\pi)) && (\top \neq \bot \wedge \mathrm{test}_f(c, f_0, (a_0^c, a_2, a_3)) > 0) \\
&= f_\top^1(s_o)((\top, \{a_0^c \to 1, a_2 \to 1, a_3 \to 1\}, \beta_\pi)) && (\top \neq \bot \wedge 1 > 0) \\
&= (\top, \mathrm{set}_f(f, 1, \sigma_\pi(s_o)), \beta_\pi) && (r \neq \bot \wedge \mathrm{test}_f(c, f, \sigma_\pi(s_o)) > 0) \\
&= (\top, \mathrm{set}_f(\{a_0^c \to 1, a_2 \to 1, a_3 \to 1\}, 1, (a_0^a, a_1)), \beta_\pi) && (\top \neq \bot \wedge \mathrm{test}_f(c, \{\dots\}, a_0^a, a_1) > 0) \\
&= (\top, \{a_0^c \to 1, a_2 \to 1, a_3 \to 1, a_0^a \to 1, a_1 \to 1\}, \beta_\pi) && (\top \neq \bot \wedge 1 > 0) \\
&= (\top, f, \beta_\pi)
\end{aligned}$$

Note that there exists no direct path from c *to* f*.*

Computing Maximal Unrolls

We will now address the problem of computing feasible unrolls of maximal length. As opposed to testing for mere feasibility, we have to take kernels and node weights into account. Consequently, unrolls can be unbounded. Note that entry and exit paths are compulsory but kernels are only optional for feasibility.

We first formally define the problem for flow networks, then we show that explicit network instantiation is not necessary in this case either. We assume node weights $\omega\colon V \mapsto \mathbb{N}_0$ given, such that:

$$\omega(u) = \begin{cases} 0 & \text{if } u \in \{s, t, i, o\} \\ n & \text{otherwise} \end{cases} \tag{5.74}$$

where $n \in \mathbb{N}_0$ denotes a unknown node weight (instead of individual node weights, total path lengths are known, which will be sufficient). The problem of computing an unroll

of maximal length is then defined as:

$$\begin{aligned}
\max \quad & \sum_{u \in V} f^{\Sigma}_{out}(u)\omega(u) && (5.75)\\
\text{s.t.} \quad & \forall u \in V : f^{\Sigma}_{in}(u) - f^{\Sigma}_{out}(u) = b_u \\
& b_u = \begin{cases} q & \text{if } u = s \\ -q & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\
& \forall u \in V : \ell(u) \leq f(u) \leq \beta(u)
\end{aligned}$$

Lemma 5.61 *An unroll is feasible if and only if Equation 5.75 has a feasible solution. It is bounded if and only if the objective value is finite.*

Proof. See Lemma 5.58 and Definition 5.57. □

Obviously, Lemma 5.59 continues to hold for mere feasibility. In addition, weights ω have to be taken into account:

Lemma 5.62 *For a scope $\mathring{s}$, let $\mathring{n}_{iok}(s_i, s_o)$ denote a network for entry state s_i and exit state s_o, which by definition includes kernels $S^K_{wcet}(\mathring{s})$. Let $\omega_\pi(\pi) = \sum_{u \in \pi} \omega(u)$. It holds that:*

$$\begin{aligned}
& \omega_\pi((s, i) \cdot \sigma_\pi(s_i) \cdot (t)) = \delta_\pi(s_i) && \text{(entry)} && (5.76)\\
& \omega_\pi((s) \cdot \sigma_\pi(s_o) \cdot (o, t)) = \delta_\pi(s_o) && \text{(exit)} && (5.77)\\
& \forall s_k \in S^K_{wcet}(\mathring{s}) : \omega_\pi((s) \cdot \sigma_\pi(s_k) \cdot (t)) = \delta_\pi(s_k) && \text{(kernels)} && (5.78)
\end{aligned}$$

Analogously, this holds true for $\mathring{n}_d$.

Proof. By definition of weights in Equation 5.74 and the accumulation of weights in path states by Definition 5.56. □

Consequently, we can compute unroll lengths by just signatures and the corresponding path length. In addition to satisfying lower capacity constraints for entry and exit states, the total path length must be maximized. We achieve this by considering entry and exit paths first, as before to satisfy implicit flow demand, then consider kernels in descending order of their lengths in order to maximize the sum of path lengths.

We keep test_f (Equation 5.70) and set_f (Equation 5.71) unchanged but define a function f^1_ω which pushes unit admissible flow over a single path $\sigma_\pi(s)$ while accumulating path length, given path state s and a tuple (ω, f, c), where ω denotes path length or infeasibility, f denotes existing flow and c denotes flow capacities, and which also returns

such a tuple as:

$$f_\omega^1 := \lambda s \,.\, \lambda(\omega, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \delta_\pi(s), \\ \text{set}_f(f, 1, \sigma_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \wedge \text{test}_f(c, f, \sigma_\pi(s)) > 0 \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.79}$$

In addition, we define a function f_ω^k, which is declared like f_ω^1 but is defined to push *maximally* admissible flow over a single path $\sigma_\pi(s)$ while accumulating path lengths scaled by the flow as:

$$f_\omega^k := \lambda s \,.\, \lambda(\omega, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \delta_\pi(s) \times \text{test}_f(c, f, \sigma_\pi(s)), \\ \text{set}_f(f, \text{test}_f(c, f, \sigma_\pi(s)), \sigma_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.80}$$

We compose f_ω^1 and f_ω^k to evaluate unrolls. Let $\sigma_{\delta_\pi} \colon \wp(S_\pi) \mapsto S_\pi^*$ order a set of path states $s_\pi \in S_\pi$ by descending length $\delta_\pi(s_\pi)$, such that for a scope $\mathring{s}$

$$f_\omega^K := \lambda \mathring{s} \,.\, [\![\sigma_{\delta_\pi}(S_{wcet}^K(\mathring{s}))]\!](f_\omega^k) \tag{5.81}$$

denotes an ordered composition of $f_\omega^k(s_k)$ for kernel states $s_k \in \sigma_{\delta_\pi}(S_{wcet}^K(\mathring{s}))$. Further, for scope $\mathring{s}$, entry u and exit v, let

$$F_\omega^{iok} := \lambda(\mathring{s}, u, v) \,.\, \left\{ f_\omega^K(\mathring{s}) \circ f_\omega^1(s_o) \circ f_\omega^1(s_i) \;\middle|\; s_i \in S_{wcet}^I(\mathring{s}, u), s_o \in S_{wcet}^O(\mathring{s}, v) \right\} \tag{5.82}$$

denote the set of all possible evaluations for unrolls and let

$$F_\omega^d := \lambda(\mathring{s}, u, v) \,.\, \left\{ f_\omega^1(s_d) \;\middle|\; s_d \in S_{wcet}^D(\mathring{s}, u, v) \right\} \tag{5.83}$$

denote the set of evaluations for direct paths.

Theorem 5.63 (Local Maximal Unroll) *Let $\mathring{s} \in \mathring{V}$ be a scope, let $u \in \text{entry}(\mathring{s})$ and $v \in \text{exit}(\mathring{s})$. Let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then we define maximal unroll length for a scope $\mathring{s}$, entry node $u \in V$ and exit node $u \in V$ as:*

$$\begin{gathered} \max_\mu \colon \mathring{V} \times V^2 \mapsto \mathbb{N}_0^{\infty,\bot} \\ \max_\mu(\mathring{s}, u, v) = \max \left\{ \omega \;\middle|\; \begin{array}{l} (\omega, f, \beta_\pi) = f_\omega(0, f_0, \beta_\pi), \\ f_\omega \in F_\omega^{iok}(\mathring{s}, u, v) \cup F_\omega^d(\mathring{s}, u, v) \end{array} \right\} \end{gathered} \tag{5.84}$$

Proof. Equation 5.82 captures the semantics of pushing unit flow over an entry and an exit path first, which satisfies flow demand — if possible — and then pushing all remaining admissible flow over all kernels in descending order of length. Since flow represents iteration repetitions, repeating longest paths first as often as possible maximizes the

accumulated path length. Equation 5.83 captures the semantics of pushing unit flow over a direct path. By Lemma 5.59 and Lemma 5.62, reduction to MAXLEN by the Ford-Fulkerson method [101] on flow networks is direct. Equation 5.84 then denotes the maximal solution for all combinations of entry, exit and direct paths. □

Example *We reconsider Figure 5.33 and maximize the unroll length from entry c to exit f given flow bounds:*

$$\beta_\pi = \{a_0^c \to \infty, a_0^a \to \infty, a_1 \to 1, a_2 \to 2, a_3 \to 2\}$$

and path states:

$$\begin{aligned} s_i &= (3, (a_0^c, a_2, a_3), c) \\ s_o &= (2, (a_0^a, a_1), a) \\ s_K &= ((5, (a_0^a, a_1, a_2, a_3), a), (5, (a_0^a, a_1, a_3), a)) \end{aligned}$$

Then composition of semantics by Equation 5.82 and Equation 5.83 yields:

$$\begin{aligned} F_\omega^{iok}(\mathring{s}, u, v) \cup F_\omega^{d}(\mathring{s}, u, v) &= F_\omega^{iok}(\mathring{s}, u, v) \cup \emptyset \\ &= \{f_\omega^K(\mathring{s}) \circ f_\omega^1(s_o) \circ f_\omega^1(s_i)\} \\ &= \{f_\omega^k(s_k^0) \circ f_\omega^k(s_k^1) \circ f_\omega^1(s_o) \circ f_\omega^1(s_i)\} \\ &= \left\{ \begin{array}{c} f_\omega^k((5, (a_0^a, a_1, a_2, a_3), a)) \circ f_\omega^k((5, (a_0^a, a_1, a_3), a)) \\ \circ f_\omega^1((2, (a_0^a, a_1), a)) \circ f_\omega^1((3, (a_0^c, a_2, a_3), c)) \end{array} \right\} \end{aligned}$$

Evaluation of $f_\omega \in F_\omega^{iok}(\mathring{s}, u, v)$, according to Equation 5.84, then yields:

$$\begin{aligned} & f_\omega(0, f_0, \beta_\pi) \\ &= (f_\omega^k(s_k^0) \circ f_\omega^k(s_k^1) \circ f_\omega^1(s_o) \circ f_\omega^1((3, (a_0^c, a_2, a_3), c)))(0, f_0, \beta_\pi) \\ &= (f_\omega^k(s_k^0) \circ f_\omega^k(s_k^1) \circ f_\omega^1((2, (a_0^a, a_1), a)))(3, \{a_0^c \to 1, a_2 \to 1, a_3 \to 1\}, \beta_\pi) \\ &= (f_\omega^k(s_k^0) \circ f_\omega^k((5, (a_0^a, a_1, a_3), a)))(3 + 2, \{a_0^c \to 1, a_2 \to 1, a_3 \to 1, a_0^a \to 1, a_1 \to 1\}, \beta_\pi) \\ &= (f_\omega^k((5, (a_0^a, a_1, a_2, a_3), a)))(5 + 5 \times 1, \{a_0^c \to 1, a_2 \to 1, a_3 \to 2, a_0^a \to 1, a_1 \to 2\}, \beta_\pi) \\ &= (10 + 5 \times 0, \{a_0^c \to 1, a_2 \to 1, a_3 \to 2, a_0^a \to 1, a_1 \to 2\}, \beta_\pi) \\ &= (10, \{\dots\}, \beta_\pi) \end{aligned}$$

Avoiding Redundant Computations

F_ω^{iok} (Equation 5.82) and F_ω^1 (Equation 5.83) can be optimized if scope tops and bottoms coincide with entries and exits. In these cases, kernels subsume entry, exit and direct paths, which is often the case in practice, and which significantly reduces computational redundancy. In the following, semantics remain unchanged. This is a purely technical modification.

To correctly test for feasibility in case of subsumption, we extend functions f_ω^1 (Equation 5.79) and f_ω^k (Equation 5.80) by a symbolic identity. Let $T \subseteq \{i, o, k, d\}$ denote symbolic identifiers. Then the *named* extension of f_ω^1 is defined as:

$$f_\omega^{T,1} := \lambda(s, T) \,.\, \lambda(\omega, f, c, T') \,. \begin{cases} \begin{pmatrix} \omega + \delta_\pi(s), \\ \text{set}_f(f, 1, \sigma_\pi(s)), \\ c, \\ T \cup T' \end{pmatrix} & \text{if } \omega \neq \bot \wedge \text{test}_f(c, f, \sigma_\pi(s)) > 0 \\ (\bot, \emptyset, c, T') & \text{otherwise} \end{cases} \tag{5.85}$$

Analogously, we define the named extension of f_ω^k as:

$$f_\omega^{T,k} := \lambda(s, T) \,.\, \lambda(\omega, f, c, T') \,. \begin{cases} \begin{pmatrix} \omega + \delta_\pi(s) \times \text{test}_f(c, f, \sigma_\pi(s)), \\ \text{set}_f(f, \text{test}_f(c, f, \sigma_\pi(s)), \sigma_\pi(s)), \\ c, \\ T \cup T' \end{pmatrix} & \text{if } \omega \neq \bot \\ (\bot, \emptyset, c, T') & \text{otherwise} \end{cases} \tag{5.86}$$

Evaluation of kernels in descending order (cf. Equation 5.81) is then defined as:

$$f_\omega^{T,K} := \lambda(\mathring{s}, T) \,.\, [\![\sigma_{\delta_\pi}(S_{wcet}^K(\mathring{s}))]\!](f_\omega^{T,k}(s_k, T)) \tag{5.87}$$

In the following let $t \in \text{top}(\mathring{s})$ and $b \in \text{bottom}(\mathring{s})$ of scope $\mathring{s}$. Then a more efficient composition of path evaluations is defined by:

$$F_\omega^{iok} := \lambda(\mathring{s}, u, v). \begin{cases} \left\{ f_\omega^{T,K}(\mathring{s}, \{i, o, k\}) \right\} & \text{if } t = u \wedge b = v \wedge |\text{bottom}(\mathring{s})| = 1 \\ \left\{ f_\omega^{T,K}(\mathring{s}, \{i, k\}) \circ f_\omega^1(s_o, \{o\}) \;\middle|\; s_o \in S_{wcet}^O(\mathring{s}, v) \right\} & \text{if } t = u \wedge (b \neq v \vee |\text{bottom}(\mathring{s})| > 1) \\ \left\{ f_\omega^{T,K}(\mathring{s}, \{o, k\}) \circ f_\omega^1(s_i, \{i\}) \;\middle|\; s_i \in S_{wcet}^I(\mathring{s}, u) \right\} & \text{if } t \neq u \wedge b = v \wedge |\text{bottom}(\mathring{s})| = 1 \\ \left\{ \begin{matrix} f_\omega^{T,K}(\mathring{s}, \{k\}) \circ f_\omega^1(s_o, \{o\}) \\ \circ f_\omega^1(s_i, \{i\}) \end{matrix} \;\middle|\; \begin{matrix} s_i \in S_{wcet}^I(\mathring{s}, u), \\ s_o \in S_{wcet}^O(\mathring{s}, v) \end{matrix} \right\} & \text{otherwise} \end{cases} \tag{5.88}$$

Giving flow evaluations identities allows to determine which types of paths have been covered by the various tests in case of subsumption. Optimization is defined analogously for direct paths: If entry and exit coincide with top and bottom, then the longest feasible kernel is already the longest direct path. Hence, in conjunction with Equation 5.88, an

optimized version of F^d_ω (Equation 5.83) is defined as:

$$F^d_\omega := \lambda(\mathring{s}, u, v) \; . \begin{cases} \emptyset & \text{if } t = u \wedge b = v \wedge |\text{bottom}(\mathring{s})| = 1 \\ \{f^1_\omega(s_d, \{d\}) \mid s_d \in S^D_{wcet}(\mathring{s}, u, v)\} & \text{otherwise} \end{cases} \tag{5.89}$$

Then we define the maximal unroll length for a scope $\mathring{s}$, entry node $u \in V$ and exit node $u \in V$ as:

$$\max{}^{\text{opt}}_\mu \colon \mathring{V} \times V^2 \mapsto \mathbb{N}_0^{\infty,\perp}$$

$$\max{}^{\text{opt}}_\mu(\mathring{s}, u, v) = \max \left\{ \omega \;\middle|\; \begin{array}{l} (\omega, f, \beta_\pi, T) = f_\omega(0, f_0, \beta_\pi, \emptyset), \\ \{i, o\} \subseteq T \vee \{d\} \subseteq T, \\ f_\omega \in f^{iok}_\omega(\mathring{s}, u, v) \cup f^d_\omega(\mathring{s}, u, v) \end{array} \right\} \tag{5.90}$$

A solution is only feasible if tests for entry and exit paths succeed, irrespective of the concrete subsumption scenario.

Example *We modify the previous example to maximize the unroll length from entry a to exit g for the scope illustrated in Figure 5.33. Since entry and exit paths coincide with kernels it holds that:*

$$\begin{aligned} & F^{iok}_\omega(\mathring{s}, u, v) \cup F^d_\omega(\mathring{s}, u, v) \\ & \quad = \{f^K_\omega(\mathring{s}, \{i, o, k\})\} \cup \emptyset \\ & \quad = \{f^k_\omega(\{i, o, k\})((5, (a^a_0, a_1, a_2, a_3), a)) \circ f^k_\omega(\{i, o, k\})((5, (a^a_0, a_1, a_3), a))\} \end{aligned}$$

Since there only exists a single kernel, evaluation of $f_\omega \in F^{iok}_\omega(\mathring{s}, u, v) \cup F^d_\omega(\mathring{s}, u, v)$ yields:

$$\begin{aligned} & f_\omega(0, f_0, \beta_\pi) \\ & \quad = f^k_\omega(\{i, o, k\})(s^0_k) \circ f^k_\omega(\{i, o, k\})(5, (a^a_0, a_1, a_3), a)(0, f_0, \beta_\pi) \\ & \quad = f^k_\omega(\{i, o, k\})(s^0_k)(5, \{a^a_0 \to 1, a_1 \to 1, a_3 \to 1\}, \beta_\pi, (\{i, o, k\})) \\ & \quad = f^k_\omega(\{i, o, k\})(5, (a^a_0, a_1, a_2, a_3), a)(5, \{a^a_0 \to 1, a_1 \to 1, a_3 \to 1\}, \beta_\pi, (\{i, o, k\})) \\ & \quad = (5 + 5 \times 0, \{a^a_0 \to 1, a_1 \to 1, a_3 \to 1\}, \beta_\pi, (\{i, o, k\})) \\ & \quad = (5, \{\dots\}, \beta_\pi, (\{i, o, k\})) \end{aligned}$$

Maximal Unroll Length to All Scope Members

We computed unrolls of maximal length from dedicated entries to dedicated exits. We are now concerned with further optimization to minimize recomputation of unrolls to multiple exits. Even more so, we generalize and show how to minimize computational overhead for unrolls to *all* member nodes in a scope. Besides eliminating unnecessary tests by means of F^{iok}_ω of Equation 5.88, this is another corner-stone optimization to keep redundancy low.

Recall that $\max_\mu$ as defined in Equation 5.84 evaluates to the maximal unroll length from s to any node in $\mathring{s}$, which, as a matter of fact, does not necessarily have to be an exit.

Definition 5.64 (Annotation Anchor) *A node $u \in V$ such that $\alpha_\pi(u) \neq \epsilon$ is referred to as* (annotation) anchor*.*

Definition 5.65 (Most Recent Anchor) *Let $s_\pi \in S_\pi$ be a path state. Then for signature $\sigma_\pi(s_\pi) = (a_0, \ldots, a_k)$, node $\alpha_\pi^{-1}(a_k) \in V$ (cf. Table 5.3 on page 132) denotes the* most recent anchor*.*

For a scope $\mathring{s}$, an entry $u \in \text{entry}(\mathring{s})$ and a node $v \in V$, we define the set of most recent anchors as:

$$\text{mra}\colon V^2 \mapsto \wp(V)$$
$$\text{mra}(u, v) = \{\alpha_\pi^{-1}(a_k) \mid \sigma_\pi(s_\pi) = (a_0, \ldots a_k), s_\pi \in S_{wcet}(u, v, a_0^u)\} \tag{5.91}$$

By definition of S_{wcet}, set $\text{mra}(u, v)$ is never empty.

Example *In Figure 5.33a it holds that* $\text{mra}(a, e) = \{a, c\}$ *and* $\text{mra}(a, g) = \{g\}$*.*

We now show that to obtain maximal unroll lengths to a node, it is sufficient to compute unroll lengths to their most recent anchors. For two path states s_π and s'_π such that $s_\pi \overset{A_\pi}{\sim} s'_\pi$ (cf. Equation 5.49), let the difference in path length be defined as:

$$d_{\delta_\pi}(s_\pi, s'_\pi) := \delta_\pi(s_\pi) - \delta_\pi(s'_\pi) \tag{5.92}$$

Recall that we refer to path states such that $\overset{A_\pi}{\sim}$ holds true, as being comparable (by length).

Lemma 5.66 *Given scope $\mathring{s}$, let $s \in \text{entry}(\mathring{s})$. For a node $v \in V$, let $u \in \text{mra}(s, v)$. Then for every path state s_π in u there exists a comparable path state s'_π in v:*

$$\forall s_\pi \in S_{wcet}(s, u, a_0^s)\colon \exists s'_\pi \in S_{wcet}(s, v, a_0^s)\colon s_\pi \overset{A_\pi}{\sim} s'_\pi \tag{5.93}$$

Proof. We consider states from node s only and u is the most recent anchor on a path from s to v. Hence, $\sigma_\pi(s_\pi) = \sigma_\pi(s'_\pi) \wedge o_\pi(s_\pi) = o_\pi(s'_\pi) \Leftrightarrow s_\pi \overset{A_\pi}{\sim} s'_\pi$. □

Lemma 5.67 *Let nodes be given as in Lemma 5.66. All length differences d between all states s_π in a most recent anchor u and a matching state s'_π in node v are equal:*

$$\forall s_\pi \in S_{wcet}(s, u, a_0^s)\colon \forall s'_\pi \in S_{wcet}(s, v, a_0^s)\colon s_\pi \overset{A_\pi}{\sim} s'_\pi \Rightarrow d_{\delta_\pi}(s_\pi, s'_\pi) = d \tag{5.94}$$

Proof. Every pair s_π, s'_π of comparable path states must lie on the same path $(s, \ldots, u, \ldots, v)$ and all pairs share the same suffix $|(u, \ldots, v)| = d$, which denotes the longest path from u to v. □

Consequently, to compute the distance from a most recent anchor to a node, it is sufficient to derive the difference from a single pair of comparable states.

Theorem 5.68 (Sparse Unrolling) *Given a scope $\mathring{s}$, an entry $s \in \mathrm{entry}(\mathring{s})$ and a node $v \in V$ such that $\max_\mu(\mathring{s}, s, v)$ denotes its maximal unroll length. We define a helper function that returns a pair of matching path states due to Lemma 5.67:*

$$m := \lambda u \,.\, \lambda v \,.\, (s_\pi, s'_\pi) \in S_{wcet}(s, u, a_0^s) \times S_{wcet}(s, v, a_0^s) \wedge s_\pi^u \overset{A_\pi}{\sim} s_\pi^v \tag{5.95}$$

Then the maximal unroll distance to node v is alternatively defined as:

$$\max_\mu^{\mathrm{sparse}} : \mathring{V} \times V^2 \mapsto \mathbb{N}_0^{\infty,\perp}$$
$$\max_\mu^{\mathrm{sparse}}(\mathring{s}, s, v) = \max \left\{ \max_\mu(\mathring{s}, s, u) + d_{\delta_\pi}(s_\pi^v, s_\pi^u) \;\middle|\; \begin{array}{l} u \in \mathrm{mra}(s, v), \\ (s_\pi^u, s_\pi^v) = \mathrm{m}(u, v) \end{array} \right\} \tag{5.96}$$

The maximal unroll distance to node v equals the maximum of maximal unroll distances to its most recent anchors u_i and the greatest iteration distance from each u_i to v.

Example *Reconsider Figure 5.33. We compute $\max_\mu^{\mathrm{sparse}}(\mathring{s}, a, f)$, given bounds as in the previous two examples. It holds that $\mathrm{mra}(a, f) = \{a\}$ and*

$$\begin{aligned} \mathrm{m}(a, f) &= (s_a, s_f) \in S_{wcet}(a, a, a_0^a) \times S_{wcet}(a, f, a_0^a) \\ &= \{((1, (a_0^a), a), (2, (a_0^a, a_1), a)\} \end{aligned}$$

Consequently, we compute:

$$\begin{aligned} \max_\mu^{\mathrm{sparse}}(\mathring{s}, a, f) &= \max_\mu(\mathring{s}, a, a) + d_{\delta_\pi}(s_f, s_a) \\ &= (f_\omega^k(s_k^0) \circ f_\omega^k(s_k^1) \circ f_\omega^1(s_o))(0, f_0, \beta_\pi) + \delta_\pi(s_f) - \delta_\pi(s_a) \\ &= 6 + \delta_\pi(s_f) - \delta_\pi(s_a) \\ &= 6 + 2 - 1 = 7 \end{aligned}$$

where s_k^0, s_k^1, s_o denote the two kernels and the exit path from node a to itself.

Given that we only unroll from scope entry to scope exits, the strategy we just proposed is only an optimization if exit nodes outnumber anchors. Otherwise, unrolling is performed unnecessarily. However, below we propose computations that rely on unrolling to all interior nodes of a scope. In that case, savings are significant.

Practical Path State Computation

In practice, we can further optimize the computation of unrolls. The model we have given so far is ultimately based on fixed point computation per node (cf. Equation 5.57). As usual, if nodes are processed in topological order, this fixed point is reached in linear time as no information is "fed back" via back edges. Moreover, a single pass in such order is sufficient to collect all possible path states originating from all entries collectively.

Another aspect is the encoding of signatures. In the given model, signatures denote paths through anchors. We recognize that all path states originating in the same entry share common prefixes of their signature. So instead of maintaining those separately, a compact representation is achieved by means of prefix trees (or tries) [8, 170] which significantly reduce the amount of information to be propagated[7].

A simple but rarely applicable optimization is the recognition of zero-valued flow bounds. Since we already know single bound valuation upon path state computation, encountering such a bound denotes invariably infeasible paths. Hence, no information has to be propagated through such points, further reducing the state set.

A concrete example where these techniques likely also have a significant impact is the derivation of symbolic representations which typically [137, 150] suffer from unnecessarily redundant representation.

5.3.3.3 Computing WCET Bounds Globally

We now extend the framework to compute a global bound on the WCET instead of just for a single scope. Intuitively, we continue to compute path states within a single scope only. Consequently, all path lengths are relative to a specific entry. As opposed to before, we now take maximal unroll lengths of subscopes that are "crossed" by iterations of the current scope into account. Recall that by Definition 5.28 on page 106, every far entry (exit) is also an entry to (exit from) its enclosing scope(s).

We assume a function

$$\max_{\mu}^{\mathring{s}} \colon \mathring{V} \times V^2 \mapsto \mathbb{N}_0^{\infty,\perp} \tag{5.97}$$

given such that $\max_{\mu}^{\mathring{s}}(\mathring{s}, u, v)$ denotes maximal unroll length for a scope $\mathring{s}$ from entry u to exit v *including all subscopes*, as opposed to $\max_{\mu}$ (Equation 5.84). Then we define a transformer $\mathrm{tf}_{\mathring{s}}$, which wraps tf_{wcet} (Equation 5.56) such that:

$$\mathrm{tf}_{wcet}^{\mathring{s}} \colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{wcet} \mapsto \mathbb{D}_{wcet})$$

$$\mathrm{tf}_{wcet}^{\mathring{s}}(\mathring{s}, u, v) = \lambda S\,.\begin{cases} \mathrm{id} & \text{if } v \in \mathrm{entry}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \left\{ \begin{pmatrix} \delta_\pi(s_\pi) + l, \\ \sigma_\pi(s_\pi), \\ o_\pi(s_\pi) \end{pmatrix} \middle| \begin{array}{l} s_\pi \in S, \\ l = \max_{\mu}^{\mathring{s}}(\mathring{t}, u, v) \end{array} \right\} & \text{if } v \in \mathrm{exit}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}_{wcet}(u)(S) & \text{otherwise} \end{cases} \tag{5.98}$$

For a node u and its predecessor v, if v is a scope entry then we neither advance path lengths nor extend signatures since the node is subject to the subscope and therefore length and annotations are already accounted for in the unroll of the subscope. If v is a

[7] In the reference algorithm we propose below, we do just that. See [8] for a graphical example.

scope exit then we extend path lengths by the unroll length of the subscope for entry u and exit v. Otherwise, tf_{wcet} applies.

To correctly account for subscope unroll lengths $\max_{\mu}^{\mathring{s}}$ in Equation 5.98, provisions have to be taken. As before, let $\mathrm{pred}^{\rightarrow}$ denote predecessors on forward edges E^F. Then predecessors "leaping" over subscopes are defined as:

$$\mathrm{pred}_{\mathring{s}}^{\rightarrow} \colon \mathring{V} \mapsto V \mapsto \wp(V)$$

$$\mathrm{pred}_{\mathring{s}}^{\rightarrow} = \lambda \mathring{s} \,.\, \lambda v \,.\begin{cases} \{u | u \in \mathrm{entry}(\mathring{t})\} & \text{if } v \in \mathrm{exit}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{pred}^{\rightarrow}(v) & \text{otherwise} \end{cases} \tag{5.99}$$

Consequently, we define path states in a program point, similar to Equation 5.57 as:

$$S_{wcet}^{\mathring{s}} \colon \mathring{V} \times V^2 \times A_\pi \mapsto S_\pi$$

$$S_{wcet}^{\mathring{s}}(\mathring{s}, s, v, a) = \begin{cases} \mathrm{tf}_{wcet}^{\mathring{s}}(\mathring{s}, s, s)(\{(0, a, s)\}) & \text{if } v = s \\ \bigsqcup \{\mathrm{tf}_{wcet}^{\mathring{s}}(\mathring{s}, u, v)(S_{wcet}^{\mathring{s}}(\mathring{s}, s, u, a)) \mid u \in \mathrm{pred}_{\mathring{s}}^{\rightarrow}(\mathring{s})(v)\} & \text{otherwise} \end{cases} \tag{5.100}$$

which computes a fixed point of longest paths in a program point, taking unrolls of maximal length of subscopes into account. Hence, $\max_\pi S_{wcet}^{\mathring{s}}(\mathring{s}, u, v, \epsilon)$ denotes the greatest possible path length in scope $\mathring{s}$ from entry u to exit v.

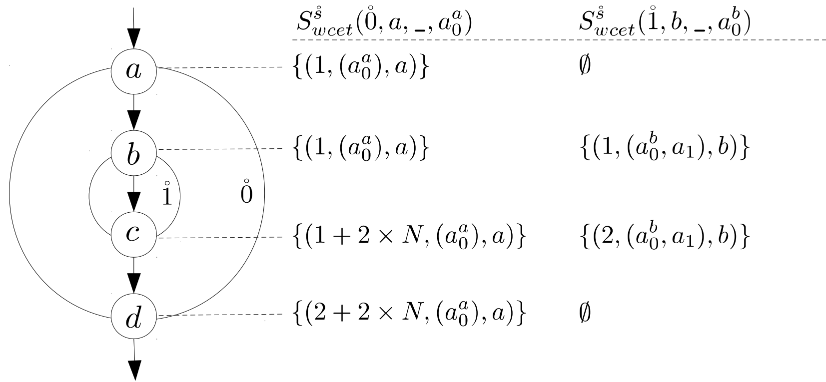


Figure 5.36: Example of global WCET computation

Example *Figure 5.36 illustrates a simple example of global WCET bound computation. We assume annotations such that* $\alpha_\pi(b) = (a_1)$ *with flow bound* $\beta_\pi(a_1) = N$. *The table to the right denotes respective path states, where* $S_{wcet}^{\mathring{s}}(\mathring{0}, a, _, a_0^a)$ *denotes states in scope* $\mathring{0}$, *from root* a *to some interior node,* $S_{wcet}^{\mathring{s}}(\mathring{1}, b, _, a_0^b)$ *denotes states in scope* $\mathring{1}$, *from root* b. *In node* b, $S_{wcet}^{\mathring{s}}(\mathring{0}, a, b, a_0^a)$ *denotes an iteration of* $\mathring{0}$ *up to but not including* b, *whereas* $S_{wcet}^{\mathring{s}}(\mathring{1}, b, b, a_0^b)$ *denotes an iteration in* $\mathring{1}$ *that includes* b *only. Consequently, in node* c, *all iterations in* $\mathring{1}$ *are known by* $S_{wcet}^{\mathring{s}}(\mathring{1}, b, c, a_0^b)$. $S_{wcet}^{\mathring{s}}(\mathring{0}, a, c, a_0^a)$ *then denotes a possible iteration in* $\mathring{0}$, *including the maximal unroll length of subscope* $\mathring{1}$.

Analogously to the definitions for just a single scope without subscopes (cf. Section 5.3.3.2), we now *redefine* partitions of path states such that:

$$S^I_{wcet}(\mathring{s}, s) = \bigcup_{u \in \mathrm{bottom}(\mathring{s})} S^{\mathring{s}}_{wcet}(\mathring{s}, s, u, a^s_0) \qquad \text{(cf. Eq. 5.58)} \tag{5.101}$$

$$S^O_{wcet}(\mathring{s}, t) = \bigcup_{u \in \mathrm{top}(\mathring{s})} S^{\mathring{s}}_{wcet}(\mathring{s}, u, t, a^u_0) \qquad \text{(cf. Eq. 5.59)} \tag{5.102}$$

$$S^K_{wcet}(\mathring{s}) = \bigcup_{t \in \mathrm{top}(\mathring{s})} \bigcup_{b \in \mathrm{bottom}(\mathring{s})} S^{\mathring{s}}_{wcet}(\mathring{s}, t, b, a^t_0) \qquad \text{(cf. Eq. 5.60)} \tag{5.103}$$

$$S^D_{wcet}(\mathring{s}, s, t) = S^{\mathring{s}}_{wcet}(\mathring{s}, s, t, a^s_0) \qquad \text{(cf. Eq. 5.61)} \tag{5.104}$$

Feasibility checks for a single iteration f^1_ω (Equation 5.79 on page 143) and for kernels f^k_ω (Equation 5.80 on page 143) remain unchanged.

It remains to define $\max^{\mathring{s}}_\mu$ to compute feasible maximal unroll lengths, just as in the previous section.

Theorem 5.69 (Maximal Unroll) *Given the redefined set of path states $S^{\mathring{s}}_{wcet}$, which now takes subscopes into account, Theorem 5.63 applies unchanged with all definitions such that*

$$\max^{\mathring{s}}_\mu \colon \mathring{V} \times V^2 \mapsto \mathbb{N}^{\infty,\perp}_0 \tag{5.105}$$

denotes the longest unroll length for a scope $\mathring{s}$ from entry u to exit v, analogously to $\max_\mu$ *(Equation 5.84 on page 143). This represents the precise worst-case path length for a scope and a pair of source and sink nodes, under the given flow bound model.*

Proof. Theorem 5.63 on page 143 applies unchanged with $S^{\mathring{s}}_{wcet}$ for S_{wcet}. □

Note that for a CFG $G = (V, E, s, t)$, the initial annotation a^s_0 for the global entry must be defined such that flow bound $\beta_\pi(a^s_0) = 1$ to avoid non-direct path composition in the (acyclic) root scope.

5.3.3.4 Computing WCET Bounds on Subgraphs

We briefly discuss how to change the problem specification from the computation of longest paths from the global CFG entry to the global CFG exit to the computation of globally longest paths to and from all interior nodes. First, we extend the framework to enable arbitrary sink nodes, then we extend it to enable source nodes.

Path Length to Arbitrary Interior Points

Intuitively, since we can compute longest paths within a scope $\mathring{s}$ from its entry to its exit, then nothing prevents us from computing longest paths from an entry of $\mathring{s}$ to the *entries* of its subscopes. Previously, all path lengths within a scope have been relative to their respective entries. We now compute *maximal offsets* to these entries. In the context of

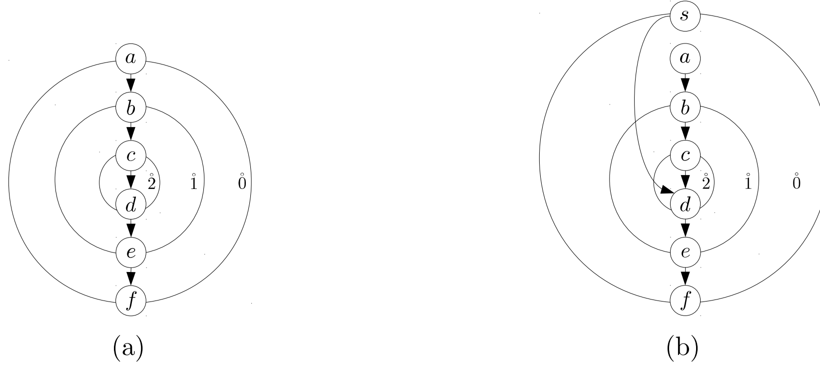


Figure 5.37: Example graphs to illustrate subgraph techniques

timing analysis, we compute an upper bound on the latest possible time a scope can be "executed" as an offset to the longest possible duration to "execute" a scope up to a specific program point. The intuition has previously been stated in [150].

We define a function $\max^{\star}_{\mu}(\mathring{s}, s, \mathring{t}, t)$, which computes the worst-case path length from source scope $\mathring{s}$ and node s to a target scope $\mathring{t}$ and node t, by computing the maximal unroll length to all entries of $\mathring{t}$ and the maximal unroll length within $\mathring{t}$ to t as:

$$\max^{\star}_{\mu} \colon \mathring{S} \times V \times \mathring{S} \times V \mapsto \mathbb{N}_0^{\infty,\perp}$$

$$\max^{\star}_{\mu}(\mathring{s}, s, \mathring{t}, u) =$$
$$\begin{cases} \max \left\{ \max^{\star}_{\mu}(\mathring{s}, s, \mathring{u}, i) + \max^{\mathring{s}}_{\mu}(\mathring{t}, i, u) \;\middle|\; \begin{array}{l} i \in \text{entry}(\mathring{s}), \\ \mathring{u} = \text{par}(\mathring{t}) \end{array} \right\} & \text{if } \mathring{s} \neq \mathring{t} \\ \max^{\mathring{s}}_{\mu}(\mathring{s}, s, u) & \text{otherwise} \end{cases} \tag{5.106}$$

Proof. $\max^{\star}_{\mu}(\mathring{t}, i, u)$ denotes the longest feasible unroll within scope $\mathring{t}$ from (entry) node i to (exit) node u. Simple induction yields longest unrolls from parent scope entry to current scope entry for all enclosing scopes and entry to terminal node u in scope $\mathring{t}$. □

Example *Consider Figure 5.37a. We assume annotations such that $\alpha_\pi(a) = (a_1)$ with flow bound $\beta_\pi(a_1) = 1$, $\alpha_\pi(b) = (a_2)$ with $\beta_\pi(a_2) = N$ and $\alpha_\pi(c) = (a_3)$ with $\beta_\pi(a_3) = M$. Then evaluation of Equation 5.106 yields:*

$$\begin{aligned} \max^{\star}_{\mu}(\mathring{0}, a, \mathring{2}, d) &= \max \left\{ \max^{\star}_{\mu}(\mathring{0}, a, \mathring{1}, c) + \max^{\mathring{s}}_{\mu}(\mathring{2}, c, d) \right\} \\ &= \max^{\star}_{\mu}(\mathring{0}, a, \mathring{1}, b) + 2 \times M \\ &= \max \left\{ \max^{\star}_{\mu}(\mathring{0}, a, \mathring{0}, b) + \max^{\mathring{s}}_{\mu}(\mathring{1}, b, c) \right\} + 2 \times M \\ &= \max^{\star}_{\mu}(\mathring{0}, a, \mathring{0}, b) + ((1 + 2 \times M + 1) \times (N - 1) + 1) + 2 \times M \\ &= 1 + ((1 + 2 \times M + 1) \times (N - 1) + 1) + 2 \times M \end{aligned}$$

Note that for the computation of path lengths to all nodes, sparse unrolling (cf. Theorem 5.68 on page 148) is specifically effective.

Path Lengths from Arbitrary Interior Points

We recognize that source nodes need not necessarily have to be scope entries as per Definition 5.28. Consequently, we can take provisions to designate arbitrary interior nodes as global entries. Recall that we already know how to compute longest paths from and to arbitrary nodes within the very same scope. Symmetrically to showing how to compute longest paths to any node above, we now sketch the idea of computing longest paths from any node.

Figure 5.37b illustrates the graphical construction by example, where we designate node d to be the global start node. Instead of the global entry a to the root scope $\mathring{0}$, an artificial entry s with node weight $\omega(s) = 0$ to scope $\mathring{0}$ and a far entry edge from s to the designated global entry node d is added. In addition, node a is ignored as an entry. Node d is by definition an entry to its enclosing scopes $\mathring{2}, \mathring{1}$. Consequently, $\max^{\star}_{\mu}(\mathring{0}, s, \mathring{0}, u)$ (Equation 5.106) denotes the globally longest path effectively starting in node d, terminating in some node u. By construction, there now exists only one feasible entry path from s to all subscopes imposing a zero-valued offset. In practice, the construction is easily achieved "virtually", without graph mutation, by adjusting Equation 5.100 accordingly to "simulate" the desired behavior.

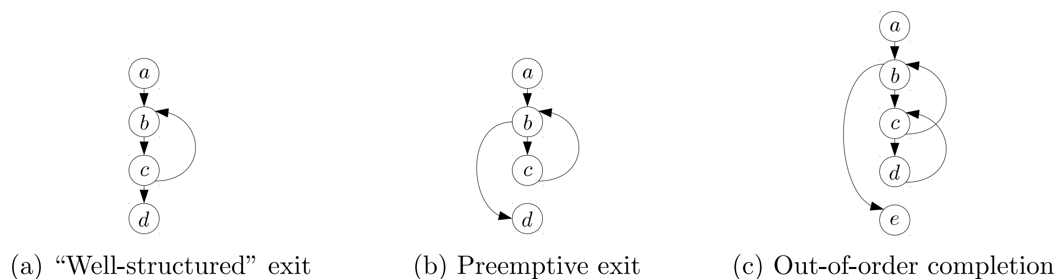
5.3.3.5 Practical Global Path Length Computation

Even though we discussed some optimizations already, the formal definitions as given above are impractical due to their recursive nature and repetitive recomputation of identical subproblems. We will now show how the problem of computing maximal path lengths to all reachable nodes as terminals can be achieved in just two passes over the CFG.

To this end, we first introduce a specialization of global topological order on a CFG to enable efficient non-recursive computations, then we propose a carefully crafted reference implementation.

Scope Order and Scope Completion

Just as topological order is optimal for fixed point computation on the DAGs that form the bodies of isolated scopes, we seek to compute a globally feasible topological order such that dependencies are satisfied optimally globally. In terms of unrolling, the maximal



(a) "Well-structured" exit (b) Preemptive exit (c) Out-of-order completion

Figure 5.38: Example CFGs to illustrate different topological orders

distance from a scope entry to its exit can only be computed once all iterations are completely known. Only then iterations of enclosing scopes can be completed. Figure 5.38 illustrates different scenarios where this is an issue. Figure 5.38a shows the ideal case: Scope bottom and exit coincide. Hence, all iterations of the inner scope are known prior to computing the maximal unroll distance from entry to exit and dependencies form iterations of the enclosing scope are satisfied implicitly. In Figure 5.38b a feasible topological order is to place node d prior to node c. Consequently, information for unrolling is still incomplete. In Figure 5.38c we assume nodes c, d to denote an innermost scope. Here, not only does node e depend on unrolling from entry b to exit b, but the scope's kernel depends on the unroll of the bottommost scope from entry c to exit c.

The usual practice is a post order traversal of the loop nesting representation (scope tree). However, this has the drawback that a recursive dependence remains and information unnecessarily has to be memoized. In addition, information propagation is only unidirectional (upwards the scope tree), preventing propagation of information into scopes prior to unrolling.

Definition 5.70 (Scope Completion) *A scope is completely known (or* complete*) if all iterations are completely known.*

This is obviously the case once all nodes in a scope and all of its subscopes have been processed. In Figure 5.38c the scope entered from node b is not complete even when its bottom is visited.

Definition 5.71 (Scope Order) Scope order $\mathring{\sigma}\colon V \mapsto \mathbb{N}_0$ *is a topological order such that all scope bottoms in all subscopes are visited prior to completing a scope.*

For a scope $\mathring{s}$, let the set of all descending scopes including $\mathring{s}$ be defined as:

$$\mathring{D}(\mathring{s}) := \{\mathring{s}\} \cup \mathrm{dsc}^+(\mathring{s}) \tag{5.107}$$

Then we define a set of "virtual bottoms" as:

$$\mathrm{vbot}(\mathring{s}) := \{b \in \mathrm{bottom}(\mathring{u})\colon \mathring{u} \in \mathring{D}(\mathring{s}) \wedge \forall \mathring{v} \in \mathring{D}(\mathring{s})\colon \nexists b' \in \mathrm{bottom}(\mathring{v})\colon b' \overset{\rightarrow}{\rightsquigarrow} b\} \tag{5.108}$$

The set contains the structurally "bottommost" scope bottoms in a scope nesting.

Example *Reconsider Figure 5.38c. Assume scopes such that* $\mathring{2} = \mathring{\gamma}(c) = \mathring{\gamma}(d)$, $\mathring{1} = \mathring{\gamma}(b)$ *and* $\mathring{1} = \mathrm{par}(\mathring{2})$*. Then* $\mathrm{vbot}(\mathring{1}) = \mathrm{vbot}(\mathring{2}) = \{d\}$.

Intuitively, we collect bottoms in case of imperfect nesting such that subscope nodes are structurally not fully enclosed. Reaching virtual bottoms in a topological order — which necessarily depends on structure — guarantees that all conceptually nested scopes are completely known.

The next issue is that of "preemptive" exits such as edge (b, d) in Figure 5.38c. We have to guarantee that all virtual bottoms are indeed processed prior to any "external"

dependencies from enclosing scopes. We can achieve the desired result by adding "virtual" dependence edges from virtual bottoms to exit edge destinations, consequently conceptually constraining the set of structurally possible topological orders.

Algorithm 5.6

```
1   let vedge G = do                                          ◁ G = (V, E)
2       for (u, v) ∈ E: (u, v) is exit edge from s̊ do
3           E ← E ∪ {(b, v) | b ∈ vbot(s̊) ∧ ¬(v ⇝ b)}
4   return (V, E)
```

Algorithm 5.6 sketches the idea. For all exit edges (u, v) of a scope $\mathring{s}$, add a virtual exit edge from its virtual bottoms b to node v. We prohibit virtual exit edges to form cycles. In that case, they are useless anyway since the target node by definition already must have been processed in some topological order. Recall that virtual exits only compensate for non-perfect nesting. Otherwise, it only ensures an additional dependency from a bottom to an external iteration.

Example *In Figure 5.38b a "virtual" edge* (c, d) *is inserted, and in Figure 5.38c a "virtual" edge* (d, e) *is inserted.*

In practice, instead of actually introducing edges to the CFG, we can "simulate" virtual edges by minor modification of Algorithm B.14 on page 258 to obtain a suitable topological order. The implementation is straight-forward and we therefore skip its detailed discussion at this point.

To summarize, counting the number of virtual bottoms while processing nodes in a topological order is a means to know when a scope is complete. The topological order must be a scope order to prevent preemptive exits. In combination, we can thus "simulate" perfectly nested scopes in an appropriately designed framework without additional computational overhead.

