DISSERTATION

Extending Optimising Compilation to Support Worst-Case Execution Time Analysis

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Doktors der technischen Wissenschaften unter Anleitung von

Ao.Univ.Prof. Dr. Peter Puschner

Institut für Technische Informatik 182.1

eingereicht an der Technischen Universität Wien,
Technisch-Naturwissenschaftliche Fakultät

von

Raimund Kirner

Matr.Nr. 9625030

A-2842 Edlitz, Sonnberg 37

Wien, im Mai 2003 .....

Extending Optimising Compilation to Support Worst-Case Execution Time Analysis

Embedded real-time systems are increasingly used in control applications. To guarantee the safe operation it is required to verify that the system can complete its tasks within their deadlines. Therefore, it is important to know the worst-case execution time (WCET) of the code running on the system. For the precise calculation of the WCET, the code has to be analysed at the object code level. The software is typically written in a language like C and translated by a compiler. Due to undecidability, the calculation of the WCET needs the knowledge of additional control-flow information – so-called flow facts. It is necessary to specify this information at the source code level in order to map the information onto the object code. For precise WCET analysis of programs optimised by the compiler, the flow facts have to be transformed in parallel to the code transformations performed during optimisation.

This thesis presents a framework to maintain correct flow facts during code optimisation. Previous solutions are either based on matching the debug information with the source code or on logging of only restricted code transformations. This thesis presents a novel concept to transform flow information in parallel to the code transformations performed by the compiler. The code transformations are abstracted to their relevant structural changes. From the structural update and the known semantic control-flow information of the performed code transformation, a safe and precise transformation of the flow information is induced. A graphical transformation framework to describe the performed code transformations supports the reader in understanding the required update of flow facts. The abstract representation of the control flow graph allows the modelling of blocks with multiple branching edges which enables the integration of this method into various compilers. The result is a framework that supports high-quality WCET analysis of optimised code.

Keywords

Real-Time Computing, Worst Case Execution Time (WCET), Timing Analysis, Flow Facts, Code Optimisation, Optimising Compilers

Erweiterung von Optimimierender Programmübersetzung zur Unterstützung von Statischer Analyse der Maximalen Programmausführungszeit

Eingebettete Computersysteme werden zunehmend für Steuerungsaufgaben eingesetzt. Um deren sicheren Betrieb zu gewährleisten, ist sicherzustellen, dass solche Systeme all ihre Aufgaben innerhalb der vorgegebenen Zeitschranken durchführen können. Es ist daher notwendig, die maximale Programmausführungszeit (Worst-Case Execution Time) des auf dem System ausgeführten Programmcodes zu kennen. Um diese WCET möglichst genau berechnen zu können, muss der Programmcode auf Objektcode-Ebene untersucht werden. Software wird heutzutage typischerweise in Programmiersprachen wie C geschrieben und mit einem Compiler übersetzt. Aufgrund von Unentscheidbarkeiten werden allerdings für die Berechnung der WCET zusätzliche Kontrollfluss-Informationen benötigt. Es ist aus praktischen Gründen erwünscht, diese Information direkt auf der Ebene des Quellcodes anzugeben. Die Kontrollfluss-Informationen müssen in diesem Fall für die WCET Analyse von der Quellcode-Ebene auf die Objektcode-Ebene transformiert werden. Um die WCET-Analyse auch für Programme, welche durch einen Compiler optimiert wurden, durchführen zu können, ist es notwendig, die Kontrollfluss-Informationen parallel zu Programm-Transformationen entsprechend zu aktualisieren.

Im Gegensatz zu früheren Arbeiten stellt diese Arbeit ein Framework zur Konstruktion von Transformationen der Kontrollfluss-Informationen bereit, das alle Arten von Codeoptimierungen unterstützt. Zuvor publizierte Ansätze stellen Beziehungen zwischen Quellcode und Objektcode über Debuginformationen beziehungsweise Traceausgaben für eingeschränkte Programmtransformationen her. In dieser Arbeit wird ein neuartiges Konzept vorgestellt, mit dem Kontrollfluss-Informationen parallel zu vom Compiler durchgeführten Programmtransformationen aktualisiert werden. Von den Programmtransformationen wird hierbei auf die relevanten strukturellen Änderungen abstrahiert. Diese strukturellen Änderungen zusammen mit der über die durchgeführte Programmtransformation bekannten Kontrollfluss-Semantik induzieren eine sichere und genaue Transformation der Kontrollfluss-Informationen. Eine graphische Repräsentation der durchgeführten strukturellen Programmänderungen erleichtert dem Leser das Verständnis der notwendigen Transformationen von Kontrollfluss-Information. Die verwendete abstrakte Darstellung des Kontrollflussgraphen erlaubt auch die Verwendung von Blöcken mit beliebig vielen Kontrollflusskanten. Damit wird die Integration der Methode in eine Vielzahl existierender Compiler unterstützt. Das Ergebnis ist ein Framework, welches hochwertige WCET Analyse von optimiertem Code unterstützt.

