Dissertation

Zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I

der Universität des Saarlandes

ILP-based Path Analysis on Abstract Pipeline State Graphs

von Diplom-Informatiker
Ingmar Jendrik Stein
aus Saarbrücken

Saarbrücken 2010

Tag des Kolloquiums:	28. Mai 2010
Dekan:	Prof. Dr. Joachim Weickert
Vorsitzender:	Prof. Dr. Raimund Seidel
Gutachter:	Prof. Dr. Dr. h. c. mult. Reinhard Wilhelm
	Prof. Dr. Sebastian Hack
Akademischer Mitarbeiter:	Dr.-Ing. Philipp Lucas

Impressum

Herstellung und Verlag: epubli GmbH, Berlin, http://www.epubli.de
Printed in Germany
ISBN: 978-3-86931-538-6

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Abstract

This thesis presents a novel approach to path analysis which is an integral part of the WCET analysis. Up to now, there have been two different methods for this step, each with its respective advantages and disadvantages. The new ILP-based path analysis on abstract pipeline state graphs supersedes the existing ones and combines the positive aspects of both but does not introduce new limitations. It provides high precision and the flexibility of user-provided annotations at the same time while opening up new possibilities for optimizations such as a new kind of persistence analysis.

Zusammenfassung

Diese Arbeit präsentiert einen innovativen Ansatz für die Pfadanalyse, ein integraler Bestandteil der WCET-Analyse. Bisher gab es zwei verschiedene Methoden für diesen Schritt, jede mit ihren spezifischen Vor- und Nachteilen. Die neue ILP-basierte Pfadanalyse auf abstrakten Pipelinezustandsgraphen ersetzt die beiden existierenden und kombiniert die positiven Aspekte, ohne neue Beschränkungen einzuführen. Sie bietet sowohl eine hohe Präzision als auch die Flexibilität benutzerbestimmter Annotationen. Darüber hinaus bietet sie neue Optimierungsmöglichkeiten wie zum Beispiel eine neuartige Persistenzanalyse.

Acknowledgements

First of all, I very much thank Prof. Dr. Dr. h. c. mult. Reinhard Wilhelm for the opportunity to write my thesis about this challenging and interesting topic. He provided me with a lot of freedom for approaching my goals.

Thanks go to Prof. Dr. Sebastian Hack for his willingness to examine this work. I am also indebted to Dr.-Ing. Philipp Lucas and Dr. Reinhold Heckmann for proof-reading parts of this work and giving valuable hints. Dr.-Ing. Florian Martin had the initial vision of the topic and had good ideas for future enhancements and improvements. Furthermore, I thank all colleagues at AbsInt Angewandte Informatik GmbH for a very pleasant working atmosphere.

Last but not least, I would like to thank my family for their support during the time of my research.

Contents

Chapter

1 Introduction

Today, microprocessors are pervasive not only in personal computers, but also in cars, planes and entertainment electronics. The information processing systems contained therein are called *embedded systems*. The programming of these systems differs significantly from ordinary application development. For example, many embedded systems have to fulfill *real-time requirements*, i. e. programs must guarantee to finish within a given timespan (deadline). If embedded systems with real-time requirements fulfill tasks relevant to security, they are subject to *hard* real-time requirements, because it may have catastrophic consequences if the maximum response time is exceeded. For example, the electronic control unit of a thrust-reverser has to comply with hard real-time, because a failure can lead to a plane crash.

To make sure that such systems work correctly, it is essential to find upper bounds for the execution time (worst-case execution time, *WCET*) of programs. In the majority of cases, it is not sufficient to only measure the runtime of a program with a given input because it is usually impossible to prove that this input leads to the maximum execution time (cf. figure 1.1). Similarly, it is often not feasible to measure the program with all possible inputs because the set of inputs may be prohibitively large. Therefore, an analysis is needed which determines the maximum runtime of a program *statically*, i. e. an analysis which calculates an upper bound for the runtime without actually running the program with any particular input. However, modern processors employ

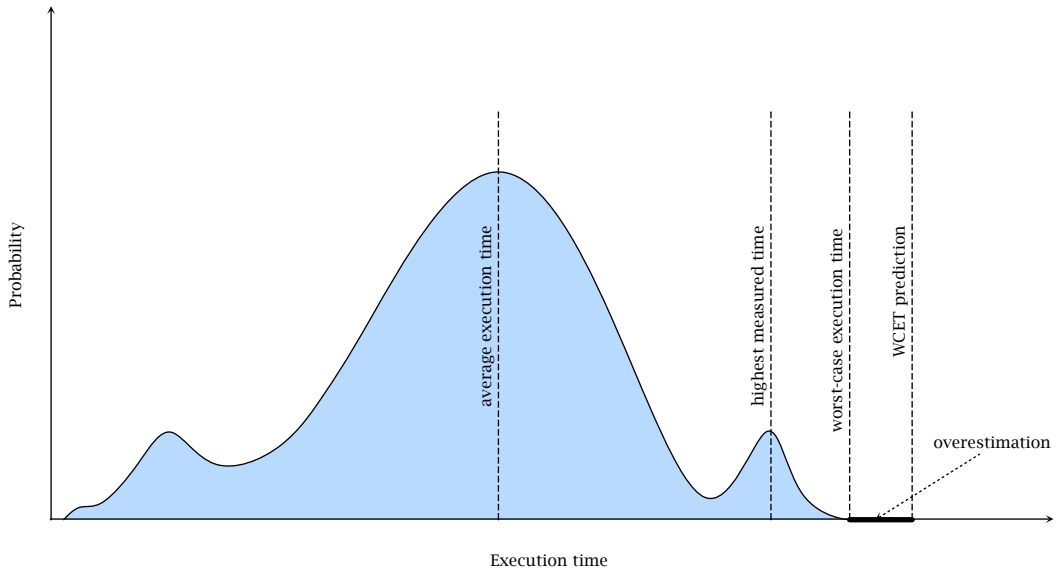


Figure 1.1.: Typical probability of observed execution times

different techniques to enhance performance that make such an analysis difficult: caches, pipelines and branch prediction. The state-of-the-art of solving this problem is a combination of abstract interpretation and integer linear programming (ILP) as it is used in the aiT component of the a^3 analysis framework by AbsInt Angewandte Informatik GmbH.

The new tool presented in this thesis enhances aiT by providing a better path analysis which replaces the existing one. It improves the precision of the worst-case execution time estimation, i. e. it reduces the amount of overestimation by up to 20 %. At the same time, it offers a high level of versatility and opens up new opportunities for further optimizations of the WCET prediction precision.

This thesis is structured as follows: the next chapter 2 presents an overview over this work and the analysis framework it is integrated into, followed by chapter 3 with the mathematical fundamentals which constitute the theoretical foundation for the following chapters. The next two chapters 4 and 5 describe the existing path analysis methods—the classical ILP-based path analysis and

its counterpart which works on abstract pipeline state graphs. A detailed description of the new analysis which improves upon the former can be found in chapter 6. Chapter 7 introduces a method to improve the WCET precision by using the results of a cache persistence analysis within the new path analysis. Implementation details, an evaluation and test results are contained in chapter 8. Chapter 9 gives an outlook on possible future work and extensions. Finally, chapter 10 summarizes the findings of this work. Appendix A includes some selected examples.

Chapter

2 Overview

The focus of this work is a novel approach to *path analysis*, which is an integral part of the WCET analysis. So far, there existed two different methods for this step, each with its respective advantages and disadvantages. The new approach tries to supersede the existing ones and combines the positive aspects of both but does not introduce new limitations.

2.1. The aiT Toolchain

The new path analysis method is a part of aiT , a modular WCET analysis framework. Figure 2.1 on page 10 depicts the components it is comprised of and shows how they interact. The individual framework modules as described in [Ferdinand and Heckmann, 2008] are:

- **Control-flow reconstruction** decodes, i. e. identifies instructions and reconstructs the control-flow graph (CFG) from the binary program.
- **Loop analysis** determines upper bounds for the number of iterations of loops.
- **Value analysis** computes value ranges for registers and memory cells and address ranges for instructions accessing memory.

- **Cache/pipeline analysis** classifies memory references as cache misses or cache hits and predicts the behavior of the program on the processor pipeline.

- **ILP generator** transforms the basic block execution times and the control flow into an integer linear program (ILP).

- **ILP solver** solves the ILP.

- **Evaluation** computes the worst-case execution path of the input program from the optimal ILP solution and computes the WCET contributions of the individual routines.

- **Visualization** generates a graph in the graph description language (GDL) to visualize the WCET path.

2.1.1. Control-flow Reconstruction

The result of this phase is a control flow graph stored in a CRL2 file. The CRL2 format is used as the data exchange format of the different phases.

CRL2 stands for **C**ontrol Flow **R**epresentation **L**anguage Version 2. This language was developed by the *Transferbereich 14* and describes the control flow graph of a program in a textual form. Design goals were efficient support of analyses and optimizations. The underlying structure is organized hierarchically: a graph consists of operations, instructions, basic blocks (cf. definition 2.1.2) and routines where the former are always contained within the latter. Example A.1.1 shows how a decoded binary program looks like when it is stored in CRL2 format.

Definition 2.1.1 (Control Flow Graph (CFG))**.** A *control flow graph* is a four-tuple $K = (V, E, s, x)$ with a set of nodes V, a set of directed edges $E \subseteq V \times V$,

a unique start node s and a unique end node x. The start node s fulfills:

$$\forall u \in V : (u, s) \notin E$$

The end node x fulfills:

$$\forall u \in V : (x, u) \notin E$$

Furthermore, a function $F : V \to P$ must exist to map nodes to program fragments. P designates the set of program fragments, e. g. given by the syntax tree representation.

Remark. The requirement for unique start and end nodes is no restriction, because each graph can simply be extended by two additional nodes. The construction of control flow graphs is described at length in [Allen, 1970].

Definition 2.1.2 (Basic Block)**.** Let $K = (V, E, s, x)$ be a control flow graph. A sequence of nodes $(n_1, \ldots, n_k)$ forms a *basic block*, if $\forall i \in \{1, \ldots, k-1\}$:

$$\begin{array}{ll} & n_i \text{ is the only predecessor of } n_{i+1} \\ \wedge & n_{i+1} \text{ is the only successor of } n_i \end{array}$$

Definition 2.1.3 (Maximal Basic Block)**.** A basic block is called *maximal* if it cannot be extended by including adjacent nodes without violating definition 2.1.2.

In the following, basic blocks are always assumed to be maximal.

Remark. A basic block has a single entry point and a single exit point. The start of a basic block may be the target of more than one branch instruction. The end of a basic block is either a branch instruction or the instruction preceding the destination of a branch instruction.

2.1.2. Loop Analysis

WCET analysis requires that upper bounds for the iteration numbers of all loops be known. aiT tries to determine the number of loop iterations by *loop bound analysis*. The loop bound analysis consists of two parts: a pattern matcher which recognizes loop patterns as generated by the most commonly used compilers and a data flow analysis which interprets the machine instructions in loop bodies to derive loop bounds [Cullmann, 2006]. Bounds for the iteration numbers of the remaining loops must be provided as user annotations.

2.1.3. Value Analysis

Value analysis tries to determine the values in the processor memory for every program point and execution context [Sicks, 1997, Fritz, 2001]. Its results are used to determine possible addresses of indirect memory accesses—important for cache analysis. The precision of the value analysis is usually so high that only a few indirect accesses cannot be determined exactly. Address ranges for these accesses may be provided by user annotations.

2.1.4. Cache/Pipeline Analysis

Pipeline analysis models the pipeline behavior to determine execution times for basic blocks of instructions. It takes into account the current pipeline state(s), in particular resource occupancies, contents of prefetch queues, grouping of instructions, and includes a cache analysis for the classification of memory references. The result is an execution time for each basic block in each distinguished execution context.

The cache/pipeline analysis uses abstract interpretation—a concept that will be described in greater detail in section 3.4. Basically, the cache/pipeline

analysis models the behavior of the pipeline of a specific processor by using abstract descriptions for the concrete pipeline states [Thesing, 2004]. The abstract pipeline states are used to solve a data flow problem on the input program and the result is an abstract pipeline state graph.

Definition 2.1.4 (Abstract Pipeline State Graph)**.** An *abstract pipeline state graph* is a weighted graph $G = (V, E, C)$, $C : E \to \mathbb{N}$, where V consists of the abstract pipeline states for the given input program. An edge weight $C((u, v))$ describes the costs in CPU cycles associated with the transition from the abstract state u to the abstract state v.

What exactly is abstracted in an abstract pipeline state highly depends on the processor architecture. Usually, the model includes abstractions for internal buffers, caches, jitter and queues. As a reference, the textual representation of an abstract pipeline state for the Motorola MPC755 is given in example A.5.1.

The pipeline analysis *splits* an abstract pipeline state into two or more successor states when it encounters imprecise information. This happens for instance when a memory access cannot be classified as *cache hit* or *cache miss*. How many successor states are generated depends on the *WCET computation mode*:

Global worst-case: all successor states are created and the pipeline analysis follows their further evolution.

Local worst-case: the pipeline immediately decides which successor likely leads to the worst-case execution time and follows the evolution of this single state. Splits may still occur; they are triggered by situations in which it is not clear which is the locally worst successor state.

The local worst-case computation mode leads to a massive reduction of the runtime of the pipeline analysis, but there is a risk that the successor state that seems to be worst from a local point of view does not lead to the global worst-case execution time.

2.1.5. Path Analysis

Using the results of the micro-architecture analyses, path analysis determines a safe estimate of the WCET by computing a worst-case path through the program.

Path Analysis with ILP Generator

The first variant of the path analysis models the program's control flow by an integer linear program so that the optimal solution to the objective function is the predicted worst-case execution time for the input program. Variables in the integer linear program correspond to basic blocks so that execution and traversal counts for every basic block and edge can be computed.

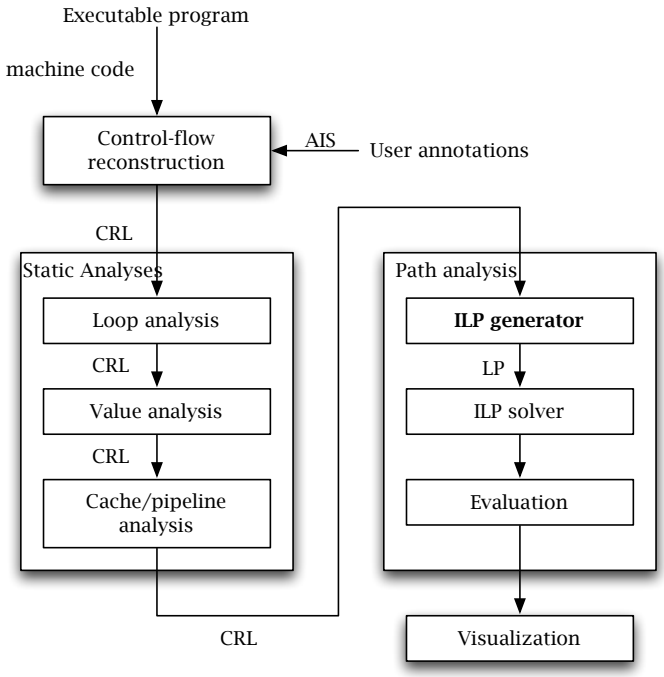


Figure 2.1.: aiT toolchain with ILP generator

Path Analysis on Abstract Pipeline State Graphs

A drawback of the path analysis with an ILP generator is that it uses the worst-case path through the pipeline states for each basic block. It therefore combines execution traces which might not represent an actual execution of the program. The resulting over-estimation can be eliminated by computing the worst-case path right from the pipeline state graph.

For this method, the toolchain is changed as follows: the ILP generator and ILP solver components are replaced by a single tool called `predan` which implements the path analysis using the abstract pipeline state graph stored in the so-called prediction file (cf. figure 2.2).

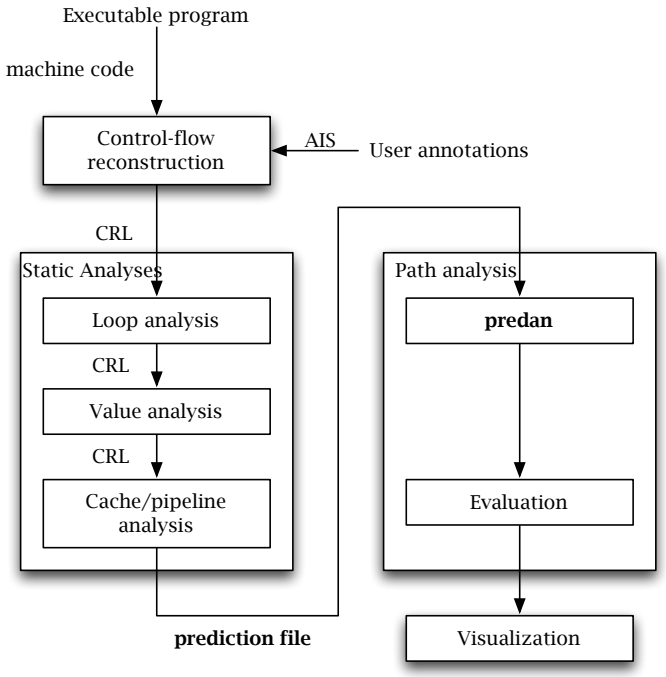


Figure 2.2.: aiT toolchain with prediction file

ILP-based Path Analysis on Abstract Pipeline State Graphs

Both path analysis variants above have some limitations: the ILP-based analysis suffers from inherent imprecisions and the path analysis on abstract pipeline state graphs does not handle loops and user annotations.

The variant that is introduced in this work overcomes these limitations by using the abstract pipeline state graph to generate an ILP which is able to incorporate loop constraints and user annotations while still providing the highest level of precision.

The toolchain is modified from the original ILP-based method as follows: the ILP generator is replaced by the new implementation which reads the prediction file in addition to the control flow graph and the evaluation step is adapted to the changed semantics of the ILP variables (cf. figure 2.3).

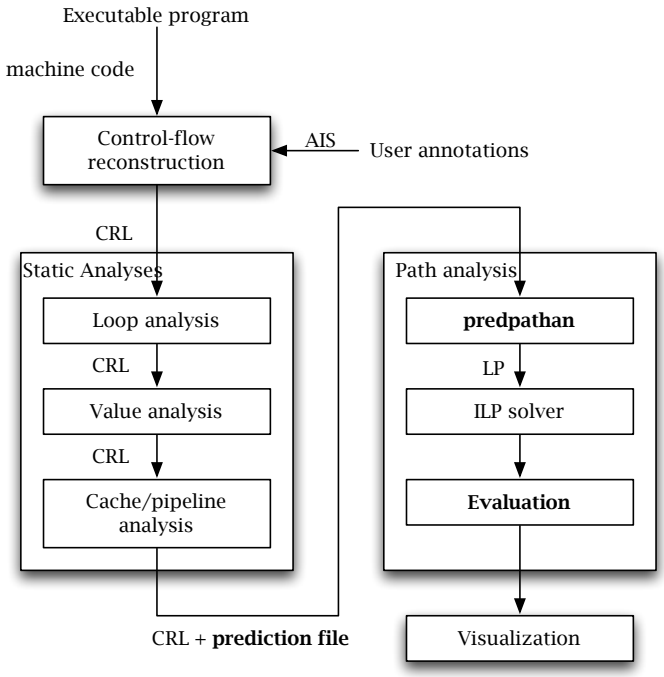


Figure 2.3.: aiT toolchain with ILP solver and prediction file

2.2. Calling Contexts

In an over-simplified view, a static program analysis computes some abstract information for every program point p. The abstract information for p has to be a correct approximation of the concrete program state at p whenever control reaches p (no matter what happened before). Thus, the abstract information for a program point p in a routine R must approximate all program states at p in all calls of R.

To be more concrete, consider a value analysis that computes an interval of possible values for every register r. The interval for r is a correct approximation of a concrete program state if it contains the value of r in this program state. Suppose now a routine R is called twice, once with parameter 0 and once with parameter 3. Then the best abstract information that can be obtained for the parameter register is the interval $[0, 3]$, which indicates that the value of the register might be 0, or 1, or 2, or 3. The precision of the analysis can be improved considerably if the analysis does not compute a single abstract value for each program point in R, but two different ones, one for each call of R. In the example considered above, these are the intervals $[0, 0]$ for the call with parameter 0 and $[3, 3]$ for the call with parameter 3. The values 1 and 2 are thus excluded successfully.

To be more general again, the analyses compute an abstract information for every pair of a program point p and a possible calling context of p. All program points in a given routine R have the same set of calling contexts. Each calling context indicates a particular way of calling R.

Chapter

3 Theoretical Background

The path analysis builds upon the results of diverse data flow analyses, one of which is the combined cache/pipeline analysis. A data flow analysis is an application of abstract interpretation on control flow graphs. Thus, this chapter gives an overview over the mathematical foundations of the concepts used in abstract interpretation. For that purpose, it introduces the fundamental terms of lattice theory, Galois theory and fixed point iteration.

The two ILP-based path analysis methods use integer linear programming as a means to solve the path analysis problem. Subsequently to the foundations of abstract interpretation, this chapter describes the structure of linear programs and the $\mathcal{NP}$-hard class of integer linear programs and points out some of their important properties. It also outlines algorithms to solve linear and integer linear programs.

3.1. Lattice Theory

Definition 3.1.1 (Partial and Total Order)**.** Let M be a set. A binary relation $\sqsubseteq \subseteq M \times M$ is called *partial order* of M, if:

1. Reflexivity:

$$\forall x \in M : x \sqsubseteq x$$

2. Transitivity:

$$\forall x,y,z \in M : x \sqsubseteq y \wedge y \sqsubseteq z \implies x \sqsubseteq z$$

3. Antisymmetry:

$$\forall x,y \in M : x \sqsubseteq y \wedge y \sqsubseteq x \implies x = y$$

The relation is called *total order* of M, if additionally:

$$\forall x,y \in M : x \sqsubseteq y \vee y \sqsubseteq x$$

A set M together with a partial order $\sqsubseteq$ is called a *partially ordered set* $(M, \sqsubseteq)$.