A Reference Algorithm For Efficient Path Analysis

In the following we document the principal outline of an efficient algorithm for worst-case path analysis for reference. To summarize, we process nodes in scope order in two passes. In a first pass, iterations are computed and every time a scope completes, its maximal unroll distances are computed. Instead of computing states from one entry at a time, we compute all states simultaneously. Since scope entries and exits denote transitions between scopes, we model entering a scope by backing up states from the parent scope (pending states) and replace them by a representative. Once scopes complete, all unrolls are computed and pending states are swapped back and updated to model leaving the scope. This leaves us with longest paths in the root scope. In a second pass, scope offsets are computed by the same strategy to compute absolute path lengths to individual nodes. In the following, we only outline the algorithms leaving special cases and error handling aside for simplicity.

Algorithm 5.7 Outline of non-recursive path analysis (pass 1)

```
 1  state: V ↦ ℘(S_π)                                        ◁ Path states
 2  pstate: V̊ ↦ V ↦ ℘(S_π)                                   ◁ Pending path states
 3  estate: V̊ ↦ V ↦ ℘(S_π)                                   ◁ Exit path states
 4  nbottom: V̊ ↦ ℕ_0                                         ◁ "Virtual bottom" count
 5  eoffset: V̊ ↦ V ↦ ℕ_0                                     ◁ Scope entry offset
 6  wcet: V ↦ ℕ_0^∞                                          ◁ Maximal path length
 7
 8  let pass1 u =                                            ◁ Designated start node
 9      nbottom ← {s̊ → |vbot(s̊)| | s̊ ∈ V̊}                    ◁ Initialize
10      u ← "top of outermost loop scope of u or u"          ◁ Effective start node
11      for v ∈ "nodes in scope order from u" do
12          state[v] ← tf_wcet(v)(prologue v)                ◁ Join/Transfer
13          while v ∈ vbot(s̊) do                             ◁ Reached "virtual bottom"
14              nbottom[s̊] ← nbottom[s̊] − 1
15              if nbottom[s̊] = 0 then                       ◁ Scope completely known
16                  epilogue s̊
17              s̊ ← par s̊                                    ◁ Parent scope
```

Algorithm 5.7 defines auxiliary arrays and the first pass. In lines 1-6, *state* holds path states in a node, *pstate* holds pending states for scope entries, *estate* holds path states in exit nodes that will be replaced by pending states upon scope completion, *nbottom* counts the number of processed virtual scope bottoms to denote completion, *eoffset* denotes scope entry offsets for the second pass and *wcet* denotes absolute path distances to individual nodes.

Function *pass1* (line 8) computes for a given designated start node the longest path to the global CFG exit. For initialization, counters of virtual bottoms *nbottom* are set up (lines 9) and an effective start node is determined (line 10). This node is the top node of the outermost scope enclosing node u if is contained in a loop, or u otherwise. We effectively start computation from here to be able to compute unrolls for all such enclosing scopes. Then we process nodes in scope order (line 11) from the effective start node. Processing generally decays into three phases: In the prologue (discussed below) path states are propagated and generally provisions are taken for conceptually entering scopes such that updating path states (line 12) is restricted to simply applying transformer tf_{wcet}. In the epilogue, scopes are conceptually left by performing unrolling and restoring original path states. In line 13, if a virtual bottom is reached, we decrease the respective counter (line 14). Once all virtual bottoms of a scope have been visited, we invoke the epilogue (line 16, details below). Since bottom nodes could be shared among scopes, all possible scopes are left at once (line 17).

Algorithm 5.8 outlines the prologue phase. Path states are initially joined as usual (line 2). Generally, we propagate all path states originating from all entries simultaneously. Therefore, at each entry, the set of path states can be partitioned (lines 3,4) into states originating in the current scope *In* and states from enclosing scopes *Ex*. If *Ex* is not empty or if the current node is an entry (line 5), then we conceptually enter a new scope.

Algorithm 5.8 Outline of non-recursive path analysis (prologue)

```
1  let prologue u =                                          ◁ Start iteration in node u
2      S ← ⊔{s ∈ state[v]: v ∈ pred→(u)}                     ◁ Propagation
3      In ← {s ∈ S: γ̊(o_π(s)) = γ̊(u)}                        ◁ Scope-local states
4      Ex ← S \ In                                           ◁ Scope-external states
5      if Ex ≠ ∅ ∨ u "is entry" then                         ◁ Actual or designated entry
6          if "u is designated entry" then
7              Ex ← Ex ∪ {(0, (a_0^s), s)}                   ◁ Initial state for global entry s
8          pstate[γ̊(u)][u] ← Ex                              ◁ Store "pending" states
9          In ← In ∪ {(0, (a_0^u), u)}                       ◁ Initial state for entry u
10     return In
```

If the current node is the designated global start point (which is also considered an entry) (line 6), then we create a path state that represents an iteration in the globally outermost scope. This effectively models a far entry from the global CFG entry (line 7). In either case, we back up states Ex (line 8) and denote them by a single representative path state modeling all parent iterations passing through this entry (line 9). Note that we do not address the issue of shared entry nodes yet. Instead, we only model an iteration of the innermost scope. The return value (line 10) now only denotes local path states.

Algorithm 5.9 Outline of non-recursive path analysis (epilogue)

```
1  let epilogue s̊ =                                                 ◁ Finish scope s̊
2      for o ∈ exit(s̊) do
3          R ← ∅
4          for i ∈ entry(s̊) do
5              R_io ← ∅                                             ◁ States to be resumed
6              l ← "unroll i to o"
7              for s ∈ pstate[s̊][i] do                              ◁ Pending states
8                  if γ̊(o_π(s)) ≠ par s̊ then                        ◁ Far entry state
9                      pstate[par s̊][i] ← pstate[par s̊][i] ∪ {s}    ◁ Export state
10                     R_io ← R_io ∪ {(l, (a_0^i), i)}              ◁ Initial state for entry i
11                 else
12                     R_io ← R_io ∪ {(δ_π(s) + l, σ_π(s), o_π(s))} ◁ Update length
13             R ← ⊔{s | s ∈ R_io ∪ R}                              ◁ Collect "resumables"
14             estate[s̊][o] ← state[o]                              ◁ Backup states
15             state[o] ← R                                         ◁ "Leave" scope s̊
```

Leaving scopes is performed in the epilogue, defined in Algorithm 5.9. Generally, maximal unrolls from all entries to one specific exit are computed, pending path states are restored and updated accordingly. Therefore, for each scope exit (line 2), we fill a set of "resumed" states R (line 3). The set R_{io} (line 5) denotes the set of path states for one specific pair of entry and exit. First, the maximal unroll length is computed (line 6), then pending states are restored one at a time (line 7). If such a state does not originate from an immediately enclosing scope (line 8), then this state denotes a path that also entered the immediately enclosing scope. Consequently, it is added to the respective set of pending states (line 9), and we create a path state to be restored, which originates in the immediately enclosing scope (line 10) along with the respective path

length. Otherwise (line 12), a pending state is simply restored by adjusting the path length. Pending states from all entries are collected (line 13). Before replacing states in the current exit node (which denote iterations in the scope to be left), we back up the original states (line 14). Replacing path states in this exit then conceptually leaves the current scope (line 15).

Algorithm 5.10 Outline of non-recursive path analysis (pass 2)

```
 1  let finish_scope s̊ =
 2      for o ∈ exit par s̊ do
 3          state[par s̊][o] ← estate[par s̊][o]                ◁ Restore original exit states
 4      for i ∈ entry s̊ do
 5          state[s̊][i] ← pstate[s̊][i]                    ◁ Restore original entry states s̊
 6          l ← 0
 7          for p ∈ entry par s̊ do                                         ◁ Parent entries
 8              l ← max(l, "unroll p to i" + eoffset[par s̊][p])
 9          eoffset[s̊][i] ← l
10          pstate[s̊][i] ← state[s̊][i]                                        ◁ Swap back
11
12  let finish_node u =
13      l ← 0
14      for i ∈ entry γ̊(u) do
15          l ← max(l, "unroll i to u" + eoffset[γ̊(u)][i])              ◁ Absolute distance
16      wcet[u] ← l
17
18  let pass2 u =
19      eoffset ← {s̊ → {u → 0 | u ∈ entry(s̊)} | s̊ ∈ V̊}
20      u ← "top of outermost loop scope of u or u"                ◁ Effective start node
21      for v ∈ "nodes in scope order from u" do               ◁ Compute absolute distances
22          Q̊ ← ϵ
23          while v ∈ top(γ̊(v)) do                                 ◁ All shared top nodes
24              Q̊ ← (s̊).Q̊                                         ◁ Inner to outer scope
25              s̊ ← par s̊
26          for s̊ ∈ Q̊ do                                                        ◁ In order
27              finish_scope s̊
28          finish_node v
```

After the first pass completes, the length of the longest path to the global CFG exit is already known. To compute absolute path distances to all nodes, a second pass is required, which is defined in Algorithm 5.10. Function *pass2* (line 18) first initializes the array of scope entry offsets *eoffset* (line 19). As before, we derive the effective start node from the designated start node, which is either the top of the topmost loop scope surrounding the designated start node or the node itself if it belongs to the root scope (line 20). The function then processes all nodes in scope order from the effective start node (line 21). If a scope top is encountered (line 23), $\mathring{\gamma}(u)$ denotes the innermost scope for a possibly shared top node. Since we propagate scope offsets top down the scope tree, we compute the sequence of scopes entered through this node in Q (line 24), from the topmost to bottommost scope (line 23-25). In this order, we first compute entry offsets in function *finish_scope* (line 27), then compute individual distances to nodes in function

finish_node (line 28).

In *finish_scope* (line 1), we successively compute values in array *eoffset* which denotes maximal path lengths to individual scope entry nodes. For a given scope $\mathring{s}$, for all scope exits of the parent scope, the original path states are restored (lines 2,3). This ensures we can unroll the parent scope as all path states originating in the parent are back in place. For each (current) scope entry (line 4), we now compute the maximal unroll from the parent entry to the current entry. Again, we need to swap states (line 5) such that path states in the entry of the current scope denote iterations of the parent scope. With correct states in place, an absolute maximal distance from all parent entries (line 7) to this entry is computed (line 8). We store this new offset (line 9) and restore original states in the entry (line 10) such that path states of the entries of scope $\mathring{s}$ are back in place.

After having computed all absolute entry distances, in *finish_node* (line 12), individual node distances are computed by computing a maximal length unroll from all entries (line 14) to a specific node (line 15). Array *wcet* then (line 16) contains the final distances.

Note that in the algorithm outline we repeatedly perform unrolling. As we learned earlier, it is sufficient to (expensively) unroll to a subset of nodes only to obtain unroll length to all nodes (cheaply). Hence, depending on the density of annotations, unrolling needs to be performed a lot less than the algorithm above suggests.

Note also that in the reference implementation, all path states are maintained for both passes. We recognize that the second pass can already be performed once a strongly connected component formed by respective loops is finished by the first pass (the outermost scope denotes an acyclic region and therefor only direct paths lead to the entries of SCC). Hence, by interleaving analysis passes, memory consumption can potentially significantly be reduced. Nevertheless, this likely has no noticeable effect on performance for realistic input sizes.

Remarks

The reference algorithm has been carefully designed to enable concurrent computation although this is not exploited here. Input data is immutable, global data is read-only and write operations to data structures are locally restricted. Potential for parallel computations are the computations of states on neighboring paths or scopes, unrolling for different entry and exit combinations or for different scopes. Also both passes can be interleaved by recognizing that once strongly-connected components have been completed in the first pass, the second pass can instantly be applied instead of performing two complete passes over the CFG. Even without concurrency, this would lower memory requirements even further as many states can be discarded early.

5.3.3.6 Evaluation

We evaluate the efficiency of the proposed reference implementation for various control flow scenarios. To this end, we evaluate the average (arithmetic mean) performance of

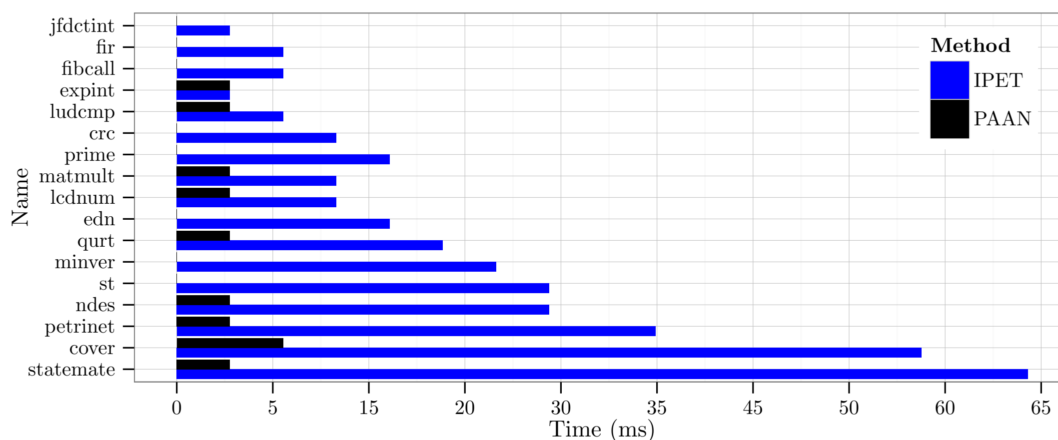


Figure 5.39: Runtimes on MRTC benchmark suite

our path analysis (called PAAN in the following) including all optimizations we proposed earlier. The aim is to demonstrate scalability characteristics for typical control flow graphs of varying sizes and topologies. We perform runtime measurements on the Mälardalen WCET benchmark suite (MRTC [97]) as well as on control flow graphs generated from random syntax trees (AST) at a sampling rate of 1 ms. The resulting CFGs range from 10 to approximately 60 000 nodes with 50 samples taken equally distributed. The ASTs are composed of four high-level language constructs IF, IFTHEN, WHILE and DOWHILE. Additional entries and exits to and from loops can be generated, as well as loop bounds and per-node WCET. Program semantics are not considered. Randomization provides the structural diversity and sizes required to provide a satisfying coverage, while the given benchmarks are comparably small and well-structured. We reuse the same framework as for the evaluation of loop detection (cf. Section 5.3.2.3 on page 107). Table 5.2 on page 118 lists all randomization parameters. Figure 5.23 illustrates loop distribution of the generator.

The experiments are carried out on a single core of an *Intel Xeon E5630* (2.53 GHz, 4 cores, 128 kB/1 024 kB/12 MB (L1/L2/L3) cache) CPU. We measure the accumulated CPU time of all phases of our path analysis including the control flow reconstruction by prenumbering (cf. Section 5.3.2.5). For IPET, the construction of the equation system is included. The resulting ILP is solved using *CPLEX* [171] (v12.4) with default arguments. WCET estimates per program point are obtained by *aiT* [172] for the *Tricore 1.3* architecture.

MRTC

Figure 5.39 shows the results for a subset of MRTC benchmarks. We compute the WCET to *all* nodes with PAAN, while IPET just solves the problem of computing WCET to the global exit node. In all cases, PAAN significantly outperforms IPET. In some cases, we solve the WCET problem from the source to all reachable nodes in less than 1 ms in some cases. These hard real-time benchmarks are comparably small in size (ca. 50

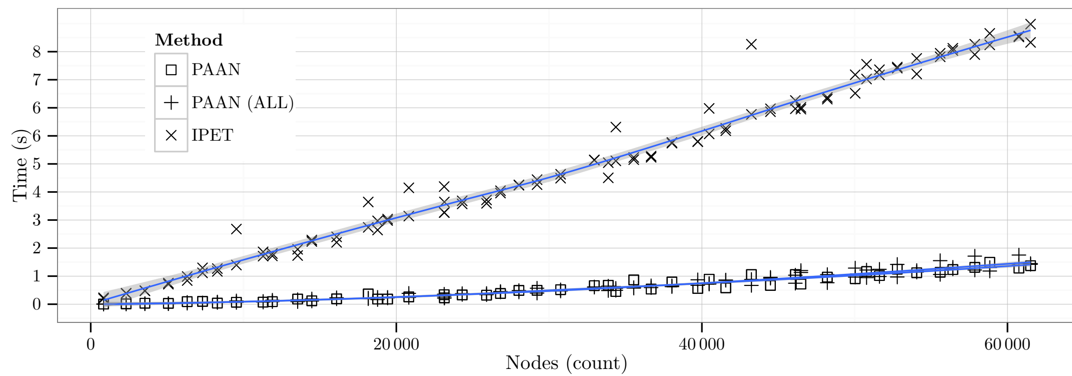


Figure 5.40: Runtime for non-degenerated CFG (DEPTH = 4, LOOP DEPTH = 3, P(IF) = 0.1, IFELSE = 0.2, P(WHILE) = 0.3, P(DOWHILE) = 0.4)

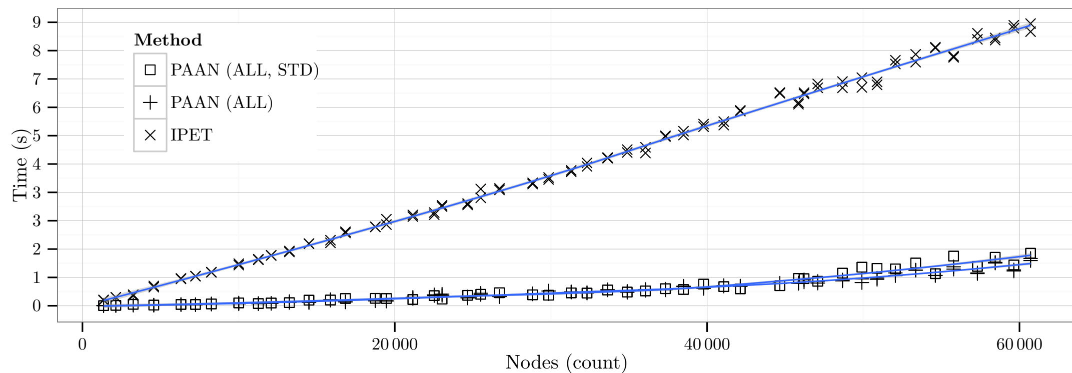


Figure 5.41: Runtime for acyclic CFG (DEPTH = 4, LOOP DEPTH = 3, P(IF) = 0.35, IFELSE = 0.65)

to 1 200 LOC). Thus, scalability for very large problem instances is not obvious. We therefore evaluate very large randomized graph instances to emphasize the differences on average.

Randomized Graphs

In Figure 5.40 we compare the computation times for inputs that are quite “typically” found in real-time benchmarks with distributions of constructs given in the title and an additional probability for flow bounds of 0.1 but a least a single flow bound per loop. PAAN denotes the time required to compute the per-task WCET, PAAN (ALL) denotes the time for the computation to all nodes and IPET denotes per-task WCET as before. As with small benchmarks, PAAN in both variations performs and scales significantly better than IPET. It is also notable that it shows significantly less variance in its time consumption and that the WCET computation to all nodes only has a marginal impact.

The time consumption for purely acyclic control flow is depicted in Figure 5.41. We now compare the average performance from Figure 5.40 for the computation of WCET to all nodes on non-degenerated graphs (PAAN (ALL, STD)) to the computation of WCET

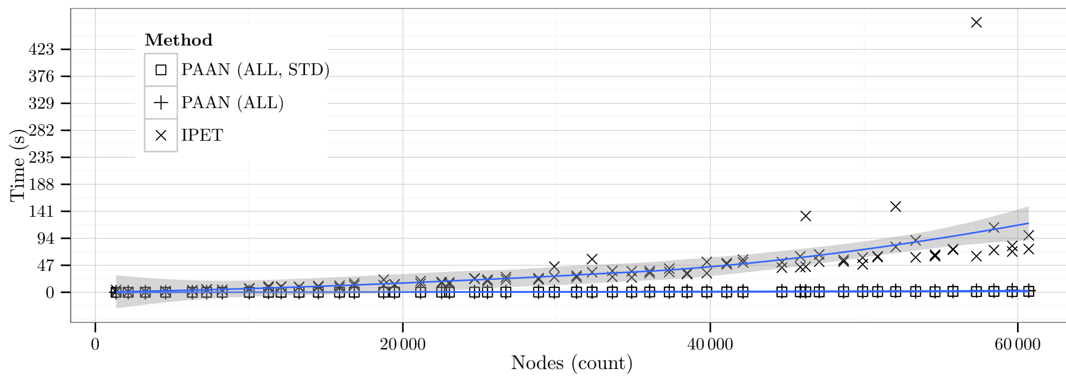


Figure 5.42: Runtime for high loop counts (DEPTH = 11, LOOP DEPTH = 10, P(IF) = 0.0, IFELSE = 0.0, P(WHILE) = 0.3, P(DOWHILE) = 0.7, P(EXIT) = 0.0, P(ENTRY) = 0.0)

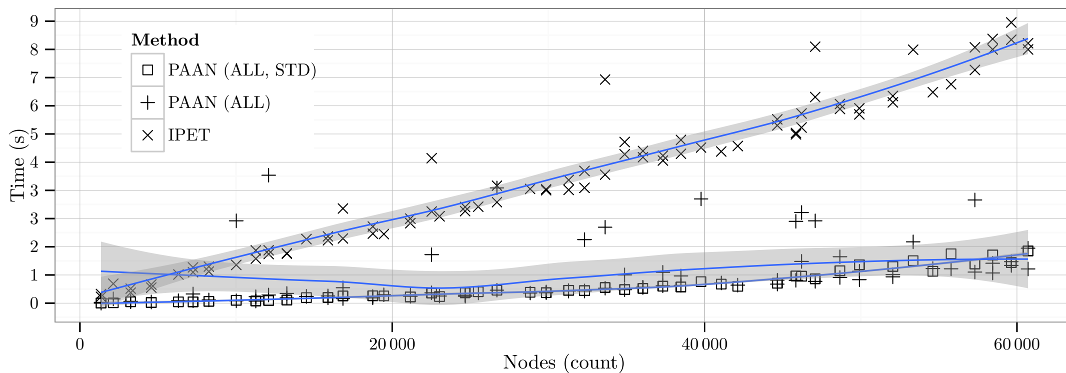


Figure 5.43: Runtime for high entry and exit counts (DEPTH = 4, LOOP DEPTH = 3, P(IF) = 0.1, IFELSE = 0.2, P(WHILE) = 0.3, P(DOWHILE) = 0.4, P(EXIT) = 1.0, P(ENTRY) = 1.0)

to all nodes for this particular graph type, which we denote by PAAN(ALL) and IPET, respectively. For acyclic graphs IPET shows a much smaller variance. The primary insight however is that the curve for PAAN (ALL) is practically identical to PAAN (ALL, STD) and shows that the complexity of PAAN only marginally depends on the number of loops.

Due to the unrolling, PAAN could potentially be sensitive to graphs that are excessive regarding their number of loops, entries to loops and loop bound specifications. Therefore, we compare the time consumption for such degenerated inputs. In the following the probabilities are identical to the standard case except for the features we add for investigation.

Figure 5.42 depicts the results for a parameters of P(WHILE) = 0.3, P(DOWHILE) = 0.7 and a maximal nesting depth of 10. In this case IPET shows to be less predictable. Again, the very high loop count has no practical effect on the scalability of PAAN. To summarize the figures so far, PAAN scales largely independently of the graph structure itself. It is now worthwhile to investigate extreme cases of irreducibility and flow bounds.

Figure 5.43 illustrates samples for cyclic graphs with loops having entries to and

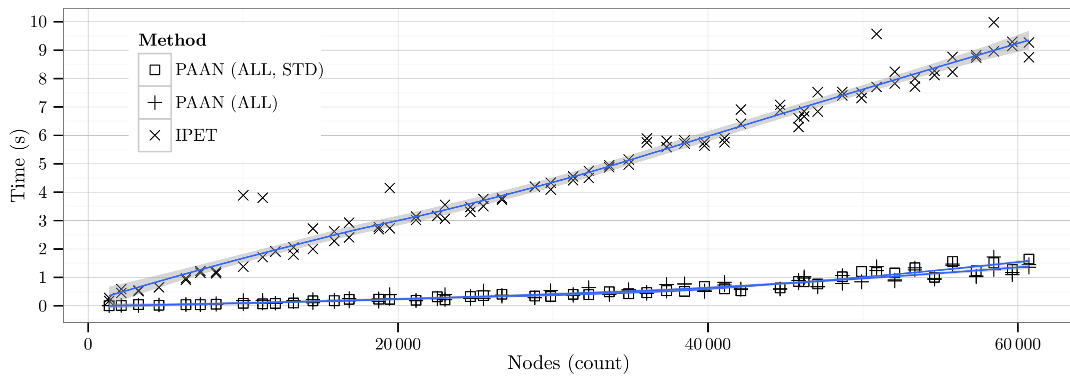


Figure 5.44: Runtime given all nodes bounded (DEPTH = 4, LOOP DEPTH = 3, P(IF) = 0.1, IFELSE = 0.2, P(WHILE) = 0.3, P(DOWHILE) = 0.4, P(BOUND) = 1.0)

exits from loops at practically all nodes (the exclusion of some nodes stems from the fact that nodes are restricted to an out-degree of 2 for technical reasons). We can now observe a significant impact for some of the inputs for PAAN as well as IPET, and a noticeable deviation from the reference (PAAN (ALL, STD)). The reason for the excessively large time consumption for PAAN in some cases is due to control flow reconstruction by prenumbering (cf. Section 5.3.2.5). Nonetheless, on average PAAN still shows very good scalability.

As opposed to that, Figure 5.44 shows results for control flow graphs where every single node is flow bounded. Since every flow bound potentially doubles the number of states, this is an approximation for the worst-case number of path states. Again, IPET is comparably slow and yields a high variance. Comparing PAAN(ALL) with its reference PAAN (ALL, STD) does not show a noticeable difference on average. The reason for this is that growth in the state space is always limited by scopes and for the root scope only the longest iteration is of relevance. Moreover, since all paths pass through the same bottom nodes, the flow bounds are quickly satisfied. Note that PAAN and IPET do not necessarily compute identical results here as in the previous cases since the semantics of both (randomized) flow fact models differ. IPET bounds are relative to specific reference points while bounds for the proposed path analysis are always relative to specific entries.

5.3.3.7 Conclusion

In this section we proposed a simple and efficient general framework for worst-case path length computation on general control flow graphs. The approach subsumes many existing explicit path analyses that are often constrained to specific subproblems. We provided a formal model supporting non-trivial flow bounds and investigated several ways to improve performance by avoiding unnecessary computations. We proposed an extension to the traditional problem of per-task WCET bound computation to compute bounds to all interior nodes. Along with the formal model, we proposed an efficient reference implementation. Evaluation suggests excellent scalability even in corner-cases.

5.3.4 Computing Best-case Execution Time Bounds

A trivial lower bound on the execution time is obviously 0. An improvement is the computation of just shortest paths, ignoring flow bounds. In this case we could simply compute SSSP (cf. Algorithm B.15 on page 259). However, this can be fairly imprecise as lower flow bounds are not taken into account otherwise. In the following we discuss the adaption of the previous framework from longest paths to shortest paths to compute BCET bounds with respect to lower flow bounds.

In Section 5.3.4.1 we discuss technical prerequisites. We then define the corresponding framework in Section 5.3.4.2, evaluate the proposal in Section 5.3.4.3 and conclude the section in Section 5.3.4.4. We assume acquaintance with the principles of WCET computation from Section 5.3.3.

5.3.4.1 Prerequisites

In the following we discuss the technical prerequisites for BCET analysis. Basic definitions from Section 5.3.3.1 on page 131 remain valid. The only semantic changes relate to the interpretation of node weights $\omega\colon V \mapsto \mathbb{N}_0$, which now denote lower bounds on the execution time of individual program points, and flow bounds $\beta_\pi\colon A_\pi \mapsto \mathbb{N}_0$, which now denote lower bounds in the minimum net flow to model minimal iteration counts. The underlying algebraic structure is the commutative semi-ring

$$(\mathbb{N}_0^{\infty,\perp}, \min, \perp, +, 0) \tag{5.109}$$

where $\mathbb{N}_0^{\infty,\perp} = \mathbb{N}_0 \cup \{\infty, \perp\}$ with min and neutral element $\perp$ for addition, and $+$ with neutral element 0 for multiplication such that:

$$\min(a, b) := \begin{cases} \min_{\mathbb{N}_0^\infty}(a, b) & \text{if } a \neq \perp \wedge b \neq \perp \\ a & \text{if } a \neq \perp \\ b & \text{otherwise} \end{cases} \tag{5.110}$$

$$a + b := \begin{cases} a +_{\mathbb{N}_0^\infty} b & \text{if } a \neq \perp \wedge b \neq \perp \\ \perp & \text{otherwise} \end{cases} \tag{5.111}$$

where $\min_{\mathbb{N}_0^\infty}$ denotes minimum and $+_{\mathbb{N}_0^\infty}$ denotes addition in $\mathbb{N}_0^\infty$. We lift operator min to path states of the *same* equivalence class and define:

$$\min_\pi\colon \wp(S_\pi) \mapsto S_\pi$$
$$\min_\pi(S) \in \{s \in S \mid \forall s' \in S\colon \delta_\pi(s) = \min(\delta_\pi(s), \delta_\pi(s'))\} \tag{5.112}$$

For an equivalence class $S = [s]$, $\min_\pi([s])$ denotes a path of minimal length. Otherwise, all definitions from Section 5.3.3.1 on page 131 continue to be valid.

5.3.4.2 Framework

In the following we derive the framework for BCET bounds from the previously defined framework for WCET bounds. First, we show how to compute iterations, then we show how unrolls are computed in general and the specific differences to WCET bounds for computing BCET bounds to all interior nodes.

Iterations

In the following we define how scope iterations for best-case path analysis are computed. The problem of computing shortest iterations has the structure of the semi-lattice:

$$(\mathbb{D}_{bcet}, \top, \sqsubseteq, \sqcup) \tag{5.113}$$

where $\mathbb{D}_{bcet} = \wp(S_\pi)$ is a set of path states, where $S_\pi \sqsubseteq S_\pi{}' \Leftrightarrow S_\pi \subseteq S_\pi{}'$ and where $S_\pi \sqcup S_\pi{}' := \{\min_\pi[s] \mid [s] \in (S_\pi \cup S_\pi{}')/\overset{A_\pi}{\sim}\}$ denotes the set of path states of different signature, origin and minimal length, according to the equivalence relation defined in Equation 5.49 on page 132. The corresponding transformer to compute iterations of minimal length is defined as:

$$\begin{aligned} \mathrm{tf}_{bcet} &\colon V \mapsto (\mathbb{D}_{bcet} \mapsto \mathbb{D}_{bcet}) \\ \mathrm{tf}_{bcet}(u) &= \lambda S\,.\,\{(\delta_\pi(s) + \omega(u), \sigma_\pi(s) \cdot \alpha_\pi(u), o_\pi(s)) \mid s \in S\} \end{aligned} \tag{5.114}$$

This is identical to the transformer for longest paths (in Equation 5.56 on page 134), except that weight ω denotes lower bounds on execution costs.

Unlike above, we directly lift the problem to the computation of iterations of minimal length including all subscopes. To this end, we assume the function

$$\min\nolimits_\mu^{\mathring{s}} \colon \mathring{V} \times V^2 \mapsto \mathbb{N}_0^{\infty,\perp} \tag{5.115}$$

to be given such that $\min_\mu^{\mathring{s}}(\mathring{s}, u, v)$ evaluates the shortest feasible unroll length in scope $\mathring{s}$ from node u to node v, and where ∞ denotes unboundedness and $\perp$ denotes infeasibility. Then we define a new transformer similar to Equation 5.98, which specifies the computation of iterations accordingly as:

$$\mathrm{tf}_{bcet}^{\mathring{s}} \colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{bcet} \mapsto \mathbb{D}_{bcet})$$

$$\mathrm{tf}_{bcet}^{\mathring{s}}(\mathring{s}, u, v) = \tag{5.116}$$
$$\lambda S\,.\begin{cases} \mathrm{id} & \text{if } v \in \mathrm{entry}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \left\{ \begin{pmatrix} \delta_\pi(s_\pi) + l, \\ \sigma_\pi(s_\pi), \\ o_\pi(s_\pi) \end{pmatrix} \,\middle|\, \begin{array}{l} s_\pi \in S, \\ l = \min_\mu^{\mathring{s}}(\mathring{t}, u, v) \end{array} \right\} & \text{if } v \in \mathrm{exit}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}_{bcet}(u)(S) & \text{otherwise} \end{cases}$$

The function either computes shortest iterations up to, but not including subscope entries, extends all iterations by unroll lengths of subscopes for subscope exits, or, otherwise, just updates path states according to Equation 5.114.

Assuming $\text{pred}_{\mathring{s}}^{\rightarrow}$ as defined in Equation 5.99 on page 150 to denote CFG predecessors such that subscopes are being "leaped over", we define path states in a program point by:

$$S_{bcet}^{\mathring{s}} \colon \mathring{V} \times V^2 \times A_\pi \mapsto S_\pi$$

$$S_{bcet}^{\mathring{s}}(\mathring{s}, s, v, a) = \begin{cases} \text{tf}_{bcet}^{\mathring{s}}(\mathring{s}, s, s)(\{(0, a, s)\}) & \text{if } v = s \\ \bigsqcup \left\{ \text{tf}_{bcet}^{\mathring{s}}(\mathring{s}, u, v)(S_{bcet}^{\mathring{s}}(\mathring{s}, s, u, a)) \mid u \in \text{pred}_{\mathring{s}}^{\rightarrow}(\mathring{s})(v) \right\} & \text{otherwise} \end{cases} \tag{5.117}$$

For a given scope $\mathring{s}$, start and terminal nodes u, v and an initial annotation a, $S_{bcet}^{\mathring{s}}(\mathring{s}, u, v, a)$ denotes the set of shortest iterations including unrolls of all subscopes.

As in the case of WCET, we define partitions of $S_{bcet}^{\mathring{s}}$ (cf. Section 5.3.3.2) to explicitly distinguish entry, exit, kernel and direct paths by the functions:

$$S_{bcet}^{I} \colon \mathring{V} \times V \mapsto \wp(S_\pi) \tag{5.118}$$

$$S_{bcet}^{O} \colon \mathring{V} \times V \mapsto \wp(S_\pi) \tag{5.119}$$

$$S_{bcet}^{K} \colon \mathring{V} \mapsto \wp(S_\pi) \tag{5.120}$$

$$S_{bcet}^{D} \colon \mathring{V} \times V^2 \mapsto \wp(S_\pi) \tag{5.121}$$

Unrolls

It remains to define $\min_{\mu}^{\mathring{s}}$ to compute respective unroll lengths. Previous definitions are straight-forward derivatives of the previous path analysis problem for worst-case path lengths. For unrolling, we have to take care not to compute shortest path unconditionally to avoid sacrificing precision by unnecessarily taking direct paths into account.

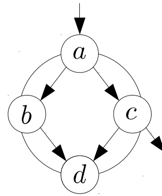


Figure 5.45: Example CFG to demonstrate BCET underestimation

Example *Figure 5.45 illustrates a scope with entry node a and exit node c. Assume a solitary lower flow bound greater* 1 *assigned to top node a. Then a direct path from a to c is feasible (and sound) but underestimates the intended unroll path length by not saturating the lower bound on admissible flow, which would allow for greater feasible path lengths by unrolling.*

The problem of computing shortest path unrolls for a single scope corresponds to MINCOST FLOW with node weights and bounds. Formally, we define this as:

$$\begin{aligned} \min \quad & \sum_{u \in V} f^{\Sigma}_{out}(u)\omega(u) && (5.122) \\ \text{s.t.} \quad & \forall u \in V \colon f^{\Sigma}_{in}(u) - f^{\Sigma}_{out}(u) = b_u \\ & b_u = \begin{cases} q & \text{if } u = s \\ -q & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\ & \forall u \in V \colon \ell(u) \leq f(u) \leq \beta(u) \end{aligned}$$

Reduction of path states to flow networks has been thoroughly discussed for WCET in Section 5.3.3.2 on page 133 and is not repeated here. As opposed to the WCET-case, the objective function is to be minimized. As before, ℓ denotes flow demand to guarantee existence of feasible entry and exit paths and β denotes an upper flow bound. Note that although β is an upper bound in a corresponding flow network, it is nonetheless a *lower* bound for the model of iteration repetitions in scope unrolls. Soundness remains a matter of correct annotation. Consequently, testing for iteration feasibility remains in principle unchanged.

Let f^1_ω (Equation 5.79) and f^k_ω (Equation 5.80) be defined as above. Let $\sigma^{-1}_{\delta_\pi} \colon \wp(S_\pi) \mapsto S^*_\pi$ order a set of path states $s_\pi \in S_\pi$ in *ascending* length $\delta_\pi(s_\pi)$. Then, for a scope $\mathring{s}$,

$$f^K_\omega := \lambda \mathring{s} \,.\, [\![\sigma^{-1}_{\delta_\pi}(S^K_{bcet}(\mathring{s}))]\!](f^k_\omega) \qquad (5.123)$$

denotes an ordered composition of f^k_ω. Further let

$$F^{iok}_\omega := \lambda(\mathring{s}, u, v) \,.\, \left\{ f^K_\omega(\mathring{s}) \circ f^1_\omega(s_o) \circ f^1_\omega(s_i) \;\middle|\; s_i \in S^I_{bcet}(\mathring{s}, u), s_o \in S^O_{bcet}(\mathring{s}, v) \right\} \qquad (5.124)$$

denote the set of all possible evaluations for non-direct unrolls from a node u to a node v, and let

$$F^d_\omega := \lambda(\mathring{s}, u, v) \,.\, \left\{ f^1_\omega(s_d) \;\middle|\; s_d \in S^D_{bcet}(\mathring{s}, u, v) \right\} \qquad (5.125)$$

denote the set of all possible evaluations for all direct paths from node u to node v. Let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then we denote the set of all unroll distances, partitioned by type, by:

$$R^{iok}_\mu := \lambda(\mathring{s}, u, v) \,.\, \left\{ \omega \;\middle|\; (\omega, f, \beta_\pi) = f_\omega(0, f_0, \beta_\pi), f_\omega \in F^{iok}_\omega(\mathring{s}, u, v) \right\} \qquad (5.126)$$

$$R^d_\mu := \lambda(\mathring{s}, u, v) \,.\, \left\{ \omega \;\middle|\; (\omega, f, \beta_\pi) = f_\omega(0, f_0, \beta_\pi), f_\omega \in F^d_\omega(\mathring{s}, u, v) \right\} \qquad (5.127)$$

Theorem 5.72 (Minimal Unroll) *Let $\mathring{s} \in \mathring{V}$ be a scope, let $u \in \text{entry}(\mathring{s})$ and $v \in \text{exit}(\mathring{s})$. Then shortest unroll length (of maximal flow) for a scope $\mathring{s} \in \mathring{V}$ and entry and*

exit nodes $u, v \in V$, respectively, is defined as:

$$\min_{\mu}^{\mathring{s}} : \mathring{V} \times V^2 \mapsto \mathbb{N}_0^{\infty,\perp}$$

$$\min_{\mu}^{\mathring{s}}(\mathring{s}, u, v) = \begin{cases} \min R_{\mu}^{iok}(\mathring{s}, u, v) & \text{if } \min R_{\mu}^{iok}(\mathring{s}, u, v) \neq \perp \\ \min R_{\mu}^{d}(\mathring{s}, u, v) & \text{otherwise} \end{cases} \tag{5.128}$$

This represents the precise best-case path length for a scope and a pair of source and sink nodes, under the given flow bound model.

Proof. Theorem 5.69 on page 151 continues to hold except for the fact that we are now maximizing repetition counts of shortest iterations and we minimize the total sum of lengths conditionally such that either the sum of kernels in descending order of lengths is feasible, and thus denotes a minimal length unroll, or at least a shortest direct path is feasible. □

BCET to Interior Points

Similarly to the worst-case problem (cf. Section 5.3.3.4), we are also interested in path lengths to all interior nodes of a given CFG. So far, $\min_{\mu}^{\mathring{s}}$ denotes the best-case path length from entry to exit of a given scope (including all subscopes). In particular, if applied to the root scope, it denotes a lower bound on the BCET of an entire task. Unfortunately, it is not as straight-forward as previously to lift the problem definition to computing best-case path lengths to arbitrary interior nodes. Care has to be taken not to overestimate the lower bound on path length "inadvertently". Recall semantics of flow bounds our case here. Informally, once a scope is "entered", flow bounds denote a minimum number of repetitions of individual program points before it can be "left" again. Inductively, we can thus compute a time bound when enclosing scopes are left. However, if an interior point is a terminal node of path computation, its enclosing scopes are never left and, hence, annotation semantics become unclear. We address this "under-specification" by guaranteed underestimation in these cases. Note that in the WCET-case, overestimation is implicit.

Recall that iteration lengths computed by $\text{tf}_{bcet}^{\mathring{s}}$ (Equation 5.116) include best-case path lengths of all subscopes. To compute absolute path lengths to interior points, we safely approximate lower bounds for the final computation (cf. Equation 5.106) in the current scope by searching for shortest paths instead of best-case paths.

To this end, we define an additional path evaluation such that

$$F_{\omega}^{io} := \lambda(\mathring{s}, u, v) \,.\, \left\{ f_{\omega}^{1}(s_o) \circ f_{\omega}^{1}(s_i) \;\middle|\; s_i \in S_{bcet}^{I}(\mathring{s}, u), s_o \in S_{bcet}^{O}(\mathring{s}, v) \right\} \tag{5.129}$$

denotes the set of all feasible evaluations for an entry and an exit path, respectively. Let F_{ω}^{d} be defined as in Equation 5.125. Analogously, we define another set of all unroll

distances as:

$$R^{io}_{\mu} := \lambda(\mathring{s}, u, v) \,.\, \left\{\omega \;\middle|\; (\omega, f, \beta_\pi) = f_\omega(0, f_0, \beta_\pi), f_\omega \in F^{io}_{\omega}(\mathring{s}, u, v)\right\} \tag{5.130}$$

where f_0 denotes initial flow. Let R^d_μ be defined as in Equation 5.127.

Intuitively, the shortest feasible path length is either denoted by a direct path or the shortest feasible unroll. Therefore, given a scope $\mathring{s} \in \mathring{V}$ and entry/exit nodes $u, v \in V$, we define:

$$\min^{\mathring{s},\text{-}1}_{\mu} \colon \mathring{V} \times V \times V \mapsto \mathbb{N}^{\infty,\perp}_0$$
$$\min^{\mathring{s},\text{-}1}_{\mu}(\mathring{s}, u, v) = \begin{cases} \min R^d_\mu(\mathring{s}, u, v) & \min R^d_\mu(\mathring{s}, u, v) \neq \perp \\ \min R^{io}_\mu(\mathring{s}, u, v) & \text{otherwise} \end{cases} \tag{5.131}$$

Finally, we define a function $\min^{\star}_{\mu}(\mathring{s}, s, \mathring{t}, t)$, which computes the minimal path length from source scope $\mathring{s}$ and node s to a target scope $\mathring{t}$ and node t, by computing the shortest path to all entries of $\mathring{t}$ and the shortest path within $\mathring{t}$ to t as:

$$\min^{\star}_{\mu} \colon \mathring{S} \times V \times \mathring{S} \times V \mapsto \mathbb{N}^{\infty,\perp}_0$$
$$\min^{\star}_{\mu}(\mathring{s}, s, \mathring{t}, u) =$$
$$\begin{cases} \min \left\{ \min^{\star}_{\mu}(\mathring{s}, s, \mathring{u}, i) + \min^{\mathring{s},\text{-}1}_{\mu}(\mathring{t}, i, u) \;\middle|\; \begin{array}{l} i \in \text{entry}(\mathring{s}), \\ \mathring{u} = \text{par}(\mathring{t}) \end{array} \right\} & \text{if } \mathring{s} \neq \mathring{t} \\ \min^{\mathring{s},\text{-}1}_{\mu}(\mathring{s}, s, u) & \text{otherwise} \end{cases} \tag{5.132}$$

5.3.4.3 Evaluation

In the following we evaluate precision and performance of our approach to BCET estimation. The test environment is identical to that of the WCET evaluation in Section 5.3.3.6 on page 159 and we use a similar method. For precision, we evaluate results from the MRTC benchmark suite [97]. Performance is evaluated by means of randomized graphs to obtain large sample sets with controlled characteristics. The implementation is an unoptimized derivative of the WCET reference algorithm from Section 5.3.3.5 on page 153.

MRTC

We evaluate a subset of MRTC benchmarks to demonstrate the loss of precision of BCET estimates over shortest-path execution time (SPET) estimates. A qualitative comparison of BCET to WCET yields obvious results, which is why we compare to SPET. For the latter, we use a modified implementation of BCET that consistently gives priority to direct paths and only falls back to full unrolling of at most a *single* iteration if no feasible direct path exists. We do not provide any details on SPET here, as it is straight-forward. Averages are computed by the arithmetic mean.

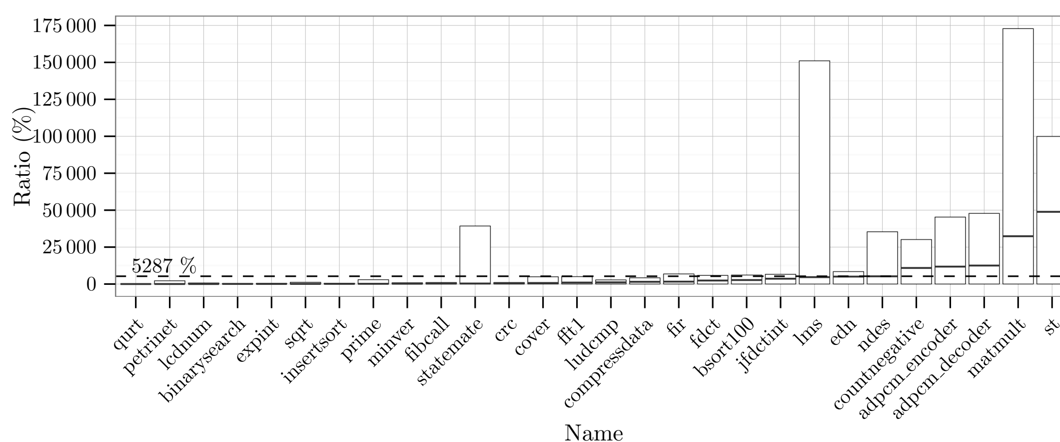


Figure 5.46: Improvement (%) in precision of BCET over shortest-path estimates

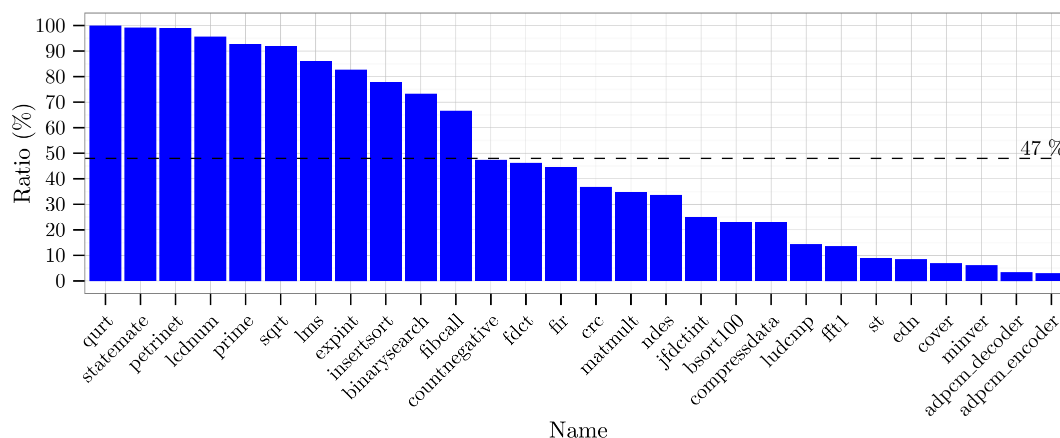


Figure 5.47: Program points (%) with non-deviating time estimates

Figure 5.46 illustrates the ratio of precision between BCET and SPET as box plots showing upper, lower and average ratios for all program points within a single benchmark. Benchmarks are ordered by ascending average difference. For all benchmarks from QURT (no difference) to ST (49×10^3 % underestimation on average), SPET underestimates BCET by ∼5 387 % on average over all benchmarks. All benchmarks yield a lower ratio bound of 0 %: there always exist program points that do not differ for SPET and BCET. For upper ratio bounds, underestimation of up to 1729×10^3 % (MATMULT) occurs. This suggests that trivial techniques to compute lower BCET bounds are usually impractical.

Figure 5.47 illustrates the ratio on the number of program points whose time bounds do *not* differ for BCET and SPET, respectively. On average, 47 % of points are not affected. Benchmark QURT yields equal results on all program points, whereas for ADPCM_ENCODER only 1 % of estimates do not differ.

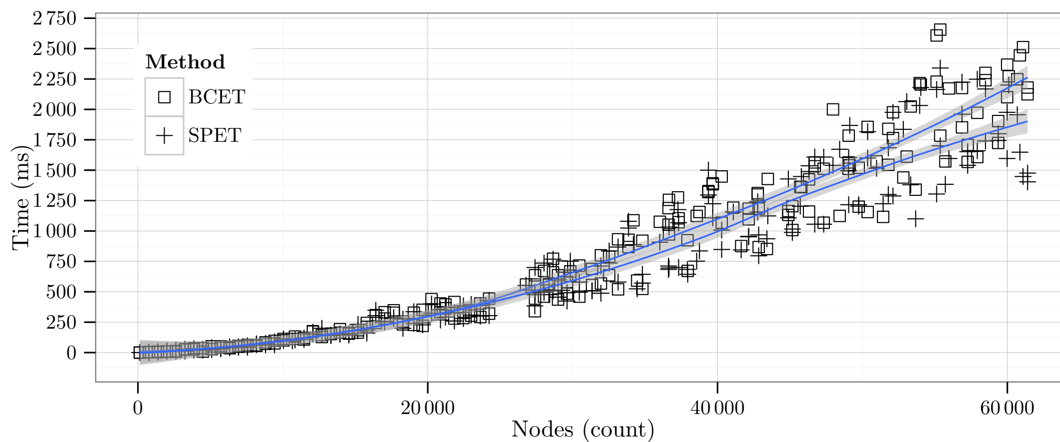


Figure 5.48: Runtime for non-degenerated CFG (DEPTH $= 4$, LOOP DEPTH $= 3$, P(IF) $= 0.1$, IFELSE $= 0.2$, P(WHILE) $= 0.3$, P(DOWHILE) $= 0.4$)

Randomized Graphs

Real-time benchmarks for qualitative comparison are not well suited for a quantitative comparison of performance due to their limited size. So we evaluate randomized CFGs from a size of approximately 100 to 60 000 nodes. As before, randomization parameters for control flow constructs and annotations are given in Table 5.2 on page 118. Note that we employed unoptimized reference implementations for both measurements.

In Figure 5.48 we relate execution time in ms to graph sizes for a "typical" distribution of control flow constructs (cf. parameters in caption), and we generate just a single bound per loop. BCET is denoted by the upper line of best fit. As an be observed, SPET is only marginally faster then BCET due to simplified unrolling. For 61 396 nodes, SPET takes 1 476 ms and BCET takes 2 124 ms. Consequently, from a performance point-of-view, a simpler but much more imprecise estimation of lower time bounds is barely justified even for very large graph instances. Similar to WCET evaluations, we did not observe significant variations for different randomization parameters.

5.3.4.4 Conclusion

In this section we derived a framework for the computation of best-case path lengths from the model for worst-case path lengths. In particular, we showed that the problem is not trivially derivable, and discussed and addressed the differences. Optimizations and the reference implementation as proposed previously remain valid with the necessary changes. Also complexity characteristics are very similar, which is why we do not repeat performance evaluation here.

5.3.5 Computing Latest Execution Time Bounds

In the two previous sections, we have been concerned with the computation of worst-case and best-case path lengths from source to terminal program points. Unfortunately, they

suffer from an inherent imprecision: What we computed are path lengths under the assumption that task execution actually terminates in these points. In this section, we address the problem of computing time bounds for interior program points under the constraint that global tasks exits remain reachable under given flow bounds.

Definition 5.73 (Latest Execution Time) Latest execution time *(LET) denotes a bound on the worst-case (best-case) execution time of a program point such that there remains a feasible path to the terminal node of the corresponding task.*

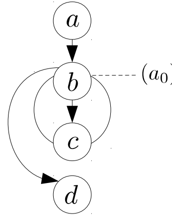


Figure 5.49: Example CFG to demonstrate difference between WCET and LET

Example *Figure 5.49 illustrates the difference between regular worst-case bounds and latest execution worst-case time bounds for a task composed of a single loop. We assume* $\forall u \in V : \omega(u) = 1$ *and* $\beta_\pi(a_0) = 2$*, and source node a. The worst-case path to node c is* $|(abcbc)| = 5$*. For the latest execution time bound, reachability of terminal node d must be guaranteed. Hence, the latest execution time path is* $|(abc)| = 3$*, since path* $(abcbd) \supseteq (abc)$ *must remain feasible.*

In the following we restrict the discussion to worst-case latest execution times. As before, we first introduce technical prerequisites which include the state space and the underlying arithmetic in Section 5.3.5.1. We discuss the underlying framework in Section 5.3.5.2, followed by an evaluation in Section 5.3.5.3 and concluding remarks in Section 5.3.5.4. As usual, we assume acquaintance with the principles of WCET computation from Section 5.3.3.

5.3.5.1 Prerequisites

S^d_π	Path states
$s^d_\pi = (\delta^d_\pi, \Delta^d_\pi, \sigma^d_\pi, o^d_\pi) \in S^d_\pi$	Path state
$\delta^d_\pi \colon S^d_\pi \mapsto \mathbb{N}_0^{\infty,\perp}$	Path length
$\Delta^d_\pi \colon S^d_\pi \mapsto (V \mapsto \mathbb{N}_0^{\infty,\perp})$	Path length map
$\sigma^d_\pi \colon S^d_\pi \mapsto A_\pi{}^*$	Path signature
$o^d_\pi \colon S^d_\pi \mapsto V$	Path origin

Table 5.4: Additional definitions for latest execution time analysis

In the following we discuss the technical prerequisites for LET analysis by defining a new state space and the underlying arithmetic. Basic definitions from Table 5.3 on

page 132 remain valid except for the replacement of path states. Here, we represent paths by means of path states $(\delta^d_\pi, \Delta^d_\pi, \sigma^d_\pi, o^d_\pi) = s^d_\pi \in S^d_\pi$, which encode path length $\delta^d_\pi\colon S^d_\pi \mapsto \mathbb{N}_0^{\infty,\perp}$, an *additional* mapping of nodes to path lengths $\Delta^d_\pi\colon S^d_\pi \mapsto (V \mapsto \mathbb{N}_0^{\infty,\perp})$, signature $\sigma_\pi\colon S^d_\pi \mapsto {A_\pi}^*$, which denotes a sequence of annotations along paths, and origin of paths $o_\pi\colon S^d_\pi \mapsto V$.

Accordingly, we discriminate path states S^d_π by the equivalence relation defined as:

$$\overset{A_\pi}{\sim}\colon S^d_\pi \times S^d_\pi$$
$$s \overset{A_\pi}{\sim} s' \Leftrightarrow \sigma^d_\pi(s^d_\pi) = \sigma^d_\pi(s^{d\prime}_\pi) \wedge o^d_\pi(s^d_\pi) = o^d_\pi(s^{d\prime}_\pi) \tag{5.133}$$

The underlying algebraic structure remains to be the commutative semi-ring

$$(\mathbb{N}_0^{\infty,\perp}, \max, \perp, +, 0) \tag{5.134}$$

as in the WCET-case (Equation 5.50 on page 132). As before, we also lift operator max to path states of the *same* equivalence class. Similar to function $\max_\pi$ (Equation 5.53 on page 133), we determine the path state of maximal length. In addition, we join all length maps Δ^d_π of all states. Hence, we define:

$$\max^{\text{let}}_\pi\colon \wp(S^d_\pi) \mapsto S^d_\pi$$
$$\max^{\text{let}}_\pi(S) = \left(\max_{s' \in S} \delta^d_\pi(s'), \bigcup_{s' \in S} \Delta^d_\pi(s'), \sigma^d_\pi(s), o^d_\pi(s)\right) : s \in S \tag{5.135}$$

For an equivalence class $S = [s]$ according to Equation 5.133, $\max^{\text{let}}_\pi([s])$ denotes a path of maximal length.

5.3.5.2 Framework

We now define the analysis framework. Except for the newly introduced additional mapping of path lengths Δ^d_π, it remains similar to the one defined for the computation of simple worst-case length in Section 5.3.3.

First we define how iterations are obtained. Then we extend semantics to complete unrolls and we discuss how LET to all interior points are computed.

Iterations

As before, the problem of computing iteration lengths has the structure of the semi-lattice:

$$(\mathbb{D}_{let}, \top, \sqsubseteq, \sqcup) \tag{5.136}$$

where $\mathbb{D}_{let} = \wp(S^d_\pi)$ is a set of path states such that for path states $S, S' \in \mathbb{D}_{let}$, where $S \sqsubseteq S' \Leftrightarrow S \subseteq S'$ and where $S \sqcup S' := \{\max^{\text{let}}_\pi[s] \mid [s] \in (S \cup S')/\overset{A_\pi}{\sim}\}$ denotes the set of path states of different signature, origin, maximal length and joined length maps.

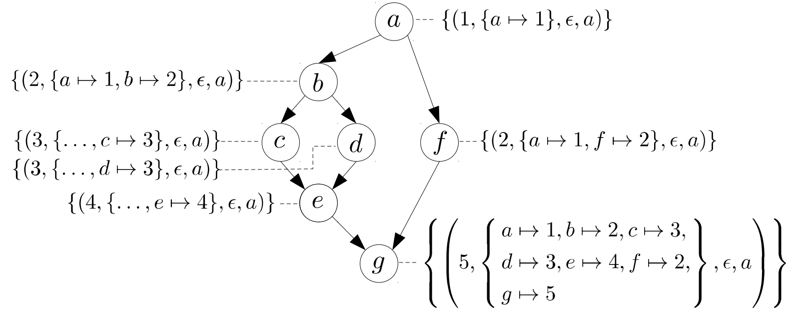


Figure 5.50: Example of LET state propagation

We define a transformer which does not only update path lengths and annotations, but which also records path length to the current node:

$$\mathrm{tf}_{let} \colon V \mapsto (\mathbb{D}_{let} \mapsto \mathbb{D}_{let})$$

$$\mathrm{tf}_{let}(u) = \lambda S \,.\, \left\{ \left(\begin{array}{l} \delta^d_\pi(s) + \omega(u), \\ \Delta^d_\pi(s)[u \to \delta^d_\pi(s) + \omega(u)], \\ \sigma^d_\pi(s) \cdot \alpha_\pi(u), \\ o^d_\pi(s) \end{array} \right) \middle|\; s \in S \right\} \tag{5.137}$$

Example *We assume a single scope without descendants and define path states in a program point as:*

$$S_{let} \colon V^2 \times A_\pi \mapsto S^d_\pi$$

$$S_{let}(s, v, a) = \begin{cases} \mathrm{tf}_{let}(u)(\{(0, a, s)\}) & \text{if } v = s \\ \bigsqcup \{\mathrm{tf}_{let}(u)(S_{let}(s, u, a)) \mid u \in \mathrm{pred}^{\to}(v)\} & \text{otherwise} \end{cases} \tag{5.138}$$

Let $\alpha_\pi = \emptyset$ and $\forall u \in V \colon \omega(u) = 1$, then Figure 5.50 illustrates the corresponding states per node $u \in V$ for $S_{let}(a, u, \epsilon)$, where $\max^{\mathrm{let}}_\pi S_{let}(a, u, \epsilon)$ denotes the longest path to u including all maximal path lengths to individual nodes on the path.

Besides path lengths, states now represent information on whether a node is on a set of paths and the maximal path length from a source node to these respective nodes.

We now define the computation of iterations with scope descendants. We assume a function

$$\max^{\mathring{s}}_{\mu^{\mathrm{let}}} \colon \mathring{V} \times V^3 \mapsto \mathbb{N}_0^{\infty,\perp} \tag{5.139}$$

such that $\max^{\mathring{s}}_{\mu^{\mathrm{let}}}(\mathring{s}, u, v, w)$ evaluates to the longest feasible unroll length in scope $\mathring{s}$ from node u (source) to node w (target) such that node v (sink) remains reachable. As usual ∞ denotes unboundedness and $\perp$ denotes infeasibility.

First, we define an additional transformer to handle entry into a scope and which differs from tf_{let} by not updating distances and annotations, but which only updates the

length map with the distance to the current node as:

$$\mathrm{tf}^{in}_{let} \colon V \mapsto (\mathbb{D}_{let} \mapsto \mathbb{D}_{let})$$
$$\mathrm{tf}^{in}_{let}(u) = \lambda S \,.\, \left\{ \begin{pmatrix} \delta^d_\pi(s), \\ \Delta^d_\pi(s)[u \to \delta^d_\pi(s)], \\ \sigma^d_\pi(s), \\ o^d_\pi(s) \end{pmatrix} \;\middle|\; s \in S \right\} \tag{5.140}$$

Second, we define a transformer to handle exit from a scope which updates a set of path states by the unroll distance of a given scope as:

$$\mathrm{tf}^{out}_{let} \colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{let} \mapsto \mathbb{D}_{let})$$
$$\mathrm{tf}^{out}_{let}(\mathring{s}, u, v) = \lambda S \,.\, \left\{ \begin{pmatrix} \delta^d_\pi(s^d_\pi) + l, \\ \Delta^d_\pi(s^d_\pi)[v \to \delta^d_\pi(s) + l], \\ \sigma^d_\pi(s^d_\pi), \\ o^d_\pi(s^d_\pi) \end{pmatrix} \;\middle|\; \begin{matrix} s^d_\pi \in S, \\ l = \max^{\mathring{s}}_{\mu^{\mathrm{let}}}(\mathring{s}, u, v, v) \end{matrix} \right\} \tag{5.141}$$

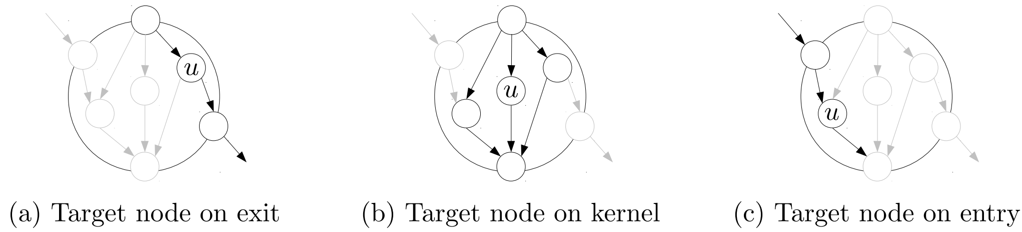
Finally, we compose the general transformer which incorporates the different cases (entry, exit and default transformation) to compute iteration lengths including subscopes as:

$$\mathrm{tf}^{\mathring{s}}_{let} \colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{let} \mapsto \mathbb{D}_{let})$$
$$\mathrm{tf}^{\mathring{s}}_{let}(\mathring{s}, u, v) = \lambda S \,.\, \begin{cases} \mathrm{tf}^{in}_{let}(u)(S) & \text{if } v \in \mathrm{entry}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}^{out}_{let}(\mathring{t}, u, v)(S) & \text{if } v \in \mathrm{exit}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}_{let}(u)(S) & \text{otherwise} \end{cases} \tag{5.142}$$

For a node u and its predecessor v, if v is a scope entry then we only update the length map with the distance up to v. If v is a scope exit then we compute a maximal unroll of the subscope and update path states accordingly. Otherwise tf_{let} applies. Function tf^{in}_{let} ensures that the length mapping contains all relevant nodes for a given scope. Function tf^{out}_{let} updates the length mapping accordingly. Note that $\max^{\mathring{s}}_{\mu^{\mathrm{let}}}(\mathring{t}, u, v, v) = \max^{\mathring{s}}_{\mu}(\mathring{t}, u, v)$ (cf. Equation 5.105 on page 151), which simply denotes the maximal feasible unroll from node u to node v.

Assuming $\mathrm{pred}^{\rightarrow}_{\mathring{s}}$ (Equation 5.99 on page 150) to denote CFG predecessors such that subscopes are being "leaped over", we define path states in a program point by:

$$S^{\mathring{s}}_{let} \colon \mathring{V} \times V^2 \times A_\pi \mapsto S^d_\pi$$
$$S^{\mathring{s}}_{let}(\mathring{s}, s, v, a) = \begin{cases} \mathrm{tf}^{\mathring{s}}_{let}(\mathring{s}, s, s)(\{(0, \emptyset, a, s)\}) & \text{if } v = s \\ \bigsqcup \left\{ \mathrm{tf}^{\mathring{s}}_{let}(\mathring{s}, u, v)(S^{\mathring{s}}_{let}(\mathring{s}, s, u, a)) \mid u \in \mathrm{pred}^{\rightarrow}_{\mathring{s}}(\mathring{s})(v) \right\} & \text{otherwise} \end{cases} \tag{5.143}$$



(a) Target node on exit (b) Target node on kernel (c) Target node on entry

Figure 5.51: Different locations of reference nodes

For a given scope $\mathring{s}$, start and terminal nodes u, v and an initial annotation a, $S^{\mathring{s}}_{let}(\mathring{s}, u, v, a)$ denotes the set of maximal iterations including unrolls of all subscopes. Note that $S^{\mathring{s}}_{let}(\mathring{s}, u, v, a) = S^{\mathring{s}}_{wcet}(\mathring{s}, u, v, a)$.

Finally, we define partitions of $S^{\mathring{s}}_{let}$ as in the case of WCET (cf. Section 5.3.3.2) such that

$$S^{I}_{let} \colon \mathring{V} \times V \mapsto \wp(S^{d}_{\pi}) \tag{5.144}$$

$$S^{O}_{let} \colon \mathring{V} \times V \mapsto \wp(S^{d}_{\pi}) \tag{5.145}$$

$$S^{K}_{let} \colon \mathring{V} \mapsto \wp(S^{d}_{\pi}) \tag{5.146}$$

$$S^{D}_{let} \colon \mathring{V} \times V^2 \mapsto \wp(S^{d}_{\pi}) \tag{5.147}$$

denote entry, exit, kernel and direct paths, respectively.

Note that a significant optimization in practice is to avoid mapping individual path distances in the root scope (acyclic control flow), since distances can directly be derived from path lengths δ^{d}_{π} without unrolling if feasibility of the global exit is guaranteed.

Unrolls

So far, we have computed worst-case path lengths exactly as in the standard case, albeit propagating additional path length information. We now address the problem of computing unrolls of maximal length under the condition that scope entries and exits remain feasible.

Assume execution of a loop represented by the scopes in Figure 5.51, which shows different placements of a target node u. We further assume all nodes are reachable within the CFG. In particular we can reach the exit from the entry. Then, if target node u is on an exit path (Figure 5.51a), we necessarily executed the entry paths and all kernels prior to reaching u, all contributing to its distance from the entry. If the u is on one of the kernels (Figure 5.51b), then the maximal distance to u is given if the entry path and all kernels, except for the very last kernel repetition that can reach u, are executed prior to reaching u, and finally the exit path is taken. If u is only on the entry path (Figure 5.51c), then its distance is simply the distance from the entry to u, while all kernels and the exit path are still to be executed, but which do not contribute to the distance of u.

In the following we refine iteration evaluations with additional parameters to extend semantics of their original versions for WCET from Section 5.3.3.2.

We extend f^1_ω (Equation 5.79) with a predicate $p \in \{0, 1\}$ to prevent length accumulation while retaining semantics of pushing unit flow along a path denoted by a path state $s \in S^d_\pi$:

$$f^{1,p}_\omega := \lambda(s,p) \,.\, \lambda(\omega, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \delta^d_\pi(s) \times p, \\ \text{set}_f(f, 1, \sigma^d_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \wedge \text{test}_f(c, f, \sigma^d_\pi(s)) > 0 \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.148}$$

In addition, we define another variant of f^1_ω (Equation 5.79) such that path length is not denoted by the entire path represented by a path state $s \in S^d_\pi$ but only by its subpath to a node $u \in V$:

$$f^{1,u}_\omega := \lambda(s,u) \,.\, \lambda(\omega, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \Delta^d_\pi(s)(u), \\ \text{set}_f(f, 1, \sigma^d_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \wedge \text{test}_f(c, f, \sigma^d_\pi(s)) > 0 \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.149}$$

Note that we assume $u \notin \text{def}(\Delta^d_\pi(s_o)) \Leftrightarrow \Delta^d_\pi(s)(u) = \bot$ and recall that $\omega + \bot = \bot$ (cf. Section 5.3.3.1). This implies that a test fails, if either there is no admissible flow left or if the target node is not on the path denoted by the respective path state.

We also define a predicated version of f^k_ω (Equation 5.80) similar to $f^{1,p}_\omega$ above:

$$f^{k,p}_\omega := \lambda s \,.\, \lambda p \,.\, \lambda(\omega, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \delta^d_\pi(s) \times \text{test}_f(c, f, \sigma_\pi(s)) \times p, \\ \text{set}_f(f, \text{test}_f(c, f, \sigma_\pi(s)), \sigma^d_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.150}$$

Analogously to f^K_ω (Equation 5.81), we lift $f^{k,p}_\omega$ to a sequence of kernels ordered by descending length:

$$f^{K,p}_\omega := \lambda(\mathring{s}, p) \,.\, [\![\sigma_{\delta^d_\pi}(S^K_{let}(\mathring{s}))]\!](f^{k,p}_\omega(p)) \tag{5.151}$$

Predicates in all versions allow to test for feasibility while preventing length accumulation. From these definitions, we can now construct an evaluation that covers the cases illustrated in Figure 5.51.

With these extended tests, we now construct evaluations for the different cases discussed earlier.

In the first case, target node u is assumed to be on an exit path and we define:

$$f_\omega^{o'ik} := \lambda(\mathring{s}, s_i, s_o, u) \,.\, f_\omega^{K,p}(\mathring{s}, 1) \circ f_\omega^{1,p}(s_i, 1) \circ f_\omega^{1,u}(s_o, u) \tag{5.152}$$

We test whether node u is on the exit path (state s_o). If it is not or if the entire path is infeasible, $f^{i'ok}$ fails. Otherwise, the full path lengths of the entry path (state s_i) and all kernels are accumulated to the distance of u in the exit path.

The second test models the case when the target node is on a kernel and we define:

$$f_\omega^{oik'} := \lambda(\mathring{s}, s_i, s_o, s_k, u) \,.\, f_\omega^{K,p}(\mathring{s}, 1) \circ f_\omega^{1,u}(s_k, u) \circ f_\omega^{1,p}(s_i, 1) \circ f_\omega^{1,p}(s_o, 0) \tag{5.153}$$

We test for exit path (state s_o) feasibility without accounting for its lengths, test for entry path (state s_i) feasibility including its complete path length, test a kernel (state s_k) for feasibility and whether it contains target node u. If it does not, the test fails. Otherwise, we accumulate the path length to node u and all remaining feasible kernel lengths.

Thirdly, we test whether the target node is on an entry path, and define:

$$f_\omega^{oi'k} := \lambda(s_i, s_o, u) \,.\, f_\omega^{1,u}(s_i, u) \circ f_\omega^{1,p}(s_o, 0) \tag{5.154}$$

This follows the familiar pattern: We test for exit path (state s_o) feasibility without accumulating path length. Then entry path (state s_i) feasibility is given if and only of the path is feasible and target node u is on this path. If so, we account for its length.

The composition of all tests (cf. Equation 5.82 on page 143), for an entry node u, an exit node v and a target node w, is then defined by:

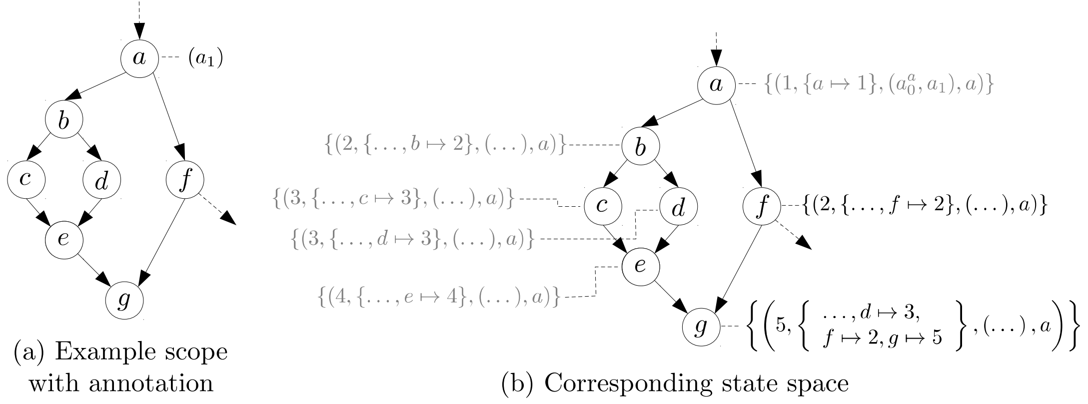
$$\begin{aligned} F_\omega^{oik} := \lambda(\mathring{s}, u, v, w).& \\ &\left\{ f_\omega^{o'ik}(\mathring{s}, s_i, s_o, u) \;\middle|\; s_i \in S_{let}^I(\mathring{s}, u), s_o \in S_{let}^O(\mathring{s}, v) \right\} \\ &\cup \left\{ f_\omega^{oik'}(\mathring{s}, s_i, s_o, s_k, u) \;\middle|\; s_i \in S_{let}^I(\mathring{s}, u), s_o \in S_{let}^O(\mathring{s}, v), s_k \in S_{let}^K(\mathring{s}) \right\} \\ &\cup \left\{ f_\omega^{oi'k}(s_i, s_o, u) \;\middle|\; s_i \in S_{let}^I(\mathring{s}, u), s_o \in S_{let}^O(\mathring{s}, v) \right\} \end{aligned} \tag{5.155}$$

By construction, this covers all possible cases for unrolling. Note that we chose this scheme for clarity. In practice, we can optimize by removing redundancy thoroughly.

As usual, we also have to account for direct paths. The set of all tests is defined as:

$$F_\omega^d := \lambda(\mathring{s}, u, v, w) \,.\, \left\{ f_\omega^{1,u}(s_d, w) \;\middle|\; s_d \in S_{wcet}^D(\mathring{s}, u, v) \right\} \tag{5.156}$$

Either target node w is on a feasible direct path, then we account for its relative path length, or the tests fail.



(a) Example scope with annotation

(b) Corresponding state space

Figure 5.52: Example of path states for LET computations

Finally, let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then the set of all unroll distances is defined as:

$$\begin{aligned} R_\mu &:= \lambda(\mathring{s}, u, v, w). \\ &\left\{\omega \;\middle|\; (\omega, f, \beta_\pi) = f_\omega(0, f_0, \beta_\pi), f_\omega \in F_\omega^{oik}(\mathring{s}, u, v, w) \cup F_\omega^{d}(\mathring{s}, u, v, w)\right\} \end{aligned} \tag{5.157}$$

Theorem 5.74 (Latest Execution Time Unroll) *The maximal unroll length for a scope $\mathring{s} \in \mathring{V}$, entry and exit nodes $u, v \in V$ and a target node $w \in V$ is denoted by:*

$$\begin{aligned} \max\nolimits^{\mathring{s}}_{\mu^{\text{let}}} &: \mathring{V} \times V^3 \mapsto \mathbb{N}_0^{\infty,\perp} \\ \max\nolimits^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{s}, u, v, w) &= \max R_\mu(\mathring{s}, u, v, w) \end{aligned} \tag{5.158}$$

This represents the precise worst-case path length for a scope to a dedicated target node w, given an entry node u and an exit node v that must remain reachable under the given flow bound model.

Proof. By construction, all possible cases for the target node w are covered explicitly (cf. Figure 5.51). For just entry node u and exit node v, correctness of the standard WCET case applies. □

Example *Figure 5.52a illustrates an example scope $\mathring{s}$ with entry a, exit f and a single annotation in node a. Figure 5.52b depicts the corresponding states space (cf. Figure 5.50) where only the non-faded states are of relevance in the following. We assume $\beta_\pi(a_1) = 2$, unit weight ω and compute LET from node a to node d with $\max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{s}, a, f, d)$. Partitions of states $S^{\mathring{s}}_{let}$ yield:*

$$\begin{aligned} S^I_{let}(\mathring{s}, a) &= \{s_i\} = \{(5, \{\dots, d \to 3, f \to 2, g \to 5\}, (\dots), a)\} \\ S^O_{let}(\mathring{s}, f) &= \{s_o\} = \{(2, \{\dots, f \to 2\}, (\dots), a)\} \\ S^K_{let}(\mathring{s}) &= \{s_k\} = \{s_i\} \\ S^D_{let}(\mathring{s}, a, f) &= \{s_d\} = \{s_o\} \end{aligned}$$

Composition of evaluation semantics yields:

$$
\begin{aligned}
&F_\omega^{oik}(\mathring{s}, a, f, d) \cup F_\omega^{d}(\mathring{s}, a, f, d) \\
&\quad = \left\{ f_\omega^{o'ik}(\mathring{s}, s_i, s_o, d), f_\omega^{oik'}(\mathring{s}, s_i, s_o, s_k, d), f_\omega^{oi'k}(s_i, s_o, d) \right\} \cup \left\{ f_\omega^{1,u}(\mathring{s}, s_o, d) \right\} \\
&\quad = \left\{ \begin{array}{l} f_\omega^{K,p}(\mathring{s}, 1) \circ f_\omega^{1,p}(s_i, 1) \circ f_\omega^{1,u}(s_o, d), \\ f_\omega^{K,p}(\mathring{s}, 1) \circ f_\omega^{1,u}(s_k, d) \circ f_\omega^{1,p}(s_i, 1) \circ f_\omega^{1,p}(s_o, 0), \\ f_\omega^{1,u}(s_i, d) \circ f_\omega^{1,p}(s_o, 0), \\ f_\omega^{1,u}(s_d, d) \end{array} \right\} = \{f_o, f_k, f_i, f_d\}
\end{aligned}
$$

Let $f_0 = A_\pi \times \{0\}$ denote initial flow. Evaluation of tests f_o, f_k, f_i and f_d starting with path lengths of 0*, initial flow f_0 and flow bounds β_π then yields:*

$$
\begin{aligned}
&\{f_o(0, f_0, \beta_\pi), f_k(0, f_0, \beta_\pi), f_i(0, f_0, \beta_\pi), f_d(0, f_0, \beta_\pi)\} \\
&\quad = \left\{ \ldots, f_\omega^{1,u}(s_d, d)(0, f_0, \beta_\pi) \right\} && \text{(direct)} \\
&\quad = \left\{ \ldots, (f_\omega^{1,u}(s_i, d) \circ f_\omega^{1,p}(s_o, 0))(0, f_0, \beta_\pi) \right\} && \text{(entry)} \\
&\qquad \cup \{(\bot, f, \beta_\pi)\} \\
&\quad = \left\{ \ldots, (f_\omega^{K,p}(\mathring{s}, 1) \circ f_\omega^{1,u}(s_k, d) \circ f_\omega^{1,p}(s_i, 1) \circ f_\omega^{1,p}(s_o, 0))(0, f_0, \beta_\pi) \right\} && \text{(kernel)} \\
&\qquad \cup \left\{ (3, f', \beta_\pi), (\bot, f, \beta_\pi) \right\} \\
&\quad = \left\{ (f_\omega^{K,p}(\mathring{s}, 1) \circ f_\omega^{1,p}(s_i, 1) \circ f_\omega^{1,u}(s_o, d))(0, f_0, \beta_\pi) \right\} && \text{(exit)} \\
&\qquad \cup \left\{ (\bot, f', \beta_\pi), (3, f', \beta_\pi), (\bot, f, \beta_\pi) \right\} \\
&\quad = \left\{ (\bot, f, \beta_\pi), (\bot, f', \beta_\pi), (3, f', \beta_\pi), (\bot, f', \beta_\pi) \right\}
\end{aligned}
$$

where $f = \{a_0^a \to 2, a_1 \to 2\}$ and $f' = \{a_0^a \to 1, a_1 \to 1\}$. Consequently, the maximal distance to node d such that scope $\mathring{s}$ can be entered and left through nodes a and f, respectively, equals:

$$
\max_{\mu^{\text{let}}}^{\mathring{s}}(\mathring{s}, a, f, d) = \max R_\mu = \max\{\bot, \bot, 3, \bot\} = 3
$$

Corollary 5.75 *For the given scenario (WCET computation), let $\mathring{0} \in \mathring{V}$ be the root scope, and let $s \in \text{entry}(\mathring{s})$ and $t \in \text{exit}(\mathring{s})$ denote global entry and exit, respectively. Then by construction $\max_{\mu^{\text{let}}}^{\mathring{s}}(\mathring{0}, s, t, t) = \max_{\mu}^{\mathring{s}}(\mathring{0}, s, t)$, where $\max_{\mu}^{\mathring{s}}$ denotes the WCET bound according to Theorem 5.69.*

Proof. $\max_{\mu^{\text{let}}}^{\mathring{s}}$ denotes the worst-case path length from an entry to an exit such that path length to a designated target node is maximized. Exit and target coincide. □

LET to Interior Points

We are now concerned with the computation of absolute LET path lengths. As before, the intuition for computing absolute path lengths is to recursively compute path lengths from entries of scopes to entries of their respective descendants as offsets to the final path

length computation within the scope containing the target node (cf. Section 5.3.3.4). For LET, we must guarantee that at least one exit of each descending scope remains feasible such that there remains a feasible path from the target node to the global exit.

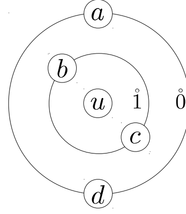


Figure 5.53: Example scenario for LET to interior nodes

Example *To support this intuition, consider Figure 5.53. We assume nodes a and d denote global entries and exits, respectively. Our intent is to compute an absolute LET path length from node a to node u. Then $\max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{0}, a, d, b)$ denotes an offset to scope $\mathring{1}$ for entry b such that global exit d remains reachable. Consequently, $\max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{0}, a, d, b) + \max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{1}, b, c, u)$ denotes the absolute path length to node u, while global exit d is guaranteed to be reachable from entry b. Reachability from exit c to exit d is implicitly given by construction.*

Lemma 5.76 (Feasible Entries) *Given a scope $\mathring{s}$ and its immediate parent $\mathring{t}$. Let $b \in \text{entry}(\mathring{s})$, $c \in \text{exit}(\mathring{s})$, $a \in \text{entry}(\mathring{t})$ and $d \in \text{exit}(\mathring{t})$. Further, we assume that all paths from entry a to exit d must pass through entry b and exit c. Then entry b is only feasible if $\max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{t}, a, d, b)$ is feasible. In particular, entry a is feasible, too.*

Proof. By definition of $\max^{\mathring{s}}_{\mu^{\text{let}}}$. □

Lemma 5.77 (Feasible Exits) *Let prerequisites be given as in Lemma 5.76. Then exit c is only feasible if $\max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{t}, a, d, c)$ is feasible. In particular, exit d is feasible, too.*

Proof. By definition of $\max^{\mathring{s}}_{\mu^{\text{let}}}$. In particular, tf^{out}_{let} (cf. Definition 5.141) requires $\max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{s}, b, c, c)$ to yield a feasible solution. □

Consequently, computing maximal unroll lengths of a scope $\mathring{s}$ guarantees both feasible entries and exits to and from descending scopes while implicitly guaranteeing reachability of exits from entries of $\mathring{s}$.

We define a function $\max^{\star}_{\mu^{\text{let}}}(\mathring{s}, s, t, \mathring{t}, u)$, which computes the (worst-case) LET path length from source scope $\mathring{s}$, with entry node s and exit node t to a target scope $\mathring{t}$ and node u by computing the maximal path length to all entries of $\mathring{t}$ from all entries of its immediate parent scope and the maximal unroll length within $\mathring{t}$ to u such that its

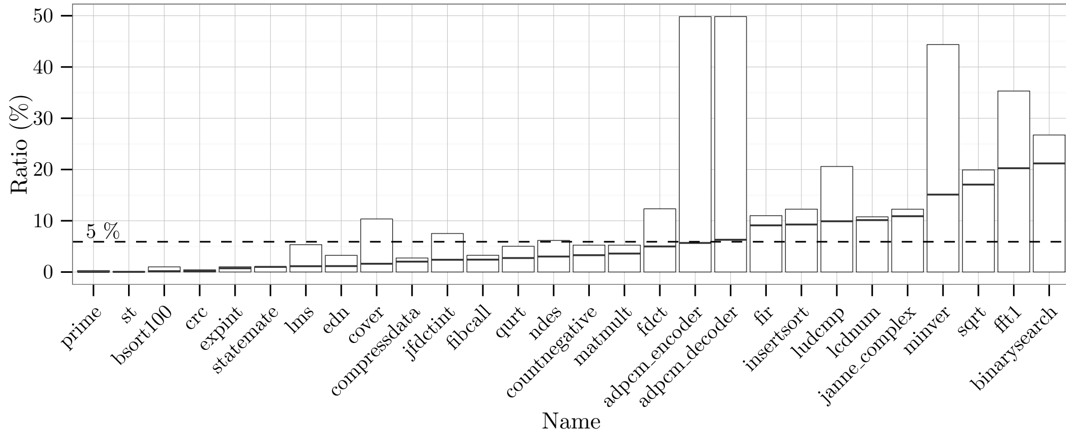


Figure 5.54: Improvement (%) in precision of LET over standard WCET estimates

respective exits are feasible, as:

$$\max^{\star}_{\mu^{\text{let}}} : \mathring{V} \times V^2 \times \mathring{V} \times V \mapsto \mathbb{N}_0^{\infty,\perp}$$

$$\max^{\star}_{\mu^{\text{let}}}(\mathring{s}, s, t, \mathring{t}, u) =$$

$$\begin{cases} \max \left\{ \max^{\star}_{\mu^{\text{let}}}(\mathring{s}, s, t, \mathring{u}, i) + \max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{t}, i, o, u) \;\middle|\; \begin{array}{l} i \in \text{entry}(\mathring{s}), \\ o \in \text{entry}(\mathring{s}), \\ \mathring{u} = \text{par}(\mathring{t}) \end{array} \right\} & \text{if } \mathring{s} \neq \mathring{t} \\ \max^{\mathring{s}}_{\mu^{\text{let}}}(\mathring{s}, s, t, u) & \text{otherwise} \end{cases} \tag{5.159}$$

5.3.5.3 Evaluation

In the following we compare our approach to LET estimation against the "standard" WCET analysis we proposed in Section 5.3.3. To this end, we use a setup as in Section 5.3.3.6 on page 159. We evaluate precision by comparison on benchmarks from the MRTC benchmark suite [97] and performance by means of randomized graphs to obtain large sample sets. Averages are computed by the arithmetic mean. The implementation is a derivative of the WCET reference algorithm from Section 5.3.3.5.

MRTC

We evaluate a subset of MRTC benchmarks to demonstrate the benefits of LET analysis on existing scenarios. Figure 5.54 illustrates the ratio of precision between standard WCET (WCET) and LET (LET) analyses for the given benchmarks as box plots depicting upper, lower and average ratio values. Precisely, the ratios denote the difference in WCET estimates per program point in loops (other program points cannot have deviating time bounds). The diagram is ordered by the average ratio of improvement of LET over WCET. For all benchmarks from PRIME (1.4 % average improvement) to BINARYSEARCH (22 % average improvement), we obtain 5 % more precise results for LET on average over all benchmarks. All benchmarks yield a lower ratio bound of 0 %: there always exist

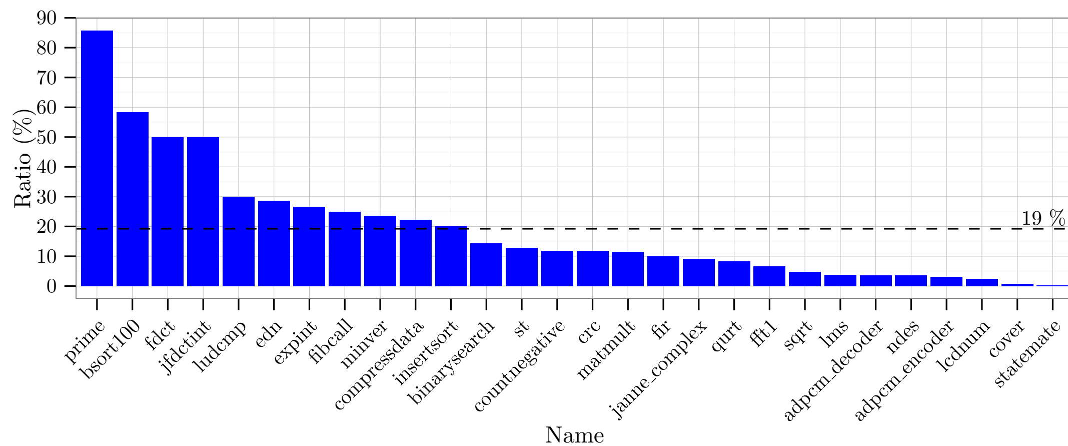


Figure 5.55: Program points (%) with non-deviating time estimates

program points that do not profit from LET. For upper ratio bounds, improvements of up to 50 % (ADPCM_ENCODER, ADPCM_DECODER) an be achieved. Figure 5.55 illustrates quantitative differences as a percentage of program points within loops that do *not* differ. On average, 19 % of program points do not yield improved estimates. Benchmark PRIME correlates with the qualitative result in Figure 5.54 in that 86 % of program points do not differ at all. For BINARYSEARCH, correlation is low: Although it yields the greatest qualitative average difference, 15 % of program points did not differ. As opposed to that, for benchmark STATEMATE, just 1 % of program points yield identical time bounds.

Randomized Graphs

Real-time benchmarks for the qualitative comparison are not well suited for a quantitative comparison of performance due to their limited size. Rather, we evaluate by scaling graph sizes of approximately 100 to 50 000 nodes in size. As before, randomization parameters for control flow constructs and annotations are given in Table 5.2 on page 118. It shall be noted that we employed unoptimized reference implementations for the following evaluations.

In Figure 5.56 we relate graph sizes with execution times (ms) given a “typical” distribution of constructs (cf. parameters in caption), and we generate just a single bound per loop. As can be seen, LET scales only marginally worse than WCET due to additional data maintenance and increased complexity for unrolling. While WCET takes from under 1 ms up to 2.22 s, LET computations take from under 1 ms up to 5.77 s for 62 336 program points. Depending on loop structure, performance of LET analysis is comparably variant. Similar to WCET evaluations, we did not observe significant variation for different randomization parameters.

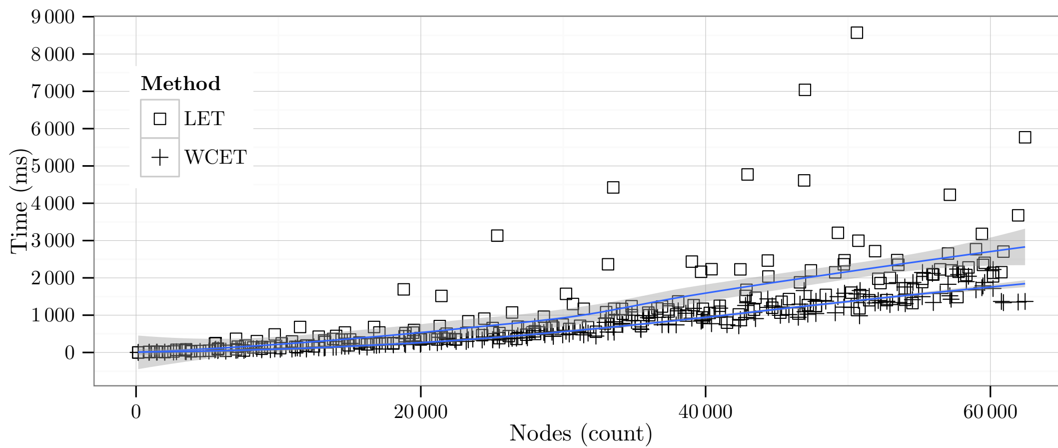


Figure 5.56: Runtime for non-degenerated CFG (DEPTH $= 4$, LOOP DEPTH $= 3$, P(IF) $=$ 0.1, IFELSE $= 0.2$, P(WHILE) $= 0.3$, P(DOWHILE) $= 0.4$)

5.3.5.4 Conclusion

In this section we proposed an alternative notion of WCET to program points by means of latest execution time. It denotes a time bound for individual program points with the additional constraint that the terminal program points must remain reachable under a given flow bound model. As opposed to that, standard WCET bounds as proposed in Section 5.3.3 consider respective (interior) program points as terminal nodes. We showed that for a set of real world benchmarks, we improved by 5 % on average over all benchmarks and up to 22 % on average per benchmark, while differences in individual program points can be considerable. An important application of LET bounds are task interference analyses in fully preemptive schedules.

5.3.6 Computing Maximum Blocking Time Bounds

In Section 3.1 we introduced blocking time as the maximal time span higher priority tasks are prevented from execution by (blocking) lower priority tasks. In the context of path analysis, we more precisely denote it as *maximum blocking time* (MBT). Blocking can implicit be caused by a deliberate scheduling decision (time-triggered, floating program points), but it can also be explicitly caused by means of synchronizing program points such as semaphores or preemption points (fixed program points). In either case, blocking must be taken into consideration in scheduling analysis.

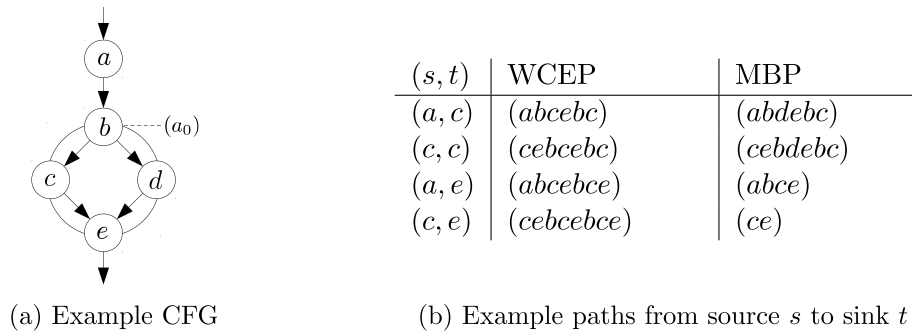
In this section we are concerned with the efficient computation of MBT and we propose a path analysis that efficiently computes, from a designated source, MBT bounds to all reachable program points. To our best knowledge, the only other approach to this problem is proposed in [173], which is, however, based on ILP and therefore incorporates its implied deficiencies (cf. Section 5.2.3). More importantly, it requires a known set of preemption points as input, which makes it unsuitable for efficient design space

exploration. We show how the reduction to the problem for known preemption points, as well as the computation to all potential preemption points.