The relation $\sqsubseteq$ has a pointwise extension for functions:

$$f \sqsubseteq g \iff \forall x : f(x) \sqsubseteq g(x)$$

Definition 3.1.2 (Upper/Lower Bound)**.** Let $(M, \sqsubseteq)$ be a partially ordered set and $N \subseteq M$. An element $x \in M$ is called an *upper bound* of N, if:

$$\forall y \in N : y \sqsubseteq x$$

x is called *least upper bound* of N ($\bigsqcup N$), if:

1. x is an upper bound of N

2. $x \sqsubseteq z$ holds for all upper bounds z of N

$\bigsqcup$ is called *union*. The least upper bound of two elements x and y is denoted by $x \sqcup y$.

The *lower* respectively *greatest lower bound* of N ($\bigsqcap N$) are defined analogously. $\bigsqcap$ is then called *intersection*.

Definition 3.1.3 (ω-Chain)**.** Let $(M, \sqsubseteq)$ be a partially ordered set. An *ω-chain*[1] of a partial order is an ascending chain of elements $x_0, x_1, x_2, \ldots$ of M with:

$$x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_i \sqsubseteq \cdots$$

If an ω-chain additionally fulfills:

$$x_0 \sqsubset x_1 \sqsubset x_2 \sqsubset \cdots \sqsubset x_i \sqsubset \cdots$$

then it is called a *strictly ascending* ω-chain. Here, $x \sqsubset y$ is defined by $x \sqsubset y \iff x \sqsubseteq y \wedge x \neq y$.

Definition 3.1.4 (Complete Partial Order)**.** Let $(M, \sqsubseteq)$ be a partially ordered set. $(M, \sqsubseteq)$ is called a *complete partial order* (CPO) if there exists a least upper bound of the set $\{x_i \mid i \in \omega\}$ for each ω-chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \cdots \sqsubseteq x_i \sqsubseteq \cdots$ with $x_i \in M$.

Definition 3.1.5 (Ascending Chain Condition)**.** A partially ordered set $(M, \sqsubseteq)$ fulfills the *ascending chain condition*, if each ω-chain is finite, i. e. has only finitely many different elements.

Remark. A partially ordered set with ascending chain condition is a complete partially ordered set.

Definition 3.1.6 (Complete Lattice)**.** A partially ordered set $(M, \sqsubseteq)$ is called *complete lattice*, if each subset of M has a least upper bound and a greatest lower bound. A lattice is written as a tuple $(M, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ with $\bot = \bigsqcap M$ and $\top = \bigsqcup M$.

Remark. A complete lattice is especially a complete partially ordered set.

[1] The ordered set $(\mathbb{N}, \leq)$ is denoted by ω.

Definition 3.1.7 (Dual Lattice)**.** Let $(M, \bot, \top, \sqsubseteq, \sqcup, \sqcap)$ be a complete lattice. The *dual lattice* is given by swapping the following symbols: $\sqsubseteq$ by $\sqsupseteq$, $\sqcup$ by $\sqcap$ and $\bot$ by $\top$.

Definition 3.1.8 (Monotonic Function)**.** A function $f : A \to B$ of two partially ordered sets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ is called *monotonic*, if:

$$\forall x, y \in A : x \sqsubseteq_A y \implies f(x) \sqsubseteq_B f(y)$$

Definition 3.1.9 (Distributive Function)**.** A function $f : A \to B$ of two complete lattices $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ is called *distributive*, if:

$$\forall x, y \in A : f(x) \sqcup f(y) = f(x \sqcup y)$$

Remark. A distributive function is always monotonic.

Definition 3.1.10 (Continuous Function)**.** A function $f : A \to B$ of two complete partially ordered sets $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ is called *continuous*, iff for all ω-chains $x_0 \sqsubseteq_A x_1 \sqsubseteq_A x_2 \sqsubseteq_A \cdots \sqsubseteq_A x_i \sqsubseteq \cdots$ in A holds:

$$\bigsqcup_{i \in \omega} f(x_i) = f\left(\bigsqcup_{i \in \omega} x_i\right)$$

Remark. A continuous function is always monotonic.

3.2. Fixed Point Iteration

In abstract interpretation, recursive systems of equations need to be solved, where the values to be defined also appear on the right side of the equation. For example, this is the case when using abstract interpretation to analyze the behavior of programs that contain loops or recursive functions.

A popular example for such a system of equations is the factorial function

$$fac(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \cdot fac(n-1) & \text{else.} \end{cases}$$

A solution is expected to fulfill the equation. To solve these recursive definitions, there exists a simple, iterative approach: starting with the least element of the solution space $\bot$, an element is inserted in the definition equation. This yields the definition for the next greater element. This process is repeated n times and so defines a function on the interval $[0, \ldots, n-1]$. The sought-after function of the natural numbers is found by forming the limit for $n \to \infty$.

Definition 3.2.1 (Prefixed Point)**.** Let $f : M \to M$ be a function. An element $x \in M$ is called *prefixed point* of f, if:

$$f(x) \sqsubseteq x$$

Definition 3.2.2 (Fixed Point)**.** Let $f : M \to M$ be a function. An element $x \in M$ is called *fixed point* of f, if:

$$f(x) = x$$

Theorem 3.2.1 (Fixed Point Iteration)**.** *Let $(M, \sqsubseteq)$ be a complete partially ordered set with the least element $\bot$ and $f : M \to M$ a continuous function. Let* $\mathrm{fix} : (M \to M) \to M$ *be defined by:*

$$\mathrm{fix}(f) = \bigsqcup_{i \in \omega} f^i(\bot)$$

Then $\mathrm{fix}(f)$ *is a fixed point of f and the least prefixed point of f. Therefore, it holds:*

1. $f(\mathrm{fix}(f)) = \mathrm{fix}(f)$
2. $\forall x \in M : f(x) \sqsubseteq x \implies \mathrm{fix}(f) \sqsubseteq x$

Remark. Because each fixed point is also a prefixed point, it follows that $\text{fix}(f)$ is the least fixed point of f.

Definition 3.2.3 (Least and Greatest Fixed Point)**.** The least fixed point of a function f is also called $lfp(f)$, the greatest fixed point $gfp(f)$.

3.3. Galois Theory

Abstract interpretation works with representatives of concrete values. The abstraction of elements of a concrete data space is carried out with the help of concepts from Galois theory. The concepts required for this work are given below. Further information and many examples can be found in [Nielson et al., 1999].

Definition 3.3.1 (Galois Connection)**.** Let $(L, \sqsubseteq)$ and $(M, \sqsubseteq)$ be complete lattices and $\alpha : L \to M$ and $\gamma : M \to L$ monotonic functions. The four-tuple (L, α, γ, M) is called *Galois connection* between the two lattices, iff:

- $\gamma \circ \alpha \sqsupseteq id_L$
- $\alpha \circ \gamma \sqsubseteq id_M$

The function α is also called *abstraction function* and the function γ is also called *concretization function*.

In a Galois connection (L, α, γ, M), multiple elements M can exist which are an abstraction of the same element of L, because the abstraction function α is not required to be injective. On the other hand, the lattice M might contain more elements than necessary for the abstraction of L. The following variation of the Galois connection is used in abstract interpretation to avoid this:

Definition 3.3.2 (Galois Insertion)**.** Let (L, α, γ, M) be a Galois connection. It is

called a *Galois insertion*, iff:

$$\alpha \circ \gamma = id_M$$

Consequently, no precision is lost when first concretizing and then abstracting an element within a Galois insertion.

3.4. Abstract Interpretation

Abstract interpretation is a general concept of program analysis and was first introduced by Cousot and Cousot in 1977 [Cousot and Cousot, 1977, Cousot and Cousot, 1992]. Because data flow analysis can be interpreted as a special case of abstract interpretation, this section presents a short overview over the theoretical framework.

Abstract interpretation aims to replace *concrete semantics* by *abstract semantics*. This is done by replacing concrete values by abstract values in such a way that both have a fixed relation, i. e. for each concrete value k, an abstract value $\overline{k}$ should exist which describes k. This is expressed by the *abstraction function*: $\overline{k} = \alpha(k)$. For each operation *op* within the concrete semantics, an abstraction $\overline{op}$ must exist, so that it holds:

$$\alpha\,(k_1\ op\ k_2) \sqsubseteq (\alpha\,(k_1))\ \overline{op}\ (\alpha\,(k_2))$$

This ensures that the chosen operation $\overline{op}$ correctly abstracts the operation *op*. $\overline{op}$ is called *abstract operation* on the abstract domain of the abstract semantics which should approximate the corresponding operation on the concrete domain. A big challenge of abstract interpretation is to choose the abstract semantics. It should be designed in such a way that calculations always terminate and that the results allow for usable conclusions on the behavior of the original program.

Example 3.4.1 (Sign Determination)**.** The following example determines the sign of arithmetic expressions with the help of abstract semantics. Allowed operators are addition (+) and multiplication (×). For the abstract semantics, the abstract domain is chosen to be $\{neg, zero, pos, ?\}$ where the question mark stands for values with an unknown sign. The abstraction function is given by the signum function σ:

$$\sigma(x) = \begin{cases} neg & x < 0 \\ zero & x = 0 \\ pos & x > 0 \end{cases}$$

The abstract operators $\oplus$ und $\otimes$ adhere to the calculation rules given in table 3.1.

$\oplus$	*pos*	*zero*	*neg*	?
pos	*pos*	*pos*	?	?
zero	*pos*	*zero*	*neg*	?
neg	?	*neg*	*neg*	?
?	?	?	?	?

$\otimes$	*pos*	*zero*	*neg*	?
pos	*pos*	*zero*	*neg*	?
zero	*zero*	*zero*	*zero*	*zero*
neg	*neg*	*zero*	*pos*	?
?	?	*zero*	?	?

Table 3.1.: Calculation rules for $\oplus$ and $\otimes$

Extended by a smallest element $\perp$, the set of abstract values represents a complete lattice. Figure 3.1 illustrates the lattice together with its ordering.

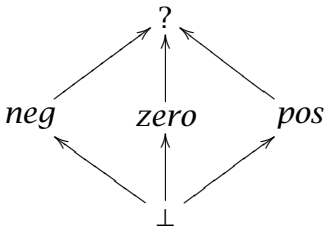


Figure 3.1.: Complete lattice of the abstract values

After the introductory example, abstract interpretation will be formally defined during the course of this section. To begin with, the concept of local consistency is needed:

Definition 3.4.1 (Local Consistency)**.** Let (L, α, γ, M) be a Galois insertion. A concrete function $f : L \to L$ and an abstract function $f' : M \to M$ are called *locally consistent*, if it holds that:

$$\forall x \in L : f(x) \sqsubseteq \gamma\left(f'\left(\alpha\left(x\right)\right)\right)$$

Figure 3.2 illustrates this relation.

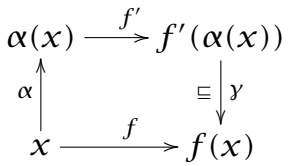


Figure 3.2.: Local consistency

With the help of this notion, abstract interpretation can now be defined as follows:

Definition 3.4.2 (Abstract Interpretation)**.** An *abstract interpretation* consists of two components:

- a Galois insertion (L, α, γ, M) and
- a pair of locally consistent functions $f : L \to L$ and $f' : M \to M$.

Instead of proving properties of function f, one can now also prove them for its abstraction f'. The two conditions above guarantee for the correctness.

Fixed point iteration is needed to compute a result if f respectively f' contain a recursion. In this regard, the following relation is useful:

Theorem 3.4.1 (Fixed Point Relation). *Given an abstract interpretation by means of the Galois insertion* (L, α, γ, M) *and the locally consistent functions* f *and* f'*. The following relations hold for the fixed points of both functions:*

- $lfp(f) \sqsubseteq \gamma(lfp(f'))$
- $gfp(f) \sqsubseteq \gamma(gfp(f'))$

3.5. Integer Linear Programming

3.5.1. Linear Programs

This section introduces the structure of Linear Programs. How they can be solved will be shown in the next section.

Definition 3.5.1 (Comparison of Vectors). Let $\Delta \in \{\leq, =, \geq\}$ be a comparison operator and let $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$. Then we define

$$\boldsymbol{a} \,\Delta\, \boldsymbol{b} \iff \boldsymbol{a}_i \,\Delta\, \boldsymbol{b}_i \quad \forall i = 1, \ldots, n$$

Definition 3.5.2 (Linear Combination). Let $\boldsymbol{x} \in \mathbb{R}^n$ be variable and let $\boldsymbol{a} \in \mathbb{R}^n$ be constant. Then $\boldsymbol{a}^{\mathrm{T}}\boldsymbol{x}$ is called a *linear combination* of x.

Definition 3.5.3 (Linear Program). Let $\boldsymbol{t} \in \mathbb{R}^d, \boldsymbol{b} \in \mathbb{R}^m, A \in \mathbb{R}^{m \times d}$ be known and constant. A *Linear Program* (LP) is the task to maximize $\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x}$ in such a way that $\boldsymbol{x} \in \mathbb{R}^d_{\geq 0} \wedge A\boldsymbol{x} \leq \boldsymbol{b}$. In short, this is written:

$$\max \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : A\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}^d_{\geq 0}$$

Definition 3.5.4. In definition 3.5.3 the function $C : \mathbb{R}^d \to \mathbb{R}$ where $C(\boldsymbol{x}) = \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x}$ is called *objective function*. The inequalities given by $A\boldsymbol{x} \leq \boldsymbol{b}$ are called *constraints*. $\boldsymbol{x}$ is said to be a *feasible* solution, if it satisfies $A\boldsymbol{x} \leq \boldsymbol{b}$. Let

$P = \left\{\boldsymbol{x} \in \mathbb{R}^d_{\geq 0} : A\boldsymbol{x} \leq \boldsymbol{b}\right\}$ be the set of feasible solutions. $\boldsymbol{x}^*$ is said to be an *optimal* solution, if $\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x}^* = \max\left\{\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : \boldsymbol{x} \in P\right\}$.

To reduce a problem of minimizing to one of maximizing, the objective function can be multiplied by -1.

There are three cases that can occur when an LP is tried to be solved:

1. $P = \emptyset$: the LP is *infeasible*.
2. $P \neq \emptyset$, but $\nexists \sup\left\{\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : \boldsymbol{x} \in P\right\}$: the LP is *unbounded*.
3. $P \neq \emptyset$, and $\exists \max\left\{\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : \boldsymbol{x} \in P\right\}$: the LP is *feasible* and has a finite solution.

To find the solution of a linear program, upper bounds of the objective function must be computed. The problem of finding the least upper bound is also an LP that is defined as follows.

Definition 3.5.5 (Primal and Dual Problem)**.** Let $\max \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : A\boldsymbol{x} \leq \boldsymbol{b}, A \in \mathbb{R}^{m\times d}$, $\boldsymbol{x} \in \mathbb{R}^d_{\geq 0}$ be a linear program. Let this program be called *primal problem*. The *dual problem* is the problem of finding the least upper bound of $\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x}$, which is defined as follows: $\min \boldsymbol{y}^{\mathrm{T}}\boldsymbol{b} : \boldsymbol{y}^{\mathrm{T}}A \geq \boldsymbol{t}^{\mathrm{T}}, \boldsymbol{y} \in \mathbb{R}^m_{\geq 0}$.

The two following theorems hold (*Duality Theorems of Linear Programming*):

Theorem 3.5.1 (Weak Duality)**.** *Let $\bar{\boldsymbol{x}}$ be a feasible solution of the primal problem $\max \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : A\boldsymbol{x} \leq \boldsymbol{b}, A \in \mathbb{R}^{m\times d}, \boldsymbol{x} \in \mathbb{R}^d_{\geq 0}$ and let $\bar{\boldsymbol{y}}$ be a feasible solution of its dual problem $\min \boldsymbol{y}^{\mathrm{T}}\boldsymbol{b} : \boldsymbol{y}^{\mathrm{T}}A \geq \boldsymbol{t}^{\mathrm{T}}, \boldsymbol{y} \in \mathbb{R}^m_{\geq 0}$. Then it holds that:*

$$\bar{\boldsymbol{y}}^{\mathrm{T}}\boldsymbol{b} \geq \boldsymbol{t}^{\mathrm{T}}\bar{\boldsymbol{x}}$$

Proof. Because $\bar{\boldsymbol{x}} \geq 0$, $\bar{\boldsymbol{y}} \geq 0$ and $\boldsymbol{b} \geq A\bar{\boldsymbol{x}}$, it holds that $\bar{\boldsymbol{y}}^{\mathrm{T}}\boldsymbol{b} \geq \bar{\boldsymbol{y}}^{\mathrm{T}}A\bar{\boldsymbol{x}}$. Since $\bar{\boldsymbol{y}}^{\mathrm{T}}A \geq \boldsymbol{t}^{\mathrm{T}}$, it follows: $\bar{\boldsymbol{y}}^{\mathrm{T}}\boldsymbol{b} \geq \boldsymbol{t}^{\mathrm{T}}\bar{\boldsymbol{x}}$. □

Theorem 3.5.2 (Strong Duality). *Let $\boldsymbol{x}^*$ be a feasible solution of the primal problem $\max \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : A\boldsymbol{x} \leq \boldsymbol{b}, A \in \mathbb{R}^{m\times d}, \boldsymbol{x} \in \mathbb{R}^{d}_{\geq 0}$ and let $\boldsymbol{y}^*$ be a feasible solution of its dual problem $\min \boldsymbol{y}^{\mathrm{T}}\boldsymbol{b} : \boldsymbol{y}^{\mathrm{T}}A \geq \boldsymbol{t}^{\mathrm{T}}, \boldsymbol{y} \in \mathbb{R}^{m}_{\geq 0}$. Then it holds that:*

$$\boldsymbol{y}^{*\mathrm{T}}\boldsymbol{b} = \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x}^* \iff \boldsymbol{x}^* \text{ and } \boldsymbol{y}^* \text{ are optimal}$$

Corollary 3.5.1. *If the primal problem is unbounded, the dual problem is infeasible.*

Corollary 3.5.2. *If there are feasible solutions of the primal and the dual problems, then there is an optimal solution. The values of the objective function of the two problems are equal for the optimal solution.*

The following Simplex algorithm exploits that corollary 3.5.2 can be used to check if a solution $\boldsymbol{x}$ of the primal problem is optimal. Starting with an initial solution, it improves the solution in each iteration until the following conditions imply optimality: $\boldsymbol{x}$ is optimal iff $\exists \boldsymbol{y}$ such that

$$A\boldsymbol{x} \leq \boldsymbol{b} \text{ (primal feasible)} \tag{3.1}$$
$$\boldsymbol{y}^{\mathrm{T}}A \geq \boldsymbol{t}^{\mathrm{T}} \text{ (dual feasible)} \tag{3.2}$$
$$\boldsymbol{y}^{\mathrm{T}}\left(A\boldsymbol{x} - \boldsymbol{b}\right) = 0 \text{ (complementary)} \tag{3.3}$$
$$\left(\boldsymbol{y}^{\mathrm{T}}A - \boldsymbol{t}^{\mathrm{T}}\right)\boldsymbol{x} = 0 \text{ (slackness)} \tag{3.4}$$
$$\left(\boldsymbol{y}^{\mathrm{T}}A - \boldsymbol{t}^{\mathrm{T}}\right)_i \cdot \boldsymbol{x}_i = 0 \text{ (binding constraints)} \tag{3.5}$$

3.5.2. Simplex Algorithm

This section introduces a non-formal description of the *Simplex* algorithm, created by George Dantzig in 1947. There is a vast amount of literature about LP solving and the Simplex algorithm available for the interested reader, e. g. [Chvátal, 1983, Schrijver, 1996, Nemhauser and Wolsey, 1988].

Each constraint of a linear program specifies a half-space in $\mathbb{R}^d_{\geq 0}$. Their intersection is the set of all feasible variable assignments. This convex area is either empty, unbounded or a polytope. An optimal solution is found in one of the vertices of this polytope. Starting with an arbitrary vertex, a better solution of the objective function is searched by following one of the outgoing edges of that vertex. This is repeated until no adjacent vertex has a better value, which means that the optimal solution has been found. Figure 3.3 illustrates this algorithm.

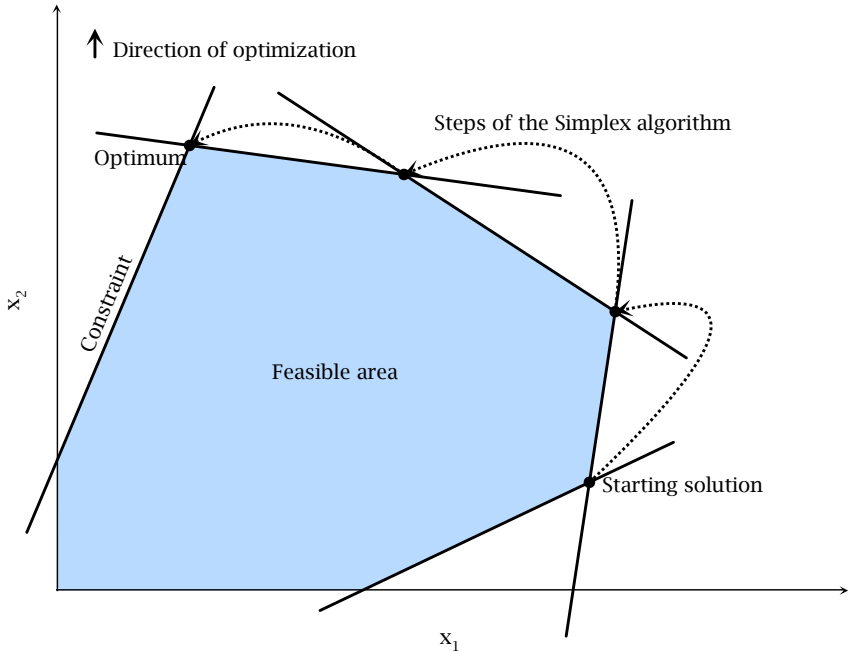


Figure 3.3.: The Simplex algorithm in $\mathbb{R}^2_{\geq 0}$

The Simplex algorithm can be used to solve large problems, since for most constraint systems, its runtime is $\mathcal{O}(n)$ for n constraints. However, Klee and Minty showed in 1972 that the worst-case runtime is exponential by giving an example (a distortion of an n-dimensional cube) where the Simplex algorithm visits all 2^n vertices before finding the optimal solution (cf. [Chvátal, 1983]), its asymptotic complexity is therefore $\mathcal{O}(2^n)$. There are better algorithms from the complexity point of view, e. g. the Ellipsoid method or the Projective

Scaling algorithm by Karmarker, which have polynomial runtime.