We first define MBT in the context of our general path analysis framework.

Definition 5.78 (Maximum Blocking Time) Maximum Blocking Time *(MBT) denotes a maximal path length from a designated CFG source node s to a sink node t such that s is only the first element and t only the final element on the corresponding* maximum blocking path *(MBP).*

MBPs model paths between preemption points which might technically be implemented such that its corresponding basic blocks might or might not be executed prior to preemption or after resumption. Hence accounting for the node weights of sources and sinks might differ. In our following proposal, this is a mere technical nuisance and adaption to the specific requirements is simple. We assume in the following that execution starts *right after* source nodes. Hence, their costs are not accounted for. This matches semantics of basic blocks where control is only transferred from the final instruction.



(s,t)	WCEP	MBP
(a,c)	$(abcebc)$	$(abdebc)$
(c,c)	$(cebcebc)$	$(cebdebc)$
(a,e)	$(abcebce)$	$(abce)$
(c,e)	$(cebcebce)$	(ce)

(a) Example CFG (b) Example paths from source s to sink t

Figure 5.57: Relating worst-case execution paths to maximum block paths

Example *Figure 5.57 illustrates the difference between worst-case paths and maximum blocking paths. Assume a CFG given as in Figure 5.57a, where the scope denotes a loop bounded by* $\beta_\pi(a_0) = 2$ *and costs are distributed such that* $\omega(a) > \omega(b) > \omega(c) > \omega(d) > \omega(e)$. *Then Figure 5.57b contains examples for various source and sink nodes* (s,t) *along with their corresponding paths. Note that the paths do not necessarily reflect that actual costs as semantics for source and sink nodes may vary.*

In the following we assume the global CFG entry to always denote the source node. Technical measures for reduction to arbitrary nodes as sources have been discussed in Section 5.3.3.4.

We follow the usual scheme. After an introduction to technical prerequisites in Section 5.3.6.1, such as state representation and arithmetic, we discuss the framework itself in Section 5.3.6.2. We conclude the section with an evaluation in Section 5.3.6.3 and concluding remarks in Section 5.3.6.4. As usual, we assume acquaintance with the principles of WCET computation from Section 5.3.3.

S_π^d	Path states
$s_\pi^d = (\delta_\pi^d, \Delta_\pi^d, \sigma_\pi^d, o_\pi^d) \in S_\pi^d$	Path state
$\delta_\pi^d \colon S_\pi^d \mapsto \mathbb{Z}^{\infty,\perp}$	Path length
$\Delta_\pi^d \colon S_\pi^d \mapsto (V \mapsto \mathbb{Z}^{\infty,\perp})$	Path length map
$\sigma_\pi^d \colon S_\pi^d \mapsto {A_\pi}^*$	Path signature
$o_\pi^d \colon S_\pi^d \mapsto V$	Path origin

Table 5.5: Additional definitions for worst-case blocking time analysis

5.3.6.1 Prerequisites

In the following we discuss the technical prerequisites for MBT analysis by defining a new state space and the underlying arithmetic. Basic definitions from Table 5.3 on page 132 remain valid except for the replacement of path states. For MBT analysis, we reuse path state definitions from LET analysis (cf. Section 5.3.5.1 on page 172) with the exception that path lengths are denoted as elements of $\mathbb{Z}^{\infty,\perp}$. Table 5.5 summarizes these functions. As before, discrimination of path states is defined by the equivalence relation (cf. Equation 5.133 on page 173):

$$\overset{A_\pi}{\sim} \colon S_\pi^d \times S_\pi^d \tag{5.160}$$

The underlying algebraic structure is now the commutative ring:

$$(\mathbb{Z}^{\infty,\perp}, \max, \perp, +, 0) \tag{5.161}$$

with max for addition and + for multiplication, with neutral elements $\perp$ and 0, respectively such that:

$$\max(a,b) := \begin{cases} \max_{\mathbb{Z}^\infty}(a,b) & \text{if } a \neq \perp \wedge b \neq \perp \\ a & \text{if } a \neq \perp \\ b & \text{otherwise} \end{cases} \tag{5.162}$$

$$a + b := \begin{cases} a +_{\mathbb{Z}^\infty} b & \text{if } a \neq \perp \wedge b \neq \perp \\ \perp & \text{otherwise} \end{cases} \tag{5.163}$$

where $\max_{\mathbb{Z}^\infty}(a,b)$ denotes maximum and $+_{\mathbb{Z}^\infty}$ denotes addition on $\mathbb{Z}^\infty$.

For our purposes, we define the following semantics: Function δ_π^d represents worst-case path lengths, which we refer to as *reference path length*. Function Δ_π^d represents *differences* in length from δ_π^d. We now lift the basic algebra to perform computations directly on this representation.

Let $V \mapsto \mathbb{Z}^{\infty,\perp}$ denote the set of all functions that map from nodes V to lengths $\mathbb{Z}^{\infty,\perp}$. Then all pairs of reference path length and difference maps are denoted by :

$$\mathbb{D}^{\delta\times\Delta} = \mathbb{D}^{\delta} \times \mathbb{D}^{\Delta} = \mathbb{Z}^{\infty,\perp} \times \left(V \mapsto \mathbb{Z}^{\infty,\perp}\right) \tag{5.164}$$

Then we define the following algebra on $\mathbb{D}^{\delta\times\Delta}$:

$$\left(\mathbb{D}^{\delta\times\Delta}, \max, 1_{\max}, +, 1_{+}\right) \tag{5.165}$$

In the following we define its corresponding operators.

Example *The intuition of operator* max *is best shown by simplifying the domain from mappings to differences to a single difference value. Without loss of generality, assume a pair* $(r, d) \in \mathbb{N}^2$ *to denote a reference length and a difference, and an operator* $\max_{\mathbb{N}^2}$*, which is a simplified version of the operator to be defined for domain* $\mathbb{D}^{\delta\times\Delta}$*. Then an example computation is the following:*

$$\begin{aligned} \max_{\mathbb{N}^2}((10,5),(7,1)) &= (\max(10,7), \max(10,7) - \max(10-5, 7-1)) \\ &= (10, 10-6) = (10, 4) \end{aligned}$$

Intuitively, we compute the maximums of the reference lengths, and differences are translated into absolute lengths for comparison and then converted back to a difference from the maximal reference length.

We now define max for domain $\mathbb{D}^{\delta\times\Delta}$. To this end, we first define a function $\text{abs}_{\delta\times\Delta}$, which returns absolute path lengths from a reference and map of differences as:

$$\begin{aligned} \text{abs}_{\delta\times\Delta} &\colon \mathbb{D}^{\delta\times\Delta} \mapsto \mathbb{D}^{\Delta} \\ \text{abs}_{\delta\times\Delta}(l, d) &= \lambda u\,.\begin{cases} l - d(u) & \text{if } u \in \text{def}(d) \\ l & \text{otherwise} \end{cases} \end{aligned} \tag{5.166}$$

Intuitively, difference $d(u)$ encodes three cases:

1. If $d(u) \in \mathbb{Z}^{\infty}$, then it denotes a path not longer than the reference. We assume $d(u) = \infty \Rightarrow l = \infty$.

2. If $u \notin \text{def}(d)$, then it denotes a path length equal to the reference (equal to $d(u) = 0$).

3. If $d(u) = \bot$, then it denotes a path that passed through node u. Note that $l - d(u) = \bot \Leftrightarrow l = \bot \vee d(u) = \bot$.

Inversely to $\text{abs}_{\delta\times\Delta}$, we define a function $\text{rel}_{\delta\times\Delta}$, which returns differences from a reference and a map of absolute lengths as:

$$\begin{aligned} \text{rel}_{\delta\times\Delta} &\colon \mathbb{D}^{\delta\times\Delta} \mapsto \mathbb{D}^{\Delta} \\ \text{rel}_{\delta\times\Delta}(l, a) &= \lambda u\,.\, l - a(u) \end{aligned} \tag{5.167}$$

We also define a function $\max_\Delta$, which returns the maximum of absolute path lengths:

$$\begin{aligned} \max_\Delta &: \mathbb{D}^\Delta \times \mathbb{D}^\Delta \mapsto \mathbb{D}^\Delta \\ \max_\Delta(a, a') &= \lambda u \,.\, \max_{\mathbb{Z}^{\infty,\perp}}(a(u), a'(u)) \end{aligned} \tag{5.168}$$

Finally, we define operator $\max_{\delta\times\Delta}$, which returns the maximum for a pair of reference lengths and differences as:

$$\max \colon \mathbb{D}^{\delta\times\Delta} \times \mathbb{D}^{\delta\times\Delta} \mapsto \mathbb{D}^{\delta\times\Delta}$$

$$\max((l,d),(l',d')) = \left(\begin{array}{l} \max_{\mathbb{Z}^{\infty,\perp}}(l,l'), \\ \left\{ \mathrm{rel}_{\delta\times\Delta} \left(\begin{array}{l} \max_{\mathbb{Z}^{\infty,\perp}}(l,l'), \\ \max_\Delta \left(\begin{array}{l} \mathrm{abs}_{\delta\times\Delta}(l,d), \\ \mathrm{abs}_{\delta\times\Delta}(l',d') \end{array} \right) \end{array} \right)(u) \;\middle|\; u \in \mathrm{def}(d) \cup \mathrm{def}(d') \right\} \end{array} \right) \tag{5.169}$$

Its neutral element is $1_{\max} = \{\perp, V \times \{\perp\}\}$.

Example *Let us investigate examples to strengthen the intuition. In a first example, we compute the maximum of a reference length of* 10 *and a difference for node* u *of 5, and a reference length of* 7 *and a difference for node* u *of* 1*:*

$$\begin{aligned} &\max((10, \{u \to 5\}), (7, \{u \to 1\})) \\ &\quad = (\max(10,7), \mathrm{rel}_{\delta\times\Delta}(\max(10,7), \max_\Delta(\mathrm{abs}_{\delta\times\Delta}(10, \{u \to 5\}), \mathrm{abs}_{\delta\times\Delta}(7, \{u \to 1\})))(u)) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \max_\Delta(\mathrm{abs}_{\delta\times\Delta}(10, \{u \to 5\}), \mathrm{abs}_{\delta\times\Delta}(7, \{u \to 1\})))(u)) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \max_\Delta(\{u \to 5\}, \{u \to 6\}))(u)) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \{u \to \max(5,6)\})(u)) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \{u \to 6\})(u)) \\ &\quad = (10, \{u \to 4\}) \end{aligned}$$

In a second example, we assume a missing mapping and relative infeasibility:

$$\begin{aligned} &\max((10, \emptyset), (7, \{u \to \perp\})) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \max_\Delta(\mathrm{abs}_{\delta\times\Delta}(10, \emptyset), \mathrm{abs}_{\delta\times\Delta}(7, \{u \to \perp\})))(u)) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \max_\Delta(\{u \to 10\}, \{u \to \perp\}))(u)) \\ &\quad = (10, \mathrm{rel}_{\delta\times\Delta}(10, \{u \to 10\})(u)) \\ &\quad = (10, \{u \to 0\}) = (10, \emptyset) \end{aligned}$$

Note that we can drop mappings that yield no difference in length.

It remains to define addition. We first define operator $+_\Delta$, which adds differences as:

$$
\begin{aligned}
+_\Delta : \mathbb{D}^\Delta \times \mathbb{D}^\Delta \mapsto \mathbb{D}^\Delta \\
d +_\Delta d' = \lambda u \,.\, \begin{cases} d(u) +_{\mathbb{Z}^{\infty,\perp}} d'(u) & \text{if } u \in \operatorname{def}(d) \cap \operatorname{def}(d') \\ d(u) & \text{if } u \in \operatorname{def}(d) \\ d'(u) & \text{otherwise} \end{cases}
\end{aligned} \tag{5.170}
$$

Recall that the absence of mappings denotes equality of the absolute path length to the reference length. Then addition on $\mathbb{Z}^{\infty,\perp}$ is defined as:

$$
\begin{aligned}
+ : \mathbb{D}^{\delta\times\Delta} \times \mathbb{D}^{\delta\times\Delta} \mapsto \mathbb{D}^{\delta\times\Delta} \\
(l, d) + (l', d') = \big(l +_{\mathbb{Z}^{\infty,\perp}} l', \big\{(d +_\Delta d')(u) \;\big|\; u \in \operatorname{def}(d) \cup \operatorname{def}(d')\big\}\big)
\end{aligned} \tag{5.171}
$$

The neutral element is $1_+ = \{0, \emptyset\}$. It is easy to see why we can avoid translation to absolute distances and back to differences for addition.

Example *Without loss of generality, assume a pair $(r, d) \in \mathbb{N}^2$ to denote a reference length and a difference, and an operator $+_{\mathbb{N}^2}$, which is defined akin to above but adapted to the given domain. Then the following equation holds:*

$$
\begin{aligned}
(n, d) +_{\mathbb{N}^2} (m, e) &= (n + m, (n + m) - ((n - d) + (m - e))) \\
&= (n + m, n + m - (n + m - d - e)) \\
&= (n + m, d + e)
\end{aligned}
$$

Finally, we lift computation of maximal lengths to path states of the *same* equivalence class and define:

$$
\max_\pi^{\text{mbt}}(S) = \max_{s' \in S} \Big(\delta_\pi^d(s'), \Delta_\pi^d(s')\Big) \cdot \Big(\sigma_\pi^d(s), o_\pi^d(s)\Big) : s \in S \tag{5.172}
$$

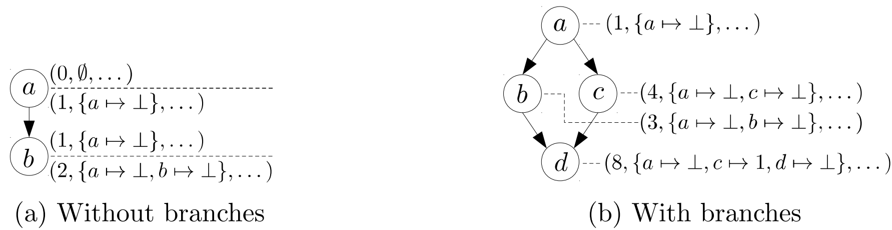
For an equivalence class $S = [s]$, $\max_\pi^{\text{let}}([s])$ denotes a reference path of maximal length and a set of differences from this path.

5.3.6.2 Framework

In the following we define the analysis framework for MBT. We first show how iterations are computed, then we extend semantics to complete unrolls. Finally, we discuss how MBT are computed to all interior points.

Intuition

We shall briefly outline the intuition. As usual, every path state denotes a path. Ultimately, we compose these states to compute a globally longest path. The issue at hand is to compose longest paths to individual nodes only from those subpaths that are feasible



(a) Without branches (b) With branches

Figure 5.58: Example computations for MBT

relative each particular sink node. For efficiency, we exploit that an individual subpath is often feasible for a number of sinks such that we can avoid unnecessary recomputation. Length map Δ_π^d fulfills two purposes in this: to mark paths as infeasible for composition with respect to specific sinks and, if the reference path is infeasible but there exists an alternative feasible but potentially shorter path, to encode their lengths as differences from the reference.

Example *Figure 5.58 illustrates example scenarios for this intuition. Figure 5.58a depicts a trivial path where tuples above and below the dashed lines denote path states just before and just after considering respective nodes. We assume unit node weights and ignore signature and origin. Once node a is visited, reference length is increased and the path is marked as being infeasible for composition (unroll) to form longer paths having node a only as their terminal node. Likewise for node b.*

Figure 5.58b illustrates a scenario with alternative paths. We assume node weights $\omega = \{a \to 1, b \to 2, c \to 3, d \to 4\}$. Path states now denote state after visiting respective nodes. As before, visiting node a marks the path infeasible and increases the reference path length. Likewise, this happens to nodes b and c. The path state after visiting node d then encodes the following information: The reference (unconditional worst-case) path length to node d equals 8. For all paths from node a to d, it holds that they are infeasible with respect to nodes a and d. With respect node b, the reference path coincides with the longest path that does not pass through node b. Hence, no difference has to be stored explicitly. With respect to node c, there exists a feasible path that does not pass through node c, but which is shorter than the reference.

More precisely the intuition is as follows: For an equivalence class of path states $[s_\pi^d]$, s_π^d denotes the unconditionally longest path of length $\delta_\pi^d(s_\pi^d)$ and map $\Delta_\pi^d(s_\pi^d)$ either encodes lengths of alternative paths in $[s_\pi^d]$ or marks state s_π^d as being infeasible with respect to individual nodes if no alternatives exist.

We refer to nodes mapped in Δ_π^d as being *known*, otherwise as *unknown*, and we refer to a path which is infeasible regarding a known node as being *relatively infeasible.*

Note that in the following, we implicitly purge differences equal to 0 from all sets where appropriate. For the sake of clarity, we never specify this removal explicitly. In practice, this is a simple optimization.

Iterations

In the following we define the computation of scope iterations. The problem of computing longest path lengths of the domain $\mathbb{D}^{\delta\times\Delta}$ has the structure of the semi-lattice defined as:

$$(\mathbb{D}_{mbt}, \top, \sqsubseteq, \sqcup) \tag{5.173}$$

where $\mathbb{D}_{mbt} = \wp(S^d_\pi)$ is a set of path states, where $S^d_\pi \sqsubseteq S^{d\prime}_\pi \Leftrightarrow S^d_\pi \subseteq S^{d\prime}_\pi$ and where $S^d_\pi \sqcup S^{d\prime}_\pi := \{\max^{\text{mbt}}_\pi[s] \mid [s] \in (S^d_\pi \cup S^{d\prime}_\pi)/\overset{A_\pi}{\sim}\}$ denotes the set of path states of different signature, origin and maximal length, according to maximal lengths as defined in Equation 5.172 and the equivalence relation defined in Equation 5.160.

The corresponding transformer to compute iterations advances the reference path length δ^d_π, marks the current node infeasible and updates annotations as usual. We define it as:

$$\begin{aligned} \text{tf}_{mbt}\colon V &\mapsto (\mathbb{D}_{mbt} \mapsto \mathbb{D}_{mbt}) \\ \text{tf}_{mbt}(u) &= \lambda S\,.\left\{ \begin{pmatrix} \delta^d_\pi(s^d_\pi) + \omega(u), \\ \Delta^d_\pi(s^d_\pi)[u \to \bot], \\ \sigma_\pi(s^d_\pi)\cdot\alpha_\pi(u), \\ o_\pi(s^d_\pi) \end{pmatrix} \middle|\; s \in S \right\} \end{aligned} \tag{5.174}$$

Although path states now potentially represent multiple paths (of different length but equal signature), due to difference encoding it is sufficient to update only the reference length.

As before, we compute maximal distances "across" scopes. Let maximal distances on $\mathbb{D}^{\delta\times\Delta}$ for a given scope including its descendants be denoted by the function:

$$\max^{\mathring{s}}_{\mu^{\text{mbt}}}\colon \mathring{V} \times V^2 \mapsto \mathbb{D}^{\delta\times\Delta} \tag{5.175}$$

such that $\max^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{s}, u, v)$ evaluates to maximal unroll length in scope $\mathring{s}$ from node u to node v.

Now we lift tf_{mbt} to take subscopes into account. To this end, we first define an additional transformer tf^{out}_{let} to model semantics upon reaching exits of subscopes. As usual, we update path lengths by unroll lengths of respective subscopes. Recall that length is denoted by the pair of reference and differences. Consequently, we define:

$$\begin{aligned} &\text{tf}^{out}_{mbt}\colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{mbt} \mapsto \mathbb{D}_{mbt}) \\ &\text{tf}^{out}_{mbt}(\mathring{s}, u, v) = \lambda S. \\ &\quad \left\{ \left(\left(\delta^d_\pi(s^d_\pi), \Delta^d_\pi(s^d_\pi) \right) + \max^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{s}, u, v) \right) \cdot \left(\sigma^d_\pi(s^d_\pi), o^d_\pi(s^d_\pi) \right) \;\middle|\; s^d_\pi \in S \right\} \end{aligned} \tag{5.176}$$

Finally, we define the scope-aware transformer (cf. Equation 5.98 on page 149) to compute iteration lengths as:

$$\mathrm{tf}^{\mathring{s}}_{mbt} \colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{mbt} \mapsto \mathbb{D}_{mbt})$$

$$\mathrm{tf}^{\mathring{s}}_{mbt}(\mathring{s}, u, v) = \lambda S \,.\begin{cases} id & \text{if } v \in \mathrm{entry}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}^{out}_{mbt}(\mathring{t}, u, v)(S) & \text{if } v \in \mathrm{exit}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}_{mbt}(u)(S) & \text{otherwise} \end{cases} \tag{5.177}$$

For a node u and its predecessor v, if v is a scope entry then do nothing. If v is a scope exit then we compute a maximal unroll of the subscope and update path states accordingly. Otherwise tf_{let} applies. Function tf^{out}_{let} updates the length mapping accordingly. Note that for $(r, d) = \max^{\mathring{s}}_{\mu^{\mathrm{mbt}}}(\mathring{t}, u, v)$, it holds that $r = \max^{\mathring{s}}_{\mu}(\mathring{t}, u, v)$ (cf. Equation 5.105 on page 151), which simply denotes the standard WCET (cf. Section 5.3.3). Map d denotes length differences from the reference unroll for individual nodes. If no such unroll exists, marks the reference unroll infeasible for each such node.

In scopes representing acyclic regions, differences or infeasibility markings need not be maintained since direct paths are the only ones to reach individual nodes. Differences and markings are only relevant if paths are subject to composition (unroll). In other words, the additional information is only relevant within loops. For clarity, we do not take this optimization into account in the formal framework definition but exploit this in the reference implementation for evaluation.

Analogously to the path analysis frameworks proposed earlier, we define path states for specific program points. Assuming $\mathrm{pred}^{\rightarrow}_{\mathring{s}}$ (Equation 5.99 on page 150) to denote CFG predecessors such that subscopes are being "leaped over", we define path states in a program point by:

$$S^{\mathring{s}}_{mbt} \colon \mathring{V} \times V^2 \times A_\pi \mapsto S^d_\pi$$

$$S^{\mathring{s}}_{mbt}(\mathring{s}, s, v, a) = \begin{cases} \mathrm{tf}^{\mathring{s}}_{mbt}(\mathring{s}, s, s)(\{(0, \emptyset, a, s)\}) & \text{if } v = s \\ \bigsqcup \left\{ \mathrm{tf}^{\mathring{s}}_{mbt}(\mathring{s}, u, v)(S^{\mathring{s}}_{mbt}(\mathring{s}, s, u, a)) \mid u \in \mathrm{pred}^{\rightarrow}_{\mathring{s}}(\mathring{s})(v) \right\} & \text{otherwise} \end{cases} \tag{5.178}$$

For a given scope $\mathring{s}$, start and terminal nodes u, v and an initial annotation a, $S^{\mathring{s}}_{mbt}(\mathring{s}, u, v, a)$ denotes the set of maximal iterations including unrolls of all subscopes.

We define partitions of $S^{\mathring{s}}_{mbt}$ as in the WCET case (cf. Section 5.3.3.2) such that

$$S^I_{mbt} \colon \mathring{V} \times V \mapsto \wp(S^d_\pi) \tag{5.179}$$

$$S^O_{mbt} \colon \mathring{V} \times V \mapsto \wp(S^d_\pi) \tag{5.180}$$

$$S^K_{mbt} \colon \mathring{V} \mapsto \wp(S^d_\pi) \tag{5.181}$$

$$S^D_{mbt} \colon \mathring{V} \times V^2 \mapsto \wp(S^d_\pi) \tag{5.182}$$

denote entry, exit, kernel and direct paths, respectively.

Unrolls

It remains to define $\max^{\mathring{s}}_{\mu^{\text{mbt}}}$ (Equation 5.175) to denote unroll lengths. Given the algebra on $\mathbb{D}^{\delta\times\Delta}$, unrolling is conceptually similar to the standard WCET case from Section 5.3.3. But, care has to be taken to correctly account for the changed path length representation.

We keep test_f (Equation 5.70) and set_f (Equation 5.71) unchanged but redefine functions f^1_ω (Equation 5.79) and f^k_ω (Equation 5.80) to accommodate to domain S^d_π.

As before, function f^1_ω pushes unit admissible flow over a network path denoted by σ^d_π and accumulates path lengths, which are now expressed as pairs of reference length δ^d_π and differences Δ^d_π. It is defined as:

$$f^1_{\delta,\Delta} := \lambda s \,.\, \lambda(\omega, d, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \delta^d_\pi(s), \\ d + \Delta^d_\pi(s), \\ \text{set}_f(f, 1, \sigma^d_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \wedge \\ & \quad \text{test}_f(c, f, \sigma^d_\pi(s)) > 0 \\ (\bot, \emptyset, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.183}$$

It is straight-forward to evaluate entry and exit paths. As usual, we just test all candidates, from which we then chose a solution of maximal length. For kernels, evaluation is more intricate. Recall from the WCET case that for evaluation we ordered kernels by descending length to maximize the product of lengths and flows. In the MBT case, difference encoding requires a unique ordering for *each* explicitly known node mapped in Δ^d_π separately: Each such node potentially has a unique set of feasible kernels of individual length. We now show how the difference representation is transformed into a mapping from nodes to sets of potential kernels of absolute length. The purpose is to reduce the problem of unrolling so that we can apply the same techniques as in the WCET-case — for each node individually. Subsequently, we then restore the original representation. In the following we refer to the composition of reference paths as *reference unrolls*.

We first define a transformation t_{wcet}, which transforms a path state in S^d_π to a path state in S_π (cf. Section 5.3.3.1) as:

$$t_{wcet} \colon S^d_\pi \times V \mapsto {S_\pi}^*$$

$$t_{wcet}(s, u) = \begin{cases} \epsilon & \text{if } u \in \text{def}(\Delta^d_\pi(s)) \wedge \\ & \quad \Delta^d_\pi(s) = \bot \\ ((\delta^d_\pi(s) - \Delta^d_\pi(s)(u), \sigma^d_\pi(s), o^d_\pi(s))) & \text{if } u \in \text{def}(\Delta^d_\pi(s)) \wedge \\ & \quad \Delta^d_\pi(s) \neq \bot \\ ((\delta^d_\pi(s), \sigma^d_\pi(s), o^d_\pi(s))) & \text{otherwise} \end{cases} \tag{5.184}$$

For a path state $s \in S^d_\pi$ and a node $u \in V$, $t_{wcet}(s, u)$ evaluates to a *singleton* sequence containing a path state $s_\pi \in S_\pi$. We distinguish three cases: i) If the paths denoted by s^d_π is infeasible relative to node u, the sequence is empty. ii) If there exists an explicit

difference value in map Δ^d_π which does not denote infeasibility, the sequence contains a path state representing absolute path length. iii) If no explicit difference is given, the sequence contains a path state with the reference path length as its absolute path length.

Example *For a path state $s \in S^d_\pi$, let $(\delta^d_\pi(s), \Delta^d_\pi(s)) = (10, \{u \to 3, v \to \bot\})$. Then the following equations hold:*

$$\begin{aligned}
t_{wcet}(s,u) &= ((\delta^d_\pi(s) - \Delta^d_\pi(s)(u), \sigma^d_\pi(s), o^d_\pi(s))) = ((7, \sigma^d_\pi(s), o^d_\pi(s))) \\
t_{wcet}(s,v) &= \epsilon \\
t_{wcet}(s,w) &= ((\delta^d_\pi(s), \sigma^d_\pi(s), o^d_\pi(s))) = ((10, \sigma^d_\pi(s), o^d_\pi(s)))
\end{aligned}$$

Before we compose individual evaluations, we first collect only path states that make up feasible kernels relative to certain nodes. Since path lengths are already known, we also already collect states in a desired order.

Let $V_\top = V \cup \{u_\top\}$ where $u_\top$ is a "dummy" node which serves as a representative for "unknown" nodes in the following. Also let $\sigma_{\delta_\pi} : S_\pi * \mapsto S^*_\pi$ order a sequence of path states in S_π by descending length $\delta_\pi(s_\pi)$. We then define a transformation t_{uniq} which computes an individually ordered sequence of WCET kernel states (of S_π) for each node that is explicitly mapped in a given set of MBT kernel states (of S^d_π) as:

$$t_{uniq} : (V_\top \mapsto {S_\pi}^*) \times \wp(S^d_\pi) \mapsto (V_\top \mapsto {S_\pi}^*)$$

$$t_{uniq}(m, S) = \begin{cases} t_{uniq}\left(\begin{array}{l} \{u \to \sigma_{\delta_\pi}(\pi \cdot t_{wcet}(s,u)) \mid (u \to \pi) \in m\} \\ \quad \cup \left\{ u \to \sigma_{\delta_\pi}(m(u_\top) \cdot t_{wcet}(s,u)) \;\middle|\; \begin{array}{c} u \in \mathrm{def}(\Delta^d_\pi(s)) \\ \wedge\, u \notin \mathrm{def}(m) \end{array} \right\}, \\ S \setminus \{s\} \end{array} \right), s \in S & \text{if } S \neq \emptyset \\ m & \text{otherwise} \end{cases} \tag{5.185}$$

Function t_{uniq} is invoked with an initially empty sequence for reference kernels and path states S^K_{mbt}, denoting all kernels for a given scope $\mathring{s}$:

$$t^K_{uniq} := \lambda \mathring{s} \, . \, t_{uniq}\left(\{u_\top \to \epsilon\}, S^K_{mbt}(\mathring{s})\right) \tag{5.186}$$

Transformation $t_{uniq}(m, S)$ proceeds recursively, one kernel state $s \in S \subseteq S^d_\pi$ at a time, constructing function m which denotes individually ordered WCET state sequences. In each iteration, every existing individual sequence π $((u \to \pi) \in m)$ is extended by an element $(t_{wcet}(s,u))$. This element denotes a WCET path state of absolute length, but only if state s is relatively feasible for node u (by ϵ otherwise). For every node u in the difference map Δ^d_π of state s $(u \in \mathrm{def}(\Delta^d_\pi(s)))$ and for which no individual sequence exists yet $(u \notin \mathrm{def}(m))$, we extend function m by a unique sequence $m(u_\top) \cdot t_{wcet}(s,u)$.

The latter consists of the reference kernel sequence so far, and an individual WCET path state which denotes this specific kernel difference if it is relatively feasible. Note that a new sequence is added regardless of whether its potential constituents turn out to be relatively infeasible. Feasibility testing will be a subsequent step.

Example *Let* $S^K_{mbt}(\mathring{s}) = \{s_1, s_2\}$ *where* $s_1 = (4, \emptyset, \sigma_1, o_1)$ *and* $s_2 = (10, \{u \to 3, v \to \bot\}, \sigma_2, o_2)$*, and where* σ_i, o_i *denote signature and origin respectively. Then evaluation yields:*

$$\begin{aligned} t_{uniq}\left(\{u_\top \to \epsilon\}, \{s_1, s_2\}\right) &= t_{uniq}\left(\{u_\top \to ((4, \sigma_1, o_1))\}, \{s_2\}\right) \\ &= t_{uniq}\left(\left\{\begin{array}{l} u_\top \to ((10, \sigma_2, o_2), (4, \sigma_1, o_1)), \\ u \to ((7, \sigma_2, o_2), (4, \sigma_1, o_1)), \\ v \to ((4, \sigma_1, o_1)) \end{array}\right\}, \emptyset\right) \end{aligned}$$

For completeness, we restate function f^k_ω (Equation 5.80 on page 143) unchanged. Recall that its purpose is to extend path lengths by scaling kernel paths lengths by maximally admissible flow.

$$f^k_\omega := \lambda s \,.\, \lambda(\omega, f, c) \,.\, \begin{cases} \begin{pmatrix} \omega + \delta_\pi(s) \times \mathrm{test}_f(c, f, \sigma_\pi(s)), \\ \mathrm{set}_f(f, \mathrm{test}_f(c, f, \sigma_\pi(s)), \sigma_\pi(s)), \\ c \end{pmatrix} & \text{if } \omega \neq \bot \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.187}$$

Transformation t_{uniq} returns a map of individually ordered sequences of kernel states. We define a function f^K_{uniq} which lifts evaluations f^k_ω to maps of kernel sequences. Effectively, we "wrap" all path states in each sequence with function f^k_ω.

$$f^K_{uniq} := \lambda \mathring{s} \,.\, \left\{ u \to [\![\pi]\!](f^k_\omega) \;\middle|\; (u \to \pi) \in t^K_{uniq}(\mathring{s}) \right\} \tag{5.188}$$

For a scope $\mathring{s}$, $f^K_{uniq}(\mathring{s})$ denotes a map of individual kernel evaluations. Node-wise, this corresponds to f^K_ω (Equation 5.81 on page 143).

For a scope $\mathring{s}$, a known node u and an initial value (ω, f, c), where ω denotes an initial path length, f denotes flow and c denotes capacity bounds, $f^K_{uniq}(\mathring{s})(u)(0, f, c)$ yields a maximal (absolute) path length. Next, we define how evaluation is carried out in general and transform the result back to differences from a reference length.

Let $\omega \in \mathbb{Z}^{\infty,\bot}$ denote the length of a reference unroll. We define a function f^K_{rel}, which performs evaluation for all non-reference kernel sequences $F_\pi = f^K_{uniq}(\mathring{s})(u)$ with $u \neq u_\top$, and returns the length difference to ω as:

$$\begin{aligned} &f^K_{rel} := \lambda \mathring{s} \,.\, \lambda(\omega, f, c). \\ &\quad \left\{ u \to \omega - \omega' \;\middle|\; (\omega', f', c) = F_\pi(0, f, c), (u \to F_\pi) \in f^K_{uniq}(\mathring{s}), u \neq u_\top \right\} \end{aligned} \tag{5.189}$$

Finally, we define a function $f^K_{\delta\times\Delta}$, which is symmetric to $f^1_{\delta\times\Delta}$ (Equation 5.183) as:

$$f^K_{\delta\times\Delta} := \lambda\mathring{s} \,.\, \lambda(\omega, d, f, c). \\ \big((\omega, d) + \big(\omega', f^K_{rel}(\mathring{s})(\omega', f, c) \;\big|\; (\omega', f', c) = f^K_{uniq}(\mathring{s})(u_\top)(0, f, c)\big)\,, f', c\big) \tag{5.190}$$

Function $f^K_{\delta\times\Delta}$ performs evaluation of the reference kernel sequence (denoted by $u_\top$) and composes a pair of reference length ω' and a map of differences $f^K_{rel}(\mathring{s})(\omega')$, and adds this pair to an initial value (ω, d) by $+_{\mathbb{D}^{\delta\times\Delta}}$. It returns a tuple (ω, d, f', c) where (ω, d) denote kernel lengths, f' denotes flow after reserving kernels and c denotes the original capacity constraints. Note that f' is just a placeholder and not used subsequently.

The composition of all evaluations (cf. Equation 5.82 on page 143) for a scope $\mathring{s}$, from node u to node v follows the familiar pattern and we define the set of all composed evaluations as:

$$F^{iok}_{\delta\times\Delta} := \lambda(\mathring{s}, u, v). \\ \big\{f^K_{\delta\times\Delta}(\mathring{s}) \circ f^1_{\delta\times\Delta}(s_o) \circ f^1_{\delta\times\Delta}(s_i) \;\big|\; s_i \in S^I_{mbt}(\mathring{s}, u), s_o \in S^O_{mbt}(\mathring{s}, v)\big\} \tag{5.191}$$

It remains to define evaluation for direct paths. The set of all evaluations of direct paths for a scope $\mathring{s}$ from node u to node v, similar to F^d_ω (Equation 5.83), is defined as:

$$F^d_{\delta\times\Delta} := \lambda(\mathring{s}, u, v) \,.\, \big\{f^1_{\delta\times\Delta}(s_d) \;\big|\; s_d \in S^D_{mbt}(\mathring{s}, u, v)\big\} \tag{5.192}$$

Computing maximal unroll distances is carried out as usual. We compute the set of solutions and determine its maximum (Equation 5.169):

Theorem 5.79 (Maximum Blocking Time Unroll) *Let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then the maximal unroll length for a scope $\mathring{s} \in \mathring{V}$, from node $u \in V$ to node $v \in V$ is defined as:*

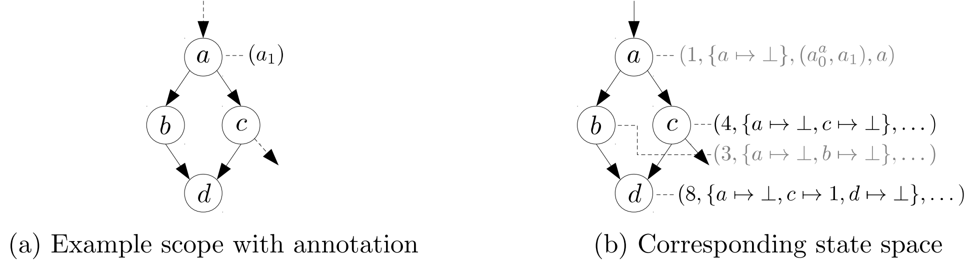
$$\max\nolimits^{\mathring{s}}_{\mu^{\text{mbt}}} : \mathring{V} \times V^2 \mapsto \mathbb{D}^{\delta\times\Delta}$$

$$\max\nolimits^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{s}, u, v) = \max\left\{(\omega, d) \;\middle|\; \begin{array}{l} (\omega, d, f, \beta_\pi) = f_{\delta\times\Delta}(0, \emptyset, f_0, \beta_\pi), \\ f_{\delta\times\Delta} \in F^{iok}_{\delta\times\Delta}(\mathring{s}, u, v) \cup F^d_{\delta\times\Delta}(\mathring{s}, u, v) \end{array}\right\} \tag{5.193}$$

This represents the precise maximally blocking path lengths for a scope and a pair of source and sink nodes, under the given flow bound model.

It is easy to see but lengthy to show that $\text{abs}_{\delta\times\Delta}(\omega, d)$ denotes unique maximal unrolls for each scope member node. For worst-case path lengths in general, see Section 5.3.3.

Example *Consider Figure 5.59a which illustrates an example scope $\mathring{s}$ with entry a, exit c and a single annotation in node a. Figure 5.59b depicts the corresponding states space where only the non-faded states are of relevance in the following (cf. example forx Figure 5.58 for details on state space construction). We assume $\beta_\pi(a_1) = 3$ and node weight $\omega = \{a \to 1, b \to 2, c \to 3, d \to 4\}$ and compute MBT from node a to node c with*



(a) Example scope with annotation (b) Corresponding state space

Figure 5.59: Example of path states for MBT computation

$\max^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{s}, a, c)$. *Partitions of states* $S^{\mathring{s}}_{mbt}$ *yield:*

$$\begin{aligned} S^I_{mbt}(\mathring{s}, a) &= \{s_i\} = \{(8, \{a \to \bot, c \to 1, d \to \bot\}, \dots)^d_\pi\} \\ S^O_{mbt}(\mathring{s}, c) &= \{s_o\} = \{(4, \{a \to \bot, c \to \bot\}, \dots)^d_\pi\} \\ S^K_{mbt}(\mathring{s}) &= \{s_k\} = \{s_i\} \\ S^D_{mbt}(\mathring{s}, a, c) &= \{s_d\} = \{s_o\} \end{aligned}$$

We will consistently assume that "...” denotes "$(a_0^a, a_2), a$" (signature, origin) for all path states. We also write "$(\dots)^d_\pi \in S^d_\pi$" and "$(\dots)_\pi \in S_\pi$" to indicate tuples representing path states.

Composition of evaluation semantics yields:

$$F^{iok}_{\delta\times\Delta}(\mathring{s}, a, c) \cup F^{d}_{\delta\times\Delta}(\mathring{s}, a, c) = \left\{ f^K_{\delta\times\Delta}(\mathring{s}) \circ f^1_{\delta\times\Delta}(s_o) \circ f^1_{\delta\times\Delta}(s_i) \right\} \cup \left\{ f^1_{\delta\times\Delta}(s_d) \right\}$$

Let $f_0 = A_\pi \times \{0\}$ *denote initial flow. Then evaluation of just the direct path yields:*

$$f^1_{\delta\times\Delta}(s_d)(0, \emptyset, f_0, \beta_\pi) = (4, \{a \to \bot, c \to \bot\}, f^d, \beta_\pi)$$

Note that this result implicitly denotes feasible direct paths of length 4 *for nodes* b, d*. Evaluation of the only possible unroll from node* a *to node* c *yields:*

$$\begin{aligned} &(f^K_{\delta\times\Delta}(\mathring{s}) \circ f^1_{\delta\times\Delta}(s_o) \circ f^1_{\delta\times\Delta}(s_i))(0, \emptyset, f_0, \beta_\pi) \\ &\quad = (\dots \circ f^1_{\delta\times\Delta}(s_i))(0, \emptyset, f_0, \beta_\pi) && \text{(entry)} \\ &\quad = (\dots \circ f^1_{\delta\times\Delta}(s_o))(8, \{a \to \bot, c \to 1, d \to \bot\}, f^i, \beta_\pi) && \text{(exit)} \\ &\quad = f^K_{\delta\times\Delta}(\mathring{s})(8+4, \{a \to \bot, c \to 1, d \to \bot\} + \{a \to \bot, c \to \bot\}, f^{oi}, \beta_\pi) \\ &\quad = f^K_{\delta\times\Delta}(\mathring{s})(12, \{a \to \bot, c \to \bot, d \to \bot\}, f^{oi}, \beta_\pi) && \text{(kernels)} \\ &\quad = \textit{(continued below)} \end{aligned}$$

Note again that there implicitly exists a feasible entry/exit combination for node b *of length* 12*.*

It remains to evaluate kernels after we have now already taken entry and exit paths into account. To make this more approachable, we dissect the invocation of $f^K_{\delta\times\Delta}$ *(Equation 5.190) and perform computations bottom up to keep noise to a minimum. First, let*

us compute $t^K_{uniq}(\mathring{s})$ (Equation 5.186) to determine individual path state sequences:

$$\begin{aligned} t^K_{uniq}(\mathring{s}) &= t_{uniq}(\{u_t \to \epsilon\}, S^K_{mbt}(\mathring{s})) && \text{(Eq. 5.186)} \\ &= t^K_{uniq}(\mathring{s}) = t_{uniq}(\{u_t \to \epsilon\}, \{(8, \{a \to \bot, c \to 1, d \to \bot\}, \dots)^d_\pi\} \\ &= \left\{ \begin{array}{l} u_\top \to ((8,\dots)_\pi), a \to \epsilon, b \to ((8,\dots)_\pi), \\ \quad c \to ((7,\dots)_\pi), d \to \epsilon \end{array} \right\} && \text{(Eq. 5.185)} \end{aligned}$$

Possible kernels exist only with respect to nodes $u_\top, b, c$, where $u_\top$ denotes the constraint reference. We convert this presentation from path state sequences to evaluation sequences with f^K_{uniq} such that:

$$f^K_{uniq}(\mathring{s}) = \left\{ \begin{array}{l} u_\top \to f^k_\omega((8,\dots)_\pi), a \to (\text{id}), b \to f^k_\omega((8,\dots)_\pi), \\ \quad c \to f^k_\omega((7,\dots)_\pi), d \to (\text{id}) \end{array} \right\} \qquad \text{(Eq. 5.188)}$$

where f^k_ω (Equation 5.187) denotes evaluation of a single relatively feasible kernel.

Now that we have obtained possible evaluation sequences, we step-wise partially evaluate Equation 5.190. First, we compute the reference unroll:

$$\begin{aligned} f^K_{uniq}(\mathring{s})(u_\top)(0, f^{oi}, \beta_\pi) &= f^k_\omega((8,\dots)_\pi)(0, f^{oi}, \beta_\pi) \\ &= (8, f^{iok}, \beta_\pi) \end{aligned}$$

Apparently, the reference kernel can only be repeated once after having reserved flow f^{io} for entry and exit paths already. Second, we evaluate all remaining individual sequences.

$$\begin{aligned} &\left(\omega', f^K_{rel}(\mathring{s})(\omega', f, c) \;\middle|\; (\omega', f', c) = (8, f^{iok}, \beta_\pi)\right) \\ &= \left(8, f^K_{rel}(\mathring{s})(8, f, c)\right) \\ &= \left(8, \left\{ \begin{array}{l} a \to 8 - \omega' \mid (\omega', f^{iok}, c) = f^K_{uniq}(\mathring{s})(a)(0, f^{io}, c), \\ b \to 8 - \omega' \mid (\omega', f^{iok}, c) = f^K_{uniq}(\mathring{s})(b)(0, f^{io}, c), \\ c \to 8 - \omega' \mid (\omega', f^{iok}, c) = f^K_{uniq}(\mathring{s})(a)(0, f^{io}, c), \\ d \to 8 - \omega' \mid (\omega', f^{iok}, c) = f^K_{uniq}(\mathring{s})(d)(0, f^{io}, c), \end{array} \right\}\right) \\ &= \left(8, \left\{ \begin{array}{l} a \to 8 - \omega' \mid (\omega', f^{iok}, c) = \text{id}(0, f^{oi}, \beta_\pi), \\ b \to 8 - \omega' \mid (\omega', f^{iok}, c) = f^k_\omega((8,\dots)_\pi)(0, f^{oi}, \beta_\pi), \\ c \to 8 - \omega' \mid (\omega', f^{iok}, c) = f^k_\omega((7,\dots)_\pi)(0, f^{oi}, \beta_\pi), \\ d \to 8 - \omega' \mid (\omega', f^{iok}, c) = \text{id}(0, f^{oi}, \beta_\pi) \end{array} \right\}\right) \\ &= \left(8, \left\{ \begin{array}{l} a \to 8 - 0, b \to 8 - 8, \\ \quad c \to 8 - 7, d \to 8 - 0 \end{array} \right\}\right) = (8, \{a \to 8, c \to 1, d \to 8\}) \end{aligned}$$

This result denotes differences in unroll length for all known nodes and therefore completely summarizes unrolling. There exists no relatively feasible kernels for nodes a and d (absolute length equals 0), and for nodes b and c absolute kernel unroll lengths equal 8

and 7*, respectively. Recall that we implicitly purge differences equal to* 0*.*

Finally, we can fully state the evaluation of f^K_{uniq} *(Equation 5.190). We resume our previous computation:*

$$
\begin{aligned}
&\textit{(continued)} \\
&= f^K_{\delta\times\Delta}(\mathring{s})(12, \{a \to \bot, c \to \bot, d \to \bot\}, f^{oi}, \beta_\pi) && \text{(kernels)} \\
&= (12, \{a \to \bot, c \to \bot, d \to \bot\}) + (8, \{a \to 8, c \to 1, d \to 8\}) \\
&= (20, \{a \to \bot, c \to \bot, d \to \bot\}) && \text{(Eq. 5.171)}
\end{aligned}
$$

Reference unroll length equals 20*, and node* b *is the only node for which a feasible unroll of equal length exists.*

Finally, we can compute the MBT unroll according to Theorem 5.79. We already computed results for direct paths and unrolls. Hence, $\max^{\mathring{s}}_{\mu^{\mathrm{mbt}}}(\mathring{s}, a, c)$ *(Equation 5.193) reduces to:*

$$
\max \left\{ \begin{array}{l} (20, \{a \to \bot, c \to \bot, d \to \bot\}), \\ (4, \{a \to \bot, c \to \bot\}) \end{array} \right\} = (20, \{a \to \bot, c \to \bot, d \to 16\}) \quad \text{(Eq. 5.169)}
$$

Nodes a *and* c *remain infeasible in all combinations, node* b *still has a maximal path length equal to the reference, and node* d *has a feasible direct path of length* 4*, which equals a difference of* 16 *to the reference.*

Note that in practice, exploiting path subsumption or sparse unrolling as proposed in Section 5.3.3.2 on page 133 potentially severely affects performance. Note also that t_{uniq} (Equation 5.185) only needs to be computed once per scope.