3.5.3. Integer Linear Programs

Many problems require the solution of a linear program to be integer, i. e. in definition 3.5.3 on page 24 it must additionally hold that $\boldsymbol{x} \in \mathbb{N}_0^d$.

This type of constraint will be particularly important for the path analysis variants in chapter 4 that use linear programs where the variables of the LP are execution counts of basic blocks or abstract pipeline states, which are naturally integers.

Definition 3.5.6 (Integer Linear Program)**.** Let $\boldsymbol{t} \in \mathbb{R}^d, \boldsymbol{b} \in \mathbb{R}^m, \boldsymbol{A} \in \mathbb{R}^{m \times d}$ be known and constant. An *Integer Linear Program* (ILP) is the task to maximize $\boldsymbol{t}^{\mathrm{T}}\boldsymbol{x}$ in such a way that $\boldsymbol{x} \in \mathbb{N}_0^d \wedge \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}$.

$$\max \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \mathbb{N}_0^d$$

The corresponding *relaxed* LP is obtained by omitting the integer requirement:

$$\max \boldsymbol{t}^{\mathrm{T}}\boldsymbol{x} : \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b}, \boldsymbol{x} \in \mathbb{R}_{\geq 0}^d$$

3.5.4. Branch and Bound Algorithm

The basic idea of the Branch and Bound algorithm is to solve the relaxed LP and then split the domain of feasibility into two sub-problems in order to satisfy the demand for integer variables. Each sub-problem is then solved until all variables are integers.

Let Ψ be an ILP and let Ψ' be the relaxed problem. If it is feasible, solving Ψ' yields a solution $\hat{\boldsymbol{x}} \in \mathbb{R}_{\geq 0}^d$.

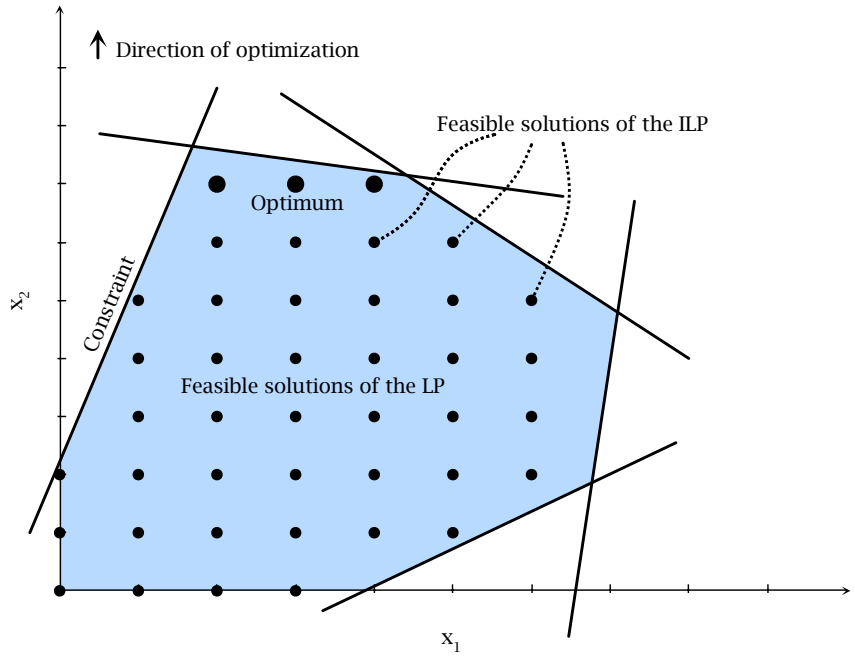


Figure 3.4.: Domain of feasibility of an ILP (grid points) and the corresponding domain of the relaxed problem (shaded area)

If $\hat{\boldsymbol{x}} \in \mathbb{Z}^d$, then $\hat{\boldsymbol{x}}$ is also a solution for Ψ. Otherwise, a coordinate $i \in \{1, \ldots, n\}$ is chosen such that $\hat{\boldsymbol{x}}_i \notin \mathbb{Z}$. Ψ' is partitioned into the two subproblems $\tilde{\Psi}_1$ and $\tilde{\Psi}_2$ by adding one of the following inequalities to Ψ':

$$\boldsymbol{x}_i \leq \lfloor \hat{\boldsymbol{x}}_i \rfloor \tag{3.6}$$

$$\boldsymbol{x}_i \geq \lceil \hat{\boldsymbol{x}}_i \rceil \tag{3.7}$$

These constraints exclude $\hat{\boldsymbol{x}}$ as a solution for $\tilde{\Psi}_1$ and $\tilde{\Psi}_2$. This method is repeated until all variables are integers.

Chapter

4 ILP-based Path Analysis

4.1. ILP

This section describes how an ILP is generated for worst-case path analysis. The first publications about path analysis using implicit path enumeration were [Li et al., 1995, Li and Malik, 1995a, Li and Malik, 1995b]. In the same year, Puschner and Koza compiled a technical report about this topic (*see* [Puschner and Koza, 1995]). The following description is based on the approach that splits the ILP-based path analysis from the micro-architecture analysis. This technique was introduced in [Theiling and Ferdinand, 1998] and improved upon in [Theiling, 2003].

4.1.1. Objective Function

The ILP-based path analysis uses nodes $n \in V$ to represent basic blocks of the source program. Let $T((u,v),c)$ be the length of the longest path from any start state of basic block u to any start state of block v in context c as determined by the combined cache and pipeline analysis (see section 2.1.4). Furthermore, let $C(e,c)$ be the *execution count*, which indicates how often control passes along edge e in context c. If one knows for a specific run of the code the execution counts $C(e,c)$ for each edge e in each context c, then

one can get an upper bound for the time of this run by taking the sum of $C(e,c) \cdot T(e,c)$ over all edge-context pairs (e,c). Thus, the task of obtaining a global WCET estimation can be solved by finding a feasible assignment of execution counts $C(e,c)$ to edge-context pairs that maximizes the objective function

$$\max: \sum_{(e,c) \in E^*} C(e,c) \cdot T(e,c) \tag{4.1}$$

The value of this sum is then the desired global WCET estimate.

In formula 4.1, $E^* \subseteq E \times U$ is the set of all edge-context pairs (e,c) that appear in the program, U denotes the set of all contexts. The ILP variables are $C(e,c)$, the values $T(e,c)$ are constant.

Figure 4.1 illustrates the representation of basic blocks in the integer linear program. It shows three basic blocks b_1, b_2 and b_3, their pipeline state graphs and the edges connecting them. The ILP for this graph would contain variables for the execution counts of the edges e_1 and e_2 in every context c.

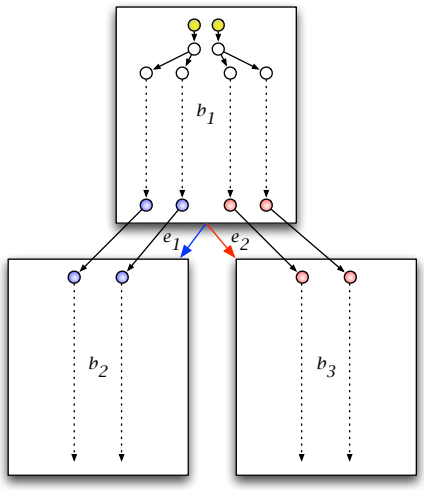


Figure 4.1.: Basic blocks with pipeline states and edges

4.1.2. Program Start Constraints

Let v_0 be the start node of the program and c_0 the start context. Since the WCET for *one* execution of the program is to be derived, the sum of the execution counts of all edges leaving v_0 is 1.

$$\sum C((v_0, w), c_0) = 1 \quad \forall (v_0, w) \in E$$

4.1.3. Structural Constraints

For all nodes, we sum up the outgoing and incoming control flow. The following constraints are generated from the CFG:

$$\forall v \in V : \sum_{((u,v),c) \in E^*} C((u, v), c) = \sum_{((v,w),c) \in E^*} C((v, w), c)$$

Infeasible Edges

The data flow analyses preceding the path analysis are able to find infeasible paths in many cases. For example, the value analysis uses its knowledge about register contents to predict the outcome of conditional branches. If it is able to prove that a branch is either taken or not taken, it marks the other outcome as infeasible.

Example 4.1.1 (Infeasible Code)**.** Let the input program be the binary program produced by the translation of the following *C* code:

```
if (i >= 0 && i < 10)
    array[i] = i;
```

```
else
    printf("Index␣out␣of␣bounds!");
```

Listing 4.1: Infeasible code

Suppose that the register corresponding to the variable i is known to contain a value in the range $[0 \ldots 9]$ in the analyzed execution context. Then, the basic block corresponding to the `else`-case is marked as infeasible.

In some cases, the micro-architecture analysis is also able to find infeasible edges, for example if the user tries to resolve a memory access by annotating an address range, but specifies an invalid range. This error will not be detected until the micro-architecture analysis reaches this memory access. As a result, the block containing the invalid access will also be infeasible.

To account for these by preventing that infeasible nodes are considered in the path analysis, an additional constraint is generated for each edge (u, v) leading to a node v that is infeasible in context c:

$$\forall u \in V \text{ s.t.}((u, v), c) \in E^* : C((u, v), c) = 0$$

4.1.4. Loop Constraints

Loop constraints bound the number of iterations of a loop. They are specified as the minimum and maximum number of iterations for each invocation of the loop (i. e. for each calling context c).

Definition 4.1.1. Let l be a loop and $e \in \text{entries}(l)$ one of its entry edges.

The *minimum loop execution count* of l via e in context c is written as $n_{\min}(e, c)$.

The *maximum loop execution count* of l via e in context c is written as $n_{\max}(e, c)$.

A loop is executed as many times as its header is executed. To limit the number of iterations of the loop per entry, the execution count of the header must be compared to the traversal counts of the loop's entry edges (cf. figure 4.2).

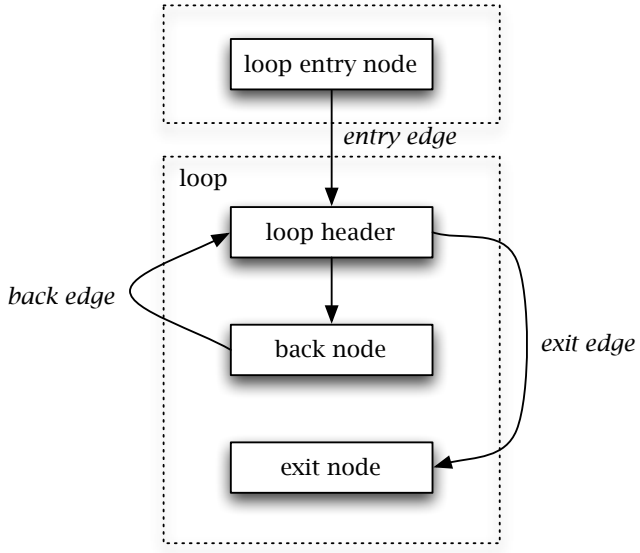


Figure 4.2.: A simple loop with all the important edges

Loop bound constraints are generated as follows for each loop l with loop header h:

$$\sum_{c \in \Gamma(h)} \sum_{n \in \text{succ}(h)} C((h,n),c) \geq \sum_{e \in \text{entries}(l)} \sum_{c \in \Gamma(e)} n_{\min}(e,c) \cdot C(e,c)$$
$$\sum_{c \in \Gamma(h)} \sum_{n \in \text{succ}(h)} C((h,n),c) \leq \sum_{e \in \text{entries}(l)} \sum_{c \in \Gamma(e)} n_{\max}(e,c) \cdot C(e,c)$$

where $\Gamma(n) : V \to \mathcal{P}(U)$ is a function that returns the set of possible contexts for a given node in the control flow graph.

The above are simplified loop constraints used to illustrate the concept. The actual implementation uses more complex and more precise constraints which are described in detail in [Theiling, 2003].

4.1.5. Time-based Loop Constraints

Some loops cannot be bounded by a fixed number of iterations. Typically, such loops implement some form of busy waiting, i. e. the loops are executed repeatedly while waiting for an external event. A busy waiting loop can be used to block a process until a certain condition is true or to stall the CPU while waiting for an I/O transfer to complete.

```
while (!can_read_io()) {
    // wait
}
```

Listing 4.2: Busy waiting loop

In order to bound this type of loop, a user can provide an annotation:

```
snippet routine "main" + 1 loop takes max 42 cycles;
```

For each such annotation, a new constraint is added to the ILP. The constraint bounds a subset of the objective function by the number of cycles given in the annotation.

The final constraint has the form

$$\sum_{x \in L} C(x) \cdot T(x) \leq A$$

where A is the time given in the annotation and L contains all edge-context pairs that are reachable when performing a depth-first search starting at the loop's start node and ending at the loop's exit.

4.1.6. User Added Constraints

Users may have additional knowledge about their program which can be used to improve the precision of the WCET analysis, e. g. they might know that two paths are mutually exclusive—a fact which may not be apparent for a static analysis which tries to detect infeasible paths. The framework allows users to add linear constraints to the ILP using annotations which look like the following

$$\texttt{flow each } c_0 * (pp_0) + \ldots + c_n * (pp_n) = c_{n+1} * (pp_{n+1}) + \ldots + c_m * (pp_m) + c_{m+1} \tag{4.2}$$

$$\texttt{flow sum } c_0 * (pp_0) + \ldots + c_n * (pp_n) = c_{n+1} * (pp_{n+1}) + \ldots + c_m * (pp_m) + c_{m+1} \tag{4.3}$$

Instead of =, the operators $\leq$ and $\geq$ may be used as well. The variables pp_i denote program points given as basic block addresses, $c_i \in \mathbb{N}$ are constant factors.

The qualifiers `each` and `sum` define whether the flow constraint applies to all contexts cumulatively or to each context separately. With `sum`, the constraint does not apply to the execution counts in individual contexts, but to the sum over the number of executions in all contexts. In contrast, the syntax 4.2 can be used to specify a constraint which applies to each context individually.

Example 4.1.2. Given the following *C* code snippet:

```
if (mode == 0)
    do_expensive_calculations();

/* ... */

if (mode == 1)
```

```
    do_other_stuff();
```

Listing 4.3: Mode-driven code

Suppose that the value analysis cannot determine the exact value of the `mode` variable, e. g. because it is initialized in initialization code which is not part of the analyzed task. The WCET analysis is then unable to deduce that the two calls to `do_expensive_calculations` and `do_other_stuff` are mutually exclusive. The resulting control flow graph is depicted in figure 4.3.

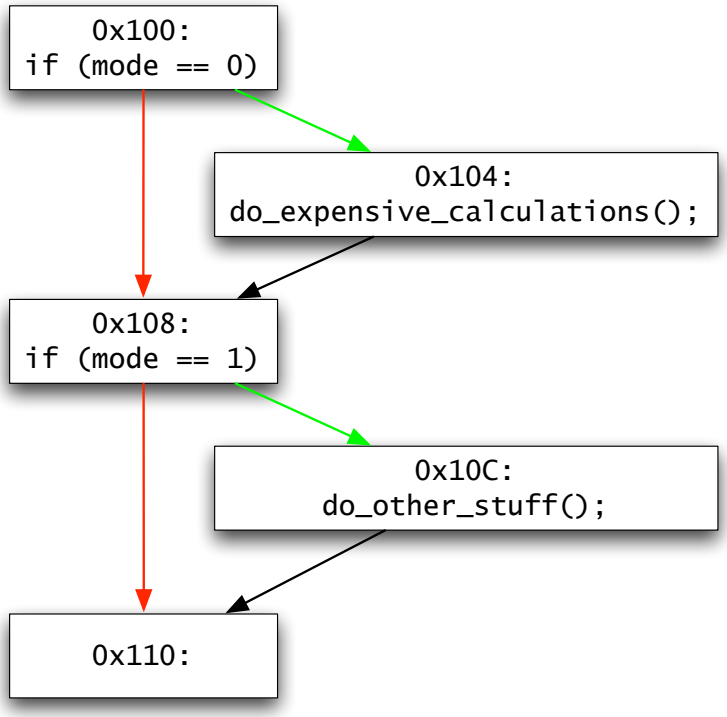


Figure 4.3.: Use-case for user constraints

The worst-case path without additional user-added constraints is given by (0x100, 0x104, 0x108, 0x10C, 0x110) although this path is not feasible (assuming that `mode` is unchanged in this program). To reduce the overestimation,

the user can add a flow-constraint like

```
flow each (0x104) + (0x10C) = (0x110)
```

The constraint means “in each context, the sum of the execution counts of blocks 0x104 and 0x10C is equal to the execution count of block 0x110”. The two possible WCET paths are now (0x100, 0x104, 0x108, 0x110) and (0x100, 0x108, 0x10C, 0x110).

Example 4.1.2 is a common scenario during the analysis of embedded control systems. Many of these systems work in different operating modes such as start-up, stand-by or shut-down. It it also often the case that the same program is deployed on many different systems where each instance uses a different mode which decides which parts of the program should be active for this instance. An introduction to operating mode specific WCET analysis can be found in [Lucas et al., 2009].

A way to detect path exclusions for mode-driven code was presented in [Stein and Martin, 2007] where flow-annotations are generated automatically to improve the precision of the WCET prediction.

Another common use-case for user constraints are non-natural loops, i. e. loops with multiple entry points (rarely generated by compilers, almost only occur with handwritten assembly code). In that case, an annotation

```
flow each (entry_1) + ... + (entry_n) = (dom)
```

bounds the execution counts of the loop entry points entry_i by the execution count of a block dom dominating all entries.

4.2. Implementation

The ILP-based path analysis in aiT was implemented by Henrik Theiling in 2002 as the tool `pathan` (short for *path an*alysis). It serves as a frontend with a user interface for the `libpathan` library which contains the algorithms and data structures.

Chapter

5 Path Analysis on Abstract Pipeline State Graphs

The path analysis on prediction files is a new approach developed by Niklas Matthies in [Matthies, 2006].

A prediction file is an optional output of the pipeline/cache analyzer which describes all possible pipeline states in a condensed form together with the external bus events. This file was mainly meant for comparing hardware traces with aiT predictions.

This abstraction was chosen since using the full representation of the state graphs (including the content of abstract pipeline states) for all basic blocks and all contexts for trace validation seemed to be impossible at that time. It turned out that for applications with moderate size the prediction files might also be used for path analysis, although the graph representation in a prediction file can take more than 1 GB size.

The approach presented in chapter 4 leads to overestimations that result from the way `pathan` computes the WCET: for each basic block the maximum number of cycles is calculated that are spent in the instructions contained in the basic block. This does not take into account that it might not be possible to spend the maximal number of cycles in each of two consecutive basic blocks (see figure 5.1).

The computed WCET using path analysis on prediction files should always be less or equal the global WCET computed with `pathan` in combination with an ILP solver. The overestimations which are avoided by the path analysis on prediction files are those, where the cache/pipeline analysis splits the states due to unknown states of the processor, e. g. due to unknown cache states as it can be seen in figure 5.1. The `pathan` approach for this example calculates a WCET of $2+5+8=15$ whereas the method based on prediction files calculates $2+1+8=11$.

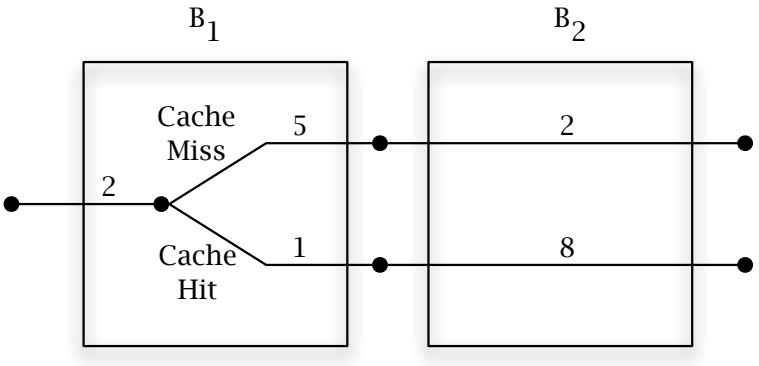


Figure 5.1.: Split due to unknown cache state

As mentioned in section 2.1.4 on page 8, the local WCET analysis avoids some of the splits due to unknown states and follows only the case taking locally more cycles, which is a cache miss in this example. But in some cases, it is locally undecidable, which decision will be the local worst-case, so there are still some split events left. Table 5.1 on the next page shows in which cases the local WCET analysis also splits, exemplified by the micro-architecture analysis for the Motorola MPC755.

Event	Split
Cache hit or miss	✗
Cache clash	✗
Cache manipulation	✓
Inexact jitter	✗
Inexact PCI jitter	✗
Option `--data-wait=perhaps` (not default)	✓
Unknown SDRAM tag register	✗
Imprecise SDRAM access	✓
Instruction `DCBT` (Data Cache Block Touch)	✓
Instruction `DCBTST` (Data Cache Block Touch for Store)	✓
Instruction `DCBST` (Data Cache Block Store)	✗
Instruction `DCBF` (Data Cache Block Flush)	✗
Arithmetic operations (IU1, IU2, SRU)	✗
Unknown branch prediction	✓

Table 5.1.: Splits during local WCET analysis for the MPC755

Due to the fact that the pipeline analysis also splits while computing the local worst-case, the problem described in figure 5.1 may still arise. Thus, the WCET computed by the path analysis on prediction files may be smaller than the WCET computed by `pathan` using the local worst-case option of the pipeline.

5.1. Prediction Files

The state graph generated by the micro-architectural analysis is stored in a so-called *prediction file*. The syntax of this textual file is defined in table 5.2 starting on page 46. It contains a sequence of the following types of elements:

Normal edges These elements (non-terminal *normal_edge*) describe an edge in the abstract pipeline state graph and consist of the IDs of the source

and target nodes, the execution time for this edge (in CPU cycles) and optionally the addresses of the associated basic block and instruction, the execution context as well as a list of events. If the code location for an edge is not specified, it belongs to the same location as the previous edge. The event list is not relevant for path analysis and is used by other tools which work on prediction files.

Subsume edges The pipeline analysis connects two states with a subsume edge (non-terminal *subsume_edge*) if an abstract state is subsumed by another abstract state (i. e. it is a subset of the other state). Subsume edges are similar to normal edges with an execution time of zero.

Node aliases Aliases (non-terminal *equal_node*) are generated by the micro-architectural analysis to identify two nodes with the same abstract state. They consist of two node IDs where the first occurs for the first time and now represents an alias for the second ID. The first node ID is called *alias node*, the second *referenced node*.

Start/end markers These elements (*node_start*, *node_end*, *edge_start* and *edge_end*) represent the start and end of the current CFG item[1], respectively.

End of block This element (non-terminal *basic_block_end*) marks the end of the current basic block.

The formal grammar of the prediction file format is given below:

start	→	*items*
items	→	*item items*
	|	ε
item	→	*node*
	|	*edge*
	|	*marker*

continued on next page

[1] here: either a node or an edge in the control flow graph.

continued from previous page

node	→	*start_node*
	|	*equal_node*
	|	*keep_node*
	|	*final_node*
start_node	→	`S` *id*
equal_node	→	`E` *id1* `:` *id2*
keep_node	→	`k` *id*
final_node	→	`F` *id*
marker	→	*node_start*
	|	*node_end*
	|	*edge_start*
	|	*edge_end*
	|	*basic_block_end*
node_start	→	`n` *node_id* `:` *context*
node_end	→	`N`
edge_start	→	`t` *edge_id* `:` *context*
edge_end	→	`T`
basic_block_end	→	`B`
edge	→	*normal_edge*
	|	*subsume_edge*
normal_edge	→	`e` *id1* `d` *id2* `c` *num code_info events*
code_info	→	`i` *adr1*
	|	`i` *adr1* `,` *adr2*
	|	`i` *adr1* `,` *adr2* `,` *ctx*
events	→	*event events*
	|	ε
event	→	*tstart*
	|	*tack*
	|	*aack*
	|	*cache*
tstart	→	`S` *a1* `,` *a2* `l` *len access_type segment_type*

continued on next page

continued from previous page

access_type	→	`r`
	|	`w`
segment_type	→	`c`
	|	`d`
tack	→	`|a`
aack	→	`|A`
cache	→	`|c` *cache_event_list*
cache_events	→	*cache_event* `,` *cache_events*
	|	ε
cache_event	→	*cache_type access_type a1* `,` *a2* `,` *s1* `,` *s2*
	|	*cache_type event_type abstract_cache tag*
cache_type	→	`i`
	|	`d`
event_type	→	`e`
	|	`l`
abstract_cache	→	`u`
	|	`a`
subsume_edge	→	`s` *id1* `:` *id2*

Table 5.2.: Prediction file syntax

The abstract pipeline state graph $G = (V, E, C)$ can be reconstructed from the prediction file as follows:

- an abstract pipeline state $v \in V$ is represented by an integer identifier, i. e. $V \subset \mathbb{N}$. The actual content of the abstract state including the abstract cache state is not needed for path analysis.

- an edge (u, v) is part of E if either a normal or a subsume edge connect the two state identifiers u and v.

- the cost $C(e)$ of an edge e is either 0 if e is a subsume edge or the number of CPU cycles as given in the prediction file.

5.2. Implementation

The implementation of the path analysis on abstract pipeline state graphs in the modular analysis framework aiT is called `predan` and was described in [AbsInt Angewandte Informatik GmbH, 2006a]. It exploits the facts that the pipeline state graph modulo loops is a *DAG* (directed acyclic graph) and that the edges in the prediction file occur in topological order. Given those two prerequisites, it can employ a very fast algorithm based on depth-first search to find the longest path.

Chapter

6 ILP-based Path Analysis on Abstract Pipeline State Graphs

Following the introduction of the existing path analyses and the theoretical background, this chapter represents the core of this thesis and presents the newly developed path analysis.

While `predan` is very fast because it uses a depth-first search on the state graph, it does not allow the user to specify additional constraints in the AIS annotations. The new approach combines the advantages of both methods, i. e. it is now possible to use user constraints together with the path analysis on prediction files.

However, the analysis time is expected to increase by an order of magnitude (dominated by the time it takes the ILP solver to solve the ILP).

The prediction files can become very large even for medium-sized programs. Therefore, `predpathan` employs several techniques to compress the pipeline state graph. The compression algorithms reduce the complexity of the state graph (and therefore of the ILP) in ways that do not alter the WCET path. In the absence of user constraints, `predpathan` produces the exact same result

as the predan approach.

6.1. Graph Compression

The following graph compression methods substitute subgraphs of the abstract pipeline state graph $S \subset G$ with equivalent subgraphs S' so that $G' = (G \setminus S) \cup S'$ and

$$\text{wcet}(G') = \text{wcet}(G) \tag{6.1}$$

Definition 6.1.1 (In-Edges)**.** The *in-edges* $\text{in}(n)$ of a node $n \in V$ are defined as

$$\text{in}(n) \coloneqq \{(m, n) \in E\}$$

Definition 6.1.2 (out-Edges)**.** The *out-edges* $\text{out}(n)$ of a node $n \in V$ are defined as

$$\text{out}(n) \coloneqq \{(n, m) \in E\}$$

Definition 6.1.3 (In-Degree)**.** The *in-degree* $\text{indeg}(n)$ of a node $n \in V$ is defined as

$$\text{indeg}(n) \coloneqq |\text{in}(n)|$$

Definition 6.1.4 (Out-Degree)**.** The *out-degree* $\text{outdeg}(n)$ of a node $n \in V$ is defined as

$$\text{outdeg}(n) \coloneqq |\text{out}(n)|$$

Definition 6.1.5 (Extended Graph)**.** The *extended graph* G^* of a graph $G =$

(V, E) is given by $G^* = (V^*, E^*)$, where

$$
\begin{aligned}
V^* &:= V \cup \{s, e\} \\
E^* &:= E \\
&\quad \cup \{(s, s') : s' \in V \wedge \text{indeg}(s') = 0\} \\
&\quad \cup \{(e', e) : e' \in V \wedge \text{outdeg}(e') = 0\} \\
s &\notin V \\
e &\notin V
\end{aligned}
$$

That is, the graph G is extended by a new start node s which has edges to all nodes which have no predecessor and a new end node e which is connected to all nodes which have no successor.

For each compression method, a proof is given that condition 6.1 holds by showing that the same is true for the respective extended graphs S^* and S'^*.

6.1.1. Chain Compression

The pipeline analysis can produce long chains of states which resemble the cyclewise evolution of the pipeline. A chain of states which have a single predecessor and a single successor can be merged into two states: one state for the start of the chain and one for the end of the chain. The cost of the edge between the start and end node is the sum of the edge weights on the chain.

Definition 6.1.6 (Chain Compression)**.** The sequence of nodes $n_1, \ldots, n_x \in V$

is called a *compressible chain* of length $x > 2$, iff

$$\begin{aligned} \text{succ}(n_i) &= \{n_{i+1}\} \quad \forall 1 \leq i \leq x-1 \\ \text{pred}(n_i) &= \{n_{i-1}\} \quad \forall 2 \leq i \leq x \end{aligned}$$

A subgraph S for the chain compression consists of the nodes in a compressible chain and the edges connecting them, i. e.

$$S = (\{n_1, \ldots, n_x\}, \{(n_i, n_{i+1}) : 1 \leq i \leq x-1\})$$

It is replaced by S' where

$$S' = (\{n_1, n_x\}, \{(n_1, n_x)\})$$

and

$$C'((n_1, n_x)) = \sum_{1 \leq i \leq x-1} C((n_i, n_{i+1}))$$

Correctness

Claim.

$$\text{wcet}(S^*) = \text{wcet}(S'^*)$$

Proof. Let $n_1, \ldots, n_x$ be a chain of length $x > 2$. Then

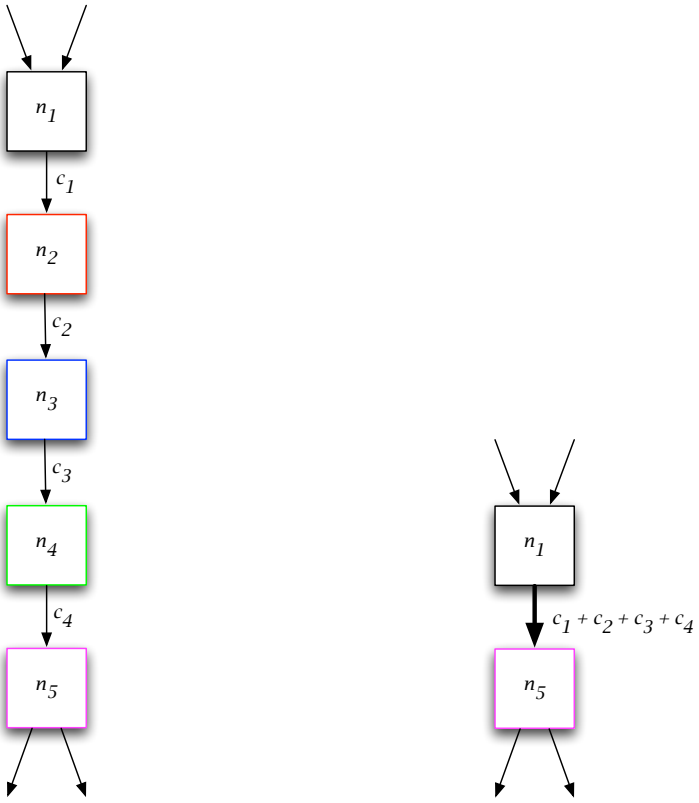
$$\begin{aligned} \text{wcet}(S^*) &= \sum_{i \in \text{in}(n_1)} C(i) + \sum_{1 \leq i \leq x-1} C((n_i, n_{i+1})) + \sum_{o \in \text{out}(n_x)} C(o) \\ &= \sum_{i \in \text{in}(n_1)} C(i) + C'(n_1, n_x) + \sum_{o \in \text{out}(n_x)} C(o) \\ &= \text{wcet}(S'^*) \end{aligned}$$

□

Runtime

In order to find the chains in an abstract pipeline state graph G, the algorithm only needs to look at each node $n \in V$ once, because it can easily be checked if n is part of a chain. n is removed by the chain compression, iff $\text{indeg}(n) = \text{outdeg}(n) = 1$. Therefore, the runtime of the compression method is $\mathcal{O}(|V|)$.

Example 6.1.1. Figure 6.1 shows the effect of the chain compression.



(a) Subgraph before applying the chain compression

(b) Subgraph after applying the chain compression

Figure 6.1.: Chain compression example

6.1.2. Basic Block Compression

For each basic block b and one of its contexts c, the *basic block graph* $G_{b,c}$ is the part of the state graph that lies in b and belongs to context c. Each basic block graph is a weighted directed acyclic graph (DAG). For the purposes of path analysis, it can be reduced to a graph which consists only of the start and end nodes of the basic block connected by edges which represent the longest paths between them. The DAG-property allows to use a very fast multiple-sources multiple-targets longest-path algorithm.

The algorithm begins by sorting the basic block graph topologically.

Definition 6.1.7 (Topological Ordering)**.** A *topological ordering* of a graph is a total ordering of its nodes which is compatible with the partial order R induced on the nodes where x comes before y ($x\ R\ y$) if there is a directed path from x to y in the graph.

Theorem 6.1.1. *A graph has a topological ordering if and only if it is a directed acyclic graph.*

The sorting algorithm is based on depth-first search:

Algorithm 1 Topological sort

1: run DFS(G), computing finish time f[v] for each vertex v
2: As each vertex is finished, insert it onto the front of a list
3: **return** the list

Runtime $\Theta(|V| + |E|)$

Algorithm 2 Depth-first search

```
1: procedure DFS(G)
2:     for all u ∈ V(G) do
3:         color[u] = white
4:     end for
5:     time = 0
6:     for all u ∈ V(G) do
7:         if color[u] == white then
8:             DFSVisit(u)
9:         end if
10:    end for
11: end procedure
1: procedure DFSVisit(u)
2:     color[u] = gray
3:     d[u] = ++time
4:     for all v ∈ adj(u) do
5:         if color[v] == white then
6:             DFSVisit(v)
7:         end if
8:         color[u] = black
9:         f[u] = ++time
10:    end for
11: end procedure
```

Runtime $\Theta(|V| + |E|)$

After sorting the graph, the longest paths can be computed be repeatedly solving the single-source multiple-targets longest-path problem for each start node s with $\text{indeg}(s) = 0$ in the basic block graph:

Algorithm 3 Single-source multiple-targets longest-path

```
1: procedure SSMT(G, s)
2:     for all vertex y ∈ G do
3:         d(s, y) = ∞                                   ▷ initialization
4:     end for
5:     d(s, s) = 0
6:     for all vertex y in a topological ordering of G do
7:         choose edge (x,y) maximizing d(s,x) + C((x,y))
8:         d(s,y) = d(s,x) + C((x,y))
9:     end for
10: end procedure
```

Runtime $\mathcal{O}(|V| + |E|)$.

Algorithm 4 Multiple-sources multiple-targets longest-path

```
1: procedure MSMT(G)
2:     for all vertex s ∈ G with indeg(s) = 0 do
3:         compute SSMT(G, s)
4:     end for
5: end procedure
```

Runtime $\mathcal{O}(|S| \cdot (|V| + |E|))$ where $S = \{n \in V : \text{indeg}(n) = 0\}$.

The results of the multiple-sources multiple-targets longest-path algorithm are stored in $d(x, y)$ where $d(x, y) = \infty$ means that y is not reachable from x, otherwise $d(x, y)$ contains the cost of the longest path from x to y.

Formally, the basic block compression can be defined as follows: let $G_{b,c} = (V_{b,c}, E_{b,c})$ be the basic block graph. Then, $G'_{b,c} = (V'_{b,c}, E'_{b,c})$ is derived from

G where

$$V'_{b,c} = \{v \in V_{b,c} : \text{indeg}(v) = 0 \vee \text{outdeg}(v) = 0\}$$
$$E'_{b,c} = \left\{(s,e) \in V^2_{b,c} : \text{indeg}(s) = \text{outdeg}(e) = 0 \wedge d(s,e) \neq \infty\right\}$$
$$C'((s,e)) = d(s,e) \quad \forall (s,e) \in E'_{b,c}$$

The correctness of the graph substitution follows directly from the construction and the correctness of the multiple-sources multiple-targets longest-path algorithm.

6.1.3. Infeasible Nodes

The pipeline analysis can mark nodes as infeasible when it finds out that a path results in inconsistent states, that is, abstract states which have no corresponding concrete state. This often occurs when the pipeline analysis handles a branch instruction and splits the abstract pipeline state for each possible successor. When the branch is resolved, all states except the one with the correctly predicted successor are marked as infeasible.

The infeasible nodes are removed from the ILP along with all paths only leading to infeasible nodes, i. e. the infeasible property is propagated backwards:

Algorithm 5 Purge infeasible nodes

```
1: while ∃i ∈ infeasible do
2:     V = V \ {i}
3:     infeasible = infeasible \ {i}
4:     for all (x, i) ∈ E do
5:         E = E \ {(x, i)}
6:         if outdeg(x) = 0 then
7:             infeasible = infeasible ∪ {x}
8:         end if
9:     end for
10: end while
```

6.1.4. ε-transition Elimination

Edges with 0 cost (e. g. subsume edges) are removed from the ILP by merging the nodes connected by them.

6.1.5. Buddy Nodes

Definition 6.1.8. The function $\text{pred} : V \to \mathcal{P}(V)$ computes the *predecessors* of a node $n \in V$:

$$\text{pred}(n) \coloneqq \{v : (v, n) \in E\}$$

The *successors* are defined analogously by $\text{succ} : V \to \mathcal{P}(V)$:

$$\text{succ}(n) \coloneqq \{v : (n, v) \in E\}$$

Nodes which have the same set of predecessors or successors with equal edge costs are called *buddy nodes* and can be merged.

$$M_{\text{pred}} = \{(u,v) : u,v \in V \wedge \text{pred}(u) = \text{pred}(v) \wedge C_{\text{pred}(u,v)}\}$$
$$M_{\text{succ}} = \{(u,v) : u,v \in V \wedge \text{succ}(u) = \text{succ}(v) \wedge C_{\text{succ}(u,v)}\}$$

where the two predicates $C_{\text{pred}} : V^2 \to \mathbb{B}$ and $C_{\text{succ}} : V^2 \to \mathbb{B}$ are defined as:

$$C_{\text{pred}(x,y)} \coloneqq \forall z \in \text{pred}(x) : C((z,x)) = C((z,y))$$
$$C_{\text{succ}(x,y)} \coloneqq \forall z \in \text{succ}(x) : C((x,z)) = C((y,z))$$

Update for $(u,v) \in M_{\text{succ}}$:

$$V' = V \setminus \{v\}$$
$$E' = E \setminus \{(x,y) \in E : x = v \vee y = v\} \cup \{(x,u) : x \in \text{pred}(v)\}$$
$$C'((x,u)) = \begin{cases} C((x,v)) & x \in \text{pred}(v) \wedge x \notin \text{pred}(u) \\ \max(C((x,u)), C((x,v))) & x \in \text{pred}(v) \wedge x \in \text{pred}(u) \end{cases}$$

Analog update for $(u,v) \in M_{\text{pred}}$.

Runtime

In order to find buddy nodes in an abstract pipeline state graph, the algorithm does not need to compare each node $n \in V$ with each other node to check if they are buddy nodes. Instead, it can test the *in-siblings* and *out-siblings*:

Definition 6.1.9 (In-Siblings)**.** The *in-siblings* $\text{siblings}_{\text{in}}(x)$ of a node $x \in V$ are defined as

$$\text{siblings}_{\text{in}}(x) \coloneqq \bigcup_{y \in \text{pred}(x)} \text{succ}(y)$$

Definition 6.1.10 (Out-Siblings)**.** The *out-siblings* $\text{siblings}_{\text{out}}(x)$ of a node $x \in$

V are defined as

$$\text{siblings}_{\text{out}}(x) := \bigcup_{y \in \text{succ}(x)} \text{pred}(y)$$

The in-siblings of n are the candidates for the buddy nodes of n with equal incoming edges and the out-siblings of n are the candidates for the buddy nodes of n with equal outgoing edges. Therefore, a small number of nodes and edges has to be visited for each node, so that the runtime of the buddy node compression algorithm is usually $\mathcal{O}(|V| + |E|)$.

The two different cases for buddy nodes are shown in figures 6.2 and 6.3 on the facing page.

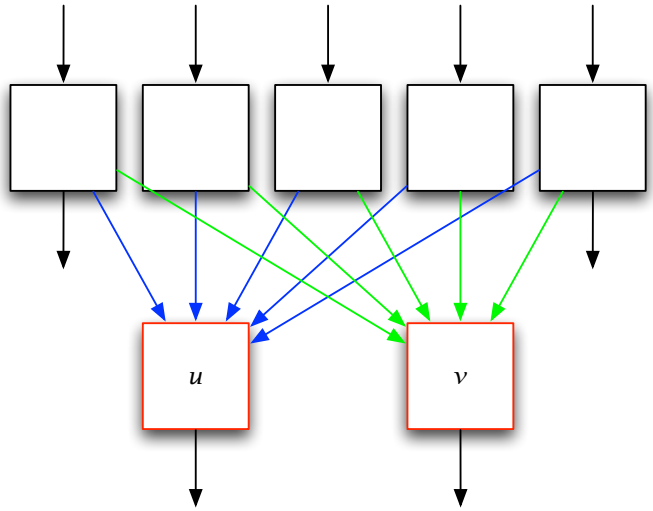


Figure 6.2.: Buddy Nodes (same incoming edges with equal costs)

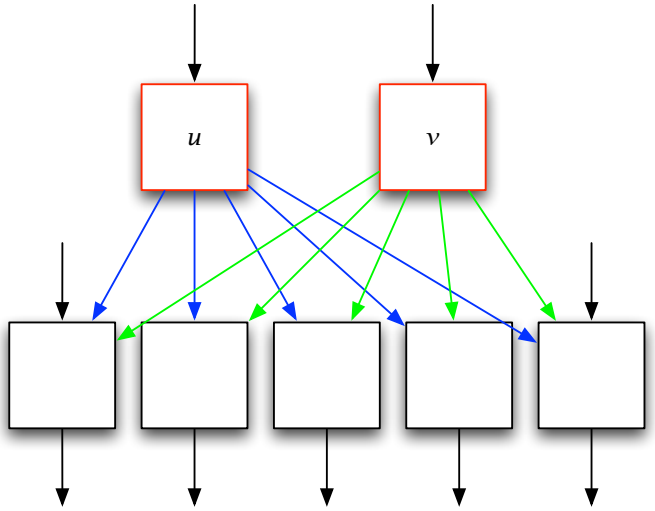


Figure 6.3.: Buddy Nodes (same outgoing edges with equal costs)

The result of merging the buddy nodes in figure 6.3 can be seen in figure 6.4.

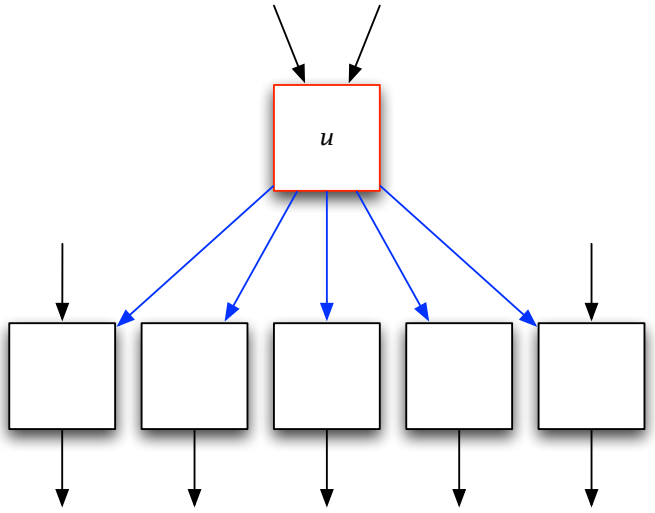


Figure 6.4.: Merged Buddy Nodes

Correctness

Claim. Merging buddy nodes does not alter the WCET.