MBT to Interior Points

When computing total path lengths from and to individual nodes, care has to be taken to account for global start and terminal nodes correctly. Recall that we assume that preemption points take effect right after each program point. Consequently, after preemption, execution resumes accordingly. The practical consequence is that designated preemption points cut the graph, effectively not allowing for any path state to propagate "across" these points.

For MBT analysis, if a node denotes a preemption point, we have to distinguish whether it represents a point of preemption or a point of resumption. Accordingly, it is either a global source node or a global terminal node for path analysis. For the sake of simplicity of the following discussion, we assume the global CFG entry node to invariably denote the global source node. Extension to subgraphs follows the same pattern as already proposed for the WCET case in Section 5.3.3.4.

Recall that transformer tf_{mbt} (Equation 5.174) marks all paths through a node as relatively infeasible. For a global sink (or source) node this invariably marks all paths to it as relatively infeasible — which is obviously unfortunate. Consequently, for the

computation of total path lengths to individual interior nodes, we have to take this into account.

To this end, we assume a set of path states

$$S^{O,T}_{mbt} : \mathring{V} \times V \mapsto \wp(S^d_\pi) \tag{5.194}$$

to represent *terminal exit path states* similar to usual exit path states S^O_{mbt}, except that $S^{O,T}_{mbt}(\mathring{s}, u)$ denotes states such that only node weight and annotations are applied according to tf_{mbt} (Equation 5.174) but paths are *not marked relatively infeasible* regarding node u. Analogously, we assume a set of *terminal direct path states*:

$$S^{D,T}_{wcet} : \mathring{V} \times V \times V \mapsto \wp(S_\pi) \tag{5.195}$$

similar to S^D_{mbt} such that $S^{D,T}_{wcet}(\mathring{s}, u, v)$ denotes path states in scope $\mathring{s}$ from node u such that tf_{mbt} is only applied partially similar to $S^{O,T}_{mbt}$.

For the sake of completeness, to adapt to these new state sets, we restate existing definitions. Otherwise, semantics remain unchanged.

We define a new set of unroll evaluations $F^{itk}_{\delta\times\Delta}$, which is similar to $F^{iok}_{\delta\times\Delta}$ (Equation 5.191), except for the replacement of S^O_{mbt} by $S^{O,T}_{mbt}$ as:

$$\begin{aligned} F^{itk}_{\delta\times\Delta} &:= \lambda(\mathring{s}, u, v). \\ &\left\{ f^K_{\delta\times\Delta}(\mathring{s}) \circ f^1_{\delta\times\Delta}(s_o) \circ f^1_{\delta\times\Delta}(s_i) \;\middle|\; s_i \in S^I_{mbt}(\mathring{s}, u), s_o \in S^{O,T}_{mbt}(\mathring{s}, v) \right\} \end{aligned} \tag{5.196}$$

Analogously, we define direct path evaluations similar to $F^d_{\delta\times\Delta}$ (Equation 5.192) as:

$$F^t_{\delta\times\Delta} := \lambda(\mathring{s}, u, v) \,.\, \left\{ f^1_{\delta\times\Delta}(s_t) \;\middle|\; s_t \in S^{D,T}_{mbt}(\mathring{s}, u, v) \right\} \tag{5.197}$$

Unsurprisingly, we also redefine $\max^{\mathring{s}}_{\mu^{\mathrm{mbt}}}$ (Equation 5.193) to adapt to these changes. Hence, let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then the maximal unroll length for a scope $\mathring{s} \in \mathring{V}$, from node $u \in V$ to *right before* node $v \in V$ is defined as:

$$\begin{gathered} \max^{\mathring{s},\mathrm{T}}_{\mu^{\mathrm{mbt}}} : \mathring{V} \times V^2 \mapsto \mathbb{D}^{\delta\times\Delta} \\ \max^{\mathring{s},\mathrm{T}}_{\mu^{\mathrm{mbt}}}(\mathring{s}, u, v) = \max \left\{ (\omega, d) \;\middle|\; \begin{array}{l} (\omega, d, f, \beta_\pi) = f_{\delta\times\Delta}(0, \emptyset, f_0, \beta_\pi), \\ f_{\delta\times\Delta} \in F^{itk}_{\delta\times\Delta}(\mathring{s}, u, v) \cup F^t_{\delta\times\Delta}(\mathring{s}, u, v) \end{array} \right\} \end{gathered} \tag{5.198}$$

In Section 5.3.3.4 we addressed the problem of computing total WCET path lengths to interior nodes. For MBT, the general idea applies almost unchanged, except for the consideration of terminal path states. Intuitively, terminal states as defined above are only relevant in the scope to which the global terminal node is mapped to. Consequently, all unroll computations except for those of this scope are unaffected. Put differently, the total length we compute corresponds to a global path having the designated sink node only as its terminal. Hence, no subpath is permitted to pass through the sink, except for

the terminal exit path.

We define a helper function, which conditionally returns either $\max^{\mathring{s},T}_{\mu^{\text{mbt}}}$ (Equation 5.198) or $\max^{\mathring{s}}_{\mu^{\text{mbt}}}$ (Equation 5.193) as:

$$\text{cmax}^{\mathring{s}}_{\mu^{\text{mbt}}} := \lambda(\mathring{s}, u)\,.\begin{cases}\max^{\mathring{s},T}_{\mu^{\text{mbt}}} & \text{if } \mathring{s} = \mathring{\gamma}(u)\\ \max^{\mathring{s}}_{\mu^{\text{mbt}}} & \text{otherwise}\end{cases} \tag{5.199}$$

Then finally, we define a function $\max^{\star}_{\mu^{\text{mbt}}}(\mathring{s}, s, \mathring{t}, t)$, similar to $\max^{\star}_{\mu}$ (cf. Equation 5.106 on page 152) for the WCET case. It computes the MBT path length from source scope $\mathring{s}$ and node s to a target scope $\mathring{t}$ and terminal node t, by computing the MBT path length to all entries of $\mathring{t}$ and the MBT path length within $\mathring{t}$ to t as:

$$\max^{\star}_{\mu^{\text{mbt}}} : \mathring{S} \times V \times \mathring{S} \times V \mapsto \mathbb{D}^{\delta\times\Delta}$$

$$\max^{\star}_{\mu^{\text{mbt}}}(\mathring{s}, s, \mathring{t}, u) =$$
$$\begin{cases}\max\left\{\begin{array}{c} \max^{\star}_{\mu^{\text{mbt}}}(\mathring{s}, s, \mathring{u}, i) \\ \quad + \text{cmax}^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{t}, u)(\mathring{t}, i, u)\end{array}\;\middle|\; \begin{array}{l} i \in \text{entry}(\mathring{s}),\\ \mathring{u} = \text{par}(\mathring{t})\end{array}\right\} & \text{if } \mathring{s} \neq \mathring{t} \\ \text{cmax}^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{s}, u)(\mathring{s}, s, u) & \text{otherwise}\end{cases} \tag{5.200}$$

where for $(\omega, d) = \max^{\star}_{\mu^{\text{mbt}}}(\mathring{s}, s, \mathring{t}, t)$, ω denotes a bound on the WCET and $d(t)$ denotes a bound on the MBT regarding node t.

Example *We reconsider the example from the introduction as illustrated in Figure 5.57. We refer to the implicit outermost most scope as* $\mathring{0}$ *and to the inner scope as* $\mathring{1}$*. Further, we assume flow bound* $\beta_\pi(a_0) = 2$ *and node weights* $\omega(a) = 5, \omega(b) = 4, \omega(c) = 3, \omega(d) = 2, \omega(e) = 1$*, denoted by the labels next to the nodes.*

As a first example, we compute an MBT bound from node a *to node* e*. It is immediately apparent that only direct paths lead to node* e*. Hence, computation yields:*

$$\begin{aligned}\max^{\star}_{\mu^{\text{mbt}}}(\mathring{0}, a, \mathring{1}, e) &= \max\left\{\max^{\star}_{\mu^{\text{mbt}}}(\mathring{0}, a, \mathring{0}, b) + \text{cmax}^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{1}, e)(\mathring{1}, b, e)\right\}\\ &= \text{cmax}^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{0}, b)(\mathring{0}, a, b) + \text{cmax}^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{1}, e)(\mathring{1}, b, e)\\ &= \max^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{0}, a, b) + \max^{\mathring{s},T}_{\mu^{\text{mbt}}}(\mathring{1}, b, e)\\ &= (5, \{a \to \bot\}) + (4 + 3 + 1, \{b \to \bot, c \to 1\})\\ &= (13, \{a \to \bot, b \to \bot, c \to 1\}) = (\omega, d)\\ &\Rightarrow d(e) = 13\end{aligned}$$

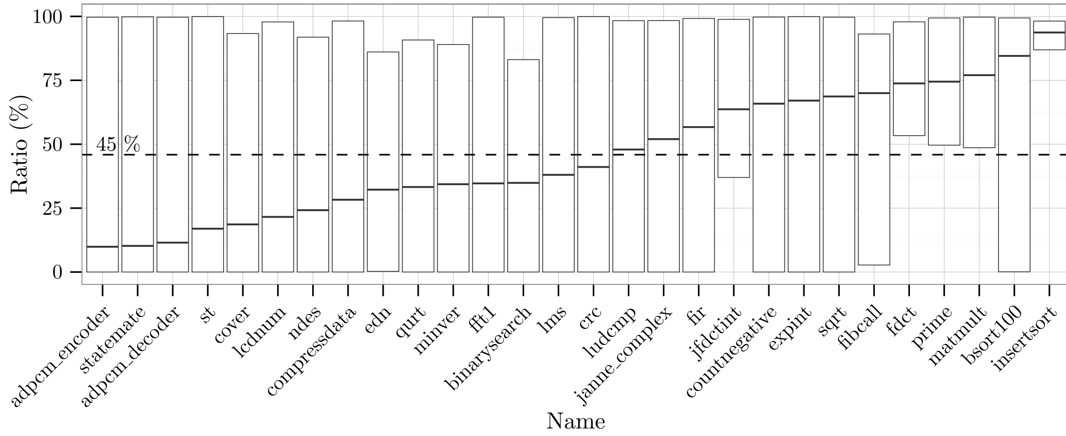


Figure 5.60: Improvement (%) in precision of MBT over WCET estimates

As a second example, computing MBT path length from node a to node c yields:

$$\begin{aligned}
\max^{\star}_{\mu^{\text{mbt}}}(\mathring{0}, a, \mathring{1}, c) &= \max^{\mathring{s}}_{\mu^{\text{mbt}}}(\mathring{0}, a, b) + \max^{\mathring{s},\text{T}}_{\mu^{\text{mbt}}}(\mathring{1}, b, c) \\
&= (5, \{a \to \bot\}) + ((8, \{b \to \bot, c \to 1, e \to \bot\}) + (7, \{b \to \bot\})) \\
&= (20, \{a \to \bot, b \to \bot, c \to 1, e \to \bot\}) = (\omega, d) \\
&\Rightarrow d(c) = 19
\end{aligned}$$

Here, the unroll is composed of an entry path "around" terminal node c and its terminal path.

5.3.6.3 Evaluation

In the following evaluation, we compare MBT against WCET analysis from Section 5.3.3 and against the ILP model as proposed in [173]. We compute WCET and MBT from the CFG entry to all program points, whereas for the ILP, only computations to a single dedicated sink node is possible; we chose the CFG exit as sink node. As test environment, we use a setup as described in Section 5.3.3.6. We evaluate precision by comparison on benchmarks from the MRTC benchmark suite [97] and performance by means of randomized graphs to obtain large sample sets. Averages are computed by the arithmetic mean. The implementation is a derivative of the WCET reference algorithm (cf. Section 5.3.3.5) with all previously proposed optimizations included.

MRTC

We evaluate on a subset of MRTC benchmarks to demonstrate the benefits of MBT over WCET on existing scenarios. Figure 5.60 illustrates the ratio of precision between WCET (WCET) and MBT (MBT) analyses for the given benchmarks as box plots depicting upper, lower and average ratio values. More precisely, the ratios denote the difference in WCET estimates per program point in loops (other program points cannot have deviating time

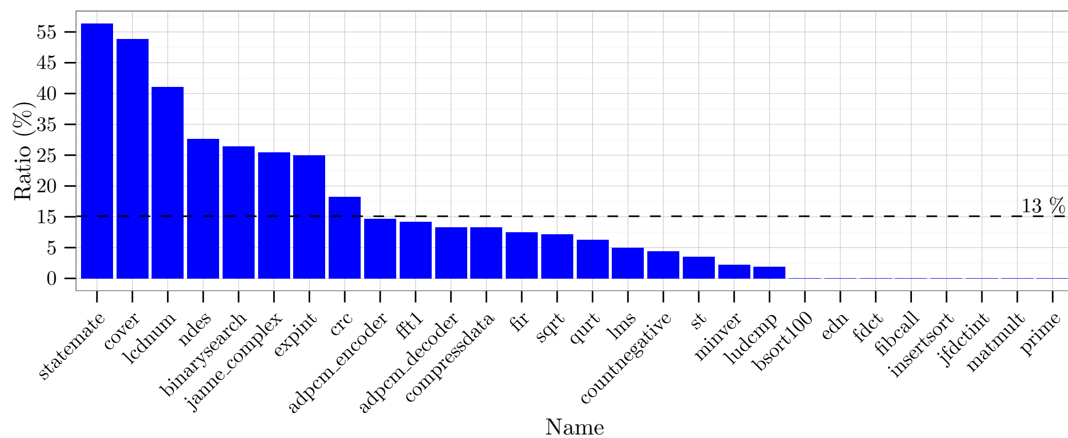


Figure 5.61: Program points (%) with non-deviating time estimates

bounds). The diagram is ordered by the average ratio of improvement of MBT over WCET analysis. For all benchmarks from ADPCM_DECODER (10 % average improvement) to INSERTSORT (93.7 % average improvement), we obtain 45 % more precise results for MBT on average over all benchmarks. Minimum and maximum bounds differ in the ranges from below 0.1 % to 100 %. Intuitively, differences are large in loops without branching bodies whereas loops with branches, low repetition counts and bottom exits yield low differences. Figure 5.61 illustrates quantitative differences as a percentage of program points within loops that do *not* differ. On average, just 13 % of program points do not yield tighter estimates by using MBT. Benchmark STATEMATE consists of deep nesting structures and accordingly 55 % of program point estimates do not differ. A range of benchmarks, such as PRIME or MATMULT, which perform numeric computations, conditional execution is rare. Hence, 100 % of program points yield tighter estimates.

Randomized Graphs

For a qualitative comparison of performance, we compare our MBT analysis with our WCET analysis and the corresponding ILP model proposed in [173] (ILP) on randomized CFG, whose parameters for control flow constructs and annotations are given in Table 5.2 on page 118. Our approaches are sampled including all necessary pre-processing such as scope tree construction. Sampling of ILP includes the generation of the ILP model. We scale sizes from 100 to circa 12 500 nodes and sample with a granularity of 1 ms. In Figure 5.62 we relate graph sizes with execution times (ms) given a “typical” distribution of constructs (cf. parameters in caption), and we generate just a single bound per loop. As already demonstrated for the WCET analysis (cf. Section 5.3.3.6), the ILP approach scales significantly worse than our WCET analysis. ILP ranges from 24 ms up to 1.42 s, whereas WCET ranges from under 1 ms up to 116 ms. Despite the increased computational complexity of MBT as opposed to WCET, we recognize that the difference in execution time is only insignificant: MBT scales from under 1 ms up to 210 ms.

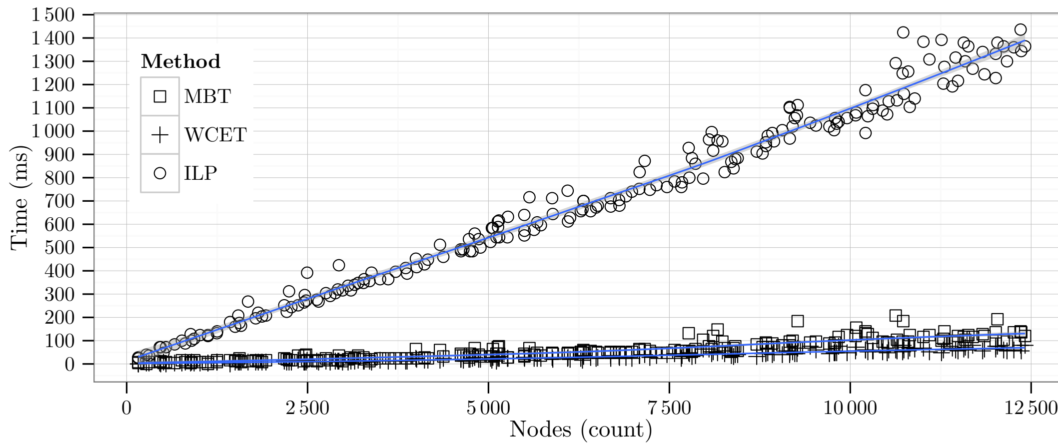


Figure 5.62: Runtime for non-degenerated CFG ($\text{DEPTH} = 4, \text{LOOP DEPTH} = 3, \text{P}(\text{IF}) = 0.1, \text{IFELSE} = 0.2, \text{P}(\text{WHILE}) = 0.3, \text{P}(\text{DOWHILE}) = 0.4$)

5.3.6.4 Conclusion

We proposed a path analysis for maximum blocking time computation, which is not only highly efficient but also the only existing non-ILP approach to this problem, to the best of our knowledge. As opposed to the ILP approach, we are not limited to fixed sets of preemption points but are able to explore the state space for all possible preemption points quickly. Efficiency is achieved by exploiting redundancy not only during computation of MBT bounds from a dedicated source to dedicated target node but in particular by exploiting the fact that for multiple target nodes computations are often redundant. We compute MBT bounds from a single source to all reachable nodes significantly quicker than the“the single source, single sink” problem is solved with the ILP approach. Moreover, we show that a carefully optimized implementation is only insignificantly slower than the computation of WCET bounds.

5.3.7 Computing Worst-Case Execution Frequencies

As a final variant of path analysis, we will be concerned not with path length bounds but with bounds on execution frequencies of individual program points. Previous approaches can be modified to compute upper bounds on the execution frequencies of nodes on specific paths but to the best of our knowledge, no efficient approach based on explicit path analysis exists to compute frequency bounds for all program points irrespective of specific paths. A related approach is referred to as *Minimum Propagation Analysis* (MPA) [149], which computes symbolic bounds on global execution frequencies from which WCET bounds can be derived. Note that with MAXLEN-based ILP approaches, only frequencies on the implicit worst-case path can be obtained.

Definition 5.80 (Worst-case Execution Frequency) *For a given set capacity constraints $\mathcal{C}$,* Worst-Case Execution Frequency *(WCEF) denotes an upper bound on the*

Figure 5.63: Example of WCEF distribution

repetition count of program points under $\mathcal{C}$ on any *path from a dedicated source to a dedicated sink node.*

Example *Figure 5.63a illustrates a schematic CFG consisting of two scopes, capacity constraints as node labels and unit node weight. Figure 5.63b illustrates the corresponding WCEF distribution: A global bound on the number of potential executions of each program point under the given flow bounds model. Note that the WCEP does not pass through the right branch within the loop. Consequently, information on potential flow in parallel paths would be lost.*

As a practical application, for the analysis of preemptive scheduling scenarios, WCEF denotes an upper bound on the number of preemptions that can occur for any specific program point. Note that for program points $u \in V$, costs ω and WCEF f, the sum $\sum_{u \in V} \omega(u) f(u)$ also denotes an upper bound on the WCET.

After technical prerequisites in Section 5.3.7.1, we discuss the respect framework in general in Section 5.3.7.2, followed its evaluation in Section 5.3.7.3 and by concluding remarks in Section 5.3.7.4. As always, we assume acquaintance with the principles of WCET computation from Section 5.3.3.

5.3.7.1 Prerequisites

S^f_π	Path states
$s^f_\pi = (\nu^f_\pi, \sigma^f_\pi, o^f_\pi) \in S^f_\pi$	Path state
$\nu^f_\pi \colon S^f_\pi \mapsto (A_\pi \mapsto \mathbb{N}_0^\infty)$	Execution frequencies
$\sigma^f_\pi \colon S^f_\pi \mapsto {A_\pi}^*$	Path signature
$o^f_\pi \colon S^f_\pi \mapsto V$	Path origin

Table 5.6: Definitions for WCEF analysis

In the following we discuss the technical prerequisites for WCEF analysis. Basic definitions from Table 5.3 on page 132 remain valid except for the replacement of path states. For WCEF, they still denote individual paths, but instead of length properties, we are solely interested in execution frequencies along such paths. Hence, we define path

states $(\nu_\pi^f, \sigma_\pi^f, o_\pi^f) = s_\pi^f \in S_\pi^f$, which encode execution frequencies $\nu_\pi^f \colon S_\pi^f \mapsto (A_\pi \mapsto \mathbb{N}_0^\infty)$ which denotes frequencies on a path, signature $\sigma_\pi \colon S_\pi^f \mapsto {A_\pi}^*$ which denotes a sequence of annotations along paths, and origin of paths $o_\pi \colon S_\pi^f \mapsto V$. Note that path infeasibility is now expressed as frequencies equal to 0.

Basic concepts of analysis remain intact. As usual, we discriminate path states S_π^f by the equivalence relation defined as:

$$\begin{aligned} \overset{A_\pi}{\sim} &\colon S_\pi^f \times S_\pi^f \\ s \overset{A_\pi}{\sim} s' &\Leftrightarrow \sigma_\pi^f(s_\pi^f) = \sigma_\pi^f({s_\pi^f}') \wedge o_\pi^f(s_\pi^f) = o_\pi^f({s_\pi^f}') \end{aligned} \tag{5.201}$$

The underlying algebraic structure — now for frequencies — is the commutative semi-ring

$$(\mathbb{N}_0^\infty, \max, \bot, +, 0) \tag{5.202}$$

as defined in Equation 5.50 on page 132. Since ν_π^f denotes a map of frequencies, we lift the algebra accordingly. Let the set of all functions that map from nodes to frequencies be denoted by:

$$\mathbb{D}^\nu = (V \mapsto \mathbb{N}_0^\infty) \tag{5.203}$$

Then we define the following algebra on $\mathbb{D}^\nu$:

$$(\mathbb{D}^\nu, \max, 1_{\max}, +, 1_+) \tag{5.204}$$

We define a helper function which applies an operator $o \in O$ element-wise as:

$$\begin{aligned} \mathrm{op_f} &\colon O \times \mathbb{D}^\nu \times \mathbb{D}^\nu \mapsto \mathbb{D}^\nu \\ \mathrm{op_f}(o, a, b) &= \lambda u \,.\begin{cases} o(a(u), b(u)) & \text{if } u \in \mathrm{def}(a) \cap \mathrm{def}(b) \\ a(u) & \text{if } u \in \mathrm{def}(a) \\ b(u) & \text{otherwise} \end{cases} \end{aligned} \tag{5.205}$$

Then maximum is defined as:

$$\begin{aligned} \max &\colon \mathbb{D}^\nu \times \mathbb{D}^\nu \times \mathbb{D}^\nu \\ \max(a, b) &= \left\{ u \to \mathrm{op_f}(\max\nolimits_{\mathbb{N}_0^\infty}, a, b)(u) \;\middle|\; u \in \mathrm{def}(a) \cup \mathrm{def}(b) \right\} \end{aligned} \tag{5.206}$$

where $1_{\max} = \emptyset$ denotes the neutral element. Similarly, we define addition[8] as:

$$\begin{aligned} + &\colon \mathbb{D}^\nu \times \mathbb{D}^\nu \times \mathbb{D}^\nu \\ a + b &= \left\{ u \to \mathrm{op_f}(+_{\mathbb{N}_0^\infty}, a, b)(u) \;\middle|\; u \in \mathrm{def}(a) \cup \mathrm{def}(b) \right\} \end{aligned} \tag{5.207}$$

where $1_+ = \emptyset$ denotes the neutral element.

[8] We assume operator + to be applicable in prefix or infix notation.

Finally, we can lift operator max to sets of path states in S^f_π of the *same* equivalence and define

$$\begin{aligned} \max^{\text{wcef}}_\pi &\colon \wp(S^f_\pi) \mapsto S^f_\pi \\ \max^{\text{wcef}}_\pi(S) &= \left(\max_{s' \in S} \nu^f_\pi(s'), \sigma^f_\pi(s), o^f_\pi(s)\right) : s \in S \end{aligned} \tag{5.208}$$

such that for an equivalence class $S = [s]$, $\max^{\text{wcef}}_\pi([s])$ denotes maximal frequencies mapped for these paths. We assume set S to denote an equivalence class according to Equation 5.201.

5.3.7.2 Framework

We now define the analysis framework for worst-case execution frequencies. Conceptually, this does not differ from previous frameworks substantially. Nevertheless, computing frequency bounds differs from path lengths enough to warrant a detailed discussion. As before, we define the framework bottom up. Starting from single iterations to unrolls to the computation of globally absolute execution frequencies.

As usual, we first show how iterations are computed, followed by the discussion of how complete unrolls are computed and how WCEF to all interior points are obtained, and we discuss how WCET bounds can be derived from bounds on WCEF.

Iterations

The problem of computing worst-case execution frequencies has the structure of the semi-lattice:

$$(\mathbb{D}_{wcef}, \top, \sqsubseteq, \sqcup) \tag{5.209}$$

where $\mathbb{D}_{wcef} = \wp(S^f_\pi)$ is a set of path states, where $S^f_\pi \sqsubseteq S^{f'}_\pi \Leftrightarrow S^f_\pi \subseteq S^{f'}_\pi$ and where $S^f_\pi \sqcup S^{f'}_\pi := \{\max^{\text{wcef}}_\pi[s] \mid [s] \in (S^f_\pi \cup S^{f'}_\pi)/\overset{A_\pi}{\sim}\}$ denotes the set of path states of different signature, origin and maximal frequencies, according to the equivalence relation defined in Equation 5.201. We define the corresponding transformer, which only updates signature σ^f_π as:

$$\begin{aligned} \text{tf}_{wcef} &\colon V \mapsto (\mathbb{D}_{wcef} \mapsto \mathbb{D}_{wcef}) \\ \text{tf}_{wcef}(u) &= \lambda S\,.\,\{(\nu^f_\pi(s), \sigma^f_\pi(s) \cdot \alpha_\pi(u), o^f_\pi(s)) \mid s \in S\} \end{aligned} \tag{5.210}$$

We assume the function

$$\max^{\mathring{s}}_{\mu^{\text{wcef}}} \colon \mathring{V} \times V^2 \mapsto \mathbb{D}^\nu \tag{5.211}$$

to be given such that $\max^{\mathring{s}}_{\mu^{\text{wcef}}}(\mathring{s}, u, v)$ evaluates to a mapping from annotation labels to maximal execution frequencies *within descendants* of scope $\mathring{s}$ from node u to node v

under given flow bounds β_π. In particular, for the root scope, $\max^{\mathring{s}}_{\mu^{\text{wcef}}}$ denotes absolute, global execution frequencies in subscopes.

We now define, how frequencies are derived from local flow bounds. To this end, we first define a lifted transformer to specify updates across subscopes as:

$$\mathrm{tf}^{\mathring{s}}_{wcef} \colon \mathring{S} \times V^2 \mapsto (\mathbb{D}_{wcef} \mapsto \mathbb{D}_{wcef})$$

$$\begin{aligned} &\mathrm{tf}^{\mathring{s}}_{wcef}(\mathring{s}, u, v) = \\ &\lambda S\,.\begin{cases} \mathrm{id} & \text{if } v \in \mathrm{entry}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \left\{ \begin{pmatrix} \nu^f_\pi(s^f_\pi) \cup f, \\ \sigma^f_\pi(s^f_\pi), \\ o^f_\pi(s^f_\pi) \end{pmatrix} \,\middle|\, \begin{array}{l} s^f_\pi \in S, \\ f = \max^{\mathring{s}}_{\mu^{\text{wcef}}}(\mathring{t}, u, v) \end{array} \right\} & \text{if } v \in \mathrm{exit}(\mathring{t}) \wedge (\mathring{t}, \mathring{s}) \in \mathring{E} \\ \mathrm{tf}_{wcef}(u)(S) & \text{otherwise} \end{cases} \end{aligned} \tag{5.212}$$

We compute maximal frequencies up to, but not including, subscope entries, extend the set of frequencies by those of the currently finished subscope, or, otherwise, just update path states according to Equation 5.210.

Assuming $\mathrm{pred}^{\rightarrow}_{\mathring{s}}$ as defined in Equation 5.99 on page 150 to denote CFG predecessors such that subscopes are being "leaped over", we define path states in a program point by:

$$S^{\mathring{s}}_{wcef} \colon \mathring{V} \times V^2 \times A_\pi \mapsto S^f_\pi$$

$$S^{\mathring{s}}_{wcef}(\mathring{s}, s, v, a) = \begin{cases} \mathrm{tf}^{\mathring{s}}_{wcef}(\mathring{s}, s, s)(\{(0, a, s)\}) & \text{if } v = s \\ \bigsqcup \left\{ \mathrm{tf}^{\mathring{s}}_{wcef}(\mathring{s}, u, v)(S^{\mathring{s}}_{wcef}(\mathring{s}, s, u, a)) \mid u \in \mathrm{pred}^{\rightarrow}_{\mathring{s}}(\mathring{s})(v) \right\} & \text{otherwise} \end{cases} \tag{5.213}$$

For a given scope $\mathring{s}$, start and terminal nodes u, v and an initial annotation a, $S^{\mathring{s}}_{wcef}(\mathring{s}, u, v, a)$ denotes the set of maximal frequencies within subscopes.

As usual, we define partitions of $S^{\mathring{s}}_{let}$ as in the WCET case (cf. Section 5.3.3.2) such that

$$S^I_{wcef} \colon \mathring{V} \times V \mapsto \wp(S^f_\pi) \tag{5.214}$$

$$S^O_{wcef} \colon \mathring{V} \times V \mapsto \wp(S^f_\pi) \tag{5.215}$$

$$S^K_{wcef} \colon \mathring{V} \mapsto \wp(S^f_\pi) \tag{5.216}$$

$$S^D_{wcef} \colon \mathring{V} \times V^2 \mapsto \wp(S^f_\pi) \tag{5.217}$$

denote entry, exit, kernel and direct paths, respectively.

Unrolls

Regarding path length, unrolling as defined in the previous analyses involved computing maximal admissible flow along the network paths denoted by state signatures such that

the length of compositions of entry, exit and kernel paths (or direct paths, alternatively) is maximized. In WCEF analysis, we are only concerned with the maximal admissible flow. Consequently, the problem to solve per scope is just MAXFLOW with node weights and bounds, which we can define as:

$$\begin{aligned} \max \quad & \sum_{u \in V} f^{\Sigma}_{out}(u) \\ \text{s.t.} \quad & \forall u \in V\colon f^{\Sigma}_{in}(u) - f^{\Sigma}_{out}(u) = b_u \\ & b_u = \begin{cases} q & \text{if } u = s \\ -q & \text{if } u = t \\ 0 & \text{otherwise} \end{cases} \\ & \forall u \in V\colon \ell(u) \leq f(u) \leq \beta(u) \end{aligned} \tag{5.218}$$

As before, ℓ denotes flow demand to guarantee existence of feasible entry and exit paths and β denotes an upper flow bound. Reduction of path states to flow networks has been thoroughly discussed for the case of WCET in Section 5.3.3.2 and is not repeated here.

Frequency is almost synonymous to network flows. The difference is that with the former we denote scope-relative flows scaled by flow of enclosing scopes. It is important to recognize that frequencies $\mathbb{D}^{\nu}$ denote maximal frequencies of subscopes only: prior to unrolling, maximal flow of current scopes is unknown. Recall that flow bounds β_{π} denote scope-local constraints only. Consequently, since $\mathbb{D}^{\nu}$ denotes maximal flow for all subscopes, unrolling encompasses not just the determination of maximal flow for a scope but also the scaling of frequencies of subscopes. We now formalize this intuition.

We keep test_f (Equation 5.70) and set_f (Equation 5.71) unchanged but define an additional function

$$\text{scale}_{\nu} := \lambda(\nu, n) \,.\, \{a \to f \times n \mid (a \to f) \in \nu\} \tag{5.219}$$

such that $\text{scale}_{\nu}(n, \nu^f_{\pi}(s^f_{\pi}))$ scales subscope frequencies ν^f_{π} by n. Then we define a function f^1_{ν} which pushes unit admissible flow over a path denoted by σ^f_{π} while scaling subscope flows accordingly. The function argument is a triple of frequencies ν, flow f and capacity bounds c and returns such a triple. It is defined as:

$$f^1_{\nu} := \lambda s \,.\, \lambda(\nu, f, c) \,.\, \begin{cases} \begin{pmatrix} \nu + \text{scale}_{\nu}(\nu^f_{\pi}(s), 1) \\ \text{set}_f(f, 1, \sigma^f_{\pi}(s)), \\ c \end{pmatrix} & \text{if } \text{test}_f(c, f, \sigma^f_{\pi}(s)) > 0 \\ (\bot, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.220}$$

As usual, we represent infeasibility by $\perp$. Analogously, we define a function f_ν^k, which pushes maximally admissible flow and which is defined as:

$$f_\nu^k := \lambda s \,.\, \lambda(\nu, f, c) \,.\, \begin{cases} \begin{pmatrix} \nu + \text{scale}_\nu(\nu_\pi^f(s), \text{test}_f(c, f, \sigma_\pi^f(s))) \\ \text{set}_f(f, \text{test}_f(c, f, \sigma_\pi^f(s)), \sigma_\pi^f(s)), \\ c \end{pmatrix} & \text{if } \nu \neq \perp \\ (\perp, \emptyset, c) & \text{otherwise} \end{cases} \tag{5.221}$$

As usual, we compose f_ν^1 and f_ν^k to evaluate unrolls. Note that we are effectively solving MAXFLOW. Hence,. kernel paths need not be ordered specifically. Consequently, all possible evaluations for scope $\mathring{s}$, entry u and exit v for unrolls are defined as:

$$\begin{aligned} &F_\nu^{iok} := \lambda(\mathring{s}, u, v). \\ &\quad \left\{ [\![S_{wcef}^K(\mathring{s})]\!](f_\nu^k) \circ f_\nu^1(s_o) \circ f_\nu^1(s_i) \;\middle|\; s_i \in S_{wcef}^I(\mathring{s}, u), s_o \in S_{wcef}^O(\mathring{s}, v) \right\} \end{aligned} \tag{5.222}$$

Similarly, all evaluations for direct paths are defined as:

$$F_\nu^d := \lambda(\mathring{s}, u, v) \,.\, \left\{ f_\nu^1(s_d) \;\middle|\; s_d \in S_{wcef}^D(\mathring{s}, u, v) \right\} \tag{5.223}$$

Theorem 5.81 (Maximal Frequencies Unroll) *Let $\mathring{s} \in \mathring{V}$ be a scope, let $u \in \text{entry}(\mathring{s})$ and $v \in \text{exit}(\mathring{s})$ and let $f_0 = A_\pi \times \{0\}$ denote initial flow. Then maximal frequencies for a scope $\mathring{s} \in \mathring{V}$ and entry and exit nodes $u, v \in V$, is defined as:*

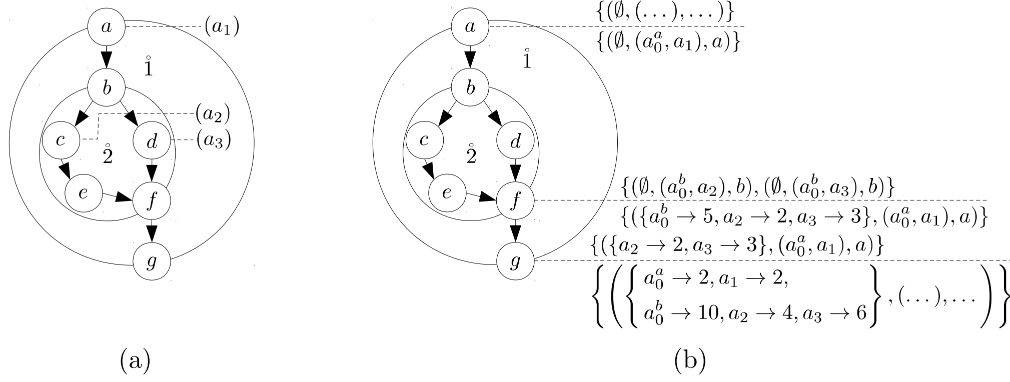
$$\begin{gathered} \max\nolimits_{\mu^{\text{wcef}}}^{\mathring{s}} : \mathring{V} \times V^2 \mapsto \mathbb{D}^\nu \\ \max\nolimits_{\mu^{\text{wcef}}}^{\mathring{s}}(\mathring{s}, u, v) = \max \left\{ \nu \cup f \;\middle|\; \begin{array}{l} (\nu, f, c) = f_\nu(\emptyset, f_0, \beta_\pi), \nu \neq \perp, \\ f_\nu \in F_\nu^{iok}(\mathring{s}, u, v) \cup F_\nu^d(\mathring{s}, u, v) \end{array} \right\} \end{gathered} \tag{5.224}$$

This represents the precise worst-case execution frequencies for a scope and a pair of source and sink nodes, under the given flow bound model.

Proof. MAXFLOW is solved as usual (cf. Theorem 5.63 on page 143). In addition, for every (local) flow f along a path π in the current scope, all maximal flows of subscopes ν "traversed" by π are scaled by f. Consequently, $f + \nu$ denotes maximal flow for the current scope and the scaled maximal flow for all subscope. Note that sets f and ν are disjoint. We determine the maximum for all possible combinations. □

Example *Figure 5.64a illustrates two interior scopes $\mathring{1}, \mathring{2}$, annotated such that $\beta_\pi(a_1) = 2, \beta_\pi(a_2) = 2, \beta_\pi(a_3) = 3$. Figure 5.64b illustrates a subset of the state space, where state sets above and below the dashed lines denote path states just before and right after transformer tf_{wcef} is applied, respectively. In nodes f and g, $\max_{\mu^{\text{wcef}}}^{\mathring{s}}$ is applied respectively such that*

$$S_{wcef}^{\mathring{s}}(\mathring{1}, a, f, a_0^a) = \{(\{a_0^b \to 5, a_2 \to 2, a_3 \to 3\}, (a_0^a, a_1), a)\}$$



(a) (b)

Figure 5.64: Example of WCEF state propagation

denotes path states right after subscope $\mathring{2}$ *has been left. If we assume a surrounding scope* $\mathring{0}$ *of entry* s*, then similarly,*

$$S^{\mathring{s}}_{wcef}(\mathring{0}, s, g, a_0^s) = \{(\{a_0^a \to 2, a_1 \to 2, a_0^b \to 10, a_2 \to 4, a_3 \to 6\}, (\ldots), \ldots\}$$

denotes path states after leaving scope $\mathring{1}$.

WCEF of All Interior Points

Let $\mathring{0}$ be a root scope where $s \in \text{entry}(\mathring{0})$ is the global CFG entry and $t \in \text{exit}(\mathring{0})$ is the global CFG exit. Then $\nu = \max^{\mathring{s}}_{\mu^{\text{wcef}}}(\mathring{0}, s, t)$ by definition represents absolute frequencies from all scopes. Recall that we assume the default annotation for entry s to denote a unit capacity bound. Consequently, only direct paths are considered in the unroll of scope $\mathring{0}$. Nevertheless, we still need to compute maximal frequencies for all interior nodes: map ν only denotes frequencies for annotation labels yet. We now map frequencies into the CFG.

For a given node $u \in V$, let $S(u)$ denote all path states in u from all entries of its respective scope $\mathring{\gamma}(u)$, which we define as:

$$S := \lambda u \,.\, \{S^{\mathring{s}}_{wcef}(\mathring{\gamma}(u), i, u) | i \in \text{entry}(\mathring{\gamma}(u))\} \tag{5.225}$$

where $S^{\mathring{s}}_{wcef}(\mathring{\gamma}(u), i, u)$ (Equation 5.213) represent path states from entry i. Recall that path signature σ^f_π denotes a sequence of annotations along a path. Then the set of all most recent annotation labels with respect to node u on paths denoted by $S(u)$ is defined as:

$$A := \lambda u \,.\, \{a_n | (a_1, \ldots, a_n) = \sigma^f_\pi(s), s \in S(u)\} \tag{5.226}$$

Theorem 5.82 *Let* $\nu = \max^{\mathring{s}}_{\mu^{\text{wcef}}}(\mathring{0}, s, t)$ *denote the mapping* $\nu\colon A_\pi \mapsto \mathbb{N}_0^\infty$ *from annotation labels to execution frequencies. Then*

$$\begin{aligned}\nu^\star &\colon V \mapsto \mathbb{N}_0^\infty \\ \nu^\star(u) &= \sum_{a \in A(u))} \nu(a)\end{aligned} \tag{5.227}$$

denotes the maximal execution frequency of a node u*.*

Proof. By definition of network flows, net flow into a node equals the sum of net flow out of its network predecessor nodes. Frequency $\nu(a)$ equals net flow out of its corresponding CFG node $v = \alpha_\pi^{-1}(a) \in V$. All paths to a node $u \in V$ must pass through a most recently annotated node v. Consequently, net flow into node u must equal the sum of net flow out its most recently annotated predecessors. □

Example *We reconsider Figure 5.64b and compute the maximum execution frequency for node* f*. Note that the example only represents a subgraph. By unrolling scope* $\mathring{1}$*, we already computed frequencies per annotation labels (cf.* $S^{\mathring{s}}_{wcef}(\mathring{0}, s, g, a_0^s)$ *which denotes path states in the implicit enclosing scope* $\mathring{0}$ *after unrolling):*

$$\nu = \max^{\mathring{s}}_{\mu^{\text{wcef}}}(\mathring{0}, s, t) = \{a_0^a \to 2, a_1 \to 2, a_0^b \to 10, a_2 \to 4, a_3 \to 6\} \tag{5.228}$$

All path states in f *are represented by:*

$$S(f) = \{(\emptyset, (a_0^b, a_2), b), (\emptyset, (a_0^b, a_3), b)\}$$

Consequently, all most recent annotation labels are denoted by:

$$A(f) = \{a_2, a_3\}$$

where $\alpha_\pi^{-1}(a_2) = c$ *and* $\alpha_\pi^{-1}(a_3) = d$*. Then the accumulated frequency in* f *is denoted by:*

$$\nu^\star(f) = \nu(a_2) + \nu(a_3) = 10$$

Note that $\nu^\star(g) = \nu(a_1) = 2$ *and* $\nu^\star(c) = \nu(a_2) = 4$ *(c.f Figure 5.63).*

From Frequencies to Time Bounds

We briefly address how WCEF bounds can be reduced to obtain WCET bounds. Let $\omega\colon V \mapsto \mathbb{N}_0$ denote WCET estimates per program point just as in the previous variants and let $\nu^\star$ be defined as in Equation 5.82. Then a global WCET bound wcet^ν per program point u is denoted by all program points potentially executing prior to u. Therefore, let $\text{pred}^*(u) := \{v \mid v \rightsquigarrow u\} \cup \{u\}$ denote all CFG nodes to reach — and including — node

u . Then we define WCET bounds as node weights scaled by frequencies as:

$$\text{wcet}^{\nu}\colon V \mapsto \mathbb{N}_0^{\infty,\perp}$$
$$\text{wcet}^{\nu}(u) = \sum_{v \in \text{pred}^*(u)} \nu^{\star}(v) \times \omega(v) \tag{5.229}$$

It is important to recognize that flow bounds for WCEF and WCET as defined in Section 5.3.3 yield different semantics.

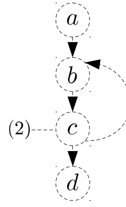


Figure 5.65: Example of different semantics of flow bounds for WCEF and WCET

Example *Consider the graph illustrated in Figure 5.65. Node c is bounded by* 2 *and we assume unit node weights. According to Equation 5.229, a bound on the WCET for node b yields* $1 \times \omega(a) + 2 \times \omega(b) + 2 \times \omega(c) = 5$. *However, the worst-case path to b equals* $|(abcbcb)| = 6$.

The reason for this is that WCET bounds by worst-case paths are sensitive to annotation location, whereas WCET bounds by WCEF are path-insensitive. When comparing both approaches directly this has to be kept in mind.

5.3.7.3 Evaluation

We evaluate the proposed WCEF framework qualitatively as well as quantitatively by evaluating WCET estimates obtained via Equation 5.229, which we will denote as just WCEF in the following, and from our proposed WCET framework from Section 5.3.3 to assess the differences between the two methods. The test environment is identical to that of the WCET evaluation in Section 5.3.3.6 and we use a similar method. For precision, we evaluate results from the MRTC benchmark suite [97]. As stated before, WCET estimates by the two methods are not directly comparable. Nonetheless, the evaluation provides an intuition of their respective quality. Averages are computed by the arithmetic mean. Performance is evaluated by means of randomized graphs to obtain large sample sets with controlled characteristics. The implementation is a derivative of the WCET reference algorithm given in Section 5.3.3.5.

MRTC

We first evaluate estimation quality by assessing a subset of MRTC benchmarks. Figure 5.66 illustrates the ratio of precision in terms of overestimation of WCEF over WCET as box plots, showing upper, lower and average ratios for all program points within a single

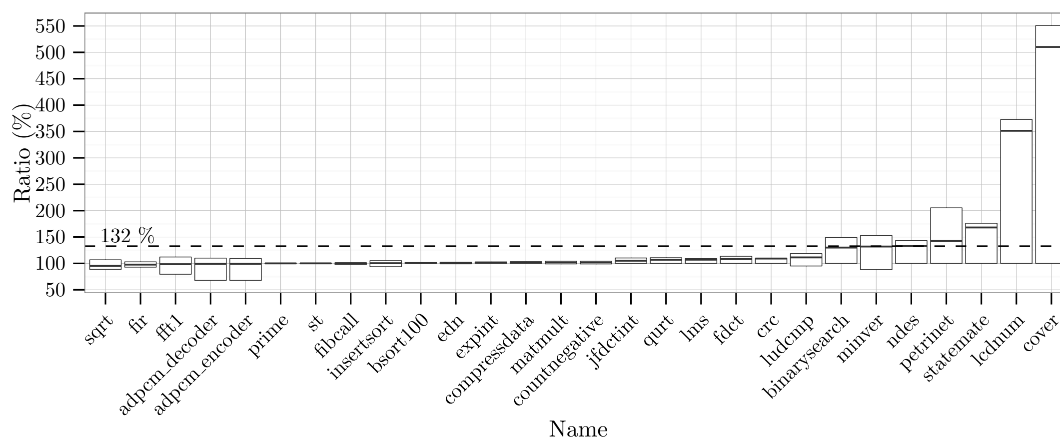


Figure 5.66: Overestimation (%) of WCEF over WCET bounds

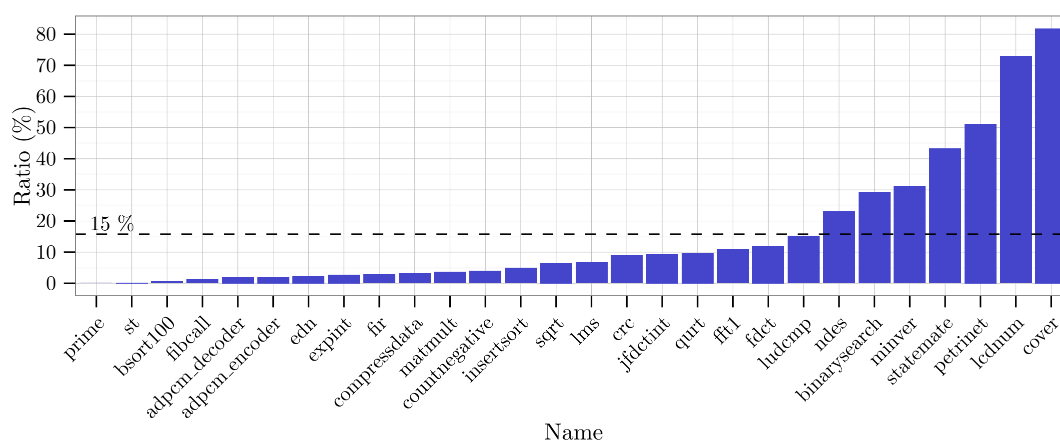


Figure 5.67: Overestimation (%) of WCEF over WCET bounds for terminal nodes only

benchmark. Benchmarks are ordered in ascending order of average deviation. On average over all benchmarks, WCEF yields 132 % overestimation. Recall from the discussion above that flow bounds have different semantics in both frameworks. Hence, WCEF is not necessarily strictly dominated by WCET in all program points, given identical sets of flow bounds; this is reflected by the diagram. WCEF for benchmark *sqrt* yields 5 % lower bounds on average. Opposed to that, benchmark COVER overestimates WCET by 510 %. Lowest WCEF estimates occurred for ADPCM_ENCODER (−22 %).

Figure 5.67 depicts WCEF overestimation just for the respective terminal program points of all benchmarks. In all cases, WCEF yields higher estimates than WCET (by 15 % on average). Overestimates range from 0 % for PRIME to 82 % for COVER.

In Figure 5.68 the percentage of program points that do *not* differ for both estimates is depicted. On average, 14 % of program points yield identical results, ranging from 55 % for EXPINT to 1 %.

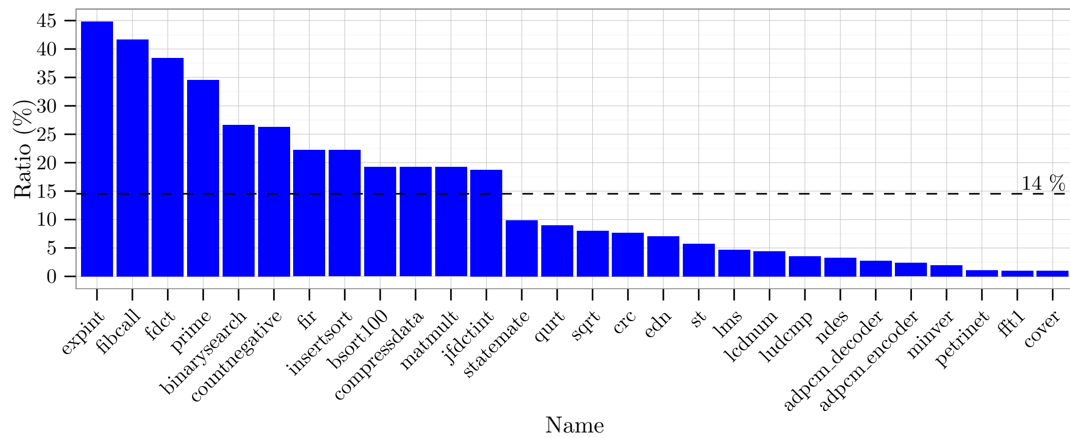


Figure 5.68: Program points (%) with non-deviating time estimates

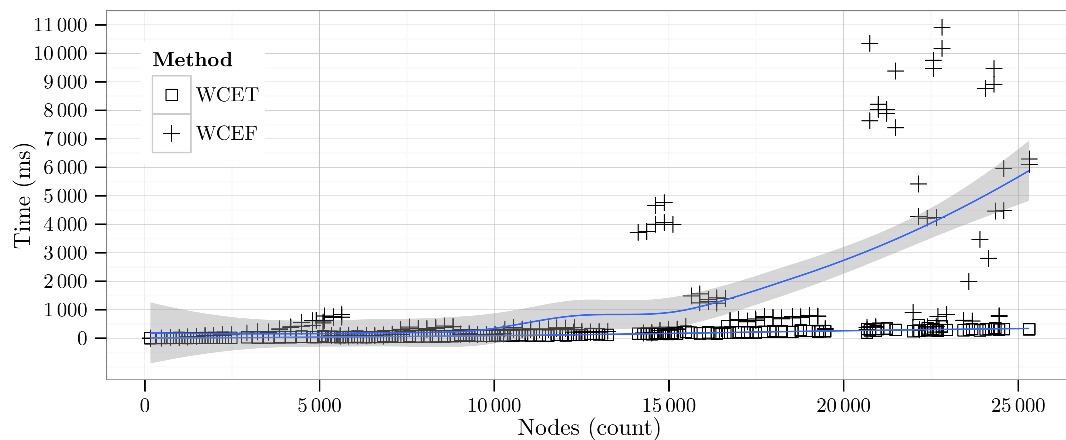


Figure 5.69: Runtime for non-degenerated CFG (DEPTH $= 4$, LOOP DEPTH $= 3$, P(IF) $= 0.1$, IFELSE $= 0.2$, P(WHILE) $= 0.3$, P(DOWHILE) $= 0.4$)

Randomized Graphs

As before (cf. Section 5.3.3.6), we argue that real-time benchmarks are not well suited for a quantitative comparison of performance due to their limited size. So we evaluate randomized CFGs from size of approximately 100 to 25 000. Randomization parameters for control flow constructs and annotations are listed in Table 5.2 on page 118.

In Figure 5.69 we relate execution time in ms to graph sizes for a "typical" distribution of control flow constructs (cf. parameters in caption), and we generate just a single bound per loop. Scalability of WCEF is significantly worse than WCET. for 25 313 nodes, WCEF takes 6 104 ms and whereas WCET takes just 320 ms. Moreover, scaling is clearly not linear for WCEF.

5.3.7.4 Conclusion

Here, we proposed a variant of our reference WCET path analysis to compute worst-case frequencies. As opposed to other path analyses in which frequencies might be derived as side-results our approach on frequency estimation is independent of specific paths but computes them only based on given flow bounds. This has the advantage of obtaining a holistic view on execution patterns within a task. This is particularly useful to identify "hot" regions within a task. This metric is can in general not be obtained by path-based analyses as proposed earlier. As we have seen, WCEF can also be used to bound WCET. However, path-based and frequency-based approaches for time estimation assume different flow bound semantics and are therefore not directly comparable in general.

5.4 Remarks

In this section we make remarks on potential future work on path analysis regarding context-sensitivity, global flow bounds, mutual path exclusion and symbolic analyses, which we have not covered explicitly so far.

As we have seen, the proposed framework is a general foundation for path analysis from which different variants besides whole-task WCET estimates can cleanly be derived. We deliberately did not address aforementioned features to keep overall formalization unified and simple. Nevertheless, throughout its design, we took care not to create unnecessary obstacles for extension. Arguably, IPET-based approaches for simple whole-task WCET estimates have the advantage that it is very easy to model, for example, mutual path exclusion. But it cannot be stressed enough that ILP-based approaches in particular yield highly limited expressiveness, preventing advances towards higher levels of integration of timing analysis phases, including — and in particular — of scheduling analysis. Recall that in particular in the case non-timing compositional architectures no viable alternative is known. Our framework is a proposal and as well as an invitation to unify non-ILP-based path analysis approaches. Future work should therefore involve the addition of features of practical relevance beyond what we proposed so far. We now address some of these specifically and give hints on their realization.

As opposed to IPET and approaches based on graph reduction, our method allows the collection of information in execution order to a greater extent. This allows path computations to be context-sensitive. As a matter of fact, we already took precautions for this case in that we introduced generic annotation labels that define path signatures. At the current state, these labels merely denote flow bounds, but we specifically took into consideration that these might identify other kinds of information, where signatures continue to discriminate state explicitly. Since analysis order is topological, it is also easy to propagate information in and out of scopes, enabling global context-sensitivity. During unrolling, we then can take this into account accordingly. As an example, above, we assumed constant execution costs ω per program point, but it is easy to define ω as a function of signatures, given they provide relevant contextual information. To some

degree context is already discriminated in our proposals in that flow bounds induce partitions of iterations. Also we explicitly discriminate entry and exit paths from kernels. For example, it would already be straight forward to distinguish first from following iterations to account for cold and hot cache contexts. It is important to recognize that if costs ω become a function of path context, it also questions the traditional order of stages in timing analysis tool chains, in that path analysis can become a potential driver of micro-architectural analysis by providing additional means to discriminate context, instead of representing just a final consolidating analysis stage. More fine-grained discrimination of loop iterations are proposed in [141]. Their constraint model is in principle also applicable to our framework with a respective adaption of unrolling. For general global flow bounds, we sketched the idea and a simple heuristic in [9]. In general, we consider this a non-trivial problem that cannot be decided just locally since flows in different scopes need to be balanced out to obtain globally maximal path lengths (see [9] for details). Note that global bounds can always be approximated with respective local bounds. Still it is an interesting problem to investigate in isolation.

An important type of context-sensitivity is mutual path exclusion from which significant gains in accuracy can be expected even under a constant cost model: mutual exclusion, along with flow bounds, reflect control flow semantics lost in the CFG abstraction. Flow bounds are necessary for path analysis while mutual exclusion may be optional. Nevertheless, we explicitly took this into account as the logical next step to enhance our framework. As with general context-sensitivity above, it is possible to extend annotations and adopt unrolling accordingly. To give a concrete hint on its realization, let annotations — besides ones that denote flow bounds — represent nodes in a conflict graph. Then states are sufficiently discriminated by definition of path states already and just unrolling needs to be adopted to compose only non-conflicting paths. Since conflicts are potentially global, annotations need to be migrated between scopes, affecting initial and restored pending path states accordingly. This equally enables the discrimination of task execution modes.

For parametric timing analysis, computing symbolic representations of path problems is of particular interest. All analyses proposed above have been carefully formalized to expose the underlying algebra. Technically, efficient construction of symbolic expressions in our framework is easily achieved by just replacing respective algebras. To the best of our knowledge, all symbolic approaches suffer from state space explosion. We argue that in many cases this is the result of naive expression construction [137, 150] which yields much redundancy that is subsequently to be eliminated by term-rewriting. Our framework circumvents such problems by eliminating redundancy early.

Undoubtedly, there likely exist equivalent ILP models to the path problems discussed so far. But we have to keep in mind that these will have to be built first, potentially requiring additional analyses and which effectively duplicate existing control flow representation in the form of linear equations. In-place analysis is not possible. Also the generated models will be static and potentially very large as the entire potential

state space must be modeled in constraints. Exploitation of dynamic redundancy is not possible. For example, to compute WCET estimates to all reachable nodes, an ILP model of at least quadratic size of the input would be required to model MAXLEN for each pair of source and potential sink node separately. In our proposal, only a minimal amount of information is maintained since we dynamically decide whether subpaths can be reused for multiple sink nodes at a time. We believe that the question is not so much whether we can find corresponding ILP models for these algorithms. But rather whether we can find algorithms as alternatives to problems traditionally modeled as ILP in the context of timing analysis to overcome the current stalemate.

5.5 Conclusion

In this chapter we proposed a general, efficient and flexible path analysis framework for timing analysis. The problem of path analysis is addressed by isolating two key aspects.

The first aspect is control-flow reconstruction. We discussed how traditional, compiler-oriented approaches to reconstruction have critical shortcomings in the context of timing analysis by creating a semantic gap between flow fact models and representation used in subsequent analyses. We argue that root cause for this is the application of traditional standard heuristics which may be ill-suited. Worse yet, existing approaches propose input transformations (e.g. to establish graph reducibility prior to reconstruction) which only widens the semantic gap. To address these problems, we first proposed an efficient parametric algorithm for loop detection which allows for structural specification as part of a flow fact model to close the semantic gap. In particular for the case of irreducibility, we proposed two alternative strategies to overcome ambiguity during recovery: Either by locally restricted enumeration of cases or by extension of structural flow facts by so-called prenumbering to maintain specific loop-structures.

The second aspect is related to path analysis itself based on structural information previously obtained. To this end, we propose a general path analysis framework which subsumes other existing approaches. As the primary use case, we showed the construction of an efficient analysis for WCET estimation, which is not limited to analysis at task granularity but allows for computation on arbitrary subgraphs and for individual program points. We proposed several optimizations for the construction of highly efficient implementations, and provided a concrete reference implementation. Further, we derived several variants of the standard WCET problem:

- Best-case Execution Time (BCET): Estimation of lower execution time bounds. In conjunction with WCET bounds, execution time intervals for individual program points can be determined.
- Latest Execution Time (LET): WCET to program points with the guarantee that a task terminal point remains reachable. This yields significantly tighter estimates for analysis cases in fully preemptive scheduling scenarios.

- Maximum Block Time (MBT): Upper bounds on the WCET for the execution from a program point to another. This provides timing estimates for fixed region deferred preemption scenarios.
- Worst-case Execution Frequencies (WCEF): Upper bounds on the repetition of individual program points independently from potential execution paths. This, for example, yields upper bounds on the number of preemptions in fully preemptive scheduling scenarios.

Together, these variants cover many existing practical problems in timing analysis for which no, only highly specialized or inefficient solutions exist. Our hypothesis is that one of the causes is the overemphasis of IPET, which might be an near-optimal choice in traditional per-task timing analysis but which is otherwise highly restrictive.

Chapter 6

Bounding Cache-related Preemption Delay

Contents

6.1 Improving Conventional CRPD Bounds **221**

6.1.1 Preliminaries 222

6.1.2 A Review of Approaches 223

6.1.3 A Refined Bound on CRPD 230

6.2 Improving CRPD Estimation with Time Bounds **237**

6.3 Evaluation **242**

6.4 Conclusion **245**

In this chapter we resume our discussion on bounding cache-related preemption delay. In Chapter 4, we focused on the computation of tight CRPD bounds for single preemptions and only addressed issues related to non-trivial task sets as far as necessary. In the following we review and compare different approaches with an emphasis on optimizing for large numbers of preemptions. We further propose two new bounds on CRPD which on the one hand improve on the current state of the art for — what we refer to as — conventional CRPD bounds and on the other hand propose a bound which explicitly exploits results from path analyses proposed in the the previous chapter.

In Section 6.1 we discuss existing CRPD bounds and propose a new bound which does not follow the conventional principles of existing approaches by more precisely modeling preemption scenarios. In Section 6.2 we propose another bound which relies on timing information from path analysis to exclude infeasible interferences. We evaluate these new bounds in Section 6.3 and conclude the chapter in Section 6.4.

6.1 Improving Conventional CRPD Bounds

In this section we discuss various approaches to bound CRPD. To this end, we first provide a brief overview of the most accurate existing conventional CRPD bounds. We

refer to this class of CRPD bounds as being conventional as they have been proposed over an extended period of time, depend on similar inputs and only differ in their degree of pessimism. We identify their common principles and their inherent weaknesses. We then propose an new alternative bound which does follow these principles and does not share sources of inaccuracy of existing approaches.

We first repeat, summarize and simplify important notions from previous chapters in Section 6.1.1. Then we review existing approaches in Section 6.1.2 and propose a new CRPD bound in Section 6.1.3.

6.1.1 Preliminaries

We first tersely summarize important notions and definitions from Chapter 3 and Chapter 4 for reference in the following discussion. Generally, we assume deadline monotonic scheduling and LRU cache replacement policy.

From scheduling theory, recall from Table 3.1 on page 23 that for a task τ_i, parameter R_i denotes response time, J_i denotes release jitter and T_i denotes period. Then the number of preemptions imposed by a higher priority task τ_j on a task τ_i is bounded by:

$$\#(i,j) := \left\lceil \frac{R_i + J_i}{T_j} \right\rceil \tag{6.1}$$

Also recall from Table 3.2 on page 27 that $\mathrm{hep}(i)$ denotes tasks of higher or equal priority than τ_i, $\mathrm{lp}(i)$ denotes tasks of lower priority than τ_i and that $\mathrm{aff}(i,j) = \mathrm{lep}(j) \cap \mathrm{hp}(i)$ denotes "affected" tasks in the context of indirect preemption of τ_i by τ_j.

We simplify the representation of static cache analysis results and denote useful cache blocks, evicting cache blocks and cache block resiliencies by the following function sets:

$$\mathrm{UCB} \subseteq V \mapsto \wp(\mathcal{M}) \tag{6.2}$$

$$\mathrm{ECB} \subseteq \wp(\mathcal{M}) \tag{6.3}$$

$$\mathrm{CBR} \subseteq V \mapsto \mathrm{Age} \tag{6.4}$$

where $\mathrm{Age} = \{0, \ldots, K-1, \infty\}$ denotes cache block age. In a formal context, UCB, ECB and CBR denote aforementioned sets. Otherwise, we just mean the concept.

Consequently, task-wise sets of UCB, ECB and CBR are denoted by:

$$\mathrm{ucb}_{\mathcal{T}} \colon \mathcal{T} \mapsto \mathrm{UCB} \tag{6.5}$$

$$\mathrm{ecb}_{\mathcal{T}} \colon \mathcal{T} \mapsto \mathrm{ECB} \tag{6.6}$$

$$\mathrm{cbr}_{\mathcal{T}} \colon \mathcal{T} \mapsto \mathrm{CBR} \tag{6.7}$$

In Chapter 4, we have been concerned with improving estimates for single preemptions and only handled bounds within task sets only as far as necessary. We briefly summarize the most important bounds for single preemptions according to our simplified notation: Bounds based on UCB only are defined as the maximal cardinality of UCB sets in

program points such that:

$$\begin{aligned} \text{crpd}^{\text{ucb}} &\colon \text{UCB} \mapsto \mathbb{N}_0 \\ \text{crpd}^{\text{ucb}}(\text{ucb}) &= \max_{u \in V} \left\{ \sum_{s \in [1,N]} \{|\text{ucb}(u)_s|\} \right\} \end{aligned} \tag{6.8}$$

Bounds using ECB-only are defined as the number of accessed cache sets scaled by cache associativity K:

$$\begin{aligned} \text{crpd}^{\text{ecb}} &\colon \text{ECB} \mapsto \mathbb{N}_0 \\ \text{crpd}^{\text{ecb}}(\text{ecb}) &= K \times \sum_{s \in [1,N]} \{s \colon |ecb_s| > 0\} \end{aligned} \tag{6.9}$$

Recall that a single evicting cache block can cause up to K cache misses in a preemptee. Consequently, for bounds combining UCBs and ECBs, a safe estimate is given by the maximal cardinality of invalidated UCBs denoted by conflicting cache sets of UCBs and ECBs over all program points:

$$\begin{aligned} \text{crpd}^{\text{ucb,ecb}} &\colon \text{UCB} \times ECB \mapsto \mathbb{N}_0 \\ \text{crpd}^{\text{ucb,ecb}}(\text{ucb}, \text{ecb}) &= \max_{u \in V} \left\{ \sum_{s \in [1,N]} \{|\text{ucb}(u)_s| \colon \text{ecb}_s \neq \emptyset\} \right\} \end{aligned} \tag{6.10}$$

CBR is an optional optimization for such combined bounds. For the sake of simplicity, we will not specifically address CBR in the following. Its application is an obvious extension to the given formulae.

6.1.2 A Review of Approaches

In this section we briefly review approaches to bound CRPD in worst-case response time analysis. Instead of providing a collection of related work, we only discuss the principle intuitions of the various techniques and briefly address their specific weaknesses. We only limit the discussion to the relevant subset of approaches. A thorough overview and the historic development of approaches is given in the seminal work of [6].

After a brief introduction, we first discuss a class of bounds akin to those already partially addressed in Chapter 4 to which we refer to as *singleton bounds*, then we discuss the current state of the art of approaches referred to as *multiset bounds*.

Introduction

Recall that UCBs in a program point denote cached memory blocks which remain cached till their next access. Any preemption between two accesses potentially evicts such UCBs, increasing the CRPD. Hence, the largest set of UCBs at one program point denotes an upper bound for evictions at any program point. The amount of possible eviction is

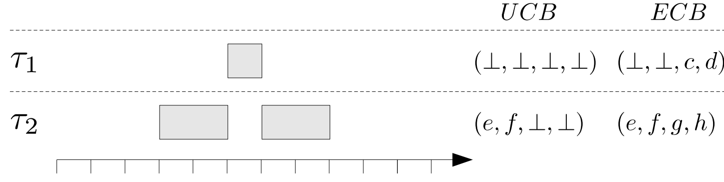


Figure 6.1: Example of UCB-only and ECB-only CRPD (K=1, N=4, BRT=1)

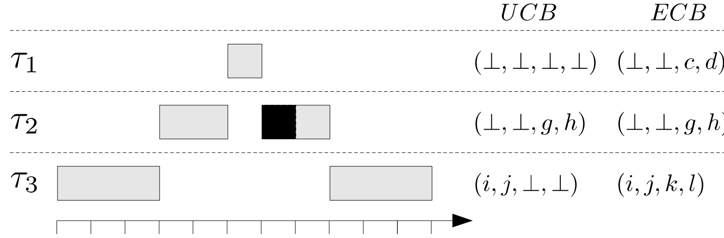


Figure 6.2: Example of UCB-only and ECB-only CRPD with nested preemption (K=1, N=4, BRT=1)

denoted by the ECBs of a preempter, which denote all memory blocks accessed during its entire execution.

Example *In Figure 6.1 task* τ_1 *preempts task* τ_2 *and causes preemption costs of* 0*. On the one hand, at most* 2 *UCBs can be evicted which therefore denotes a safe upper bound on the actual preemption costs. On the other hand, at most* 2 *ECBs of* τ_1 *evict blocks in* τ_2*, which therefore also denotes a safe upper bound on preemption costs. Taking both sets, UCB and ECB, into account yields a precise bound.*

In case of nested preemption, preemption costs may be indirect and care must be taken not to underestimate actual costs.

Example *In Figure 6.2 task* τ_3 *is indirectly preempted by task* τ_1*, causing preemption costs of* 2 *only due to evictions in the intermediate task* τ_2*. Hence, only taking* τ_1 *and* τ_3 *into account in isolation may be unsafe.*

Singleton Bounds

Let $\gamma_{i,j}$ denote an upper bound on the CRPD for task τ_i due to τ_j. Then a bound on the WCRT of τ_i including CRPD imposed by all higher priority tasks $\text{hp}(\tau_i)$ is denoted by:

$$R_i = C_i + \sum_{j \in \text{hp}(i)} \#(i,j) \times (C_j + \gamma_{i,j}) \tag{6.11}$$

Bound $\gamma_{i,j}$ has been previously proposed to be derived from UCB, ECB or combinations thereof. Some of these approaches yield unsafe bounds which have subsequently been corrected [6]. We briefly review the corrected variants.

Using just UCB is a safe bound on CRPD despite nested preemption is given as follows:

Definition 6.1 (UCB-only [88]) *A safe CRPD bound considering only UCB is given by:*

$$\gamma_{i,j} := BRT \times \max_{k \in \mathrm{aff}(i,j)} \mathrm{crpd}^{\mathrm{ucb}}(\mathrm{ucb}_\tau(k)) \tag{6.12}$$

with $\mathrm{crpd}^{\mathrm{ucb}}$ *defined by Equation 6.8.*

Example *In Figure 6.2 this yields precise bounds for the preemption of* τ_3 *by* τ_1*:*

$$\max_{k \in \mathrm{aff}(1,3)} \mathrm{crpd}^{\mathrm{ucb}}(\mathrm{ucb}_\tau(k)) = \max\{2, 2\} = 2$$

Similarly, CRPD can be bounded by ECB alone. In this case indirect costs need not be taken into account since ECBs always denote an upper bound on possible evictions.

Definition 6.2 (ECB-only [41]) *A safe CRPD bound considering only ECB is given by:*

$$\gamma_{i,j} := BRT \times \mathrm{crpd}^{\mathrm{ecb}}(\mathrm{ecb}_\tau(j)) \tag{6.13}$$

with $\mathrm{crpd}^{\mathrm{ecb}}$ *defined by Equation 6.9.*

Example *In Figure 6.2 this yields precise bounds for the preemption of* τ_3 *by* τ_1*:*

$$\mathrm{crpd}^{\mathrm{ecb}}(\mathrm{ecb}_\tau(j)) = 2$$

An obvious improvement is to take the actual interference of UCBs and ECBs into account. In Figure 6.1 the intersection of UCB and ECB yield a precise bound. In case of nested preemption, care has to be taken to account for all possible cases of indirection. Two conceptually symmetric approaches are known which solve the problem of indirect preemptions by computing supersets of either all sets of UCB of all possible preemptees by a preempter, or of all sets of ECBs of all possible preempters of a preemptee. In both cases, the respective UCB and ECB sets are eventually intersected. Simple intersection without previous transformation potentially misses indirect delays from nested preemptions.

The first technique is dubbed *UCB-union*. In the UCB-only approach above, the largest set of UCB over all program points of all preemptees is computed. The worst-case preemption cost for UCB-union, however, is the largest element in the cross-product of all preemptees. For example, in Figure 6.3, ECBs of τ_1 intersect with UCBs (at specific program points) in τ_2 and τ_3.

Definition 6.3 (UCB-union) *Let the projection of all UCB of affected tasks onto all program points of a preemptee* τ_i *be defined as:*

$$\mathrm{ucb}^{\times}_{\tau}(i) = \left\{ u \rightarrow \left\{ \bigcup_{k \in \mathrm{aff}(i,j)} \bigcup_{v \in V} \mathrm{ucb}_\tau(k)(v) \right\} \middle| \; u \in \mathrm{def}(\mathrm{ucb}_\tau(i)) \right\} \tag{6.14}$$

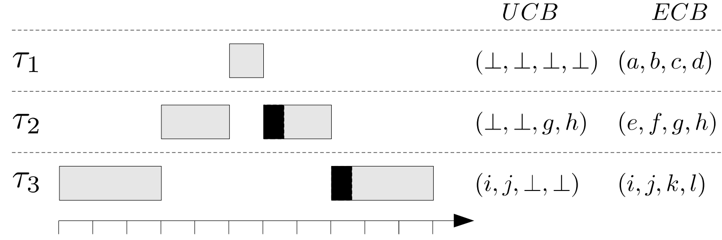


Figure 6.3: Example of imprecise UCB-union and precise ECB-union (K=1, N=4, BRT=1)

Then a safe CRPD bound considering combined UCB and ECB is defined as:

$$\gamma_{i,j} := BRT \times \mathrm{crpd}^{\mathrm{ucb,ecb}}\left(\mathrm{ucb}_\tau^\times(i), ecb_\tau(j)\right) \tag{6.15}$$

where $\mathrm{crpd}^{\mathrm{ucb,ecb}}$ *is defined according to Equation 6.10.*

Note that projection $\mathrm{ucb}_\tau^\times$ is only one possibility. In the original proposal [174] of this approach, UCB are projected onto a single set. We altered the definition here to adapt to $\mathrm{crpd}^{\mathrm{ucb,ecb}}$.

Example *For the example illustrated in Figure 6.3, we ignore the distinction of specific preemption points for the sake of simplicity. For the preemption by* τ_3*, actual preemption costs equal* 4 *and UCB-union yields a safe but not precise upper bound of* 6 *by computing:*

$$\gamma_{3,2} = \mathrm{crpd}^{\mathrm{ucb,ecb}}((i, j, \bot, \bot), (\bot, \bot, g, h)) = 0$$
$$\gamma_{3,1} = \mathrm{crpd}^{\mathrm{ucb,ecb}}((i, j, g, h), (a, b, c, d)) = 4$$

Symmetrically, we define a technique referred to as *ECB-union.* Here, ECB of all potential preempters are joined to compute interference with a preemptee. However, as we already pointed out for Figure 6.2, the worst-case may not be a direct preemption of task τ_j in the preemptee τ_i but a preemption in a task τ_k with $k \in \mathrm{aff}(i,j)$.

Definition 6.4 (ECB-union [94]) *A safe CRPD bound considering UCB and combined ECB is defined as:*

$$\gamma_{i,j} := BRT \times \max_{k \in \mathrm{aff}(i,j)} \mathrm{crpd}^{\mathrm{ucb,ecb}}\left(\mathrm{ucb}_\tau(k), \bigcup_{h \in \mathrm{hep}(j)} \mathrm{ecb}_\tau(h)\right) \tag{6.16}$$

where $\mathrm{crpd}^{\mathrm{ucb,ecb}}$ *is defined according to Equation 6.10.*

Intuitively, we account for the preemption by a task τ_j which has itself already been preempted by tasks τ_h. The worst-case, however, need not be a direct preemption of τ_i but might be any preemption of an intermediate task τ_k by τ_j that itself preempted τ_i

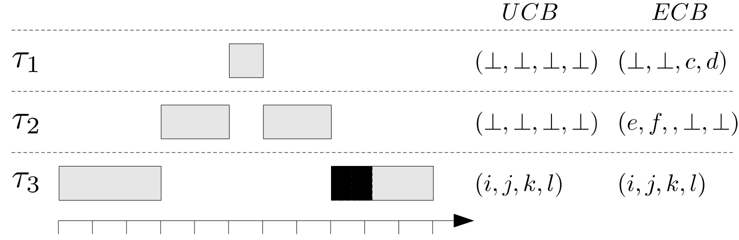


Figure 6.4: Example of precise UCB-union and imprecise ECB-union (K=1, N=4, BRT=1)

Example *Figure 6.4 illustrates an example scenario with an actual preemption cost of* 2*. ECB-union yields total preemption cost f* 6 *by computing:*

$$\begin{aligned}\gamma_{3,1} :=& \max\left\{\text{crpd}^{\text{ucb,ecb}}((i,j,k,l),(\bot,\bot,c,d)), \text{crpd}^{\text{ucb,ecb}}((\bot,\bot,\bot,\bot),(\bot,\bot,c,d))\right\} \\ =& \max\{2,0\} = 2 \\ \gamma_{3,2} :=& \max\left\{\text{crpd}^{\text{ucb,ecb}}((i,j,k,l),(e,f,c,d)),\right\} \\ =& \max\{4\} = 4\end{aligned}$$

Note that in Figure 6.3, ECB-union yields a precise bound as opposed to UCB-union, and in Figure 6.4 UCB-union yields a precise bound as opposed to ECB-union. Such overestimation is the result of accounting for the same evictions multiple times by forming supersets.

The bound we originally proposed in Section 4.6.4 is similar to the ECB-union approach which has only been proposed recently [94]. Instead of accounting for indirect preemption by joining ECB sets of preempting tasks, we accumulate direct preemption costs only but allow for an optimized treatment of successive interaction in non-direct mapped caches. We restate our bound here without CBR for reference only.

Definition 6.5 (ECB-union*) *A safe CRPD bound considering UCB and only interacting ECB is defined as:*

$$\gamma_{i,j} := BRT \times \sum_{k \in \text{aff}(i,j)} \text{crpd}^{\text{ucb,ecb}} \left(\text{ucb}(k), \bigcup_{h \in \text{hep}(j)}^{\star} \text{ecb}(h) \right) \tag{6.17}$$

where $\bigcup^\star$ *only joins interacting cache sets (cf. Section 4.6.4) and where* $\text{crpd}^{\text{ucb,ecb}}$ *is defined by Equation 6.10.*

Similar to UCB-union and ECB-union, this bound suffers from potentially accounting for the same evictions repeatedly.

ECB-union and ECB-union* can be combined with CBR (cf. Section 4.5.3) to enhance precision. In the following in order to simplify the discussion, we will not be concerned with matters of interaction and constrain ourselves to direct-mapped caches.

Multiset Bounds

We refer to the previous approaches as singleton bounds since we compute a single bound which must be safe for *all* preempting jobs (cf. Equation 6.11). Several authors [94, 175, 176] proposed the discrimination of individual preemptions by maintaining multisets of preemption costs that *individual* preempting jobs can impose. In the following we only discuss the most precise of these approaches.

Let $\gamma^m_{i,j}$ denote a bound that denotes the accumulated costs of individual preemptions of a task τ_i by a task τ_j. Then WCRT for τ_i is denoted by:

$$R_i = C_i + \sum_{j \in \text{hp}(i)} \left(\#(i,j) \times C_j + \gamma^m_{i,j}\right) \tag{6.18}$$

Intuitively, $\gamma^m_{i,j}$ is computed as follows: For preemptions of τ_i by τ_j, multiset M contains the costs of all direct and indirect preemptions that τ_j possibly imposes on τ_i. This set may be larger than the actual number of jobs of τ_j in the response time of τ_j. Hence, only the sum of the $\#(i,j)$ largest values in M denote a safe upper bound on the total preemption costs.

As the number of preempters increases, the number of scenarios grows exponentially, potentially accounting for the very same evictions many times. For example, in Figure 6.5, only τ_1 imposes (indirect) preemption costs onto τ_3. Since $\#(3,1) = 2$, this leads to an unnecessarily imprecise bound since costs attributed to τ_1 stem from τ_1 preempting τ_2 which itself only repeats $\#(3,2) = 1$ times in the response time of τ_3. Therefore, we [94] recognize that any task τ_k with $k \in \text{aff}(i,j)$ can not be preempted more often than $\#(k,j)$ times by a preempter τ_j and does itself not preempt τ_i more often than $\#(i,k)$ times.

Definition 6.6 (Multiset [6]) *Let $\gamma^s_{i,j}$ denote any singleton bound as discussed above, then multiset $M_{i,j}$, which denotes preemption costs of a preempter τ_j for a preemptee τ_i, is defined as:*

$$M_{i,j} := \bigcup_{k \in \text{aff}(i,j)} \left\{\gamma^s_{k,j}\right\}^{\#(k,j) \times \#(i,k)} \tag{6.19}$$

Let $\max^n : S \mapsto S$ *denote the n largest elements in a set S. Then*

$$C_{i,j} = \max^{\#(i,j)} M_{i,j} \tag{6.20}$$

denotes the $\#(i,j)$ largest values in multiset $M_{i,j}$. A bound according to Equation 6.18 is then defined as:

$$\gamma^m_{i,j} := \text{BRT} \times \sum C_{i,j} \tag{6.21}$$

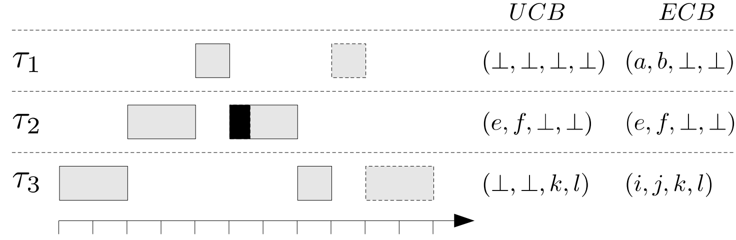


Figure 6.5: Example for effectiveness of multiset approaches (K=1, N=4, BRT=1)

For the singleton bound $\gamma^s_{i,j}$, we can apply either UCB-union or ECB-union approaches as discussed above. For UCB-union (Equation 6.15), it is defined as:

$$\gamma^s_{i,j} := \text{crpd}^{\text{ucb,ecb}}\left(\text{ucb}^{\times}_{\tau}(i), \text{ecb}_{\tau}(j)\right) \tag{6.22}$$

For ECB-union (Equation 6.16), it is defined as:

$$\gamma^s_{i,j} := \text{crpd}^{\text{ucb,ecb}}\left(\text{ucb}_{\tau}(i), \bigcup_{h\in\text{hep}(j)} \text{ecb}_{\tau}(h)\right) \tag{6.23}$$

Multiset approaches do not tighten the bounds on individual preemptions per se. Instead they accumulate individual preemption costs as opposed to applying a single upper bound to all preemptions as in previous approaches.

Example *Figure 6.5 illustrates the improvement of the multiset CRPD bounds over singleton CRPD bounds. We assume ECB-union. Then singleton preemption costs $\gamma^s_{i,j}$ are denoted by:*

$$\gamma^s_{3,1} := \max\left\{\text{crpd}^{\text{ucb,ecb}}((\bot,\bot,k,l),(a,b,\bot,\bot))\right\} = \max\{0\} = 0$$
$$\gamma^s_{2,1} := \max\left\{\text{crpd}^{\text{ucb,ecb}}((e,f,\bot,\bot),(a,b,\bot,\bot)),\right\} = \max\{2\} = 2$$

Then multiset $M_{i,j}$ yields:

$$M_{3,1} = \{\gamma^s_{3,1}\}^{\#(3,1)\times\#(3,3)} \cup \{\gamma^s_{2,1}\}^{\#(2,1)\times\#(3,2)} = \{0,0\} \cup \{2\}$$

Total preemption costs for task τ_1 preempting τ_3 is then bounded by:

$$\gamma^m_{3,1} = \sum \max{}^{\#(3,1)} M_{3,1} = \sum\{2,0\} = 2 \tag{6.24}$$

With singleton UCB-union or ECB-union, preemption costs are bounded by 4 *since indirect preemption costs of τ_1 preempting τ_3 is accounted for twice.*

Multiset approaches cannot prevent overestimation in the singleton bounds they employ internally, but in cases of potentially many preemptions (high CPU utilization), they are more precise than their purely singleton counterparts.

6.1.3 A Refined Bound on CRPD

We propose a new bound on CRPD which precisely accounts for block evictions. In the singleton approaches discussed above, supersets are composed, which either represent safe upper bounds on evicted or evicting memory blocks, respectively. Consequently, identical evictions are accounted for multiple times. Multiset mitigates overestimation by discriminating evictions of individual preempting jobs. Nonetheless, pessimism in estimating costs of nested preemptions remain by ultimately relying on the same singleton bounds.

The fundamental restriction of existing bounds is the lack of contextual information to precisely bound evictions. Response time according to Equation 6.11 or Equation 6.18 is ultimately composed of CRPD estimates of just a preemptee τ_i and a preempter τ_j such that indirect evictions in tasks τ_k of intermediate priority must be safely bounded. Actual evictions in all tasks preempted by τ_j, however, depend on specific nesting scenarios, which are unknown by only taking τ_i and τ_j into account in isolation. The approach we are to propose in the following enumerates all possible preemption scenarios, precisely accounting for evictions. Accordingly, response time is not based on estimates for a specific preempter but by accumulating all preemptions costs for a single preemptee:

$$R_i = C_i + \sum_{j \in \mathrm{hp}(i)} (\#(i,j) \times C_j) + \gamma_i^b \tag{6.25}$$

In the following we introduce the basic principles and start from a naive and imprecise bound which we successively refine.

First, we introduce the underlying basic data structure "bucket". Then we show how a complete enumeration of all preemption scenarios can be achieved. To achieve tight bounds, we then successively optimize the enumeration approach.

Buckets

Instead of collecting preemption costs in a multiset attributed to a specific preempter which represents its own preemption costs as well as those of all its lower priority tasks cumulatively, we maintain "buckets" that precisely account eviction costs to all individual tasks potentially involved in a preemption nesting.

Definition 6.7 (Bucket) Buckets $B\colon \mathcal{T} \mapsto \mathbb{N}_0{}^*$ *are a mapping from tasks to preemption costs and represent worst-case preemption costs in a specific preemption context. Joining of buckets* $b, b' \in B$ *is defined as task-wise concatenation such that:*

$$b \cup b' := \left\{\tau \to \oplus(b,b')(\tau) \;\middle|\; \tau \in \mathrm{def}(b) \cup \mathrm{def}(b')\right\} \tag{6.26}$$

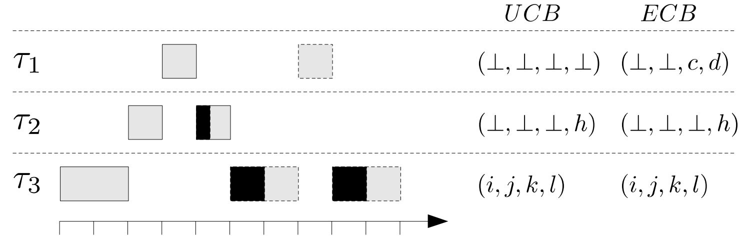


Figure 6.6: Example of overhead by naive enumeration (K=1, N=4, BRT=1)

where

$$\oplus \colon B \times B \mapsto (\mathcal{T} \mapsto \wp(\mathbb{N}_0))$$

$$\oplus(b, b') = \lambda\tau \, . \begin{cases} b(\tau) \cdot b'(\tau) & \textit{if } \tau \in \operatorname{def}(b) \cap \operatorname{def}(b') \\ b(\tau) & \textit{if } \tau \in \operatorname{def}(b) \\ b'(\tau) & \textit{otherwise} \end{cases} \tag{6.27}$$

Example *Let* $b = \{\tau_1 \to (a), \tau_2 \to (b)\}$ *and* $b' = \{\tau_1 \to (a)\}$ *then* $b \cup b' = \{\tau_1 \to (a, a), \tau_2 \to (b)\}$.

The purpose of buckets is similar to multisets. We use sequences just for technical reasons.

Enumeration

Instead of computing bounds for just a pair of tasks, where indirect preemptions are taken into account implicitly, our approach is based on enumeration of all scenarios explicitly. A naive bound based on this idea is given by:

Definition 6.8 (Enumeration) *Let response time be defined as in Equation 6.25. Then a bound on CRPD by enumeration of all preemptions is defined as:*

$$\gamma_i^b := BRT \times c(i) \tag{6.28}$$

where

$$c \colon \mathcal{T} \mapsto \mathbb{N}_0$$

$$c(j) = \sum_{h \in \operatorname{hp}(j)} \#(j, h) \times \Big(c(h) + \operatorname{crpd}^{\text{ucb,ecb}} \left(\operatorname{ucb}_\tau(j), \operatorname{ecb}_\tau(h) \right) \Big) \tag{6.29}$$

For every task, we accumulate preemption costs due to higher priority tasks. This bound is trivially safe by considering all possible combinations but is highly pessimistic as it overestimates the number of preemptions as well as the number of evictions.

Example *Consider Figure 6.6, which illustrates an example with total preemption costs of* 5*. Enumeration for the preemption if task* τ_3 *according to Equation 6.29 yields:*

$$
\begin{aligned}
c(1) &= 0 \\
c(2) &= \Big(\#(2,1) \times \Big(c_1 + \mathrm{crpd}^{\mathrm{ucb,ecb}}\left((\bot,\bot,\bot,h),(\bot,\bot,c,d)\right)\Big)\Big) = 1 \times 1 \\
c(3) &= \#(3,2) \times \Big(c(2) + \mathrm{crpd}^{\mathrm{ucb,ecb}}\left((i,j,k,l),(\bot,\bot,\bot,h)\right)\Big) \\
&\quad + \#(3,1) \times \Big(c(1) + \mathrm{crpd}^{\mathrm{ucb,ecb}}\left((i,j,k,l),(\bot,\bot,c,d)\right)\Big) \\
&= 1 \times (1 \times 1 + 1) + 2 \times 2 = 6
\end{aligned}
$$

Evictions due to task τ_1 *are taken into account three times although during the response time of task* τ_3 *only* 2 *jobs are possible and it is not taken into account that in the nested preemption* τ_1 *and* τ_2 *evict the very same blocks in* τ_3*.*

Instead of directly accumulating costs, we can rewrite Equation 6.29 to fill buckets which are evaluated only after all costs have been accumulated.

Definition 6.9 (Bucket-based Enumeration) *Let response time be defined as in Equation 6.25. Then a bound on CRPD by collecting all preemptions costs in buckets is defined as:*

$$\gamma_i^b := BRT \times \sum_{j \in \mathrm{hp}(i)} \sum B(j) \colon B = c(i) \tag{6.30}$$

where

$$
\begin{aligned}
c &\colon \mathcal{T} \mapsto \mathcal{B} \\
c(j) &= \bigcup_{h \in \mathrm{hp}(j)} \Big(c(h) \cup \Big\{h \to \mathrm{crpd}^{\mathrm{ucb,ecb}}\left(\mathrm{ucb}_\tau(j), \mathrm{ecb}_\tau(h)\right)\Big\}\Big)^{\#(j,h)}
\end{aligned}
\tag{6.31}
$$

Example *We reconsider the example illustrated in Figure 6.6. Buckets* c_3 *according to Equation 6.31 yields:*

$$
\begin{aligned}
c(1) &= \emptyset \\
c(2) &= \{\tau_1 \to (1)\}^{\#(2,1)} \\
c(3) &= \{c(2) \cup \{\tau_2 \to (1)\}\}^{\#(3,2)} \cup \{\tau_1 \to (2)\}^{\#(3,1)} \\
&= \{\{\tau_1 \to (1)\} \cup \{\tau_2 \to (1)\}\} \cup \{\tau_1 \to (2,2)\} \\
&= \{\tau_1 \to (1,2,2), \tau_2 \to (1)\}
\end{aligned}
$$

Accumulating costs according to Equation 6.30 yields a cost estimate of 6 *— equal to the previous example.*

In the following we improve the bucked-based enumeration approach and first address overestimation of preemption counts and then address overestimation due to duplicate evictions.

Limiting Preemption Counts

While the number of possible scenarios is exponential to the number of tasks, the actual number of possible preemptions is limited by the maximal number of jobs of a preempter within the response time of a preemptee. We therefore must guarantee the following invariant: A bucket, at any time during composition, is guaranteed never to contain more elements than globally feasible.

Assumption 6.10 *Let τ_i be a preemptee and $B \in \wp(B)$ a set of buckets representing preempters. Then it must hold that:*

$$\forall k \in \mathrm{hp}(i)\colon |B(k)| \leq \#(i,k) \tag{6.32}$$

Let task τ_i denote a task for which CRPD is to be computed. We introduce two helper functions. The first function removes from buckets $\tau_k \in \mathrm{def}(B)$, with $\tau_k \in \mathrm{hp}(i)$, all but the $\max^{\#(i,k)}$ largest preemption costs with respect to a task τ_i. We define it as:

$$\mathrm{rm}_i := \lambda B \, . \left\{ k \to \max{}^{\#(i,k)}\, C \;\middle|\; (k \to C) \in \mathrm{def}(B) \right\} \tag{6.33}$$

This guarantees Assumption 6.10 for a set of buckets in general.

The second function duplicates cost values in a similar fashion to multisets above. Let task τ_i denote a task for which CRPD is to be computed. Let $\tau_j \in \mathrm{hp}(i)$ be a preemptee, let $\tau_h \in \mathrm{hp}(j)$ be its direct preempter and let $\tau_k \in \mathrm{hep}(h)$ denote either τ_h or tasks $\mathrm{hp}(h)$ that indirectly preempt τ_j by nesting in τ_h. Let B denote the corresponding buckets denoting the singleton cost of one job of τ_h preempting τ_j directly and costs of all jobs of tasks τ_k preempting τ_j indirectly. Then we define a function mu_i which first duplicates all costs in B $\#(j,h)$ times and afterwards constrains costs by Equation 6.33. We define it as:

$$\mathrm{mu}_i := \lambda(j,h,B) \, . \, \mathrm{rm}_i \left\{ k \to C^{\#(j,h)} \;\middle|\; (k \to C) \in \mathrm{def}(B) \right\} \tag{6.34}$$

Locally, any indirect preemption by nesting in τ_h or direct preemption by τ_h repeat at most $\#(j,h)$ times within the response time of τ_j. Globally, no preemption can occur more often than $\#(i,j)$ times within the response time of τ_i.

Enumerating (and accumulating) *all* preemptions is safe by construction since all possible evictions are accounted for at least once. When purging preemption costs from a bucket, however, we lose information which might not be redundant.