Proof. Let $u, v \in V$ be buddy nodes with the same outgoing edges with equal costs. The WCET of the uncompressed extended subgraph S^* is given by

$$\begin{aligned}
\text{wcet}(S^*) &= \max_{b \in \{u,v\}} \left(\max_{e \in \text{in}(b)} C(e) + \max_{x \in \text{succ}(b)} \left(C(b,x) + \max_{e \in \text{out}(x)} C(e) \right) \right) \\
&= \max_{b \in \{u,v\}} \left(\max_{e \in \text{in}(b)} C(e) + \max_{x \in \text{succ}(u)} \left(C(u,x) + \max_{e \in \text{out}(x)} C(e) \right) \right) \\
&= \max_{b \in \{u,v\}} \left(\max_{e \in \text{in}(b)} C(e) \right) + \max_{x \in \text{succ}(u)} \left(C(u,x) + \max_{e \in \text{out}(x)} C(e) \right) \\
&= \max_{x \in \text{pred}(\{u,v\})} C((x,u)) + \max_{x \in \text{succ}(u)} \left(C(u,x) + \max_{e \in \text{out}(x)} C(e) \right) \\
&= \text{wcet}(S'^*)
\end{aligned}$$

Analog proof for buddy nodes with the same incoming edges. □

6.1.6. Chain Combination

The chain compression does not collate chains of pipeline states which cross basic block boundaries. If, however, some chains that span several basic blocks have the same start and end nodes, only the chain with the highest aggregate cost has to be added to the ILP (cf. figure 6.5 on page 65).

The formal specification of the *chain combination* algorithm requires the following definition:

Definition 6.1.11 (uv-chain)**.** A sequence of nodes $(n_1, \ldots, n_x)$ with $x > 2$ is

called a uv-chain of length x, iff

$$\begin{aligned} n_1 &= u \\ n_x &= v \\ n_2 &\in \text{succ}(u) \\ n_{x-1} &\in \text{pred}(v) \\ \text{succ}(n_i) &= \{n_{i+1}\} \quad \forall 2 \leq i \leq x-1 \\ \text{pred}(n_i) &= \{n_{i-1}\} \quad \forall 2 \leq i \leq x-1 \end{aligned}$$

$\mathfrak{C}(u,v)$ designates the set of all chains starting at u and ending at v.

Using this definition, the chain combination is given in algorithm 6:

Algorithm 6 Chain combination

```
1: while ∃(u, v) ∈ V × V : |𝔠(u, v)| > 1 do
2:     m = ()                ▷ compute the uv-chain m with the maximum cost
3:     for all k ∈ 𝔠(u, v) do
4:         if Σ_{1≤i≤|k|−1} C((k_i, k_{i+1})) > Σ_{1≤i≤∈|m|−1} C((m_i, m_{i+1})) then
5:             m = k
6:         end if
7:     end for
8:     for all k ∈ 𝔠(u, v) do                ▷ remove all other uv-chains
9:         if k ≠ m then
10:            remove k from G
11:        end if
12:    end for
13: end while
```

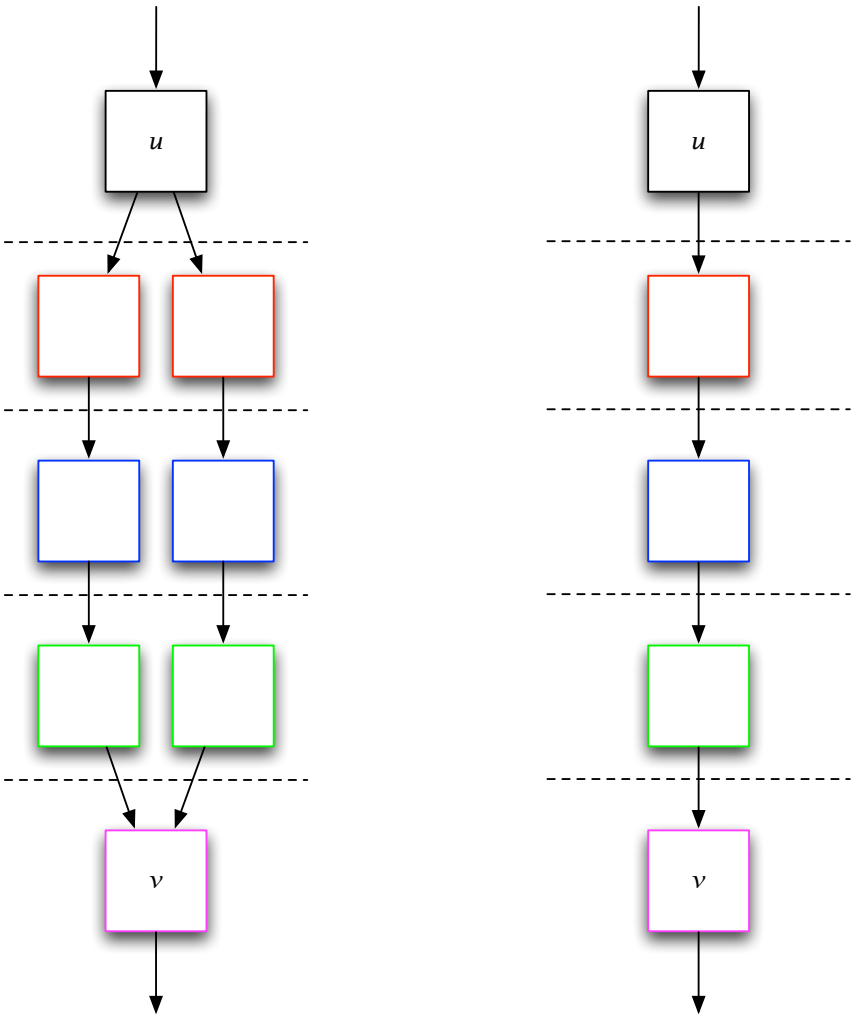
Claim. The chain combination does not alter the WCET.

Proof. Let $u, v \in V$ with $|\mathfrak{C}(u,v)| > 1$. The relevant subgraph of G is $S =$

$\{u, v\} \cup \{n \in k : k \in \mathfrak{C}(u, v)\}$.

$$
\begin{aligned}
& \text{wcet}(S^*) \\
= & \max_{e \in \text{in}(u)} C(e) + \max_{k \in \mathfrak{C}(u,v)} \left(\sum_{1 \le i \le |k|-1} C((k_i, k_{i+1})) \right) + \max_{e \in \text{out}(v)} C(e) \\
= & \text{wcet}(S'^*)
\end{aligned}
$$

□



(a) Combinable chains (b) Combined chains

Figure 6.5.: Chain combination example

6.1.7. Fixed Point

The optimization phases are executed in a loop until the abstract pipeline state graph is irreducible, i. e. until the fixed point is reached. This is because the execution of one compression algorithm may open up new opportunities

for further compression using other algorithms. For example, merging buddy nodes may create new chains that can be reduced by chain compression. Usually, this converges quite quickly. The termination is guaranteed by the fact that each step can only reduce the number of nodes and never increase it.

6.1.8. Lossy Compression

If the user is willing to exchange WCET precision for analysis time, he can use the `--lossy` switch advertised by the tool. This switch enables some optimizations that further compress the state graph but do so at the expense of precision, i. e. the predicted WCET increases. For example, the definition of *buddy nodes* in section 6.1.5 on page 58 is changed by dropping the requirement for equal edge costs. In *lossy* mode, the edges are merged by computing the maximum costs. This option is disabled by default.

6.1.9. Inter-block Compression

In addition to this, the user can get a faster result with the `--fast` option which enables optimizations that span several basic blocks. By default, compression algorithms such as the chain compression stop at the basic block boundary, so that each block is represented by at least one variable in the ILP. That way, a WCET contribution can be calculated for each block when reconstructing the critical path. If the only requirement is a figure for the global WCET and the visualization of the WCET contributions for the individual blocks may be incomplete, this option can reduce the size of the ILP and speed up the solving process.

6.2. Loop and User Constraints

Loop constraints and user constraints are computed as described in sections 4.1.4 and 4.1.6, respectively. They have in common that they operate on items of the control flow graph: loop constraints correlate loop entry edges and loop headers while user constraints are linear constraints on basic blocks. In order to represent these constraints in the ILP generated from the abstract pipeline state graph, each edge is associated with a CFG item in the prediction file (cf. table 5.2).

This association defines a reverse mapping function $m : I \times U \to \mathcal{P}(E)$ which maps pairs of CFG items (I) and contexts to sets of edges in the abstract pipeline state graph.

The simple loop constraints become:

$$\sum_{c \in \Gamma(h)} \sum_{n \in \mathrm{succ}(h)} C(m((h,n),c)) \geq \sum_{e \in \mathrm{entries}(l)} \sum_{c \in \Gamma(e)} n_{\min}(e,c) \cdot C(m(e,c))$$
$$\sum_{c \in \Gamma(h)} \sum_{n \in \mathrm{succ}(h)} C(m((h,n),c)) \leq \sum_{e \in \mathrm{entries}(l)} \sum_{c \in \Gamma(e)} n_{\max}(e,c) \cdot C(m(e,c))$$

Time-based loop constraints become:

$$\sum_{x \in L} C(m(x)) \cdot T(x) \leq A$$

where $C : \mathcal{P}(E) \to \mathbb{N}$ is the canonical extension of C for sets, i. e.

$$C(s) \coloneqq \sum_{e \in s} C(e)$$

6.3. Predictability

It is assumed that there is a connection between the *predictability* of a hardware architecture and the compressibility of the respective abstract pipeline state graphs.

The term *predictability* is still an active research topic. The current state-of-the-art of designing predictable hardware architectures is described in [Thiele and Wilhelm, 2004, Wilhelm et al., 2009a] and [Wilhelm et al., 2009b]. Roughly speaking, good predictability of an architecture implies that a pipeline analysis has to split rarely. Inversely, a pipeline analysis for an architecture with bad predictability needs to split very often because of imprecise information to handle all possible cases.

This thesis establishes the hypothesis that the graph compression algorithms can mitigate the effects of bad predictability to some extent. This hypothesis is tested empirically in section 8.2.2.

Chapter

7 Cache Persistence Analysis

7.1. Cache Analysis

A cache analysis is necessary to provide a tight WCET estimation for systems with instruction or data caches. It would be an overly pessimistic assumption that all accesses miss the cache that would lead to a huge overestimation.

This section sums up the cache analysis as described in [Ferdinand, 1997] to understand how the new path analysis can use its results to provide an additional increase of precision not available to the other path analysis methods.

A *cache* can be characterized by three major parameters:

- *capacity* is the number of bytes it may contain
- *line size* is the memory quantum in bytes that is transferred from memory to the cache in one transfer. The cache can hold at most $n = capacity / linesize$ lines.
- *associativity* is the number of cache locations where a particular line may reside. $n / associativity$ is the number of *sets* of a cache.

If a line can reside in any cache location, then the cache is called *fully associa-*

tive. If a line can reside in exactly one location, then it is called *direct mapped*. If a line can reside in exactly A locations, then the cache is called *A-way set associative*.

In the case of an associative cache, a cache line has to be selected for replacement when the cache is full and the processor requests further data. This is done according to a replacement policy. The following description assumes a *LRU* (Least Recently Used) policy.

The domain for the abstract interpretation consists of *abstract cache states*. In the following, a cache is a set of cache lines $L = \{l_1, \ldots, l_n\}$ and $S = \{s_1, \ldots, s_m\}$ denotes a set of memory blocks.

Definition 7.1.1 (Concrete Cache State)**.** A *concrete cache state* is a function $c : L \to S$. C denotes the set of all concrete cache states.

Definition 7.1.2 (Abstract Cache State)**.** An *abstract cache state* $\tilde{c} : L \to \mathcal{P}(S)$ maps cache lines to sets of memory blocks. $\tilde{C}$ denotes the set of all abstract cache states.

The update function for abstract cache states is depicted in figure 7.1.

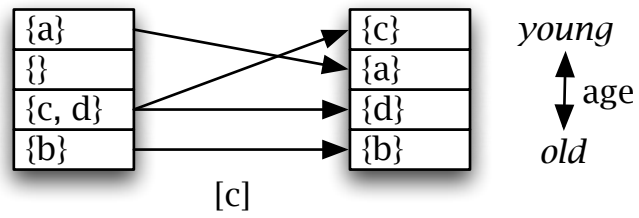


Figure 7.1.: Update of an abstract fully associative cache

7.1.1. Must Analysis

The must analysis determines a set of memory blocks that must be in the cache at a given program point upon any execution. It uses abstract cache states where the positions of the memory blocks in the abstract cache state are *upper* bounds of the ages of the memory blocks in the concrete states it represents.

The join function which combines the information from different control flow paths is similar to set *intersection*. Two abstract cache states are combined by keeping only those memory blocks which are contained in both states and assigning them the *oldest* of the two ages (see figure 7.2).

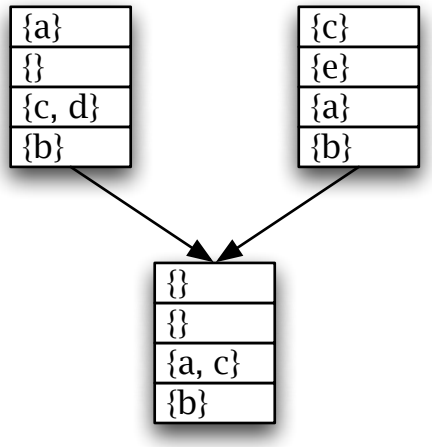


Figure 7.2.: Join function for the must analysis

7.1.2. May Analysis

The may analysis determines all memory blocks that may be in the cache at a given program point. It can be used to guarantee the absence of a memory block in the cache. It uses abstract cache states where the positions of the memory blocks in the abstract cache state are *lower* bounds of the ages of the

memory blocks in the concrete states it represents.

The join function is similar to set *union*. Two abstract cache states are combined by merging the memory blocks from both states and assigning them the *youngest* of the two ages (see figure 7.3).

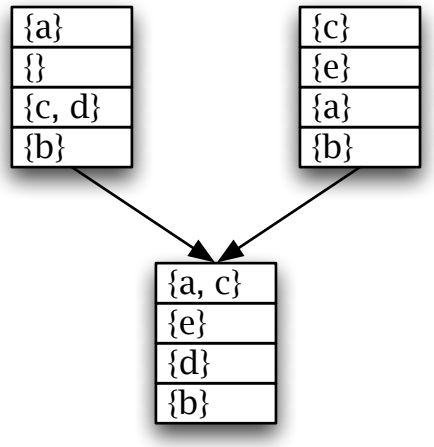


Figure 7.3.: Join function for the may analysis

7.1.3. Persistence Analysis

Cache persistence analysis is a way to improve the precision of the must and may based cache analysis. Its goal is to determine the *persistence* of a cache line, i. e., the *absence* of the possibility that a cache line l is removed from the cache. If there is no possibility to remove l from the cache, then the first access to l may result in a cache miss, but *all* further accesses to l are cache hits.

To exemplify this, consider a small loop containing conditional code (cf. figure 7.4). The may and must cache analyses cannot classify the access to the conditional code as a sure hit or sure miss and therefore the pipeline analysis splits in each iteration of the loop since it may happen that the conditional

code is executed for the first time in this iteration. The path analysis then computes a longest path which may contain several cache misses. The cache persistence analysis provides the additional information that the conditional code cannot be replaced in the cache during the execution of the loop after it was loaded the first time. Using this information the WCET analysis can conclude that only the first execution of the conditional code can be a cache miss, whereas all other executions will be cache hits.

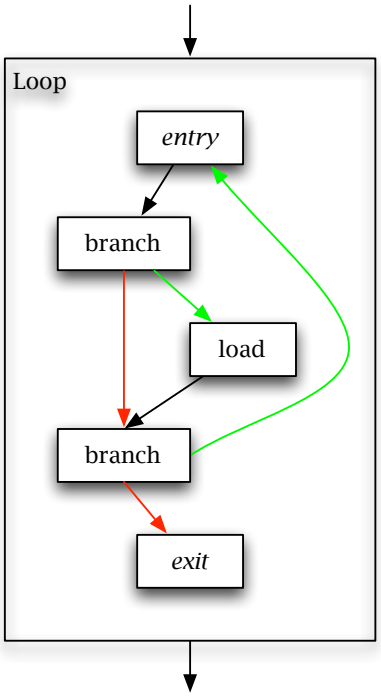


Figure 7.4.: Code that benefits from the cache persistence analysis

An abstract cache state for the persistence analysis is a combination of the states for the must and may analyses: the positions of the memory blocks are *upper* bounds of the ages of the memory blocks (like *must*) and the join function is similar to set *union* (like *may*). In addition to the memory blocks which may be in the cache, the abstract cache state also collects all memory blocks which may have been evicted from the cache in a special line $l_\perp$, i. e. whenever a block is about to be evicted from the abstract cache, it is added to

$l_\perp$. This special line is used when classifying an access to a cache line l: the access cannot be persistent if $l \in l_\perp$.

7.2. Precise Use of Cache Persistence Analysis

Until now, the results of a persistence analysis could not be directly used in the pipeline analysis of aiT because the persistence analysis provides information for a set of accesses to a memory location, but not for each access as usually required by aiT. The new path analysis enables a precise use of the results of the persistence analysis in a safe way.

With the traditional ILP-based path analysis, the persistence information is not usable because it is lost after the pipeline analysis annotates each basic block with the computed WCET. Using the pipeline state graph from the prediction file, it is now possible to add additional constraints to the ILP expressing that only one execution of the conditional code can be a cache miss.

The prediction file format has been extended with two new events. When the cache analysis classifies a cache access as persistent, the pipeline analysis splits the current pipeline state into two: one state for the cache hit, one for the cache miss. The edge leading to the cache miss is annotated with an “m” event, the edge leading to the cache hit is annotated with an “n” event. The events carry two additional parameters, the scope identifier (called `scope_id`) and the cache line.

cache_event	→	*pers_event scope_id* , *cache_line*
pers_event	→	`m` | `n`

Table 7.1.: Extended prediction file syntax for persistence events

7.3. Automatic Persistence Scopes

The persistence analysis is performed locally inside given *scopes*. This is because very few cache lines would be classified as persistent if the analysis is performed for the whole program. Therefore, the persistence analysis can be be restricted to smaller program snippets, where the likelihood of a cache line being evicted by later cache allocations is small. Generally speaking, the probability of a cache access being classified as persistent is higher if the snippet is smaller.

The user can add persistence scopes manually via annotations when spotting a program snippet which is likely to benefit from a persistence scope. Nonetheless, this can be difficult to get right or to be exhaustive. To alleviate this problem, a pre-analysis is performed when the persistence analysis is enabled. This pre-analysis implements a heuristic which tries to guess suitable persistence scopes by marking routines that

1. are a loop, and
2. do not call routines that belong to another persistence scope, and
3. have at least two different paths containing at least one load instruction or call a sub-routine that fulfills this criterion

7.4. Persistence Constraints

Definition 7.4.1 (Persistence identifier)**.** Let s be a persistence scope and l a cache line, then the tuple (s, l) is called a *persistence identifier*.

Let P be the set of persistence identifiers, E the set of edges. Then, a new

constraint is generated for each $i \in P$

$$\sum_{e \in M_i} c(e) \leq 1$$

where $c(e)$ denotes the execution count of an edge $e \in E$, M_i the set of edges with "m" events and persistence identifier i.

The state graph compression algorithms pay special attention not to remove any node which is part of such a persistence constraint.

Note that these constraints allow at most one cache miss for each persistence identifier. Ideally, they would only allow the first access to be a cache miss, however this cannot easily be expressed as a linear constraint.

7.5. Generalization

The precise use of the persistence analysis is just one instance of a whole class of problems which can be solved with the ILP-based path analysis on abstract pipeline state graphs.

The generalization is a hardware event for which the micro-architectural analysis can compute a set E of locations where that event might possibly occur, but only n of the $m = |E|$ events are actually feasible. The general constraints have the form

$$\sum_{e \in E} c(e) \leq n$$

For the persistence analysis, the event is a persistent miss with persistence identifier i, $E = M_i$ and $n = 1$. Other cases which might fall into this problem

class are:

- non-LRU caches
- TLB misses
- writebacks

Definition 7.5.1 (Timing Anomaly)**.** Intuitively, a timing anomaly is a situation where the local worst-case does not entail the global worst-case. For instance, a cache miss—the local worst-case—may result in a shorter execution time, than a cache hit, because of scheduling effects. For a formal definition of a *timing anomaly*, see [Reineke et al., 2006].

Definition 7.5.2 (Compositional Architecture)**.** If the absence of timing anomalies can be proven for a given hardware architecture, it is called a *compositional architecture*.

On compositional architectures, TLB misses and writebacks can simply be counted whenever they might occur in the program, and if a penalty time for a single miss/writeback can be quantified in processor cycles, the WCET can just be incremented by the product of the event count and the penalty cycles. However, this simple counting method is not safe for architectures with timing anomalies. In order to safely bound the number of these events on non-compositional architectures, they can be incorporated into ILP constraints similar to the cache persistence analysis results.

To precisely predict memory writebacks, for example, one could imagine an enhanced cache analysis which calculates a set of possibly evicted memory blocks for each cache access. The cache analysis would need to know both the minimum and maximum ages of the memory blocks to identify a range of accesses which cause the replacement of the same cache item.

Consider an abstract cache state containing a memory block m with an age in

[*associativity* − 3, *associativity* − 1] followed by 3 accesses to the same cache set (all misses). m is either evicted by the first, second or third access, but only exactly once. This information could be used to generate an ILP constraint as described above.

Chapter

8 Implementation and Evaluation

This chapter first describes the implementation of the ILP-based path analysis on abstract pipeline state graphs and then goes on to evaluate several aspects of the analysis, such as runtime performance, precision and ILP solvers.