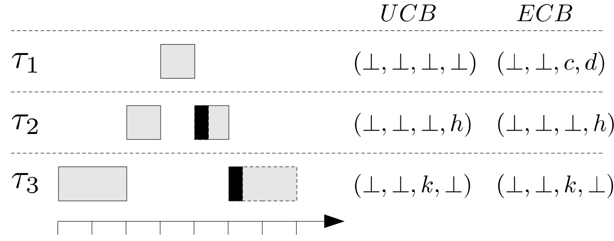


Figure 6.7: Example of underestimation by constrained buckets (K=1, N=4, BRT=1)

Example *Consider the illustration in Figure 6.7. Task τ_1 preempts τ_3 at most once but enumeration yields two scenarios. Constraining the bucket of τ_1 underestimates that the single preemption by τ_1 evicts blocks in both τ_2 and τ_3.*

Therefore, instead of accounting for direct preemptions only, we have to take potential nestings of a preemption by a task τ_h into account.

Let $\pi \in \wp(\mathcal{T})$ denote a potential nesting of a task τ_h preempting a task τ_i such that $\forall \tau_k \in \pi \colon \tau_k \in \mathrm{aff}(i, h)$. Then we define preemption costs of τ_h as the accumulated costs over π as:

$$\mathrm{co} := \lambda(\pi, h) \, . \left\{ h \to \sum_{k \in \pi} \mathrm{crpd}^{\mathrm{ucb,ecb}} \left(\mathrm{ucb}_\tau(i), \mathrm{ecb}_\tau(k) \right) \right\} \tag{6.35}$$

Finally, we can define a bound on CRPD that takes Assumption 6.10 properly into account.

Definition 6.11 (Constrained-bucket Enumeration) *Let response time be defined as in Equation 6.25. Then a safe bound on CRPD for a preemptee τ_i is given by:*

$$\gamma_i^b := BRT \times \sum_{j \in \mathrm{hp}(i)} \sum B(j) \colon B = c_i(i, \emptyset) \tag{6.36}$$

where

$$c_i \colon \mathcal{T} \times \wp(\mathcal{T}) \mapsto \mathcal{B}$$

$$c_i(j, \pi) = \mathrm{rm}_i \left(\bigcup_{h \in \mathrm{hp}(j)} \mathrm{rm}_i \left(\mathrm{mu}_i \left(j, h, B \cup \mathrm{co} \left(\pi \cup \{j\}, h \right) \right) \colon B = c_i \left(h, \pi \cup \{j\} \right) \right) \right) \tag{6.37}$$

denotes a set of buckets such that Assumption 6.10 holds.

As in Equation 6.31, we enumerate all cases recursively. Set $c_i(j, \pi)$ denotes a set of buckets representing the maximal constrained preemption costs in all scenarios of higher priority tasks. During recursion, task τ_j always denotes a preemptee and τ_h denotes its direct preempter. For every preemption by τ_h, buckets $B = c_i(h, \pi \cup \{j\})$ denote preemption costs of higher priority tasks. Then preemption costs in the current task τ_j are composed of B and the costs $\mathrm{co}(\pi \cup \{j\}, h)$ that τ_h imposes on τ_j and all lower priority tasks π. All preemptions of τ_h and higher priority tasks may repeat $\#(j, h)$ times

during the response time of τ_j (mu_i). However, globally, no preemption repeats more often than $\#(i,j)$ times (rm_i). The latter is true for each preempter τ_h individually, as well as for all scenarios relative to τ_j collectively. Hence, scenarios are purged twice.

Example *We reconsider the example in Figure 6.6 with an actual preemption cost of* 5 *and compute buckets according to Equation 6.37. In the following we list the respective return values of the recursive invocation starting from* $c_3(3,\emptyset)$*:*

$$
\begin{aligned}
c_3(1,\{3,2\}) &= c_3(1,\{3\}) = \emptyset \\
c_3(2,\{3\}) &= \mathrm{rm}_3\left(\bigcup_{h\in\{1\}} \mathrm{rm}_3\left(\mathrm{mu}_3(2,h,B\cup \mathrm{co}(\{3,2\},h)\colon B = c_3(h,\{3,2\})\right)\right) \\
&= \mathrm{rm}_3\left(\mathrm{rm}_3\left(\mathrm{mu}_3(2,1,\{\tau_1\to(3)\})\right)\right) \\
&= \max^{\#(3,h)}\left(\max^{\#(3,h)}\left(\{\tau_1\to(3)\}^{\#(2,h)}\right)\right) \\
&= \{\tau_1\to(3)\} \\
c_3(3,\emptyset) &= \mathrm{rm}_3\left(\bigcup_{h\in\{2,1\}} \mathrm{rm}_3\left(\mathrm{mu}_3(3,h,B\cup \mathrm{co}(\{3\},h)\colon B = c_3(h,\{3\})\right)\right) \\
&= \mathrm{rm}_3\begin{pmatrix} \mathrm{rm}_3\left(\mathrm{mu}_3(3,2,B\cup \mathrm{co}(\{3\},2)\colon B = c_3(2,\{3\})\right) \\ \cup\,\mathrm{rm}_3\left(\mathrm{mu}_3(3,1,B\cup \mathrm{co}(\{3\},1)\colon B = c_3(1,\{3\})\right)\end{pmatrix} \\
&= \mathrm{rm}_3\begin{pmatrix} \mathrm{rm}_3\left(\mathrm{mu}_3(3,2,\{\tau_1\to(3)\}\cup \mathrm{co}(\{3\},2))\right) \\ \cup\,\mathrm{rm}_3\left(\mathrm{mu}_3(3,1,\mathrm{co}(\{3\},1))\right)\end{pmatrix} \\
&= \mathrm{rm}_3\begin{pmatrix} \mathrm{rm}_3\left(\mathrm{mu}_3(3,2,\{\tau_1\to(3),\tau_2\to(1)\})\right) \\ \cup\,\mathrm{rm}_3\left(\mathrm{mu}_3(3,1,\{\tau_1\to(2)\})\right)\end{pmatrix} \\
&= \max^{\#(3,h)}\begin{pmatrix} \max^{\#(3,h)}\left(\{\tau_1\to(3),\tau_2\to(1)\}^{\#(3,2)}\right) \\ \cup\,\max^{\#(3,h)}\left(\{\tau_1\to(2)\}^{\#(3,1)}\right)\end{pmatrix} \\
&= \max^{\#(3,h)}\left(\{\tau_1\to(3),\tau_2\to(1)\}\cup\{\tau_1\to(2,2)\}\right) \\
&= \{\tau_1\to(3,2),\tau_2\to(1)\}
\end{aligned}
$$

Accumulated preemption costs in τ_i *equal* 6*. Note that from* 3 *possible preemption scenarios of* τ_1 *just* 2 *are globally feasible with respect to* τ_3*.*

Avoiding Duplicate Preemption Costs

Although we minimize bucket sizes in Equation 6.37 to only account for a globally feasible number of preemptions by individual jobs, we still overestimate the amount of evictions single preemptions can cause. For example, in Figure 6.7, we account for the eviction of UCB l by ECB h and d, separately, although in case of a nesting of τ_1 in τ_2, block l can only be evicted once. However, we recognize that if we can ensure a preemption by τ_1 is nested within τ_2 then we can exclude all the UCB from consideration by τ_1 that have

been (or will be) already evicted by lower priority tasks that form the nesting. Recursive enumeration allows just that.

Axiom 6.12 *Given Equation 6.37, for any invocation of $c_i(j, \pi)$, set π denotes the set of tasks forming a nesting into which τ_j is embedded.*

Consequently, in Equation 6.35 evictions need only be taken into account for those UCB that have not been (or will be) already evicted by lower priority tasks in a respective nesting.

We define an alternative to $\mathrm{crpd}^{\mathrm{ucb,ecb}}$ which denotes the set of evicted (useful) cache blocks instead of just its cardinality as:

$$\mathrm{crpd}^{\mathrm{ucb,ecb}}_{\mathcal{M}} \colon \mathrm{UCB} \times ECB \times \wp(\mathcal{M}) \to \wp(\mathcal{M})$$
$$\mathrm{crpd}^{\mathrm{ucb,ecb}}_{\mathcal{M}}(\mathrm{ucb}, \mathrm{ecb}, M) = \max_{u \in V} \left\{ \bigcup_{s \in [1,N]} \left\{ \max{}^{K} \left(\mathrm{ucb}(u)_s \setminus M\right) \colon \mathrm{ecb}_s \neq \emptyset \right\} \right\} \tag{6.38}$$

where set M denotes UCBs to be excluded from consideration. For brevity, we define:

$$\mathrm{ev} := \lambda(j, h, M)\,.\,\mathrm{crpd}^{\mathrm{ucb,ecb}}_{\mathcal{M}}(\mathrm{ucb}_\tau(j), \mathrm{ecb}_\tau(h), M) \tag{6.39}$$

Accordingly, we redefine Equation 6.35 to adapt and define accumulated costs for a preempter τ_h along nesting path π with exclusion M as:

$$\mathrm{co} := \lambda(\pi, h, M)\,.\left\{ h \to \sum_{k \in \pi} |\mathrm{ev}(k, h, M)| \right\} \tag{6.40}$$

Let functions rm_i (Equation 6.33) mu_i (Equation 6.34) be unchanged.

Definition 6.13 (Context-sensitive Constrained-bucket Enumeration) *Let response time be defined as in Equation 6.25. Then a safe bound on CRPD for a preemptee τ_i is given by:*

$$\gamma_i^b := BRT \times \sum_{j \in \mathrm{hp}(i)} \sum B(j) \colon B = c_i(i, \emptyset, \emptyset) \tag{6.41}$$

where

$$c_i \colon \mathcal{T} \times \wp(\mathcal{T}) \times \wp(\mathcal{M}) \mapsto \mathcal{B}$$
$$c_i(j, \pi, M) =$$
$$\mathrm{rm}_i \left(\bigcup_{h \in \mathrm{hp}(j)} \mathrm{rm}_i \left(\mathrm{mu}_i \begin{pmatrix} j, \\ h, \\ B \cup \mathrm{co}\left(\pi \cup \{j\}, h, M\right) \end{pmatrix} \colon B = c_i \begin{pmatrix} h, \\ \pi \cup \{j\}, \\ M \cup \mathrm{ev}(j, h, \emptyset) \end{pmatrix} \right) \right) \tag{6.42}$$

Intuitively, we propagate already (or to be) evicted cache blocks "upwards" the nesting hierarchy to avoid duplicate evictions, then collect and constrain buckets while unwinding the recursion. This removes the inherent overestimation of existing bounds while keeping the number of assumed preemptions minimal. Note that this bound can be combined with CBR by propagating the aging of UCBs instead of just the UCBs themselves to achieve greater precision in case of non-direct-mapped caches.

Example *We reconsider the example in Figure 6.6 with an actual preemption cost of* 5 *and compute buckets according to Equation 6.42. In the following we list the respective return values of the recursive invocation starting from* $c_3(3, \emptyset, \emptyset)$ *just as in the previous example:*

$$c_3(1, \{3, 2\}, \{l, h\}) = c_3(1, \{3\}, \{k, l\}) = \emptyset$$

$$\begin{aligned}
c_3(2, \{3\}, \{l\}) &= \mathrm{rm}_3 \left(\bigcup_{h \in \{1\}} \mathrm{rm}_3 \left(\mathrm{mu}_3(2, h, B \cup \mathrm{co}(\{3, 2\}, h, \{l\}) \colon B = c_3(h, \{3, 2\}, \{l, h\}) \right) \right) \\
&= \mathrm{rm}_3 \left(\mathrm{rm}_3 \left(\mathrm{mu}_3(2, 1, \{\tau_1 \to (2)\} \right) \right) \\
&= \dots \\
&= \{\tau_1 \to (2)\} \\
c_3(3, \emptyset) &= \mathrm{rm}_3 \left(\bigcup_{h \in \{2,1\}} \mathrm{rm}_3 \left(\mathrm{mu}_3(3, h, B \cup \mathrm{co}(\{3\}, h, \emptyset) \colon B = c_3(h, \{3\}, \{l\}) \right) \right) \\
&= \mathrm{rm}_3 \left(\begin{array}{l} \mathrm{rm}_3 \left(\mathrm{mu}_3(3, 2, B \cup \mathrm{co}(\{3\}, 2, \emptyset) \colon B = c_3(2, \{3\}, \{l\}) \right) \\ \quad \cup\, \mathrm{rm}_3 \left(\mathrm{mu}_3(3, 1, B \cup \mathrm{co}(\{3\}, 1, \emptyset) \colon B = c_3(1, \{3\}, \{l\}) \right) \end{array} \right) \\
&= \dots \\
&= \max{}^{\#(3,h)} \left(\{\tau_1 \to (2), \tau_2 \to (1)\} \cup \{\tau_1 \to (2, 2)\} \right) \\
&= \{\tau_1 \to (2, 2), \tau_2 \to (1)\}
\end{aligned}$$

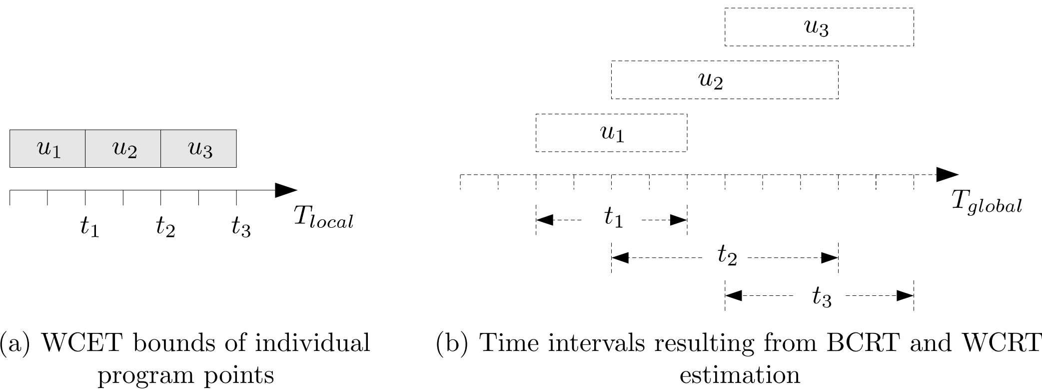
Accumulated preemption costs in τ_i *equal* 5*, which is precise for the given example. Note that examples in Figure 6.3 (imprecise UCB-union) and Figure 6.4 (imprecise ECB-union) are also precisely estimated by this approach.*

6.2 Improving CRPD Estimation with Time Bounds

Conventional CRPD bounds assume that preemptions can occur at arbitrary points within a task. We can improve such estimations by recognizing that jobs of preempters are constrained by well-defined time-frames in which they are being activated. Consequently, not all program points of a preemptee are subject to the same preemption. In Chapter 5, we showed how to compute lower and upper execution time bounds for individual program points. We exploit the availability of this information to exclude UCBs from eviction by jobs of higher priority tasks in CRPD estimation.

We first introduce the idea intuitively. Then we specify time bounds formally and show how they are applied to tighten the class of multiset CRPD bounds.

Intuition



(a) WCET bounds of individual program points

(b) Time intervals resulting from BCRT and WCRT estimation

Figure 6.8: Mapping from task-local execution time bounds to system time

The example in Figure 6.8 illustrates the intuition. Figure 6.8a outlines the execution of a task which is composed of program points $\{u_1, u_2, u_3\}$ with associated completion times $\{t_1, t_2, t_3\}$, respectively. Time T_{local} denotes task-local execution time in which potential preemption is not taken into account. Given a set of higher priority tasks, we can map the local time instants into time intervals in the global system time by estimating BCRT and WCRT for each program point, respectively. Figure 6.8b illustrates the corresponding intervals in time T_{global}, in which given program points may complete despite preemption. Preemption occurs periodically in T_{global}. Consequently, we can derive sets of UCBs potentially evicted by specific jobs of higher priority tasks. In particular, this allows to exclude worst-case sets of UCBs from consideration for some preemptions. Recall from above that otherwise only a single worst-case program point is taken into account. Note that we simplified the example: completion time in T_{local} are also time intervals $[\text{BCET}^{est}, \text{WCET}^{est}]$ from which we derive intervals $[\text{BCRT}, \text{WCRT}]$, respectively.

Time-bounded Cache Interference

For a task $\tau_i \in \mathcal{T}$, let V_i denote its program points. Let

$$\text{tb}_i \colon V_i \mapsto (\mathbb{N}_0^\infty)^2 \tag{6.43}$$

be a mapping from program points to intervals, where lower bounds denote BCRT and upper bounds denote WCRT for each such program point individually.

Theorem 6.14 *Let T_j denote the period of a higher priority task τ_j. Then a program point $u \in V_i$ may only be a preemption point if and only if $T_j \in \text{tb}_i(u)$.*

Proof. We assume no blocking. Then τ_j preempts τ_i instantly at T_j. By definition of BCRT (Equation 3.21 on page 30) and WCRT (Equation 3.18 on page 30), a program point u may have already been completed or may not have been executed if $T_j \notin \mathrm{tb}_i(u)$, respectively. □

For BCRT, we can assume no CRPD since in the best case no evictions occur. For WCRT any of the aforementioned bounds on CRPD can be assumed. We recognize that under this assumption, time bounds tb denote program points u subject to preemption and therefore also denotes UCBs $ucb(u)$ subject to eviction by a specific job of a preempting task.

An Upper Bound on Multiset CRPD

Let $t\colon \mathrm{tb}_i$ denote time bounds of task τ_i, let τ_j be a preempter and let $r \in \mathbb{N}_0$ denote job τ_j^r. Then the set of program points potentially subject to preemption is denoted by:

$$V_{i,j}^{\mathrm{tb}}(t, r) := \{u \mid (r \times T_j) \in t(u), u \in V_i\} \tag{6.44}$$

Accordingly, we wrap $V_{i,j}^{\mathrm{tb}}$ and define the function of UCBs subject to eviction by job τ_j^r as:

$$\mathrm{ucb}_{i,j}^{\mathrm{tb}}(t, r) := \lambda u \,.\, \left\{\mathrm{ucb}_{\mathcal{T}}(i)(u) \colon u \in V_{i,j}^{\mathrm{tb}}(t, r)\right\} \tag{6.45}$$

where $\mathrm{ucb}_{\mathcal{T}}(i)(u)$ denotes UCBs of task τ_i in program point u. Consequently, the time-bounded singleton ECB-union CRPD bound (cf. Definition 6.16) for a preemption of job τ_j^r is denoted by:

$$\gamma_{i,j}^{\mathrm{tb}}(t, r) := \mathrm{crpd}^{\mathrm{ucb,ecb}} \left(\mathrm{ucb}_{i,j}^{\mathrm{tb}}(t, r), \bigcup_{h \in \mathrm{hep}(j)} \mathrm{ecb}_{\mathcal{T}}(h) \right) \tag{6.46}$$

We now compute a multiset (cf. Definition 6.6) from singleton bounds, which allows to discriminate individual preempting jobs. Recall that above multisets represent all preemption costs within the response of a task. We generalize this notion and compute multisets with respect to individual program points. For task τ_i, let $C_i^u \colon V \mapsto \mathbb{N}_0^\infty$ denote WCET per program point.

We further denote the response time of a program point in τ_i by a function $\bar{R}_i$, which we will define below. Then the number of preemptions of a task τ_j within the response time of program point $u_i \in V_i$ in τ_i, given time bounds t, is defined as:

$$\#(t, i, u_i, j) := \left\lceil \frac{\bar{R}_i(t)(u_i) + J_i}{T_j} \right\rceil \tag{6.47}$$

Let $x_i \in V_i$ denote the terminal program point of a task τ_i. Then the corresponding multiset of preemption costs is defined as:

$$M^{\text{tb}}_{i,j}(t, u_i) := \bigcup_{k \in \text{aff}(i,j)} \left(\left\{ \gamma^{\text{tb}}_{k,j}(t, r) \colon r \in [0, \#(t, k, x_k, j) - 1] \right\}^{\#(t,i,u_i,k)} \right) \tag{6.48}$$

where r denotes the jobs τ_j^r that preempt any task $\tau_k \in \text{aff}(\tau_i, \tau_j)$ and where $\gamma^{\text{tb}}_{k,j}$ denotes the corresponding time-bounded singleton preemption costs. Such preemptions repeat at most $\#(t, \tau_k, x_k, \tau_j)$ times during the response time of a task τ_k which in turn repeat at most $\#(t, \tau_i, u_i, \tau_k)$ times during the response time of node u_i of task τ_k.

As usual, CRPD is bounded by the $\#(t, \tau_i, u_i, \tau_j)$ largest preemption costs in multiset $M^{\text{tb}}_{i,j}$. We therefore define the time-bounded multiset bound on CRPD as:

$$\gamma^m_{i,j}(t, u_i) := \text{BRT} \times \sum \max\nolimits^{\#(t,i,u_i,j)} M^{\text{tb}}_{i,j}(t, u_i) \tag{6.49}$$

Then WCRT, given time bounds t, with respect to a single program point $u_i \in V_i$ in τ_i, is denoted by the fixed point of:

$$\bar{R}_i \colon (V \mapsto (\mathbb{N}_0^\infty)^2) \mapsto (V \mapsto \mathbb{N}_0^\infty)$$

$$\bar{R}_i^{(n+1)}(t) = \lambda u_i \,.\, C_i^u(u_i) + \sum_{j \in \text{hp}(i)} \left(\left\lceil \frac{\bar{R}_i^{(n)}(t)(u_i) + J_i}{T_j} \right\rceil \times C_j^u(x_j) + \gamma^m_{i,j}(t) \right) \tag{6.50}$$

where $\bar{R}_i^{(0)}(u) = C_i(u)$. Note that $\bar{R}_i$, just like R_i (Equation 6.18), is monotonously increasing. We denote the fixed point by $\bar{R}_i(t)$.

It is important to recognize that time bounds t denote response times that we already (somehow) computed, and whose purpose it is to exclude UCBs from eviction by specific preempting jobs. Otherwise they do not interact with the current round of response time computations carried out in Equation 6.50. To summarize, the only differences from a standard multiset approach is the generalization to individual program points and the consideration of time bounds within the singleton bound and within the multiplication of preemption costs to compose the multiset.

A Lower Bound On Task Interference

Response times $\bar{R}_i$ denote WCRT per program point given upper and lower response time bounds per program point. Time bounds increase the precision of $\bar{R}_i$. For initial time bounds $t^{(0)} = \{V_i \times \{[0, \infty]\}\}$, $\bar{R}_i$ yields unconstrained response times such that $\bar{R}_i(t^{(0)})(x_i)$, where $x_i \in V_i$ denotes the terminal program point of task τ_i, equals regular ECB-union multiset response times (cf. Definition 6.6 on page 229).

We recognize that in general, for initial time bounds $t^{(m)}$, response times $\bar{R}_i(t^{(m)})$ denote improved upper response time bounds as the number of potentially interfering

program points is reduced. Formally, time-bounds $t^{(m+1)}$ can be derived from $t^{(m)}$ as:

$$t^{(m+1)} = \lambda u \, . \, [\inf(t^{(m)}(u)), \bar{R}_i(t^{(m)})(u)] \tag{6.51}$$

where $\inf(t^{(m)}(u))$ denotes the lower bound of the interval $t^{(m)}(u)$. Note that we assumed above that this lower bound equals 0. A more precise lower bound is given by considering BCRT, which we denote by $\bar{R}_i^\star$. Recall that BCRT is independent of CRPD. Similar to WCRT bounds above, we generalize BCRT bounds to individual nodes and define:

$$\bar{R}_i^\star \colon (V \mapsto \mathbb{N}_0^\infty)$$
$$\bar{R}_i^{\star,(n+1)} = \lambda u \, . \, C_i^\star(u) + \sum_{j \in hp(i)} \left(\left\lceil \frac{\bar{R}_i^{\star,(n)}(u) - J_j}{T_j} \right\rceil - 1 \right) C_j^\star(u) \tag{6.52}$$

where $\bar{R}_i^{\star,(0)}(u) = R_i$ and where $C_i^\star(u)$ denotes BCET bounds per program point.

Definition 6.15 (Initial Response Time Bounds) *For a task τ_i, a safe initial response time bound t_i for $\bar{R}_i$ (Equation 6.52) is denoted by the minimal BCRT of all preceding nodes as lower bound and an unbounded upper bound, defined as:*

$$t_i^{(0)} = \lambda u \, . \left[\min_{v \in \mathrm{pred}(u)} \bar{R}_i^\star(v), \infty \right] \tag{6.53}$$

Lower response time bounds are constant and independent of existing response time bounds. Upper response time bounds recursively depend on the result of time bounded CRPD estimation.

Definition 6.16 (General Response Time Bounds) *For a task τ_i, the smallest possible worst-case response time bounds are denoted by the fixed point of:*

$$t_i^{(m+1)} = \lambda u \, . \left[\min_{v \in \mathrm{pred}(u)} \bar{R}_i^\star(v), \bar{R}_i(t_i^m)(u) \right] \tag{6.54}$$

Every upper response time bound depends on the WCRT given existing response time bounds. Note that t_i is monotone since in each iteration a non-increasing number of potential preemption points is taken into account.

Finally, we can compose the definition of time bounded response time.

Definition 6.17 (Time-bounded Response Time) *For task τ_i and terminal node $x_i \in V_i$, the worst-case response time with reduced cache interference is denoted by:*

$$\underaccent{\bar}{R}_i = \sup(t_i(x_i)) \tag{6.55}$$

where $\sup(t_i(x_i))$ denotes the upper bound of the interval denoted by $t_i(x_i)$. Note that $\underaccent{\bar}{R}_i$ is not necessarily a global optimum.

Remarks

We also extended time-bounded response time analysis to exploit worst-case execution frequencies (cf. Section 5.3.7) in addition to response time intervals. The rationale is that program points cannot be preemption points more often than they are executed within their respective response time intervals. However, we found this to yield only insignificant improvements. Intuitively, program points in loops typically yield the highest UCB counts but execution frequencies then often far exceed the possible number of preemptions. In addition, adjacent program points in loops often tend to have similar sets of UCBs. Exclusion of a single point therefore only yields insignificant reduction — if at all — in worst-case interference. Therefore, we do not address this possibility here since complexity of formulation and practical gain are at odds, and the venerable reader might sure be glad to have made it.

6.3 Evaluation

In the following we evaluate the proposed approaches of time-bounded multiset and bucket-based CRPD estimation, where we compare our results to existing approaches for various benchmarks from the Mälardalen WCET Benchmarks (MRTC) [97], which comprises of typical real-time applications, and the PapaBench [98] benchmark suite, which models tasks of an Unmanned Aerial Vehicle (UAV).

For all benchmarks, we compute WCRT estimates. For a given task set ordered by WCET estimate, we model the preemption of the longest running task by a successively increasing set of preempters. We compare the following approaches: WCRT according to Equation 6.11 without CRPD (NONE) and with ECB-union (ECB, Definition 6.16), WCRT according to Equation 6.18 with ECB-union multiset (ECBMS, Definition 6.6), WCRT according to Equation 6.25 with buckets (BUCK, Equation 6.42) and time bounded WCRT for ECB-union multiset according to Equation 6.52 (BECBMS). We found in all samples UCB-union approaches to be inferior, so we exclude them. Note that [94] reports UCB-union approaches to be competitive for DC-UCB. Here, we rely on the standard definition of UCB.

To adapt to benchmark sizes and to provoke greater cache interference, we chose a direct-mapped cache of size of 1 kB with line size of 8 B. Note that associativities $K > 1$ are not relevant here as our focus is to evaluate task sets with high preemption counts instead of focusing on the reduction of estimates for single preemptions. We estimate WCET (per program point) of benchmarks with path analyses for latest execution times WCET (LET) and BCET path analysis. We assume a BCET estimate per instruction of 50 % of the WCET estimate for the lack of BCET pipeline analysis.

We scale task periods to achieve specific processor utilization for fixed-priority periodic scheduling without CRPD. We evaluate WCRT for a constant utilization of 40 % without taking CRPD into account as well as determine breakdown utilization for given CRPD estimates.

Name	Size (B)	WCET (cyc)	BCET (cyc)	UCB	ECB
bs	132	287	112	13	17
fibcall	52	432	26	6	7
lcdnum	254	1 344	131	32	32
insertsort	206	4 150	126	13	25
fdct	2 478	10 569	5 283	128	128
select	910	11 045	377	86	113
sqrt	234	14 959	21	24	29
cnt	276	44 311	18 334	16	37
edn	3 052	112 660	52 738	128	128
fft1	4 000	115 160	1 044	128	128
crc	978	122 776	101	63	66
st	1 410	414 183	198 376	73	88
bsort 100	162	597 946	2 902	12	20
matmult	518	809 582	403 957	29	55
lms	1 664	1 477 995	50 197	101	127

Table 6.1: Properties of MRTC benchmarks

Table 6.1 list the utilized MRTC task set including its properties ordered by WCET. Benchmark sizes are given in bytes of the binary image[1].

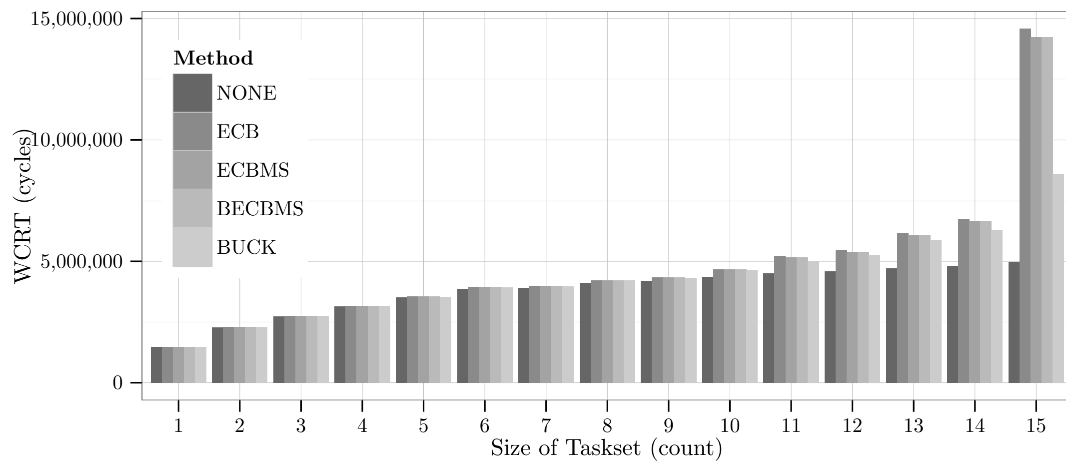


Figure 6.9: Absolute WCRT estimates for various approaches

Figure 6.9 illustrates absolute WCRT estimates for the various approaches. As can be seen, CRPD overhead is substantial for all but small set sizes. Notably, the addition of benchmark *bs* as the last additional benchmarks yields considerable overhead. In this specific use case precision of estimates increases monotonically from NONE to BUCK.

Figure 6.10 illustrates the ratio of improvement of BECBMS and BUCK over ECBMS for the same benchmarks as in Figure 6.9. Numbers right of the bars denote the ratio for BUCK, numbers left from the bars denote BECBMS. From 10 preempters on, pessimism of MS and BECMS increases significantly, up an extreme case of 40 % above BUCK. We

[1]Sizes differ insignificantly from properties listed in Table 4.2 on page 73 due to a different version of the underlying platform.

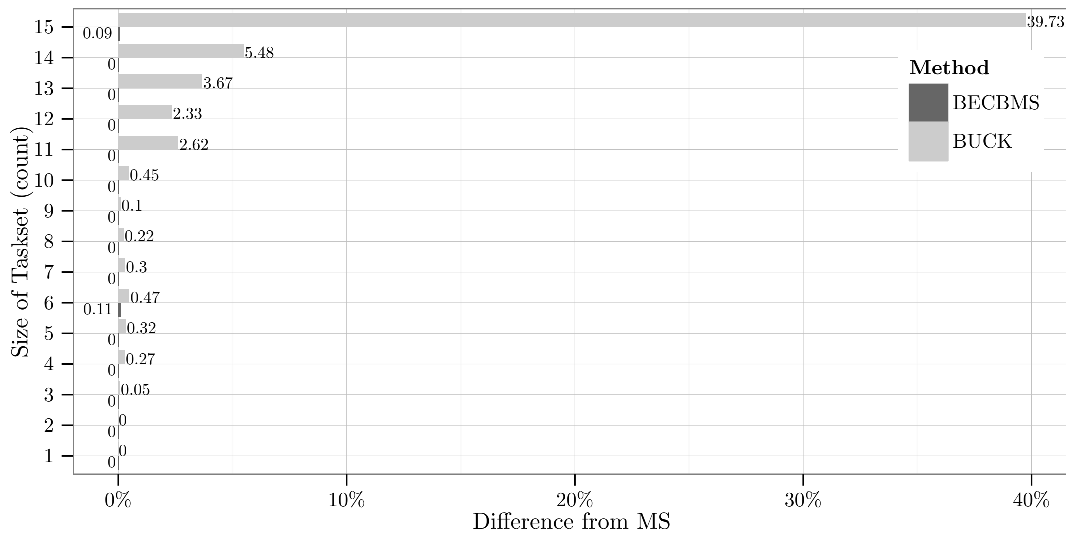


Figure 6.10: Ratio of WCRT estimate from ECB-union multiset (MS) to time-bounded ECB-union multiset (BECBMS) and bucket-based CRPD (BUCK)

recognize that time bounds in BECBMS only yield an insignificant impact on overall precision in this specific use case.

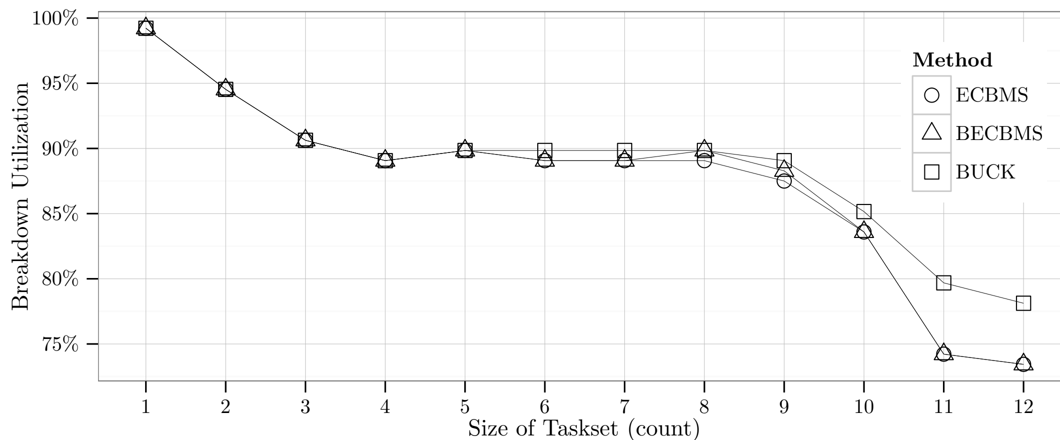


Figure 6.11: Breakdown utilization for MRTC task set of increasing size

To estimate breakdown utilization, we normalized utilization by adjusting task periods accordingly. Figure 6.11 illustrates respective results. For increasing task set sizes, we recognize BUCK to dominate the other approaches in this use case. BECBMS improves up to ∼2 % (8-9 tasks) over MS, whereas BUCK improves up to ∼5 % over MS (12 tasks).

For additional use cases, we also evaluated task from the PapaBench benchmark suite. Tasks along with their properties are listed in Table 6.2 ordered by descending WCET estimate. As opposed to MRTC, these benchmarks are control tasks with small footprint, short execution times and low loop repetition counts.

Accordingly, Figure 6.12 illustrates breakdown utilization similar to Figure 6.11. Most notably, BUCK does not necessarily dominate multiset-based approaches in this use case

Name	Size (B)	WCET (cyc)	BCET (cyc)	UCB	ECB
I6	108	92	24	8	14
T7	164	121	30	16	21
T5	166	157	19	10	21
I5	260	162	50	8	33
I4	338	250	50	12	42
T10	462	413	53	34	61
T6	752	553	25	14	95
T12	700	660	289	16	73

Table 6.2: Properties of PapaBench tasks

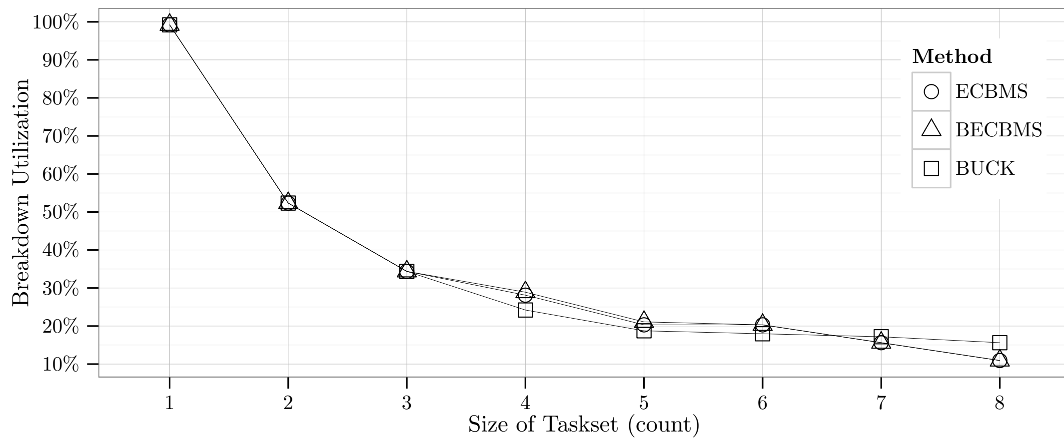


Figure 6.12: Breakdown utilization for PapaBench task set of increasing size

($\sim -3\,\%$ for 4 tasks). For increasing set sizes though, overhead inherent to "non-bucket" approaches shows ($\sim$3 %).

6.4 Conclusion

In this chapter we proposed two new approaches to bound CRPD for WCRT analysis. We first addressed the problem of inherent overestimation in conventional CRPD bounds, which are too pessimistic in case of deep preemption nestings. In the proposes "bucket"-based approach, we explicitly enumerate all nesting scenarios which by itself largely overestimates actual preemption costs. We then refined this approach and demonstrated that it improves, potentially significantly (up to $\sim$40 % in our use cases), upon the current state of the art. We then reviewed the possibility to associate time intervals with potential cache contents to tighten CRPD estimates by exploiting results from path analyses proposed in Chapter 5. For the given use cases, we observed improvements of up to $\sim$2%. Results may vary depending on the scenario so greater improvements can be expected for long-running low-priority tasks. Overall we proposed two new techniques to bound CRPD that do not follow the conventional line of approaches and outperform the latter in many cases.

Chapter 7

Conclusion

Contents

7.1 Summary of Contributions **247**
7.2 Future Work **249**
7.3 Conclusion **249**

In this chapter we summarize our contributions in Section 7.1, give hints on potential future work in Section 7.2 and conclude this thesis in Section 7.3.

7.1 Summary of Contributions

We identified the critical aspect in traditional static timing analysis as representation and flow of information between micro-architectural analysis and scheduling analysis, which are conceptually fundamentally different. We contributed analyses and conceptual, theoretical, and practical improvements, and new solutions to several domain-specific problems in this context. The main contributions are as follows:

Cache Analysis We discussed the role of cache analysis in timing analysis and the interfacing of such micro-architectural analysis with scheduling analysis by means of cache content summaries UCB, ECB and CBR. The intuition is to bound CRPD by computing summaries and incorporate them in schedulability tests while bypassing path analysis. We discussed the construction of a precise cache analysis for k-way set-associative caches with LRU replacement policy and we showed how these summary metrics can cheaply and precisely be derived from analysis results without the need for separate analyses. Along the discussion, we pointed out potential sources of inherent imprecision in the original definitions. We then proposed how UCB and CBR computations can be optimized for instruction caches by reducing the number of sample points to basic block boundaries, and we discussed task interaction in non-direct mapped caches and how to avoid unnecessary overestimation by excluding infeasible interaction.

Path Analysis The core contribution of this thesis is the proposal of a general path analysis. We thoroughly discussed the problem of in general path analysis with a focus on traditional methods of graph theory as well as in the context of static timing analysis. We put existing approaches into context and discussed their specific strengths and weaknesses. We concluded that methods inspired from traditional definitions and approaches do not conceptually fit the principles of program analysis well in general, and that the problem of preserving structural mappings of flow facts remains to be acute. To remedy these problems we reconsidered the problem holistically instead of focusing on specific subproblems. All solutions proposed follow specific design goals: i) *Efficiency*: Besides clarity of formal models, practical efficiency is of overarching concern. ii) *Simplicity*: The proposed solutions are technically simple and generalized — theoretically and practically — as far as possible or sensible. iii) *Immutability*: All input is supposed to be immutable. This implies that no assumption about graph layouts, such as reducibility, is made which avoids structural transformation that potentially distorts structural relation of input and flow facts. iv) *Conception*: As opposed to existing approaches, we specifically took concepts of traditional program analysis into account which allows for simple formal and practical combination for other program analyses. This specifically allows the introduction of path-sensitivity.

First, we focused on the reconstruction of loops in control flow. We proposed a highly efficient algorithm with parametric heuristics for loop identification. To solve problems related to irreducible graphs, we proposed two variants: i) *Enumeration*: We show how a context-sensitive representation of scope nestings is efficiently generated if no additional structural information through annotations is available. ii) *Prenumbering*: We proposed how annotations can be provided to guide loop identification deterministically..

Second, we proposed a general framework for path analysis. We provided its corresponding formal model, and we initially showed how it is applied to the traditional problem of whole-task WCET estimation. Along the discussion, we proposed numerous optimizations and generalized the algorithm to computations on subgraphs and how to efficiently derive solutions for all reachable program points. We also proposed a highly efficient reference algorithm. We then showed how a generalized framework can be adopted to efficiently solve other important problems in the domain of timing and scheduling analysis: i) *BCET*: We showed that BCET estimation is not symmetric to WCET estimation and discussed the differences accordingly. ii) *LET*: We introduced a new metric *latest execution time* which specifically denotes latest possible executions of program points in fully preemptive scheduling . iii) *MBT*: For scheduling with deferred preemption, proper placement of preemption points or the general availability of blocking times, we proposed how maximum blocking times can be derived efficiently for all reachable program points. iv) *WCEF*: All previous proposals are path-sensitive. We show how to efficiently derive maximal execution frequencies independent of paths.

CRPD We improved upon the state of the art of CRPD application in scheduling analysis in two ways. First, we proposed bucket-based CRPD estimation which is not based on summary computation of UCB or ECB, but is ultimately based on enumeration of preemption scenarios which avoids inherent imprecision in existing approaches. Second, instead of bypassing path analysis, we propose how to exploit time bounds for program points to exclude infeasible preemption scenarios to tighten CRPD estimates.

7.2 Future Work

Accounting for CRPD in the context of static program analysis highlights the shortcomings in the traditional tool chain of program analyses to obtain execution time estimates. A major problem that has to be addressed in the near future is the incorporation of scheduling analysis into timing analysis. Traditional methods to account for CRPD are limited to timing compositional or at least constant-bounded timing compositional architectures — a property only given for simple architectures whose use is becoming increasingly rare. Scheduling analysis must not be decoupled from micro-architectural analysis. Traditional numeric scheduling analysis is, just as ILP-based path analysis, a misfit in this regard. Methods based on abstract interpretation need to be found. Ironically, in the domain of multicore WCET analysis, abstract interpretation of shared bus schedules is already done [177] and is therefore conceptually already ahead of unicore multitask analysis in this regard. Its application to problems of task scheduling is potentially just a single abstraction step away — but only with a sufficiently powerful path analysis. In this thesis we used a constraint model sufficient enough to ensure soundness of estimates. We purposely did not advance the constraint model to mutual exclusion of paths and global flow bounds, which only contribute positively to tightness but not to soundness of estimates. So this remains an open problem.

7.3 Conclusion

Established methods for static timing analysis are not necessarily a good fit for existing and foreseeable problems in the context of multi-task timing analysis. The objective in this thesis has been to contribute various proposals for improvements in the interfacing from per-task program analysis to scheduling analysis. We showed that despite the maturity of traditional techniques, there still is room for improvement. Nevertheless, it is strikingly obvious that traditional thinking on how tooling for timing analysis is supposed to be composed requires a second though. Increasingly, we recognize that technical advances and available tools for timing analysis part ways. Removing the traditional separation of timing and scheduling analysis is therefore worth the endeavor.

Appendix A

Notations and Conventions

Contents

A.1 Mathematical Notation **251**

A.2 Pseudo-code Language **253**

This chapter serves as a general reference for mathematical notations and pseudo-code listings.

A.1 Mathematical Notation

In the following, we define basic mathematical notations used in this thesis which might differ from standard notation and semantics.

Sets Let $S = \{a_1, a_2, \dots, \}$ denote an (unordered) set. The empty set is denoted by $\emptyset = \{\}$. $\wp(S) = 2^S = \{\emptyset, \{a_1\}, \{a_1, a_2\}, \dots\}$ denotes the power set. The usual set operators $(\cup, \cap, \setminus, \times)$ apply. Sets can be partitioned into *equivalence classes*, which are induced by an binary *equivalence relation* $\sim$. We write $[a] = \{s \mid s \in S \wedge a \sim s\}$ to denote an equivalence class. Equivalence classes are either disjoint or equal. The *quotient set*, denoted by $S/\!\sim$, is the set of all equivalence classes induced by the equivalence relation. A set of class representatives is a set of exactly one element of each equivalence class $\{s \mid [s] \in S/\!\sim\}$. We denote an arbitrary element from a set by $\text{any}(S) = s \in S$. A *multiset* like $S = \{a, a, b, b, c\}$ is an unordered set that allows repeated elements. We denote the n greatest (smallest) elements of a set S by $\max^n\colon \wp(S) \mapsto \wp(S)$ ($\min^n\colon \wp(S) \mapsto \wp(S)$).

Tuples A tuple $T = (a_1, \dots, a_k)$ is a sequence of elements such that $T \in S^k$. The empty tuple is denoted by $\epsilon = ()$. Every symbolic element of a tuple is implicitly assumed to be a function that, if applied to the tuple, evaluates to its value. Thus, given a tuple $T \in \mathbb{N}^k$ with $T = (a_1, a_2, \dots, a_k) = (1, \dots, K)$, there exist functions $a_1\colon \mathbb{N}^k \mapsto \mathbb{N}$, $a_2\colon \mathbb{N}^k \mapsto \mathbb{N}, \dots$ such that $a_1(T) = 1$, $a_2(T) = 2$, etc. Cardinality is denoted by $|T| = k$. Inversion is denoted by $(a_1, a_2, \dots)^{-1} = (\dots, a_2, a_1)$. Set inclusion ($\in$), subset ($\subseteq$) and cardinality ($|\cdot|$) operators equally apply to tuples with their intuitive meaning. Tuples $A = (a_1, a_2, \dots)$

and $B = (b_1, b_2, \dots)$ are equal, if $\forall a_i \in A \colon \forall b_i \in B \colon a_i = b_i$. For convenience, we use "_" to "match" any tuple element, such that for a set $S = \{(a, b), (a, c), (b, c)\}$, we can construct sets such that $\{s \mid (a, _) \in S\} = \{(a, b), (a, c)\}$. We allow for implicit conversion from tuples to sets and therefore alternatively refer to tuples as ordered sets.