8.1. Implementation

Here is a list of programs and libraries that were implemented as part of this thesis.

`predpathan` The path analysis on abstract pipeline state graphs is implemented in a tool called `predpathan`. The total implementation consists of approximately 7800 lines of code.

`libpredfile` This library encapsulates the reading and writing of prediction files. It it a shared component of the cache/pipeline analysis, `predpathan`, a trace validation tool and other programs operating on prediction files. The implementation totals approximately 3000 lines of code.

`predsolve2crl2` This is the tool which handles the visualization of path analysis results. A description follows in section 8.1.5 on page 85. The implementation totals approximately 1400 lines of code.

The programming language used to develop the tools and libraries is C++ [International Organization for Standardization, 2003]. They are documented throughout with DOXYGEN [van Heesch, 2007] and can be compiled using the GNU Compiler Collection [FSF (Free Software Foundation), 2005] or Microsoft VisualC++. The resulting binaries are tested on GNU/Linux, Microsoft Windows and MacOS X.

8.1.1. Platforms

The `predpathan` analysis has been integrated into the aiT WCET analysis framework as a mostly platform-independent module. At the time of this writing, it supports the analysis of the following hardware platforms:

1. AMD Am486 DX4
2. ARM7
3. Texas Instruments TMS320VC33
4. HC11
5. HCS12 (STAR12)
6. i386
7. LEON2
8. LEON3
9. Renesas M32C
10. Motorola M68020
11. PowerPC MPC55xx
12. PowerPC MPC5xx
13. PowerPC MPC603e
14. PowerPC MPC7448
15. PowerPC MPC755
16. PowerPC PPC750
17. Infineon TriCore (TC1766, TC1796 and TC1797)
18. Infineon PCP2 (TriCore Peripheral Control Processor)
19. V850

8.1.2. Prediction File Library

Because prediction files (also called *TRC files* for historical reasons) tend to be quite big, they are compressed on-the-fly using the well-known *zlib* library [Roelofs and Gailly, 2010]. Their textual representation is more or less human readable and contains a fair bit of redundancy which allows for good compression factors (cf. table 8.1).

File	Uncompressed [bytes]	Compressed [bytes]	Reduction
dcbf	27 759	7791	72 %
do_char_008	49 078	12 504	75 %
dry2_1	406 807	116 443	71 %
edn	54 459 951	8 864 774	84 %
loop3	7 377 700	1 817 575	75 %
minmax	461 981	129 561	72 %
morswi	3 519 312	561 813	84 %

Table 8.1.: Prediction file compression

8.1.3. ILP Solvers

Solving integer linear programs is an $\mathcal{NP}$-hard problem. There are many different ILP solvers available which all implement different sets of heuristics to speed up the solving process. Because of that, one solver may solve a particular problem very quickly but might have problems with others. Thus, aiT offers the user the choice of a set of ILP solvers:

lp_solve

lp_solve [lp_solve, 2008] is a free (LGPL) linear (integer) programming solver based on the revised simplex method and the branch-and-bound method for the integers. It was originally developed by Michel Berkelaar at Eindhoven University of Technology and is now maintained by the new developers Kjell Eikland and Peter Notebaert.

CLP+CBC

CLP [CLP, 2009] is a high quality open-source LP solver and is available under the Common Public License (CPL) 1.0. Its main strengths are its Dual and Primal Simplex algorithms. It also has a barrier algorithm for Linear and Quadratic objectives. The branch-and-bound algorithm is implemented in the *CBC* [CBC, 2009] part. Both are sub-projects of *COIN-OR* [COIN-OR, 2009], the Computational Infrastructure for Operations Research.

GLPK

The *GLPK* [GLPK, 2009] (GNU Linear Programming Kit) is a callable library for solving large-scale linear programming (LP), mixed integer programming (MIP), and other related problems. It is available under the GNU Public License (GPL).

CPLEX

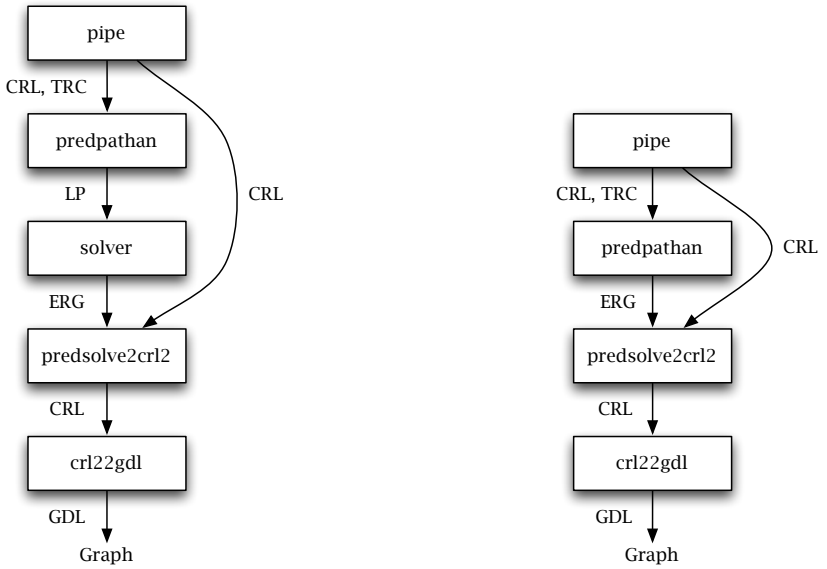
CPLEX [CPLEX, 2008] is a commercial mixed integer optimizer by ILOG S.A. which employs state-of-the-art algorithms and techniques to solve difficult

mixed integer programs, including problems with quadratic terms in the objective function and/or constraints.

The `predpathan` tool has C++ interfaces for lp_solve and CPLEX to pass the ILP in-memory to the solver libraries. However, it can also write LP files in formats suitable for the respective command line tools (cf. table 8.2). See [LPFF, 2008] and [ILOG S. A., and ILOG, Inc., 2006] for a comparison of these file formats. Figure 8.1 shows how the in-memory toolchain differs from the standard toolchain using LP files.

Solver	In-Memory	File
lp_solve	✓	✓
CLP+CBC	✗	✓
GLPK	✗	✓
CPLEX	✓	✓

Table 8.2.: ILP solvers and their interfaces to `predpathan`



(a) external ILP solver (b) internal ILP solver

Figure 8.1.: `predpathan` toolchain

8.1.4. ILP Solver Optimization

`predpathan` generates ILPs which are always of the same type. This fact can be used to provide hints to the ILP solver. The solvers must be able to solve any generic integer linear program but might perform better if they know some properties of the program. For example, the `predpathan`-generated programs benefit from a presolving phase in which the solver pre-processes the program to simplify it before feeding it to the actual solver.

The following settings empirically proved to be advantageous for the lp_solve solver (the descriptions of the options are taken from the lp_solve reference guide):

PRESOLVE_ROWS Presolve rows.

PRESOLVE_LINDEP Eliminate linearly dependent rows.

PRESOLVE_REDUCEGCD Reduce (tighten) coefficients in mixed integer models based on greatest common divisor (GCD) argument.

PRESOLVE_ROWDOMINATE Idenfify and delete qualifying constraints that are dominated by others, also fixes variables at a bound.

PRESOLVE_COLDOMINATE Deletes variables (mainly binary), that are dominated by others (only one can be non-zero).

PRESOLVE_IMPLIEDSLK Converts qualifying equalities to inequalities by converting a column singleton variable to slack. The routine also detects implicit duplicate slacks from inequality constraints, fixes and removes the redundant variable. This latter removal also tends to reduce the risk of degeneracy. The combined function of this option can have a dramatic simplifying effect on some models.

PRESOLVE_COLFIXDUAL Variable fixing and removal based on considering signs of the associated dual constraint.

PRESOLVE_BOUNDS Does bound tightening based on full-row constraint information. This can assist in tightening the objective function bound, eliminate variables and constraints. At the end of presolve, it is checked if any variables can be deemed free, thereby reducing any chance that degeneracy is introduced via this presolve option.

Furthermore, it turned out to be favorable to disable the scaling algorithm and to use the Devex pricing [Harris, 1973] as the pivot rule.

The commercial CPLEX solver has a much more sophisticated auto-detection for problem properties and finds the best settings automatically. However, `predpathan` indicates a MIP emphasis so that CPLEX's MIP optimizer emphasizes optimality over feasibility. This is because `predpathan` requires the solution to be optimal, a feasible but sub-optimal solution is not necessarily an upper bound for the worst-case execution time.

The same settings can also be used for `pathan`-generated ILPs which have basically the same structure but are naturally much smaller.

8.1.5. Visualization

After the ILP is solved, its results need to be mapped back to the control flow graph, i. e. the calculated critical path (the path which leads to the WCET prediction) is visualized so that the user can examine it. In case the prediction shows that the allotted time limit might be exceeded, the visualization is instrumental in finding the program points which contribute the most to the overall WCET.

The tool which implements the integration of the ILP results into the CRL2 graph is called `predsolve2crl2`. Figure 8.1 shows its place in the `predpathan` toolchain. The inputs for this phase are the CRL2 graph and the optimal ILP solution in the ERG format. An example for an ERG file is given in A.3.1.

The ERG file contains the value of the objective function along with the values (execution counts) for the ILP variables (representing edges). In order to map ILP variables to CFG items, `predpathan` uses the following scheme for the variable names:

`e_`*source_target_item_context_time*

During parsing the ERG file, `predsolve2crl2` decomposes the variable names into the following components:

source the source node of this edge in the abstract pipeline state graph.

target the target node of this edge in the abstract pipeline state graph.

item the corresponding item (node or edge) in the control flow graph.

context the context this edge belongs to.

time the cost of this edge in processor cycles.

Using this information, the edge costs are assigned to CFG items. As a next step, `predsolve2crl2` computes the cumulative WCET contributions for each routine, i.e. the contribution of each routine including the subroutines it calls.

The resulting graph annotated with the critical path and the cumulative WCET information is transformed into the Graph Description Language (GDL, [GDL, 2010]) with the help of the tool `crl22gdl`. The GDL graph can be viewed with `aiSee` [aiSee, 2010] or similar graph visualization programs.

8.1.6. Memory Usage

`predpathan` compresses the abstract pipeline state graph while reading the prediction file in order to keep the memory usage low. It uses two separate

graph structures: a local graph to store the nodes belonging to the current basic block in the current context and a global graph which keeps the compressed nodes. When an end-of-block marker is encountered in the prediction file, the chain compression and the basic block compression are applied on the local graph. Afterwards, the compressed local graph is merged into the global graph and the local graph is cleared.

Therefore, the memory usage increases continuously when reading the prediction file, but the maximum usage is usually only the sum of the size of the largest basic block and the *compressed* graph preceding this block (cf. figure 8.2 on the following page). It would be much higher if the *uncompressed* abstract pipeline state graph was completely read into memory before applying the compression algorithms. Uncompressed graphs can be so large that they do not fit into the main memory of standard PCs anyway, so this process only now enables the analysis in these cases.

The remaining compression methods are executed after the last block has been read because they require a complete graph. For example, infeasible nodes can only be removed in a complete graph because this algorithm would otherwise remove the nodes which do not have any successor in the current local graph, if the successors appear later in the prediction file.

Compression method	Scope
Chain compression	local + global
Basic block compression	local
Buddy nodes	global
Infeasible nodes	global
Chain combination	global
ϵ-transition elimination	global

Table 8.3.: Local vs. global graph compression

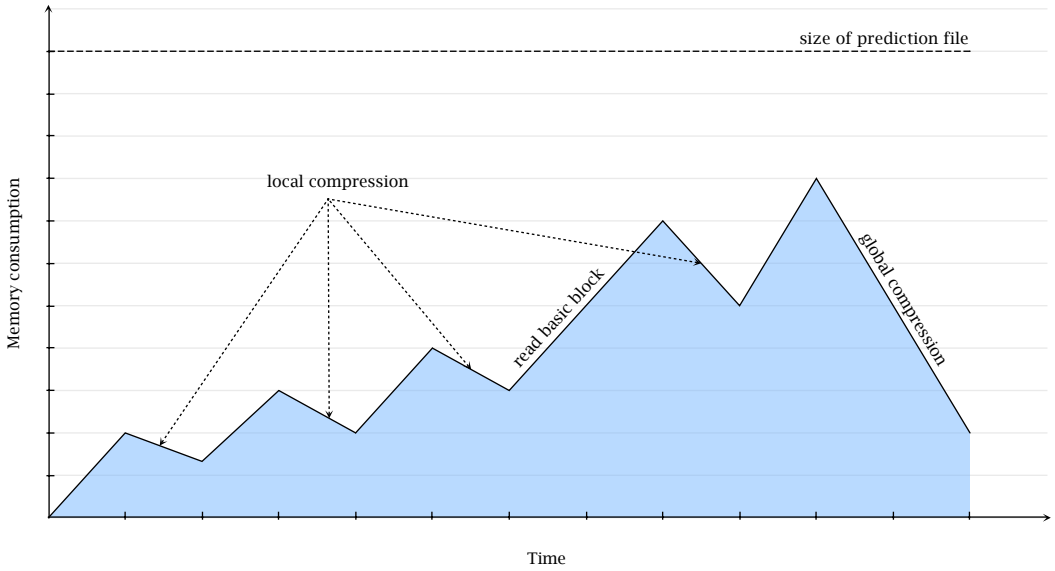


Figure 8.2.: Schematic graph of the memory consumption while reading a prediction file

8.2. Evaluation

This section quantifies the gain in precision offered by the ILP-based path analysis on abstract pipeline state graphs compared to the other path analysis methods. It also evaluates the effectiveness of the different graph compression algorithms and the graph compression as a whole. Furthermore, it examines the complexity of the generated ILP and of the particular constraint classes. Last but not least, it proves the advantages of the cache persistence analysis for some example programs and compares the features of all three presented path analyses.

8.2.1. Precision

Table 8.4 compares the precision of the different path analysis methods presented in chapters 4, 5 and 6. The benchmark is the IOM application which

includes user-added flow constraints.

Path Analysis Variant	WCET [cycles]	Relative	Time
predpathan (lossy, with flow constraints)	1 025 993	158 %	4.4 h
predpathan (without flow constraints)	790 196	122 %	42.9 h
predan	790 196	122 %	8.5 min
pathan (with flow constraints)	770 931	119 %	36.6 min
predpathan (fast, with flow constraints)	648 950	100 %	30.5 h
predpathan (with flow constraints)	648 950	100 %	38.0 h

Table 8.4.: Results for the different path analysis variants for the IOM application

8.2.2. Graph Compression

One design goal of `predpathan` was to reduce the abstract pipeline state graph so that the resulting ILP becomes small enough to be solvable in a reasonable amount of time. Table 8.5 shows the results of the algorithms presented in section 6.1 in *non-lossy* mode for a set of example programs.

The compression works best for the edn example which has very long basic blocks containing arithmetic instructions, so that the chain compression and basic block compression methods are able to reduce the graph by a large margin (cf. table 8.7 on page 92). In contrast to that, the minmax program has short basic blocks and many calls/branches so that it cannot be compressed as well.

All tests above were performed with the pipeline analysis for the PowerPC MPC755. Table 8.6 on the following page compares the graph compression results for a number of hardware architectures.

Program	Uncompressed		Compressed		Ratio[1]
	Nodes	Edges	Nodes	Edges	
minmax	14 792	22 259	1719	3290	86 %
morswi	52 230	66 940	4556	5170	92 %
drhystone	10 167	13 467	1244	1669	88 %
prime	39 599	56 473	2842	4181	93 %
fac	3129	4802	247	397	92 %
edn	753 507	1 223 574	9746	18 892	99 %

Table 8.5.: Results of the graph compression for several example programs

[1] combined ratio for nodes and edges

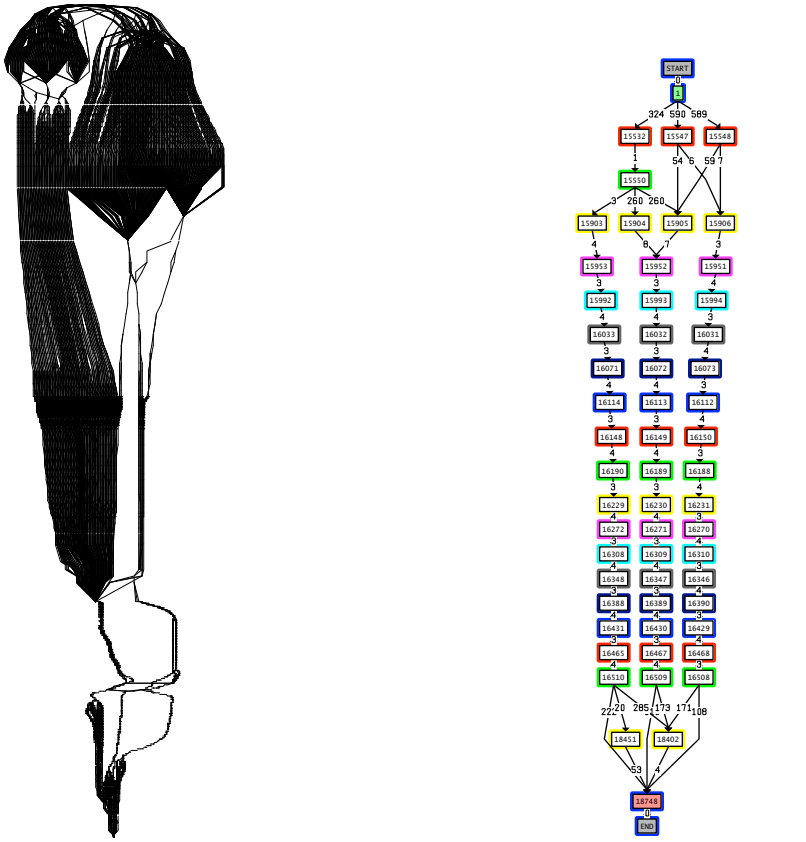
Processor	Compiler	Compression Ratio	
		drhystone	minmax
PowerPC MPC755	DiabData 5.3.1.0	88 %	86 %
HC11	Cosmic	64 %	67 %
i386	Intel	98 %	97 %
LEON3	GCC 3.4.4	97 %	95 %
Renesas M32C	IAR 2.11a	62 %	54 %
Motorola M68020	GCC 4.2.1	59 %	61 %
TriCore TC1797	Hightec 3.4.5.1	81 %	76 %
V850E1F	Greenhills	67 %	56 %

Table 8.6.: Graph compression comparison for various hardware architectures

As can be seen in table 8.6, the compression works better on architectures which are supposed to have “bad predictability”: MPC755 (complex pipeline, instruction and data caches), i386 (decoupled instruction fetch and decode), LEON3 (caches with valid bits for cache lines) and TriCore TC1797 (complex memory hierarchy, instruction and data caches) show the highest compression ratios. The simpler architectures with “good predictability” have lower ratios:

HC11 (no cache, compositional), Renesas M32C (no cache), M68020 (instruction cache only) and V850E1F (no cache) are all below the 70 % mark.

Figure 8.3 shows the abstract pipeline state graphs for a single example before and after compression.



(a) Uncompressed (36.595 states) (b) Compressed (59 states)

Figure 8.3.: Abstract pipeline state graph of `do_char_008` before and after compression

Table 8.7 on the next page shows the contribution of each compression algorithm to the overall graph compression and the number of rounds needed to reach the fixed point.

	minmax	morswi	drhystone	prime	fac	edn	∅
Chain compression	34.16%	9.38%	65.48%	59.94%	36.70%	43.71%	*41.56%*
Basic block compression	31.32%	4.90%	19.15%	13.24%	23.52%	46.06%	*23.03%*
Buddy nodes	31.82%	2.08%	11.99%	24.92%	29.44%	9.72%	*18.33%*
Infeasible nodes	1.79%	83.62%	1.26%	0.24%	4.42%	0.11%	*15.24%*
Chain combination	0.21%	0.01%	2.10%	1.66%	5.68%	0.40%	*1.68%*
ϵ-transition elimination	0.71%	0.01%	0.03%	0.00%	0.24%	0.00%	*0.16%*
Fixed point	4	11	9	7	8	135	*29*

Table 8.7.: Breakdown of the graph compression by algorithm

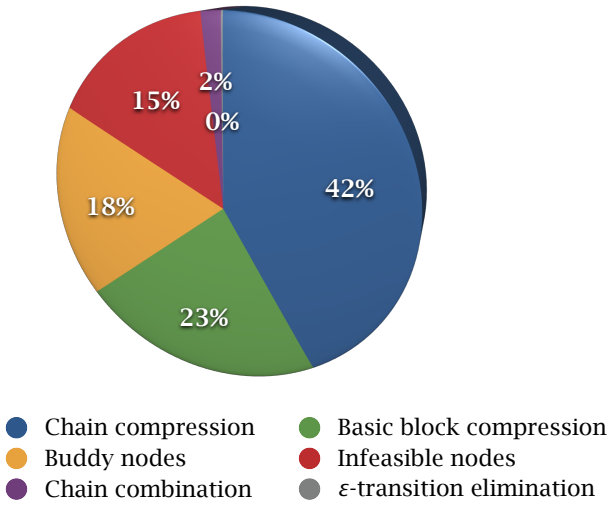


Figure 8.4.: Average contribution of each compression algorithm

As can be seen in figure 8.4, the local compression methods (chain and basic block compression) account for the majority of the graph reduction, but the

global methods also contribute significantly to a further refinement of the pipeline stage graph.

8.2.3. ILP Complexity

The different constraint types have a different impact on the time needed to solve the ILP. Although it is impossible to say exactly how a single constraint affects the solving time, given the heuristics which differ between the various solver implementations, the following is a rough complexity estimate for the constraint types (ordered from *cheap* to *expensive*):

1. Structural constraints: if the ILP only consists of structural constraints, it is a *maximum-cost network flow program*. This type of problem can be solved efficiently as it does not allow for many alternatives during the branch-and-bound process. The structural constraints are therefore the cheapest constraint class.

2. Loop constraints: if the loop bodies are small, loop constraints have a quite local effect, i. e. the corresponding nodes are close to each other in terms of the length of the paths between them. As the loop constraints are confined to a single routine (the loop routine), the influence on the solving time is usually small.

3. Persistence constraints: the cost of a persistence constraint depends on the number of accesses inside the persistence scope. As a rule of thumb, a persistence constraint is cheap, if its persistence scope is small.

4. Time-based loop constraints: a constraint of this class replicates a part of the objective function, which is generally considered bad for the solver. Again, the dimension of the adverse effect on the solving time depends on the size of the loop body (including its children in the call graph, i. e. subroutines).

5. User constraints: flow constraints can span multiple routines and have the largest potential to have non-local effects. They might increase the solving time by an order of magnitude because they increase the search-space for the branch-and-bound step.

8.2.4. ILP Solver Comparison

Table 8.8 on the next page is a performance comparison of the `predpathan` toolchains (cf. figure 8.1 on page 83) using different ILP solvers. The three test programs are all parts of the IOM application:

IOM1 small code snippet consisting of 1 routine, 4 basic blocks and 37 machine instructions. The prediction file contains 3255 items. The generated ILP has 70 variables and 31 constraints.

IOM2 medium-sized code snippet consisting of 14 routines, 86 basic blocks, 2 loops and 412 machine instructions. The prediction file contains 398 297 items. The generated ILP has 18 744 variables and 6482 constraints.

IOM3 large code snippet consisting of 25 routines, 338 basic blocks, 9 loops and 1584 machine instructions. The prediction file contains 24 464 997 items. The generated ILP has 1 418 138 variables and 869 462 constraints.

The tests were performed on an Intel®Core™2 Duo E8400 clocked at 3.0 GHz with 8 GB of main memory.

Solver[1]	IOM1	IOM2	IOM3
glpsolve	0.56 s	4.20 s	33.76 h
clpsolve	0.56 s	1.39 s	29.57 h
CPLEX (external)	0.56 s	1.45 s	41.04 min
CPLEX (in-memory)	0.56 s	1.33 s	39.89 min
none	0.55 s	0.92 s	2.62 min

Table 8.8.: ILP solver performance comparison

[1] for external solvers, the given time includes the time needed to generate and write the ILP file with `predpathan`

Table 8.8 illustrates that the solvers don't scale equally well with the problem size. For small problems, there is virtually no difference between all solvers. For large problems, however, the well-engineered heuristics of CPLEX outperform all alternatives.

8.2.5. Cache Persistence Analysis

Table 8.9 shows the effectiveness of the cache persistence analysis in combination with `predpathan`.

Program	w/o Persistence [cycles]	w/ Persistence [cycles]	Improvement
simple	5990	5612	6.31 %
FCGU	23 016	21 483	6.66 %
IOM	707 483	647 386	8.49 %

Table 8.9.: Results of the cache persistence analysis

8.2.6. Features

The final feature matrix for the three path analysis variants can be concluded from the above results and is given by table 8.10.

Variant	Scope	User con.	Persis-tence	Loops	Busy waiting	Precision	Speed
`pathan`	block	✓	✗	✓	✓	low	slow
`predan`	global	✗	✗	✗	✗	high	fast
`predpathan`	global	✓	✓	✓	✓	highest	slow

Table 8.10.: Feature matrix of the different path analysis variants

Chapter
9 Outlook

The following chapter presents several ideas how the `predpathan` technology might be adapted for further use-cases and how it might be improved in the future.

9.1. SQL-based Node Storage

`predpathan` is already pretty smart about keeping graphs in memory—it only stores the compressed graph and allocates enough memory to hold the uncompressed graph for the largest basic block. However, there are micro-architectures with bad predictability where a static pipeline analysis needs to split very often and generates a huge state graph for certain input programs. In this case, the tool needs to page out parts of the graph to a mass-storage medium because it cannot keep all nodes in-memory. In order to achieve this, we have experimented with an SQLite-based node storage.

SQLite describes itself as “a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.” [SQlite, 2010]

A simplified SQL schema to store the abstract pipeline state graph is pictured in figure 9.1. It proved too expensive to load each node from the database

when it is needed for the graph compression algorithms and to write it back after modification. Therefore, the algorithms were broken down into several larger operations which have been formulated directly as SQL queries to reduce the number of transfers from/to the database.

It remains to be seen if this storage backend can be optimized enough to be competitive with the standard backend. Although the SQLite database was configured as an in-process, in-memory database which only swaps to disk when the graph grows very large, its performance was only sufficient for small examples.

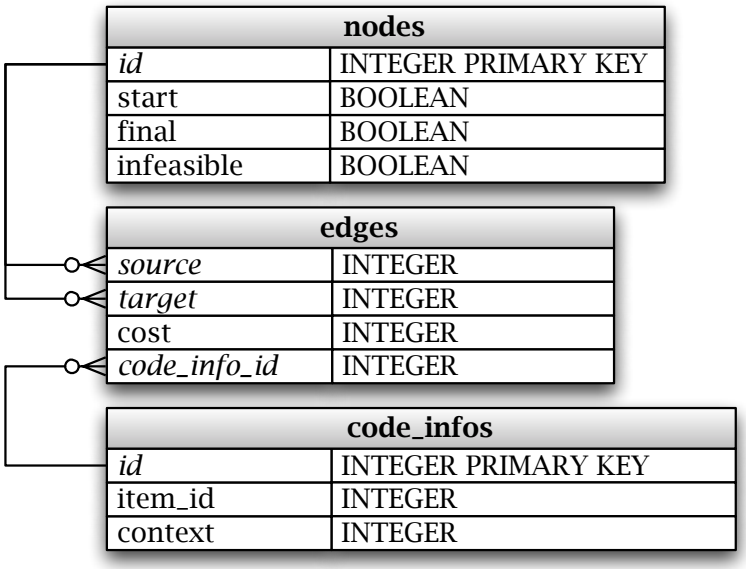


Figure 9.1.: SQL schema

9.2. More Architectures

In its current state, `predpathan` already supports a long list of hardware platforms. This ever-expanding list will be extended by even more architectures in the near future, e. g. with the Infineon C16x/ST10.

9.3. More Constraints

As described in section 7.5 on page 76, it is possible to use the ILP-based path analysis on abstract pipeline states to improve the analysis of certain hardware events. As future work, one could explore which events also fall into this class in addition to the ones already mentioned.

9.4. Parallelization

A means to reduce the analysis runtime is to take advantage of multi-core processors to speed up `predpathan`. In order to benefit from the parallelism, the analysis needs to be broken down into smaller subproblems which can be run in parallel.

A natural way to partition the graph compression is to process each basic block in a dedicated worker thread—possibly from a thread pool. Because the basic block graphs are independent from each other, a worker thread does not require additional locking to operate on the graph data. After a thread has finished the compression, it needs to acquire a lock for the global graph in order to merge it with the newly-compressed subgraph.

Further research is necessary to find out if the basic block level is a good level of granularity. For input programs with many small basic blocks, the overhead to copy the uncompressed graphs to thread-local memory, locking and the increased memory usage can be significant.

Some ILP solvers are already optimized to exploit the opportunities of shared-memory multi-core machines. For example, CPLEX is able to parallelize the process of solving nodes of the branch-and-cut tree and offers a special parallel barrier optimizer.

9.5. Detecting Timing Anomalies

Besides calculating the longest path, the ILP-based path analysis on abstract pipeline state graphs could be used to detect timing anomalies. For some split events, the pipeline analysis is able to designate one edge as the *local* worst-case. The prediction file format could be extended to mark all other edges as being a *non-local* worst-case. After solving the ILP, the evaluation component could inspect the calculated path to see if it contains any *non-local* edge. Any non-local edge which is part of the critical path indicates a timing anomaly.

9.6. Best-Case Execution Time

For various reasons, customers might also be interested in the *best*-case execution time (BCET). This extension could easily be added to `predpathan`: the optimization direction of objective function of the ILP needs to be changed from max to min and the lossy graph compression methods must be changed to compute the minimum edge costs. The loop analysis needs to compute *minimum* iteration counts and the user needs to specify minimum execution times for busy-waiting loops.

Chapter 10 Summary

The preceding chapters presented a new approach to path analysis—an important component in the WCET analysis complex. Two previously existing methods have been combined into a new algorithm which is able to replace both of them. The flexibility and precision that it offers proved to be superior to both older path analyses.

The gain in precision with the new path analysis increases the range of programs that can be analyzed with a static WCET analysis. The increased precision lowers the computed upper bound for the worst-case execution time, so that programs whose WCET prediction exceeded the allocated time slice using the older path analysis methods might now become certifiable if the more precise WCET is smaller than the hard deadline.

In those cases where the static WCET analysis yields results that are above any measured run-times, companies often resorted to measurement-based methods. Measurements are of course unsafe, because they do not offer any guarantees. Therefore, the new path analysis also increases the safety because fewer people have to rely on hardware measurements.

The implementation has successfully been deployed at several clients of AbsInt Angewandte Informatik and has been used for the certification of avionics software.

Appendix

A Examples

The following pages contain a listing of examples.

A.1. CRL2 File

Example A.1.1 (CRL2 file)**.** Listing A.1 contains an example CRL2 file (slightly edited for brevity). The corresponding control flow graph is illustrated in figure A.1 on page 108.

```
// -*- Mode: CRL -*-

crl
    specification  'f375656e-a41e-4623-aac9-b5dbb261c4bd'
    implementation '18399358-21ba-45b1-8339-33592c28f594'
    version 2 1 5 1003000 120770;

attributes global
    attribute_change_code,
    attribute_safety_code,
    clock_rate,
    compiler_name,
    decoder_name,
    input_file_name,
    mapping,
    reader_name,
    start: routine[];

attributes routine
    address,
```

```
    end: block,
    name,
    section,
    start: block,
    surface_address;

attributes block
    address,
    buddy: block,
    surface_address,
    type: enum;

attributes edge
    linear: bool,
    source: block,
    target: block,
    type: enum;

attributes instruction
    address: address<64>,
    surface_address,
    width: unsigned<64>;

attributes operation
    cat,
    conditional,
    dst,
    ext,
    genname,
    mnemonic: symbol,
    op,
    op_id,
    predicted_taken,
    src,
    target,
    type;

attributes data
    address,
    byte_order,
    executable,
    file_size,
    mem_size,
    name,
    readable,
    surface_address,
    type,
    writable;
```

```
global g1: attribute_change_code=4, attribute_safety_code=4, clock_rate=0x2625a00..0
    x2625a00, compiler_name="Tasking␣Tricore␣C/C++␣compiler␣v2.0r3", decoder_name="
    Infineon␣TriCore",
    input_file_name="main.elf",
    mapping="VIVU-4,len=inf,def_unroll=2",
    reader_name="ELF␣32", start=1*[ r0 ];

routine r0: address=0xd4000018,  name="main", section=".text.main", surface_address="
    0xd4000018" {
    pag context c5: ;
    block b0 (start): {
        edge e6 (linear) -> b2: ;
    }
    block b1 (end): ;
    block b2: address=0xd4000018, surface_address="0xd4000018" {
        edge e8 (true) -> b7: ;
        edge e9 (false, linear) -> b3: ;
        instruction i10 0xd4000018:2: surface_address="0xd4000018" {
            operation o11 "sub.a␣a10,␣8": cat=0*{}, dst=1*[ 'a10' ], ext=3*[ 2*{
                genname='AGPR', op=1*[ 'a10' ] }, 2*{ genname='AGPR', op=1*[ 'a10' ]
                 }, 2*{ genname='Const',
                    op=1*[ 8 ] } ], genname='suba', op=3*[ 'a10', 'a10', 8 ], op_id=0
                        x20, src=3*[ 1='a10', 8 ];
        }
        instruction i12 0xd400001a:2: surface_address="0xd400001a" {
            operation o13 "mov␣d4,␣d4": cat=0*{}, dst=1*[ 'd4' ], ext=2*[ 2*{ genname
                ='DGPR', op=1*[ 'd4' ] }, 2*{ genname='DGPR', op=1*[ 'd4' ] } ],
                genname='mov',
                op=2*[ 'd4', 'd4' ], op_id=2, src=2*[ 1='d4' ];
        }
        instruction i14 0xd400001c:2: surface_address="0xd400001c" {
            operation o15 "mov.aa␣a4,␣a4": cat=0*{}, dst=1*[ 'a4' ], ext=2*[ 2*{
                genname='AGPR', op=1*[ 'a4' ] }, 2*{ genname='AGPR', op=1*[ 'a4' ] }
                 ], genname='movaa',
                op=2*[ 'a4', 'a4' ], op_id=0x40, src=2*[ 1='a4' ];
        }
        instruction i16 0xd400001e:2: surface_address="0xd400001e", 4, 0x20, 0
            xd0009ff8 ] {
            operation o17 "ld.w␣d15,␣[a10]": cat=1*{ mem_read }, dst=1*[ 'd15' ], ext
                =3*[ 2*{ genname='DGPR', op=1*[ 'd15' ] }, 2*{ genname='AGPR', op
                =1*[ 'a10' ] }, 2*{ genname='Const',
                    op=1*[ 0 ] } ], genname='ldw', op=4*[ 'd15', 'a10', 0, 'Mem' ],
                        op_id=0x54, src=4*[ 1='a10', 0, 'Mem' ];
        }
        instruction i18 0xd4000020:4: surface_address="0xd4000020", 4, 0x20, 0
            xd0009ffc ] {
```

```
        operation o19 "ld.w␣d0,␣[a10]␣+4": cat=1*{ mem_read }, dst=1*[ 'd0' ],
            ext=3*[ 2*{ genname='DGPR', op=1*[ 'd0' ] }, 2*{ genname='AGPR', op
            =1*[ 'a10' ] }, 2*{ genname='Const',
                op=1*[ +4 ] } ], genname='ldw', op=4*[ 'd0', 'a10', +4, 'Mem' ],
                    op_id=0x9000009, src=4*[ 1='a10', +4, 'Mem' ];
    }
    instruction i20 0xd4000024:4: surface_address="0xd4000024" {
        operation o21 "jlt␣d0,␣d15,␣0xd4000034␣<0xd4000034>": cat=2*{ branch,
            taken }, conditional=1, ext=4*[ 1=2*{ genname='DGPR', op=1*[ 'd0' ]
            }, 2*{ genname='DGPR',
                op=1*[ 'd15' ] }, 2*{ genname='Const', op=1*[ 0xd400003a ] } ],
                    genname='j_cond', op=4*[ 'lt', 'd0', 'd15', 0xd400003a ],
                    op_id=0x3f, predicted_taken=0, src=4*[ 'lt', 'd0', 'd15', 0
                    xd4000034 ], target=0xd4000034,
            type='branch';
    }
}
block b3: address=0xd4000028, surface_address="0xd4000028" {
    edge e23 (true, linear) -> b7: ;
    instruction i24 0xd4000028:4: surface_address="0xd4000028" {
        operation o25 "lea␣a4,␣[a10]␣+0": cat=0*{}, dst=1*[ 'a4' ], ext=3*[ 2*{
            genname='AGPR', op=1*[ 'a4' ] }, 2*{ genname='AGPR', op=1*[ 'a10' ]
            }, 2*{ genname='Const', op=1*[
                    +0 ] } ], genname='lea', op=3*[ 'a4', 'a10', +0 ], op_id=0
                        xa000049, src=3*[ 1='a10', +0 ];
    }
    instruction i26 0xd400002c:4: surface_address="0xd400002c" {
        operation o27 "lea␣a5,␣[a10]␣+4": cat=0*{}, dst=1*[ 'a5' ], ext=3*[ 2*{
            genname='AGPR', op=1*[ 'a5' ] }, 2*{ genname='AGPR', op=1*[ 'a10' ]
            }, 2*{ genname='Const', op=1*[
                    +4 ] } ], genname='lea', op=3*[ 'a5', 'a10', +4 ], op_id=0
                        xa000049, src=3*[ 1='a10', +4 ];
    }
    instruction i28 0xd4000030:2: surface_address="0xd4000030" {
        operation o29 "mov.aa␣a4,␣a4": cat=0*{}, dst=1*[ 'a4' ], ext=2*[ 2*{
            genname='AGPR', op=1*[ 'a4' ] }, 2*{ genname='AGPR', op=1*[ 'a4' ] }
             ], genname='movaa',
            op=2*[ 'a4', 'a4' ], op_id=0x40, src=2*[ 1='a4' ];
    }
    instruction i30 0xd4000032:2: surface_address="0xd4000032" {
        operation o31 "mov.aa␣a5,␣a5":  cat=0*{}, dst=1*[ 'a5' ], ext=2*[ 2*{
            genname='AGPR', op=1*[ 'a5' ] }, 2*{ genname='AGPR', op=1*[ 'a5' ] }
             ], genname='movaa',
            op=2*[ 'a5', 'a5' ], op_id=0x40, src=2*[ 1='a5' ];
    }
}
block b7: address=0xd4000034, surface_address="0xd4000034" {
    edge e128 (true) -> b8: ;
```

```
        instruction i129 0xd4000034:2: surface_address="0xd4000034" {
            operation o130 "mov␣d2,␣d2": cat=0*{}, dst=1*[ 'd2' ], ext=2*[ 2*{
                genname='DGPR', op=1*[ 'd2' ] }, 2*{ genname='DGPR', op=1*[ 'd2' ] }
                 ], genname='mov',
                op=2*[ 'd2', 'd2' ], op_id=2, src=2*[ 1='d2' ];
        }
        instruction i131 0xd4000036:2: surface_address="0xd4000036" {
            operation o132 "j␣0xd4000082␣<0xd4000082>": cat=2*{ branch, taken }, ext
                =1*[ 2*{ genname='Const', op=1*[ 0xd4000082 ] } ], genname='j', op
                =1*[ 0xd4000082 ],
                op_id=0x3c, src=1*[ 0xd4000082 ], target=0xd4000082, type='branch';
        }
    }
    block b8: address=0xd4000082, surface_address="0xd4000082" {
        edge e134 (true) -> b1: ;
        instruction i135 0xd4000082:2: surface_address="0xd4000082" {
            operation o136 "ret": cat=2*{ return, taken }, genname='ret', op_id=0
                x9000, type='return';
        }
    }
}

data d308: address=0xd4000008, byte_order='x0123', executable=1, file_size=8,
    mem_size=8, name=".text.libc.csa_areas", readable=1, surface_address="0xd4000008
    ", type='code', writable=0;

end
```

Listing A.1: CRL2 description of a control flow graph

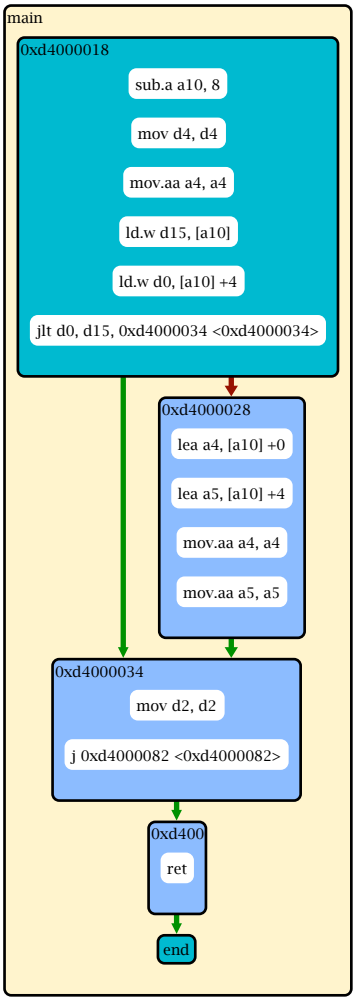


Figure A.1.: Control flow graph for listing A.1

A.2. Prediction File

Example A.2.1 (Prediction file)**.** Listing A.2 contains an excerpt from a prediction file.

```
S1
n66:0
e1d17c16i0x80000114,0x80000114,0
e17d28c10i0x80000114,0x80000118,0
e28d61c32i0x80000114,0x8000011a,0
e61d80c18i0x80000114,0x8000011e,0
e80d83c2i0x80000114,0x80000122,0
k83
N
t69:0
e83d89c5i0x80000114,0x80000124,0
k89
T
t68:0
e83d94c5i0x80000114,0x80000124,0
k94
T
B
n76:0
e94d106c12i0x80000126,0x80000126,0
e106d117c10i0x80000126,0x8000012a,0
k117
N
t80:0
e117d144c26i0x80000126,0x8000012e,0
k144
T
B
n4:0
e144d159c15i0x80000094,0x80000094,0
e159d186c26i0x80000094,0x80000098,0
e186d204c17i0x80000094,0x8000009c,0
e204d222c17i0x80000094,0x800000a0,0
e222d236c13i0x80000094,0x800000a2,0
e236d246c9i0x80000094,0x800000a4,0
e246d274c27i0x80000094,0x800000a6,0
k274
N
t6:0
e274d294c19i0x80000094,0x800000a8,0
k294
```

```
T
B
n86:0
e294d307c13i0x80000134,0x80000134,0
k307
N
t87:0
e307d311c3i0x80000134,0x80000136,0|a
k311
T
B
n90:0
e89d328c17i0x80000138,0x80000138,0
k328
N
t92:0
e328d332c3i0x80000138,0x8000013a,0
k332
T
n110:0
N
t91:0
e328d335c3i0x80000138,0x8000013a,0
k335
T
B
n95:0
e335d343c8i0x8000013c,0x8000013c,0
e343d356c12i0x8000013c,0x8000013e,0
e356d366c9i0x8000013c,0x80000140,0
k366
N
t99:0
e366d393c26i0x8000013c,0x80000142,0
k393
T
B
n18:0
e393d408c15i0x800000d4,0x800000d4,0
e408d435c26i0x800000d4,0x800000d8,0
e435d453c17i0x800000d4,0x800000dc,0
k453
N
t21:0
e453d459c5i0x800000d4,0x800000e0,0
k459
T
t20:0
```

```
e453d464c5i0x800000d4,0x800000e0,0
k464
T
B
```

Listing A.2: Representation of an abstract pipeline state graph in a prediction file

A.3. ERG File

An ERG file stores a solution for an ILP. It consists of two sections:

1. the value of the objective function
2. the ILP variables and their values in the solution