Graphs Graph $G = (V, E)$ is composed of a finite set of *vertices* (or *nodes*) V and *edges* (or *arcs*) E that represents relations between vertices. In an *undirected graph*, $E \subseteq \{\{u, v\} \mid u, v \in V\}$ is a set of unordered pairs. In a *directed graph* (or *digraph*), $E \subseteq V^2$ is a set of ordered pairs. If $(u, v) \in E$, vertices u and v are *adjacent* to each other and the edge is said to be *incident* upon u and v. The *degree* of a node denotes the number of incident edges. For a digraph, *indegree* and *outdegree* is distinguished accordingly. A path is a sequence of vertices $\pi = (u_1, \dots, u_n)$ of *length* n such that $(u_i, u_{i+1}) \in E$. Path $(u_1, \dots, u_n)$ is *closed* if $u_n = u_1$. We write $u \rightsquigarrow v$, if there exists a path from u to v. The empty path is denoted by ϵ. A *cycle* is a path such that $u \rightsquigarrow u$. A graph is *connected*, if there exists a path between any two nodes. A *tree* is a connected graph with $|V| - 1$ edges. A *disjoint set* of trees is a *forest*.

Functions For a function $\mathrm{f} \colon X \mapsto Y$, its *domain* is $\mathrm{dom}(\mathrm{f}) = X$, its *co-domain* is $\mathrm{codom}(\mathrm{f}) = Y$, its *range of definition* is $\mathrm{def}(\mathrm{f}) \subseteq X$ and its image is $\mathrm{img}(\mathrm{f}) \subseteq (Y)$. A discrete function $\mathrm{f} \colon X \mapsto Y$ is a set of tuples $\mathrm{f} = \{(x_i, y_j), \cdots \mid x_i \in X, y_j \in Y\}$ denoting the mapping relation. If the mapping is incomplete such that $\forall x \in X \colon \exists y \in Y \colon (x, y) \notin \mathrm{f}$, the function is *partially defined* and we implicitly denote this by $\mathrm{f}(x) = \bot$, unless stated otherwise. The notation $f[x \to y]$ denotes:

$$f[x \to y](z) := \begin{cases} y & \text{if } x = z \\ f(z) & \text{otherwise} \end{cases}$$

Partially applying a sequence of arguments to a function is denoted by operator $[\![\cdot]\!]$. For example, let $f = \lambda x \,.\, \lambda y \,.\, x + y$, then $[\![(1, 2, 3)]\!](f) = (f(3) \circ f(2) \circ f(1))$, which is equivalent to $\lambda y \,.\, f(3, f(2, f(1, y)))$.

Function composition is denoted by operator $\circ$ such that for functions $\mathrm{f} \colon X \mapsto Y$ and $\mathrm{g} \colon Y \mapsto Z$, $(\mathrm{g} \circ \mathrm{f})(x) = \mathrm{g}(\mathrm{f}(x)) = z$. Operator $\mapsto$ is right-associative: $\mathrm{f} \colon X \mapsto Y \mapsto Z$ equals $\mathrm{f} \colon X \mapsto (Y \mapsto Z)$. We assume implicit argument "unpacking" such that $\mathrm{f}((x, y))$ equals $\mathrm{f}(x, y)$, unless stated otherwise. The *identity function* id maps to itself $\mathrm{id(x) = x}$. The *indicator function* 1_S denotes existence in a set S such that:

$$1_S(s) := \begin{cases} 1 & \text{if } s \in S \\ 0 & \text{otherwise} \end{cases}$$

Hence, $\sum_{s \in S} 1_S(s)$ denotes the number of occurrences of s in set S.

Note that function symbols composed of multiple characters are typeset in roman ("fun"), whereas single character symbols are optionally written slanted (f).

Lambda Calculus A convenient way to express *higher-order functions* (functions as arguments or return values from functions) is the *lambda calculus* [178]. We write the *lambda term* $f := \lambda x \,.\, y$, where x is *bound* and y is *free*, such that the application $f(a)$ replaces all occurrences of x in y by a. Examples: $(\lambda x . x{+}1)(1) = 2$, $(\lambda x . x{+}z)(1) = 1{+}z$ and $(\lambda x \,.\, \lambda y \,.\, x + y)(1)(2) = 3$.

A.2 Pseudo-code Language

The pseudo-code language we are using in code listings is inspired by different existing programming languages, such as Haskell, Lisp and OCaml. We allow for the mixture of pure mathematical notation with typical programming language constructs such as conditional branches and loop constructs. The semantics is not purely functional. We use indentation instead of curly braces ("{...}") or similar to denote grouped consecutive blocks of execution. Depending on whether it supports clarity, we introduce explicit type declaration. Otherwise, types are static nonetheless but are skipped if inference is obvious. Higher order functions are possibly used but generally avoided to allow for a simple translation into "imperative" programming languages. Generally, neither syntax or semantics is very strict to enhance readability.

Comments are denoted by "$\triangleleft$" and do not span multiple lines.

Arithmetic expressions are written in usual infix notation and conventional mathematical notation for set comprehension etc. is used as usual. Comparison is denoted by operators $<, \leq, =, \neq, \geq, >$. Assignment of values to variables is denoted by operator $\leftarrow$. Access to element n in map (array) x is written as $x[n]$. If a tuple is explicitly declared (i.e. $T\colon (A, B)$), then we assume the implicitly definition of functions to access tuple elements (For tuple $t \in T$, function A denotes the first element in tuple t by $A(t)$).

Binding/rebinding of values to symbols representing variables does not occur. An exception is function definition: we write, for example, $\mathbf{let} f\ x\ y = x + y$ to emphasize the introduction of a new function. Function invocation is typically written without explicit parenthesis for simplicity. Hence, $f(x, y)$ equals $f\ x\ y$.

For manipulation of frequently used data structures such as queues, we define the following functions. Let $s = (x_1, \ldots, x_n)$ be some ordered sequence (array, tuple, string, etc.) then the following relations (not strictly mathematical) hold:

$$
\begin{aligned}
\mathrm{car}(s) &= x_1 \\
\mathrm{cdr}(s) &= (x_2, \ldots, x_n) \\
\mathrm{top}(s) &= x_n \\
\mathrm{pop}(s) &= x_n \wedge s = (x_1, \ldots, x_{n-1}) && \text{(Note: side-effect)} \\
\mathrm{deq}(s) &= x_1 \wedge s = (x_2, \ldots, x_n) && \text{(Note: side-effect)} \\
\mathrm{any}(s) &= x_i \colon 1 \leq i \leq n
\end{aligned}
$$

Note that appending elements to a queue (or stack "push") is usually denoted by concatenation $(s \cdot (x_{n+1}))$.

Very high level semantics are written in italics and given in plain words, if the precise semantics is not relevant to the understanding of algorithms.

Appendix B

Reference Implementations of Basic Graph Algorithms

Contents

B.1 Breadth-first Search **255**
B.2 Maximum Flow **256**
B.3 Single-source Shortest Paths **257**
B.4 Topological Sort **258**
B.5 Single-source Shortest Paths on Directed Acyclic Graphs **258**
B.6 Depth-first Search **258**

In this chapter, we list reference implementations of classic problems from graph theory in the pseudo language outlined in Appendix A. In the core chapters, we discuss modifications of these algorithms, so it might be of interest how we implemented the original versions in the first place to support understanding. We only provide algorithm listings along with their description here, along with references to more thorough discussions. Note that we often deviate from standard textbook implementations.

B.1 Breadth-first Search

Breadth-first Search [103] (BFS) visits nodes in a digraph G starting from a root node s "level-wise". The level denotes the distance, in terms of node count, on a shortest path from the root node. It returns an array P encoding a path tree which denotes shortest paths from every node towards the root node. Algorithm B.11 specifies the corresponding algorithm. Function bfs is invoked with a source node s and a digraph G (line 1) and returns a path tree P (line 12). The algorithm is initialized (lines 2-4) with a first-in first-out (FIFO) queue, a set of marked (visited) nodes M and a set of path predecessors P, where $P[u]$ denotes the preceding node for some node u. The algorithm repeats until all reachable nodes have been marked, that is, no additional nodes have been queued (line 5). The oldest element of the queue is removed (line 6) and all its adjacent nodes

Algorithm B.11 Breadth-first Search

1. **let** bfs $s\,G =$ **do** $\triangleleft\ G = (V, E, s, t)$
2. $\quad Q \leftarrow (s)$
3. $\quad M \leftarrow \{s\}$
4. $\quad P \leftarrow \{(u, \bot) \mid u \in V\}$
5. $\quad$ **while** $Q \neq \epsilon$ **do**
6. $\quad\quad u \leftarrow \text{car}\ Q$
7. $\quad\quad$ **for** $v \in \text{pred}\ u$ **do**
8. $\quad\quad\quad$ **if** $v \notin M$ **then**
9. $\quad\quad\quad\quad M \leftarrow M \cup \{v\}$
10. $\quad\quad\quad\quad Q \leftarrow Q \cdot (v)$
11. $\quad\quad\quad\quad P[v] \leftarrow u$
12. $\quad$ **return** P

(line 7) that have not been marked yet (line 8), are marked, queued and its preceding path node is recorded (lines 9-11).

B.2 Maximum Flow

Computing maximum flow by the Ford-Fulkerson method [101] is performed by repeatedly searching for a path from source to sink via BFS in the residual network R of the flow network G and sending as much flow as possible over the corresponding path P in the original network until flow can no longer be increased. This is equivalent to a disconnected sink in the residual network. The algorithm returns an array F denoting flow along edges.

Algorithm B.12 Maximum Flow via Ford-Fulkerson Method

1. **let** maxflow $G =$ **do** $\triangleleft\ G = (V, E, s, t, c)$
2. $\quad F \leftarrow \{(u, v) \rightarrow 0) \mid (u, v) \in E\}$
3. $\quad R \leftarrow (V, \{(u, v) \in E \mid c(u, v) - F[(u, v)] > 0\})$
4. $\quad P \leftarrow \text{bfs}\ R$
5. $\quad$ **let** pushflow $u\ v\ P\ a =$
6. $\quad\quad a \leftarrow \min a\ (c(u\ v) - F[(u\ v)])$
7. $\quad\quad$ **if** $P[u] \neq \bot$ **do**
8. $\quad\quad\quad a \leftarrow \text{pushflow}\ P[u]\ u\ P\ a$
9. $\quad\quad F[(u, v)] \leftarrow F[(u, v)] + a$
10. $\quad\quad$ **return** a
11. $\quad$ **while** pushflow $P[t]\ t\ P\ \infty$ **do**
12. $\quad\quad R \leftarrow (V, \{(u, v) \in E \mid c(u, v) - F[(u, v)] > 0\})$
13. $\quad\quad P \leftarrow \text{bfs}\ R$
14. $\quad$ **return** F

Algorithm B.12 outlines this method. Function maxflow in invoked with a flow network G (line 1) and returns (line 14) flow along an edge. Initially, all flows equal 0, we compute the residual network R, which contains all the edges of G that do not saturate their respective capacity bound, and an initial path is computed via bfs.

The algorithm loops (line 11), repeatedly pushing flow, recomputing the residual network R and looking up a path in R (lines 11-13). Function pushflow (line 5) in invoked with a predecessor node u of a node v on the path encoded in path tree P, a maximal flow a. The maximal flow is bounded by the minimum of all residual flows. Thus, given a maximal flow a, for the edge (u, v), the new maximal flow is the minimum of a and the residual flow (line 6). If the flow network source has not been reached yet (line 7), repeat this step recursively for the preceding edge along the path tree P. Hence, once recursion terminates, a equals the maximal admissible flow along the path from source to sink. Unwinding the recursion, thus, involves adjusting the net flow accordingly (line 9) and passing on the minimum flow (line 10).

B.3 Single-source Shortest Paths

Computing single-source shortest paths is classically implemented according to [104]. Here, we deviate slightly from the original to increase similarity with the reference implementation of BFS in Section B.1. For a edge-weighted digraph G, the algorithm returns an array of distances D from source node s and an array modeling a path tree P denoting the path from any node to s. Algorithm B.13 lists the implementation. For

Algorithm B.13 Single-source Shortest Paths

```
 1  let sssp G = do                                        ◁ G = (V, E, s, t, ω)
 2      Q_min ← (s)
 3      M ← {s}
 4      D ← {(u → ∞) | u ∈ V \ {s}} ∪ {(s → 0)}
 5      P ← {(u, ⊥) | u ∈ V}
 6      while Q_min ≠ ϵ do
 7          u ← deq Q_min
 8          for v ∈ succ u do
 9              if D[u] + ω u v < D[v] then
10                  D[v] ← D[u] + ω u v
11                  P[v] ← u
12              if v ∉ M then
13                  M ← M ∪ {v}
14                  Q_min ← σ_D(Q_min · (v))
15      return (D, P)
```

source node s and graph G (line 1), we initialize a queue $Q_{\min}$ ordered by ascending distance D from source node s (line 2), a set of visited nodes M with s (line 3), an array of distances D from s (line 4) and a path tree P (line 5). While $Q_{\min}$ is not empty (line 6), select node u with the shortest distance to s (line 7) from $Q_{\min}$. For its successor nodes v (line 8), we update their distances to s if the path through node u including the distance from u to v denotes by weight ω is shorter then any previous path, and updates the path tree P accordingly (lines 9-11). If successor node v is unvisited yet (line 12), mark v visited (line 13) and enqueue v (line 14) where σ_D sorted $Q_{\min}$ in descending order of distances denoted by array D.

B.4 Topological Sort

This algorithm orders nodes of an acyclic digraph in topological order [103]. Intuitively, edges denote dependencies, so we successively remove nodes without dependencies from the graph, which in turn leads to new nodes without dependencies. It returns a string S containing nodes in such order. Algorithm B.14 specifies the algorithm. For a graph G

Algorithm B.14 Topological Sort

```
1  let topo G = do                                  ◁ G = (V, E)
2      S ← ϵ
3      Q ← ϵ
4      for u ∈ V do
5          I[u] ← deg_in u
6          if I[u] = 0
7              Q ← Q · (u)
8      while Q ≠ ϵ do
9          u ← deq Q
10         S ← S · (u)
11         for v ∈ succ u do
12             I[v] ← I[v] − 1
13             if I[v] = 0
14                 Q ← Q · (v)
15     return S
```

(line 1), we initialize a string of nodes S and a queue Q of nodes without predecessors (lines 1,2), and for every node u (line 5), compute its respective indegree I and enqueue all nodes of indegree equal to 0 (lines 4-7). While queue Q is not empty (line 8), remove a node from Q (line 9) and append it to string S (line 10). For all successors v of node u (line 11), decrease indegree I (line 12) and enqueue v if its indegree now equals 0.

B.5 Single-source Shortest Paths on Directed Acyclic Graphs

Given topological order, compute single-source shortest paths on a DAG can be computed efficiently by only taking preceding nodes into account. Note that longest paths can be computed by just negating weights. Similar to the general algorithm for single-source shortest paths, this implementation returns an array denoting distances from a source node and a path tree denoting the shortest paths explicitly. Algorithm B.15 lists the corresponding algorithm. For a weighted DAG G (line 1), we initialize a path tree P and an array denoting distances per node D (lines 1,2). For all nodes v in topological order (line 4), we compute the shortest distance to v by adding the shortest distance to a preceding node u and edge weight ω, and we update the path tree accordingly (lines 5-8).

B.6 Depth-first Search

Depth-first Search traverses a digraph from a given root node along a spanning tree consisting of tree edges T. All other edges are being classified as back B, forward F or

Algorithm B.15 DAG Single-source Shortest Paths

```
1  let dagsssp G = do                                  ◁ G = (V, E, s, t)
2      P[s] ← ⊥
3      D ← {(u → ∞) | u ∈ V \ {u}} ∪ {(s → 0)}
4      for v ∈ topo G do
5          for u ∈ pred v do
6              if D[u] + ω u v < D[v] then
7                  D[v] = D[u] + ω u v
8                  P[v] = u
9      return (D, P)
```

cross C edges. Nodes are being labeled according to their discovery and finishing time, respectively, represented as an interval I.

Algorithm B.16 Non-recursive Depth-first Search

```
1  let dfs G = do                                      ◁ G = (V, E, s, t)
2      Q ← ((s, succ s))
3      M = {(u → WHITE) | u ∈ V \ {s}} ∪ {(s → GRAY)}
4      T ← B ← F ← C ← ∅
5      I = {u → [∞, ∞]: u ∈ V}
6      t ← 0
7      while Q ≠ ϵ do
8          (u, S) ← pop Q
9          while S ≠ ∅ do
10             v ← pop S
11             if M[v] = WHITE then
12                 T ← T ∪ {(u, v)}
13                 Q ← Q · ((u, S))
14                 u ← v
15                 S ← succ v
16                 M[u] ← GRAY
17                 I[u][0] ← t ← t + 1
18             else if M[v] = GRAY then
19                 B ← B ∪ {(u, v)}
20             else
21                 if I[u][0] ≤ I[v][0] then
22                     F ← F ∪ {(u, v)}
23                 else
24                     C ← C ∪ {(u, v)}
25         M[u] ← BLACK
26         I[u][1] ← t ← t + 1
27     return (T, B, F, C, I)
```

Algorithm B.16 specifies a respective implementation. Function dfs is invoked with digraph G (line 1) and returns a tuple of disjoint sets of tree T, back B, forward F and cross edges C, and an interval I representing discovery and finishing time of nodes (line 27). During traversal of G nodes are marked as either non-visited (WHITE), visited but unfinished (GRAY), or as finished (BLACK). A node is unfinished if there exists an unvisited adjacent nodes. A stack maintains the set of unfinished nodes. For initialization, the stack Q is initialized with s and its set of adjacent nodes (line 2). All nodes but s are

marked unvisited and s is marked unfinished (line 3). Lines 4-6 initialize edge partitions, time intervals and the counter for time stamps.

Function dfs proceeds until all reachable nodes have been visited (line 7). It repeats the following actions. A node u — we will refer to u as the current node — and its successors S are popped off stack Q (line 8). While there exist unvisited successors (line 9), remove a successor v from S (line 10) and check its marking. If v is not yet visited (line 11), classify edge (u, v) as tree edge (line 12), push the current node u along with its remaining successors S onto the stack for later processing (line 13), set v to be the current node — along with its respective successors — and mark it unfinished (lines 14-16), and increase and store its time stamp t (lines 17). If v is unfinished (line 18), (u, v) is a back edge. If v is finished (line 20), (u, v) is a forward edge, if u is a descendant in the subtree of T rooted in v (lines 21,22). Otherwise, (u, v) is a cross-edge (line 24). If the current node u has no unfinished successors left (line 25), it is marked finished, labeled with its timestamp in increased and assigned (lines 26).

Appendix C

On Linear Programming

Linear Programming (LP) is a branch of constraint programming to solve convex optimization problems [99]. As opposed to “imperative” programming languages, not the way towards a certain result is specified in terms of a specialized algorithm, but only the “shape” of the solution is modeled and finding a solution is subject to a generalized optimization algorithm. Linear programming refers to the fact these models are specified in terms of linear equations.

Given a matrix $A \in \mathbb{R}^n \times \mathbb{R}^m$, a row vector $c \in \mathbb{R}^n$, a column vector $b \in \mathbb{R}^m$ and a column vector of “unknowns” $x \in \mathbb{R}^n$. The (canonical) form of a general LP problem is denoted by:

$$\begin{aligned} \min \quad & cx \\ \text{s.t.} \quad & Ax \geq b \\ & x \geq 0 \end{aligned} \tag{C.1}$$

Equation C.1 is the *objective function*, A is a *constraint matrix*, x is a set of *decision variables* and the matrix rows, which are constrained by b, are referred to as *constraints*. An LP can only have one of three possible solutions: i) *Feasible*: An optimal solution exists and the *objective value* denotes its valuation. ii) *Infeasible*: No solution exists at all. iii) *Unbounded*: The optimal solution is infinite.

LP can be turned into a maximization problem by negating the objective coefficients c. Note that although the objective value is optimal, decision variables might be valuated non-deterministically. LP is well-understood and efficient algorithms exist to solve its general form as well as specialization of this problem [179].

One particularly important type of LP in our context are *integer linear programs* (*ILP*). Here, all decision variables must be whole numbers ($x \in \mathbb{Z}^n$).

The general ILP is NP-complete [99] and the typical strategy is based on successive constraining of an identical (relaxed) LP, based on branch-and-bound [179] (branch-and-cut) techniques to obtain feasible solutions.

Again, for variations of ILP, efficient algorithms exist. One particularly important type of ILP are those with a constraint matrix which is *totally unimodular*.

Definition C.1 (Total Unimodularity [99]) *A matrix A is* totally unimodular *(TUM) if and only if every square matrix $A' \subseteq A$ has a determinant of -1, 0 or 1.*

A totally unimodular LP is guaranteed to yield an integer solution, hence an equivalent ILP can be solved efficiently by known LP techniques. As an example, the MAXFLOW network flow problem as specified in Definition 5.4 is TUM.

Bibliography

[1] Edward A. Lee. Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report. Technical Report UCB/EECS-2007-72, EECS Department, University of California, Berkeley, May 2007.

[2] P. Marwedel. *Embedded System Design.* Springer, Secaucus, NJ, USA, 2011.

[3] Edward A. Lee. Absolutely Positively on Time: What Would It Take? *IEEE Computer*, 38(7):85–87, 2005.

[4] Reinhard Wilhelm, Jakob Engblom, et al. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *ACM Transaction on Embedded Computing Systems*, 7(3), 2008.

[5] Edward A. Lee and Yang Zhao. Reinventing Computing for Real Time. In *Proceedings of the 12th Monterey Conference on Reliable Systems on Unreliable Networked Platforms*, pages 1–25. Springer, 2007.

[6] S. Altmeyer. *Analysis of Preemptively Scheduled Hard Real-time Systems.* epubli GmbH, 2013.

[7] Jan Kleinsorge, Heiko Falk, and Peter Marwedel. A Synergetic Approach to Accurate Analysis of Cache-related Preemption Delay. In *Proceedings of the 7th International Conference on Embedded Software*, EMSOFT '11. ACM, October 2011.

[8] Jan Kleinsorge, Heiko Falk, and Peter Marwedel. Simple Analysis of Partial Worst-case Execution Paths on General Control Flow Graphs. In *Proceedings of the 9th International Conference on Embedded Software*, EMSOFT '13. ACM, October 2013.

[9] Jan Kleinsorge and Peter Marwedel. Computing Maximum Blocking Times with Explicit Path Analysis under Non-local Flow Bounds. In *Proceedings of the 10th International Conference on Embedded Software*, EMSOFT '14. IEEE, October 2014.

[10] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of

Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.

[11] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. In *Language Design for Reliable Software*, pages 77–94, 1977.

[12] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[13] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[14] S. Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.

[15] John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[16] Patrick Cousot and Radhia Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 269–295, 1992.

[17] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2005.

[18] Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, 1973.

[19] Jan Reineke. *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University, 2009.

[20] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[21] Yong-Fong Lee, Barbara G. Ryder, and Marc E. Fiuczynski. Region Analysis: A Parallel Elimination Method for Data Flow Analysis. *IEEE Transaction on Software Engineering*, 21(11):913–926, November 1995.

[22] R. Kramer, R. Gupta, and M. L. Soffa. The Combining DAG: A Technique for Parallel Data Flow Analysis. *IEEE Transaction on Parallel and Distributed Systems*, 5(8):805–813, August 1994.

[23] Florian Martin. PAG - An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[24] Frances E. Allen and John Cocke. Graph Theoretic Constructs For Program Control Flow Analysis. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1972.

[25] Johannes Kinder, Florian Zuleger, and Helmut Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 214–228, Berlin, Heidelberg, 2009. Springer.

[26] Henrik Theiling. Control Flow Graphs For Real-Time Systems Analysis. Ph. D. Thesis, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[27] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications*. Springer, Santa Clara, CA, USA, 2004.

[28] Marko Bertogna, Orges Xhani, Mauro Marinoni, Francesco Esposito, and Giorgio C. Buttazzo. Optimal Selection of Preemption Points to Minimize Preemption Overhead. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[29] Constantine D. Polychronopoulos. Toward Auto-scheduling Compilers. *The Journal of Supercomputing*, 2(3):297–330, 1988.

[30] Alan Burns. Preemptive Priority-Based Scheduling: An Appropriate Engineering Approach. In *Advances in Real-Time Systems*, pages 225–248. Prentice Hall, 1994.

[31] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transaction on Industrial Informatics*, 9(1):3–15, 2013.

[32] Robert I. Davis and Marko Bertogna. Optimal Fixed Priority Scheduling with Deferred Pre-emption. In *Proceedings of the Real-time Systems Symposium*, pages 39–50, 2012.

[33] Sanjoy K. Baruah. The Limited-Preemption Uniprocessor Scheduling of Sporadic Task Systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 137–144. IEEE Computer Society, 2005.

[34] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[35] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[36] John P. Lehoczky, Lui Sha, et al. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[37] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[38] Fengxiang Zhang and Alan Burns. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Transaction on Computers*, 58(9):1250–1258, 2009.

[39] E. Bini, G. Buttazzo, and G. Buttazzo. A Hyperbolic Bound for the Rate Monotonic Algorithm. In *Proceedings of the Euromicro Conference on Real-Time Systems*, pages 59–66, 2001.

[40] Mathai Joseph and Paritosh K. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5), 1986.

[41] N. Audsley, A Burns, M. Richardson, K. Tindell, and A J. Wellings. Applying new Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Engineering Journal*, 8(5):284–292, Sep 1993.

[42] Reinder J. Bril, Johan J. Lukkien, and Rudolf H. Mak. Best-case Response Times and Jitter Analysis of Real-time Tasks with Arbitrary Deadlines. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 193–202. ACM, 2013.

[43] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9): 1175–1185, September 1990.

[44] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building Timing Predictable Embedded Systems. *ACM Transactions on Embedded Computing Systems*, 13(4):82:1–82:37, March 2014.

[45] Greger Ottosson and Mikael Sjodin. Worst-Case Execution Time Analysis for Modern Hardware Architectures. In *In Proceedings ACM SIGPLAN Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 47–55, 1997.

[46] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978, July 2009.

[47] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Saarland University, 2005.

[48] Stephan Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *5th International Workshop on Worst-Case Execution Time Analysis*, volume 1

of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2007. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[49] Stephan Wilhelm and Björn Wachter. Symbolic state traversal for WCET analysis. In *Proceedings of the International Conference on Embedded Software*, pages 137–146. ACM, 2009.

[50] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[51] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[52] Jörn Schneider. *Combined schedulability and WCET analysis for real-time operating systems.* PhD thesis, Shaker, 2003.

[53] Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU.* PhD thesis, Saarland University, 2012.

[54] Christoph Berg. PLRU Cache Domino Effects. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2006. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[55] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A Fast and Precise Static Loop Analysis Based on Abstract Interpretation, Program Slicing and Polytope Models. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 136–146, Washington, DC, USA, 2009. IEEE Computer Society.

[56] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis using Abstract Execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, December 2006.

[57] Florian Martin Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 1998.

[58] Christian Ferdinand, Reinhold Heckmann, and Bärbel Franzen. Static Memory and Timing Analysis of Embedded Systems Code. In *Proceedings of the 3rd European*

Symposium on Verification and Validation of Software Systems, volume 07-04 of *TUE Computer Science Reports*, 2007.

[59] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[60] Reinhard Wilhelm, Sebastian Altmeyer, Claire BurguiÃšre, Daniel Grund, Jörg Herter, Jan Reineke, Björn Wachter, and Stephan Wilhelm. Static Timing Analysis for Hard Real-Time Systems. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 5944 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2010.

[61] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *In Proceedings of the IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, volume 6399 of *Lecture Notes in Computer Science*, pages 35–46. Springer, 2010.

[62] Björn Lisper. SWEET – A Tool for WCET Flow Analysis (Extended Abstract). In *Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer, 2014.

[63] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-based WCET Analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, The Netherlands, June 2001.

[64] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound - A Conceptually New Tool for Worst-Case Execution Time Analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time Analysis*, volume 8 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[65] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A Timing Analyzer for Embedded Software. *Science of Computer Programming*, 69 (1-3):56–67, December 2007.

[66] Maurice V. Wilkes. The Memory Gap and the Future of High Performance Memories. *SIGARCH Computer Architecture News*, 29(1):2–7, March 2001.

[67] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[68] Ulrich Drepper. What Every Programmer Should Know About Memory, 2007.

[69] L. A. Belady. A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[70] Gernot Gebhard. Timing Anomalies Reloaded. In *Proceeedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[71] Christoph Cullmann. Cache Persistence Analysis: A Novel Approach. In *Proceedings of International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES, pages 121–130, New York, NY, USA, 2011. ACM.

[72] Christian Ferdinand and Reinhard Wilhelm. Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[73] Jan Staschulat and Rolf Ernst. Scalable Precision Cache Analysis for Real-Time Software. *ACM Transaction on Embedded Computing Systems*, 6(4), 2007.

[74] Daniel Sandell, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Static Timing Analysis of Real-Time Operating System Code. In *Proceedings of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, volume 4313 of *Lecture Notes in Computer Science*, pages 146–160. Springer, 2004.

[75] Mingsong Lv, Nan Guan, Yi Zhang, Qingxu Deng, Ge Yu, and Jianming Zhang. A Survey of WCET Analysis of Real-Time Operating Systems. In *Proceedings of the IEEE International Conference on Embedded Software and Systems*, pages 65–72. IEEE, 2009.

[76] Andrea Bastoni, Bjorn B. Brandenburg, and James H. Anderson. Cache-Related Preemption and Migration Delays: Empirical Approximation and Impact on Schedulability. In *Proceedings 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Brussels, Belgium, July 2010.

[77] Johan Stärner and Lars Asplund. Measuring the Cache Interference Cost in Preemptive Real-time Systems. *SIGPLAN Notices*, 39(7):146–154, June 2004.

[78] J. V. Busquets-Mataix, J. J. Serrano, et al. Adding Instruction Cache Effect to Schedulability Analysis of Preemptive RealTime Systems. In *Proceedings of the Internation Conference on Real-Time and Embedded Technology and Applications Symposium*, 1996.

[79] Sascha Plazar, Jan Kleinsorge, Heiko Falk, and Peter Marwedel. WCET-aware Static Locking of Instruction Caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 44–52, San Jose, CA, USA, April 2012.

[80] Isabelle Puaut and David Decotigny. Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 114–123. IEEE Computer Society, 2002.

[81] Antonio Martí Campoy, Francisco Rodríguez-Ballester, and Rafael Ors Carot. Using Dynamic, Full Cache Locking and Genetic Algorithms for Cache Size Minimization in Multitasking, Preemptive, Real-Time Systems. In *Theory and Practice of Natural Computing*, volume 8273 of *Lecture Notes in Computer Science*, pages 157–168. Springer, 2013.

[82] Heiko Falk and Jan C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Design Automation Conference*, pages 732–737, San Francisco / USA, July 2009.

[83] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *Proceedings of the 1997 European Conference on Design and Test*, EDTC '97, pages 7–11. IEEE, 1997.

[84] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad Allocation for Concurrent Embedded Software. *ACM Transaction on Programming Languages and Systems*, 32(4):13:1–13:47, April 2010.

[85] Manish Verma, Klaus Petzold, Lars Wehmeyer, Heiko Falk, and Peter Marwedel. Scratchpad Sharing Strategies for Multiprocess Embedded Systems: A First Approach. In *Proceedings of the 3rd Workshop on Embedded Systems for Real-Time Multimedia*, ESTImedia, pages 115–120. IEEE Computer Society, 2005.

[86] Sascha Plazar, Paul Lokuciejewski, and Peter Marwedel. WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*, pages 78–88, Dublin / Ireland, June 2009.

[87] Frank Mueller. Compiler Support for Software-Based Cache Partitioning. In *Proceedings of the Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 125–133. ACM, 1995.

[88] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. *IEEE Transactions on Computers*, 47(6):700–713, June 1998.

[89] Sebastian Altmeyer and Claire Burguière. A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay. In *Proceedings of the International Conference on Euromicro Conference on Real-Time Systems*, 2009.

[90] Hiroyuki Tomiyama and Nikil D. Dutt. Program Path Analysis to Bound Cache-Related Preemption Delay in Preemptive Real-Time Systems. In *Proceedings of the International Conference on Hardware/Software Codesign*, pages 67–71, 2000.

[91] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–11, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[92] Yudong Tan and Vincent Mooney. Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-Time Systems. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems*, volume 3199 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2004.

[93] Sebastian Altmeyer, Claire Maiza, et al. Resilience Analysis: Tightening the CRPD bound for set-associative caches. In *Proceedings of International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, 2010.

[94] Sebastian Altmeyer, Robert I. Davis, and Claire Maiza. Improved Cache Related Pre-emption Delay Aware Response Time Analysis for Fixed Priority Pre-emptive Systems. *Real-Time Systems*, 48(5):499–526, 2012.

[95] Hemendra Singh Negi, Tulika Mitra, et al. Accurate Estimation of Cache-Related Preemption Delay. In *Proceedings of the International Conference on Hardware/-Software Codesign and System Synthesis*, 2003.

[96] Paul Lokuciejewski. *A WCET-Aware Compiler- Design, Concepts and Realization.* VDM Verlag, Saarbrücken, Germany, 2007.

[97] Jan Gustafsson, Adam Betts, et al. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*, 2010. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[98] Fadia Nemer, Hugues Cassé, et al. PapaBench: a Free Real-Time Benchmark. In *Proceedings of the International Workshop On Worst-Case Execution Time Analysis*, 2006.

[99] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[100] L. R. Ford and D. R. Fulkerson. Maximal Flow through a Network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[101] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[102] Laura Ciupală. Incremental Algorithms for the Minimum Cost Flow Problem. In *Proceedings of the 15th WSEAS International Conference on Computers*, pages 212–216, Stevens Point, Wisconsin, USA, 2011. World Scientific and Engineering Academy and Society (WSEAS).

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

[104] E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[105] Laura Ciupală. About Flow Problems in Networks with Node Capacities. *WSEAS Transactions on Computers*, 8(8):1266–1275, August 2009.

[106] Esko Nuutila. *Efficient Transitive Closure Computation in Large Digraphs.* PhD thesis, Helsinki University of Technology, 1995.

[107] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[108] Carl Offner. Notes on Graph Algorithms Used in Optimizing Compilers, 2013.

[109] Ravi Sethi. Testing for the Church-Rosser Property. *Journal of the ACM*, 21(4): 671–679, October 1974.

[110] Matthew S. Hecht and Jeffrey D. Ullman. Flow Graph Reducibility. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing*, STOC '72, pages 238–250, New York, NY, USA, 1972. ACM.

[111] G. Ramalingam. Identifying Loops in Almost Linear Time. *ACM Transaction on Programming Languages and Systems*, 21(2):175–188, March 1999.

[112] Johan Janssen and Henk Corporaal. Making Graphs Reducible with Controlled Node Splitting. *Proceedings of the Transactions on Programming Languages and Systems*, 19(6):1031–1052, 1997.

[113] Sebastian Unger and Frank Mueller. Handling Irreducible Loops: Optimized Node Splitting versus DJ-Graphs. *ACM Transaction on Programming Languages and Systems*, 24(4):299–333, July 2002.

[114] Larry Carter, Jeanne Ferrante, and Clark D. Thomborson. Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger. In *Proceedings of the Internationl Symposium on Principles of Programming Languages*, pages 106–114. ACM, 2003.

[115] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[116] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2007.

[117] Robert Endre Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9(3):355–365, 1974.

[118] Ken Kennedy and Linda Zucconi. Applications of a Graph Grammar for Program Control Flow Analysis. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 72–85, New York, NY, USA, 1977. ACM.

[119] M. Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3-4):141–153, January 1980.

[120] Paul Havlak. Nesting of Reducible and Irreducible Loops. *Proceedings of the Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[121] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A New Algorithm for Identifying Loops in Decompilation. In *Proceedings of the Static Analysis Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer, 2007.

[122] AM. Erosa and L.J. Hendren. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, May 1994.

[123] M. Howard Williams and G. Chen. Restructuring Pascal Programs Containing Goto Statements. *The Computer Journal*, 28(2):134–137, 1985.

[124] Bjarne Steensgaard. Sequentializing Program Dependence Graphs for Irreducible Programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmond, WA, 1993.

[125] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying Loops Using DJ Graphs. *ACM Transaction on Programming Languages and Systems*, 18 (6):649–658, November 1996.

[126] Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practiced and Experience*, 1(2):105–133, 1971.

[127] Christopher A. Healy, Mikael Sjödin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting Timing Analysis by Automatic Bounding of Loop Iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.

[128] Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET Bounds by Abstract Execution. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis*. OCG, July 2011.

[129] Yves Bertot, Benjamin Grégoire, and Xavier Leroy. A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis. In *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2006.

[130] Gogul Balakrishnan and Thomas W. Reps. WYSINWYX: What you see is not what you eXecute. *ACM Transactions on Programming Languages and Systems*, 32(6), 2010.

[131] Raimund Kirner and Peter P. Puschner. Timing Analysis of Optimised Code. In *Proceedings of the 8th International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 100–105. IEEE Computer Society, 2003.

[132] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[133] Benedikt Huber, Wolfgang Puffitsch, and Peter Puschner. Towards an Open Timing Analysis Platform. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis*, pages 5–14, Vienna, Austria, 2011. OCG.

[134] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–88, 2004.

[135] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF - A Language for WCET Flow Analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*. OCG, June 2009.

[136] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the IEEE International Conference on Automation Science and Engineering*, CASES '01, pages 132–140. ACM, 2001.

[137] Benedikt Huber, Daniel Prokesch, and Peter Puschner. A Formal Framework for Precise Parametric WCET Formulas. In *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis*, volume 23 of *OASIcs*, pages 91–102, 2012.

[138] Björn Lisper. Principles for Value Annotation Languages. In *Proceedins of the 14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–10, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[139] Robert Endre Tarjan. A Unified Approach to Path Problems. *Journal of the ACM*, 28(3):577–593, 1981.

[140] S. C. Kleene. Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[141] Jakob Engblom and Andreas Ermedahl. Modeling Complex Flows for Worst-Case Execution Time Analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, pages 163–174, November 2000.

[142] Pascal Montag and Sebastian Altmeyer. Precise WCET Calculation in Highly Variant Real-time Systems. In *Proceedings of the Conference on Design, Automation & Test in Europe*, pages 920–925. IEEE, 2011.

[143] A. C. Shaw. Reasoning about Time in Higher-Level Language Software. *IEEE Transaction on Software Engineering*, 15(7):875–889, January 1989.

[144] P. Puschner and Ch. Koza. Calculating the Maximum, Execution Time of Real-time Programs. *Real-Time Systems*, 1(2):159–176, September 1989.

[145] G. Pospischil, P. Puschner, A. Vrchoticky, and R. Zainlinger. Developing Real-Time Tasks with Predictable Timing. *IEEE Software*, 9(5):35–44, September 1992.

[146] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, 1995.

[147] R. Chapman. *Static Timing Analysis and Program Proof*. University of York, 1995.

[148] Antoine Colin and Guillem Bernat. Scope-Tree: A Program Representation for Symbolic Worst-Case Execution Time Analysis. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, 2002.

[149] Stefan Bygde, Andreas Ermedahl, and Björn Lisper. An Efficient Algorithm for Parametric WCET Calculation. In *Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 13–21. IEEE Computer Society, 2009.

[150] Ernst Althaus, Sebastian Altmeyer, and Rouven Naujoks. Precise and Efficient Parametric Path Analysis. In *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2011.

[151] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1):53–70, Jan 1999.

[152] Friedhelm Stappert and Peter Altenbernd. Complete Worst-case Execution Time Analysis of Straight-line Hard Real-time Programs. *Journal of Systems Architecture*, 46(4):339–355, February 2000.

[153] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

[154] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Proceedings of the Design Automation Conference*, 1995.

[155] Gabriel Y. Handler and Israel Zang. A Dual Algorithm for the Constrained Shortest Path Problem. *Networks*, 10(4):293–309, 1980.

[156] Y.P. Aneja and K. Nair. The Constrained Shortest Path Problem. *Naval Research Logistics Quarterly*, 25:549–555, 1978.

[157] Pascal Raymond. A General Approach for Expressing Infeasibility in Implicit Path Enumeration Technique. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 8:1–8:9, New York, NY, USA, 2014. ACM.

[158] Claire Maiza-Burguière and Christine Rochange. History-Based Schemes and Implicit Path Enumeration. In *Proceedings of the 6th International Workshop On Worst-Case Execution Time Analysis*, July 2006.

[159] Mingsong Lv, Zonghua Gu, Nan Guan, Qingxu Deng, and Ge Yu. Performance Comparison of Techniques on Static Path Analysis of WCET. *Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, 1: 104–111, 2008.

[160] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. *ACM Transaction Design Automation of Electronic Systems*, 4(3):257–279, July 1999.

[161] Amine Marref. *Predicated Worst-Case Execution-Time Analysis*. PhD thesis, University of York, 2009.

[162] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 223–232. IEEE Computer Society, 2005.

[163] Paul Feautrier. Parametric Integer Programming. *RAIRO Recherche Op'erationnelle*, 22(3):243–268, 1988.

[164] Sebastian Altmeyer, Christian Hümbert, Björn Lisper, and Reinhard Wilhelm. Parametric Timing Analysis for Complex Architectures. In *Proceedings of the 14th International Conference on Real-Time Computing Systems and Applications*, 2008.

[165] Jan Reineke and Johannes Doerfert. Architecture-Parametric Timing Analysis. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Application Symposium*, pages 189–200. IEEE, April 2014.

[166] Benedikt Huber and Martin Schoeberl. Comparison of Implicit Path Enumeration and Model Checking Based WCET Analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis*, volume 10 of *OpenAccess Series in Informatics (OASIcs)*, pages 1–12, Dagstuhl, Germany, 2009. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[167] Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 334–347. Springer, 2004.

[168] Reinhard Wilhelm. Why AI + ILP Is Good for WCET, but MC Is Not, Nor ILP Alone. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 309–322, 2004.

[169] Loukas Georgiadis, Luigi Laura, Nikos Parotsidis, and Robert Endre Tarjan. Loop Nesting Forests, Dominators, and Applications. In *Proceedings of the 13th International Symposium on Experimental Algorithms*, pages 174–186, 2014.

[170] Donald E. Knuth. *The Art of Computer Programming, Vol. 3.* Addison Wesley, 1998.

[171] IBM ILOG CPLEX Optimizer, Last 2010. URL `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`.

[172] *AbsInt Advanced Analyzer User Documentation.* AbsInt Angewandte Informatik GmbH, 2010.

[173] Sebastian Altmeyer, Claire Maiza-Burguière, and Reinhard Wilhelm. Computing the Maximum Blocking Time for Scheduling with Deferred Preemption. In *Proceedings of the International Conference on Software Technologies for Future Dependable Distributed Systems*, 2009.

[174] Yudong Tan and Vincent John Mooney III. Timing Analysis for Preemptive Multitasking Real-time Systems with Caches. *ACM Transaction on Embedded Computing Systems*, 6(1), 2007.

[175] Stefan M. Petters. Scheduling Analysis with Respect to Hardware Related Preemption Delay. In *Proceedings of the Workshop on Real-Time Embedded Systems*, 2001.

[176] Jan Staschulat, Simon Schliecker, and Rolf Ernst. Scheduling Analysis of Real-Time Systems with Precise Modeling of Cache Related Preemption Delay. In *Proceedings*

of the 17th Euromicro Conference on Real-Time Systems, ECRTS '05, pages 41–48, Washington, DC, USA, 2005. IEEE Computer Society.

[177] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. Static Analysis of Multi-Core TDMA Resource Arbitration Delays. *Real-Time Systems*, 50(2):pp185–229, March 2014.

[178] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[179] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

Index

abstract interpretation, 7
abstract syntax tree, 90
abstraction function, 16
analysis
  dynamic, 7
  fixed point , 12
  static, 7
anchor, 147
arrival time, 24

basic block, 19
best-case execution time, 30
best-case response time, 30
blocking time, 24
breadth-first search, 83, 255
bucket, 230

cache
  arithmetic, 43
  block reload time, 42
  block resilience, 56
  capacity miss, 42
  compulsory miss, 42
  conflict miss, 42
  line, 42
  locking, 51
  miss, 41, 42
  miss penalty, 42
  replacement policy, 43
  replacement policy, 41
  set, 42
  set associativity, 42
  write miss, 42
  write/allocate, 42
  write/no-allocate, 42
cache-related preemption delay, 50, 51
chain, *see* lattice
  condition, 14
chained blocking, 32
Church-Rosser transformation, 87
computation tree, 9
concretization function, 16
condensation graph, 85
constistency, *see* locally consistent
context-switch cost, 49
control flow, 8
  analysis, 19
  graph, 10
  path, 10
  reconstruction, 36
critical instant, 29
critical section, 31

data flow, 8
deadline, 24
deadline monotonic scheduling, 27
deadlock, 33
depth-first search, 85, 258
deterministic, 9
directed acyclic graph, 84
dominance relation, 85
domino effect, 35

earliest deadline first, 27
equivalence relation, 251
equivalence class, 251

evicting cache block, 54

finishing time, 24
fixed point, 13
  greatest, 13
  least, 13
flow
  constraint, 95
  fact, 36, 95
  frequency, 97
  network, 81
flow network
  residual, 82

Galois
  connection, 16
  insertion, 16
graph reduction, 85

halting problem, 9
hyper period, 24

implicit path enumeration technique, 100
integer linear program, 261
inter-arrival time, 23
interpretation, 8
iteration, 89
  scope, 106
  type, 106

job, 22

lambda calculus, 253
language, 8
lateness, 24
latest execution time, 172
lattice, 12
  chain, 14
  semi, 12
linear Programming, 261
live cache state, 60
local consistency, 16
locality principle, 41
longest paths, 84
loop, 105
  bounds, 36
  depth, 89
  entry path, 90
  exit path, 90
  kernel, 90
  natural, 88
  nesting forest, 89
  nesting relation, 89

maximum blocking
  path, 184
  time, 184
maximum flow problem, 82
maximum path length, 84
may analysis, 47
may set, 47
meet-over-all-paths, 12
memory gap, 40
minimal fixed point, 14
minimum cost flow problem, 83
minimum flow problem, 82
minimum path length, 83
model checking, 9
monotonicity, 13
must analysis, 48
must set, 46
mutex, 31
mutual exclusion, 22, 31

narrowing, 14
node splitting, 88

optimal instant, 30

partial order, 12
partially ordered set, 12
path analysis, 36
path expression, 96
  bounded, 98
precedence constraint, 22
preemption, 50
  interaction, 68

preemption point, 26
priority ceiling protocol, 33
priority inheritance protocol, 32
priority inversion problem, 31
program point, 8
program analysis, *see* analysis

quotient set, 251

rate monotonic scheduling, 27
reaching cache state, 60
reducibility, 88
residual flow network, 82
resource constraint, 22
response time, 24
response time
  analysis, 29
reverse postorder, 87

scheduling
  aperiodic, 23
  cooperative, 26
  deferred preemption, 25
  periodic, 23
  policy, 21
  problem, 21
  sporadic, 23
scope, 105
  bottom, 106
  entry, 106
  exit, 106
  immediate parent, 110
  order, 154
  parent, 110
  top, 106
  tree, 106
semantics, 8
  collecting, 10, 11
  fixed point, 12
  path, 10
  path-based collecting, 11
  trace, 9
semaphore, 31
single-source shortest paths, 83, 257
  directed acyclic graph, 258
soundness, 16
starting time, 24
strongly connected component, 85
subgraph, 85
synchronization, 31
syntax, 8

task, 21
  set, 22
termination, 9
timing anomaly, 35
topological order, 84
topological sort, 258
total unimodularity, 262
trace, 9
transfer function, 8
transformer, *see* transfer function
transitive closure, 85

unroll, 133
useful cache block, 52
  definitely cached, 54
utilization factor, 28

value analysis, 36

widening, 14
working set, 41
worst-case execution frequency, 204
worst-case response time, 29