Example A.3.1 (ERG file)**.** Listing A.3 contains an example for an ERG file.

```
Value of objective function:              881
e_677_759_39_0_40                         1
e_307_311_87_0_3                          0
e_948_968_6_2_19                          1
e_117_144_80_0_26                         0
e_1300_1320_124_0_19                      1
e_328_332_92_0_3                          0
e_328_335_91_0_3                          1
e_294_307_86_0_13                         0
e_759_763_41_0_3                          0
e_759_766_40_0_3                          1
e_335_366_95_0_29                         1
e_664_677_37_0_13                         1
e_763_1050_58_0_45                        0
e_1210_1300_123_0_50                      1
e_459_759_39_0_58                         0
e_514_644_4_1_124                         1
e_1199_1206_116_0_7                       1
e_274_294_6_0_19                          0
e_311_1192_110_0_30                       0
e_89_328_90_0_17                          1
e_83_89_69_0_5                            1
e_83_94_68_0_5                            0
e_1_83_66_0_78                            1
```

```
e_393_453_18_0_58                          1
e_1196_1225_121_0_15                       0
e_1225_1300_123_0_35                       0
e_1103_1110_107_0_6                        1
e_332_1192_110_0_32                        0
e_1192_1196_112_0_3                        0
e_1192_1199_111_0_3                        1
e_766_785_45_0_18                          1
e_487_514_30_0_26                          1
e_982_1050_58_0_41                         1
e_785_818_49_0_32                          1
e_644_664_6_1_19                           1
e_464_487_26_0_22                          1
e_1050_1065_59_0_14                        1
e_453_459_21_0_5                           0
e_453_464_20_0_5                           1
e_968_982_56_0_14                          1
e_366_393_99_0_26                          1
e_818_948_4_2_124                          1
e_1206_1210_117_0_3                        1
e_1065_1103_106_0_38                       1
e_144_274_4_0_124                          0
e_94_117_76_0_22                           0
e_1110_1192_110_0_25                       1
e_18446744073709551615_1_0_4294967295_0    1
e_1320_18446744073709551614_0_4294967295_0 1
```

Listing A.3: ILP solution stored in an ERG file

A.4. GDL File

Example A.4.1 (GDL file)**.** Listing A.4 contains an example for a GDL file (slightly edited for brevity).

```
graph: {
    title: "Crl2Gdl␣Graph"
    classname 3: "Basic␣Block␣Edges"
    graph: {
    title: "r0/*main*/"
    label: "main"
    graph: {
        title: "/*r0:main*/b2/*main*/"
        label: "0xd4000018"
```

```
        info1: "Source: minmax.c:29"
        info2: "0xd4000018"
        node: {
            title: "i10/*main*/"
            label: "sub.a a10, 8"
            info1: "Source: minmax.c:29"
            info2: "0xd4000018 = virt(0xd4000018)"
        }
        node: {
            title: "i12"
            label: "mov d4, d4"
            info1: "Source: minmax.c:30"
            info2: "0xd400001a = virt(0xd400001a)"
        }
        node: {
            title: "i14"
            label: "mov.aa a4, a4"
            info1: "Source: minmax.c:30"
            info2: "0xd400001c = virt(0xd400001c)"
        }
        node: {
            title: "i16"
            label: "ld.w d15, [a10]"
            info1: "Source: minmax.c:31"
            info2: "0xd400001e = virt(0xd400001e)"
        }
        node: {
            title: "i18"
            label: "ld.w d0, [a10] +4"
            info1: "Source: minmax.c:32"
            info2: "0xd4000020 = virt(0xd4000020)"
        }
        node: {
            title: "i20"
            label: "jlt d0, d15, 0xd4000034 <0xd4000034>"
            info1: "Source: minmax.c:32"
            info2: "0xd4000024 = virt(0xd4000024)"
        }
    }
    edge: { source: "i20" target: "i34" thickness: 4 class: 3 }
    edge: { source: "i20" target: "i24" thickness: 4 class: 3 }
    graph: {
        title: "/*r0:main*/b3"
        label: "0xd4000028"
        info1: "Source: minmax.c:32"
        info2: "0xd4000028"
        node: {
            title: "i24"
```

```
            label: "lea␣a4,␣[a10]␣+0"
            info1: "Source:␣minmax.c:32"
            info2: "0xd4000028␣=␣virt(0xd4000028)"
        }
        node: {
            title: "i26"
            label: "lea␣a5,␣[a10]␣+4"
            info1: "Source:␣minmax.c:32"
            info2: "0xd400002c␣=␣virt(0xd400002c)"
        }
        node: {
            title: "i28"
            label: "mov.aa␣a4,␣a4"
            info1: "Source:␣minmax.c:32"
            info2: "0xd4000030␣=␣virt(0xd4000030)"
        }
        node: {
            title: "i30"
            label: "mov.aa␣a5,␣a5"
            info1: "Source:␣minmax.c:32"
            info2: "0xd4000032␣=␣virt(0xd4000032)"
        }
    }
    edge: { source: "i30" target: "i34" thickness: 4 class: 3 }
    graph: {
        title: "/*r0:main*/b4"
        label: "0xd4000034"
        info1: "Source:␣minmax.c:35"
        info2: "0xd4000034"
        node: {
            title: "i34"
            label: "mov␣d2,␣d2"
            info1: "Source:␣minmax.c:35"
            info2: "0xd4000034␣=␣virt(0xd4000034)"
        }
        node: {
            title: "i36"
            label: "j␣0xd4000082␣<0xd4000082>"
            info1: "Source:␣minmax.c:35"
            info2: "0xd4000036␣=␣virt(0xd4000036)"
        }
        }
        edge: { source: "i36" target: "i40" thickness: 4 class: 3 }
        graph: {
            title: "/*r0:main*/b5"
            label: "0xd4000082"
            info1: "Source:␣minmax.c:36"
            info2: "0xd4000082"
```

```
            node: {
                title: "i40"
                label: "ret"
                info1: "Source: minmax.c:36"
                info2: "0xd4000082 = virt(0xd4000082)"
            }
        }
        edge: { source: "i40" target: "/*r0:main*/b1" thickness: 4 class: 3 }
        node: {
            title: "/*r0:main*/b1"
            label: "end"
            info1: ""
            info2: "/*r0:main*/b1"
        }
    }
}
```

Listing A.4: Graph Description Language example

A.5. Abstract Pipeline State

Example A.5.1 (Abstract Pipeline State)**.** Listing A.5 contains the textual representation of an abstract pipeline state (including the abstract cache) for the Motorola MPC755.

```
Jitter: [1.5, 2.0]=0x18
Fetch and Branch Prediction Unit:
================================
State: ignore(0x194, 3)
Instruction index: 0
Prediction[0]: NONE, branch: NONE, ctx: NONE
Prediction[1]: NONE, branch: NONE, ctx: NONE
SPR knowledge: LR: 1, CTR: 1, CR: 1
external stall: 0
SpecSplitState: invalid

Dispatch Unit:
==============
Free shadow registers: (GPR: 5, FPR: 6, CTR: 1, CR: 0, LR: 1)

Completion Unit:
================
CQ: empty
```

```
Retirement Delay Count: 0

Integer Unit 1:
===============
Reservation Station: NONE
Working Stage: NONE, cycles 0

Integer Unit 2:
===============
Reservation Station: NONE
Working Stage: NONE, cycles 0

System Register Unit:
=====================
Reservation Station: NONE
Working Stage: NONE, cycles 0
EIEIO: NONE

Floating Point Unit:
====================
Reservation Station: NONE
Working Stage[0]: NONE, cycles 0
Working Stage[1]: NONE
Working Stage[2]: NONE
Pipeline is not blocked.

Load/Store Unit:
================
Reservation Station[0]: NONE
Reservation Station[1]: NONE
Effective Address Stage: NONE
Access Stage: NONE
Store Queue[0]: [NONE-NONE](0), index: NONE
Store Queue[1]: [NONE-NONE](0), index: NONE
Store Queue[2]: [NONE-NONE](0), index: NONE
State: idle
Load/Store Clash Index: 0, recheck required: yes
Number of Accesses: 0
Memory Index: 0
Store is not prioritized.
ICache Busy: 0, DCache Busy: 0

BU_IC: 0x30(2) cacheable CL: 0x30, FF: 1, FGET: 4, FGO: 1(0) BUSY=0
BU_DC: [ 0x400004, 0x400004 ] (4) write, DCLASH: 0, DGET: 1, DGO=1(0) BUSY=0
BU_ACC: [ ( SRC: Write, ADDR: [ 0x3ffff8--0x3ffff8 ] , LEN: 4, STATE: AACK, CNT: 1 ),
          ( SRC: Write, ADDR: [ 0x3ffffc--0x3ffffc ] , LEN: 4, STATE: TS, CNT: 1 )]
BU_IDATA: 0,  BU_DDATA: 0
BU_IDO: 0,  BU_DDO: 0, BU_DELAY: 1
```

```
BU_TS([0x3ffff8,0x3ffff8], RW:0[data], Len:4)
CSU/DBSM: State=NONE Addr=[NONE,NONE], Len=0..7 RW=0 Ctr=0 ta_n=1
CSU/Reg: Ctr=0
CSU/CE: AACK=PATCH, TAG_REG=[0x3fffe8,0x3fffe8] (invalid), DW:0 Purge:0
        INSERV: PATCH, TSPATCH=0, TYPATCH=00,
        eodOutPatch: 0(0), curLen=0..7, curTy=02
        aackBE_r=1(1) Reads=0
        DataSelect=PATCH, EodOccursFirst=0, AACK_EDC_slave=1
        TS_EDC_slave=1, TS_BE_W=1, TS_BE_R=1
        EDCengine=IDLE, PipedRead=1, CurAddr=[0x3fffe8,0x3fffe8]
        TyCPU=01, EODoutEDCslave=0
CSU/SD: RowOpen=[ 0x0--0x0, 0x0--0xffff, 0x0--0xffff, 0x0--0xffff ]
        PageHit=0, State=ROWACT, doActivate=0, doPrecharge=0
        TSsdramIdl=1, Cmd=DC, WaitRAC=0, stopApcb=0
        TSsdram=1, WaitRD=0, WaitWR=0, BrstCntR=0, BrstCntW=0
        singleTerm=0, TA_BE_V_O=1
CSU/PCI: R_TA=1, PPCState=IDLE, PCIState=IDLE
         Busy=0, Gnt=1, curAddr=[NONE,NONE], Write,
         brst=0, reqCnt=0, STScnt=0, TAread=1, TAwrite=1
         mrxCnt=0, mtxCnt=0, cycles=[0-0]
CSU/VHDL: GO=(S:1, P:1, R:1) AACK=(S:1, P:1, R:1) TA=(S:1, P:1, R:1)
          STOPwrite=0, BE=(TS:1, AACK=1, TA=1(1), EODIN:0)
          DS_TA:0 DS_AACK:0
NEXTBRANCH: 1
BRANCH: TFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
SPEC: (63, 63)
BRANCHES: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0]
REL_REF_ACCESS[0]: NONE
REL_REF_CONTEXT[0]: NONE
REL_REF_ACCESS[1]: NONE
REL_REF_CONTEXT[1]: NONE
PERS_ID: -1
ACT_CTX: 0
NOP_CNT: 0
NOP_SPEC_CNT: 0

Instruction Cache:
must:
  1: {{0x20}{}{}{}{}{}{}{}}
 11: {{0x160}{}{}{}{}{}{}{}}
 12: {{0x180}{}{}{}{}{}{}{}}
Data Cache:
must:
127: {{0x3fffe0}{}{}{}{}{}{}{}}
Evictions: 0
```

Listing A.5: MPC755 pipeline state

List of Tables

List of Figures

Listings

List of Algorithms

Bibliography

[AbsInt Angewandte Informatik GmbH, a] AbsInt Angewandte Informatik GmbH. AbsInt: Advanced Compiler Technology for Embedded Systems [online]. Available from: http://www.absint.com/.

[AbsInt Angewandte Informatik GmbH, b] AbsInt Angewandte Informatik GmbH. aiT: Worst-Case Execution Time Analyzers [online]. Available from: http://www.absint.de/ait/.

[AbsInt Angewandte Informatik GmbH, 2006a] AbsInt Angewandte Informatik GmbH (2006a). *Path Analysis on Prediction Files in aiT for PowerPC MPC755*.

[AbsInt Angewandte Informatik GmbH, 2006b] AbsInt Angewandte Informatik GmbH (2006b). *Worst-Case Execution Time Analyzer aiT for PowerPC MPC755 (Hurricane Chip Set)*.

[aiSee, 2010] aiSee Graph Layout Software [online]. (2010). Available from: http://www.aisee.com/.

[Allen, 1970] Allen, F. E. (1970). Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19.

[CBC, 2009] CBC [online]. (2009). Available from: http://www.coin-or.org/projects/Cbc.xml.

[Chvátal, 1983] Chvátal, V. (1983). *Linear Programming*. W. H. Freeman and Company.

[CLP, 2009] CLP [online]. (2009). Available from: `http://www.coin-or.org/projects/Clp.xml`.

[COIN-OR, 2009] COIN-OR [online]. (2009). Available from: `http://www.coin-or.org/index.html`.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixed Points. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles. ACM Press.

[Cousot and Cousot, 1992] Cousot, P. and Cousot, R. (1992). Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547.

[CPLEX, 2008] ILOG CPLEX [online]. (2008). Available from: `http://www.ilog.com/products/cplex`.

[Cullmann, 2006] Cullmann, C. (2006). Statische Berechnung sicherer Schleifengrenzen auf Maschinencode. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes, Saarbrücken. Available from: `http://babylon2k.de/publications/diplom.pdf`.

[Esterel Technologies,] Esterel Technologies. SCADE Suite – The Standard for the Development of Safety-Critical Embedded Software in the Avionics Industry [online]. Available from: `http://www.esterel-technologies.com/products/scade-suite/overview.html`.

[Ferdinand, 1997] Ferdinand, C. (1997). *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes.

[Ferdinand and Heckmann, 2008] Ferdinand, C. and Heckmann, R. (2008).

Worst-Case Execution Time – A Tool Provider's Perspective. In *11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing ISORC 2008, Orlando, Florida, USA*.

[Ferdinand et al., 1999] Ferdinand, C., Kästner, D., Langenbach, M., Martin, F., Schmidt, M., Schneider, J., Theiling, H., Thesing, S., and Wilhelm, R. (1999). Run-Time Guarantees for Real-Time Systems - The USES Approach. In *Proceedings of Informatik '99 - Arbeitstagung Programmiersprachen*, pages 410–419.

[Freescale Semiconductor, Inc., 2001] Freescale Semiconductor, Inc. (2001). *MPC750 RISC Microprocessor Family User's Manual*. Available from: `http://www.freescale.com/files/32bit/doc/ref_manual/MPC750UM.pdf`.

[Freescale Semiconductor, Inc., 2005] Freescale Semiconductor, Inc. (2005). *MPC755 RISC Microprocessor Hardware Specifications*, 7. edition. Available from: `http://www.freescale.com/files/32bit/doc/data_sheet/MPC755EC.pdf`.

[Fritz, 2001] Fritz, N. (2001). Generische Value-Analyse für Maschinenprogramme. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes.

[FSF (Free Software Foundation), 2005] FSF (Free Software Foundation). GNU Compiler Collection [online]. (1987–2005). Available from: `http://gcc.gnu.org`.

[GDL, 2010] Graph Description Language in a Nutshell [online]. (2010). Available from: `http://www.aisee.com/gdl/nutshell/`.

[GLPK, 2009] GLPK [online]. (2009). Available from: `http://www.gnu.org/software/glpk/`.

[Harris, 1973] Harris, P. M. (1973). Pivot Selection Methods of the Devex LP Code. *Mathematical Programming*, 5(1):1–28.

[ILOG S. A., and ILOG, Inc., 2006] ILOG S. A., and ILOG, Inc. (2006). *ILOG CPLEX 10.0 File Formats*. Available from: `http://www.lix.polytechnique.fr/~liberti/teaching/xct/cplex/reffileformatscplex.pdf`.

[International Organization for Standardization, 2003] International Organization for Standardization (2003). *ISO/IEC 14882:2003: Programming Languages — C++*. American National Standards Institute, International Organization for Standardization, Second edition. Available from: `http://www.iso.org/iso/catalogue_detail.htm?csnumber=38110`.

[Kästner and Wilhelm, 2002] Kästner, D. and Wilhelm, S. (2002). Generic control flow reconstruction from assembly code. In *ACM SIGPLAN Notices*, volume 37, pages 46–55.

[Li and Malik, 1995a] Li, Y.-T. S. and Malik, S. (1995a). Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*.

[Li and Malik, 1995b] Li, Y.-T. S. and Malik, S. (1995b). Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems (LCT-RTS)*, number 30(11) in SIGPLAN Notices, pages 88–98, La Jolla, California, USA. ACM Press.

[Li et al., 1995] Li, Y.-T. S., Malik, S., and Wolfe, A. (1995). Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*.

[LPFF, 2008] LP file format [online]. (2008). Available from: `http://lpsolve.sourceforge.net/5.5/lp-format.htm`.

[lp_solve, 2008] lp_solve [online]. (2008). Available from: `http://lp_solve.sourceforge.net`.

[Lucas et al., 2009] Lucas, P., Parshin, O., and Wilhelm, R. (2009). Operating

Mode Specific WCET Analysis. In Seidner, C., editor, *Proceedings of the 3rd Junior Researcher Workshop on Real-Time Computing (JRWRTC)*, pages 15–18.

[Martin, 1999] Martin, F. (1999). *Generating Program Analyzers*. PhD thesis, Universität des Saarlandes.

[Martin et al., 1998] Martin, F., Alt, M., Wilhelm, R., and Ferdinand, C. (1998). Analysis of Loops. In Koskimies, K., editor, *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94. Springer.

[Matthies, 2006] Matthies, N. (2006). Präzise Bestimmung längster Programmpfade anhand von Zustandsgraphen unter Berücksichtigung von Schleifen-Nebenbedingungen. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes.

[May et al., 1994] May, C., Silha, E., Simpson, R., and Warren, H., editors (1994). *The PowerPC Architecture – A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Second edition.

[Nemhauser and Wolsey, 1988] Nemhauser, G. and Wolsey, L. (1988). *Integer and Combinatorial Optimization*. John Wiley and Sons, New York.

[Nielson et al., 1999] Nielson, F., Nielson, H. R., and Hankin, C. L. (1999). *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Puschner and Koza, 1995] Puschner, P. and Koza, C. (1995). Computing Maximum Task Execution Times with Linear Programming Techniques. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria.

[Reineke et al., 2006] Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., and Becker, B. (2006). A Definition and

Classification of Timing Anomalies. In *Proceedings of 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Available from: http://rw4.cs.uni-saarland.de/~reineke/publications/TimingAnomaliesWCET06.pdf.

[Roelofs and Gailly, 2010] Roelofs, G. and Gailly, J.-I. zlib – A Massively Spiffy Yet Delicately Unobtrusive Compression Library [online]. (2010). Available from: http://www.zlib.net.

[Schlickling, 2005] Schlickling, M. (2005). Generisches Slicing auf Maschinencode. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes, Saarbrücken. Available from: http://rw4.cs.uni-sb.de/~schlickling/GenStaSlicing.pdf.

[Schrijver, 1996] Schrijver, A. (1996). *Theory of Linear and Integer Programming*. John Wiley and Sons.

[Sharir and Pnueli, 1981] Sharir, M. and Pnueli, A. (1981). Two Approaches to Interprocedural Data Flow Analysis. In Muchnick, S. S. and Jones, N. D., editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall.

[Sicks, 1997] Sicks, M. (1997). Adreßbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes.

[Souyris et al., 2005] Souyris, J., Pavec, E. L., Himbert, G., Jégu, V., Borios, G., and Heckmann, R. (2005). Computing The Worst Case Execution Time Of An Avionics Program By Abstract Interpretion. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET2005)*, pages 21–24. Available from: http://artist.cs.uni-sb.de/WCET05/Papers/WCET2005Proceedings.pdf.

[SQlite, 2010] SQLite – a self-contained, serverless, zero-configuration, trans-

actional SQL database engine [online]. (2010). Available from: `http://sqlite.org/`.

[Stein, 2006] Stein, I. (2006). ILP-based Path Analysis on Prediction Files. Technical report, AbsInt Angewandte Informatik GmbH.

[Stein and Martin, 2007] Stein, I. and Martin, F. (2007). Analysis of Path Exclusion at the Machine Code Level. In Rochange, C., editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. Available from: `http://drops.dagstuhl.de/opus/volltexte/2007/1196`.

[Theiling, 2003] Theiling, H. (2003). *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Saarland University.

[Theiling and Ferdinand, 1998] Theiling, H. and Ferdinand, C. (1998). Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, pages 144–153, Madrid, Spain. IEEE Computer Society Press.

[Theiling et al., 2003] Theiling, H., Martin, F., Schneider, J., and Schmidt, M. (2003). Specification of the Standard for a File Format used for Exchanging Results of Different Parts of a Run-Time Analysis (ERD). Technical report, Universität des Saarlandes, AbsInt Angewandte Informatik GmbH.

[Thesing, 2004] Thesing, S. (2004). *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Universität des Saarlandes.

[Thiele and Wilhelm, 2004] Thiele, L. and Wilhelm, R. (2004). Design for Timing Predictability. *Real-Time Systems*, 28:157–177.

[van Heesch, 2007] van Heesch, D. (2007). *doxygen – Manual for version 1.6.2*.

Available from: `http://www.doxygen.org`.

[Wilhelm et al., 2009a] Wilhelm, R., Ferdinand, C., Cullmann, C., Grund, D., Reineke, J., and Triquet, B. (2009a). Designing Predictable Multicore Architectures for Avionics and Automotive Systems. In *Workshop on Reconciling Performance with Predictability (RePP)*.

[Wilhelm et al., 2009b] Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., and Ferdinand, C. (2009b). Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-critical Embedded Systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978.

[Wilhelm, 2001] Wilhelm, S. (2001). Generische Rekonstruktion von Kontrollflußgraphen aus Assemblerprogrammen. Diplomarbeit an der Universität des Saarlandes FB 6.2 (Wilhelm), Universität des Saarlandes.

Index

D

E

F

G

I

L

N

O

P

R

S

T

U

W

Z