Schlüsselwörter

(Harte) Echtzeitsysteme, Maximale Programmausführungszeit (WCET), Zeitanalyse, Kontrollfluss-Fakten, Codeoptimierungen, Optimierende Compiler

Acknowledgements

This thesis was carried out during my employment as research and teaching assistant at the Institut für Technische Informatik, Real-Time Systems Group, at the Vienna University of Technology.

First of all I would like to thank my mentor and professor, Ao.Univ.Prof. Dr. Peter Puschner, for his valuable support and helpful suggestions. Besides his professional support I want to thank him for being a friend who always finds time for insightful discussions. I would also like to thank O.Univ.Prof. Dr. Hermann Kopetz for giving me the opportunity to work in a well-equipped and stimulating environment. I'm grateful to my secondary advisor Ao.Univ.Prof. Dr. Andreas Krall for several discussions related to compiler technologies.

Further, I want to thank Jan Gustafsson from the Mälardalen University, Jakob Engblom from Uppsala University, Guillem Bernat from University of York, and Frank Furrer for their interest on my work and the stimulating technical discussions we had. Special thanks goes to Jan Gustafsson, because after reading his PhD thesis, I knew that *abstraction* would be the key strategy to set up the theory for this thesis.

Thanks also to all my colleagues from the department of Technische Informatik, Vienna University of Technology, who have given me their technical support as well as a really pleasent working environment. I would particularly like to express my thanks and appreciation to Pavel Atanassov for his close cooperation and friendship and also, Christopher Temple, Thomas M. Galla, Thomas Losert and Andreas Steininger for interesting discussions about the topic of the thesis.

Besides Peter Puschner and Pavel Atanassov, I also want to thank my colleagues Johann Blieberger, Bernd Burgstaller, and Bernhard Scholz from the local WCET team of the Vienna University of Technology for fruitful discussions and exchange of information.

For the time consuming proof reading of preliminary versions of this thesis special thanks to Thomas M. Galla, Maria Nassey, and Wilfried Steiner.

I would like to express my gratitude to my parents for their support all the time. My friends, who were always close to me, appertain sincere thanks. Finally I would like to thank my girlfriend Lili Zhai for her patience and motivations and her special sense of humour all the time.

Contents

O friend, unseen, unborn, unknown,
Student of our sweet English tongue,
Read out my words at night, alone:
I was a poet, I was young.

James E. Flecker, *36 Poems* (1910)

Chapter 1

Introduction

Computer systems have become an essential part of human life. A well-known application domain for computer systems is the combination of personal computers and servers to assist people in their professional activities by processing and storing data. However, computer systems are used more and more to control their environment. Such computer systems are equipped with sensors and actuators to interact with the environment. In the growing market of *embedded computing*, computer systems are integrated into devices to perform controlling tasks. A current trend in computer science is *ubiquitous computing* (often called *pervasive* or *ambient computing*). One of the basic ideas of ubiquitous computing is that the computer hardware will completely disappear from the view of the user. Ubiquitous computing systems consist of a potentially large number of small networked components. They typically have to directly interact with their physical environment, as they rarely have traditional human-computer interfaces.

Computer systems that interact with their physical environment have to be designed to fulfil the temporal requirements of this environment. Besides the numerical correctness of a calculated result, the timeliness of this result becomes an additional requirement. Such computer systems are called *real-time systems*. Important parameters to reason about the timeliness of a real-time system are the best-case execution time (BCET) and the worst-case execution time (WCET) of internally performed tasks. The WCET of a program code typically depends on the specific target hardware and the possible values of the input parameters. The development of concepts for WCET analysis tools that can provide a safe upper bound for the WCET has become an active research field over the last fifteen years.

In general, WCET analysis can be done dynamically, i.e., by doing runtime measurements, or statically, i.e., by using analytical methods. The work presented in this thesis is primarily intended to be used within static WCET analysis frameworks, but it can also be applied to hybrid approaches that combine static and dynamic methods.

1.1 Motivation and Contributions of the Thesis

Initially, software for embedded systems written directly in assembly code. The increasing system complexity and the reduced time-to-market requirements demand the use of more accurate software development tools. Today, most of the embedded software is written in a language like C or C++. A current trend is to use more abstract development tools like MATLAB/Simulink that generate code automatically.

The representation level at which a program is developed is relevant when performing WCET analysis. Due to undecidability, it is proven to be impossible to calculate the WCET of a program in general (equivalent to solving the Halting Problem). Therefore, WCET analysis approaches often use additional flow information to search for the longest execution trace through a program. These pieces of flow information are called *flow facts*. To calculate a tight bound for the WCET it is required to calculate flow facts at object code level, as this is necessary to consider the specific properties of the target hardware in the WCET analysis. If the flow facts are given manually, it is of advantage to present them at the same level where the program is developed. This frees the developer from the error-prone task of translating them manually to the level where WCET analysis is performed. Also, if flow facts are mainly extracted automatically by a tool, it is preferable to perform this analysis at higher program representation levels, as there is typically more information available about the possible program behaviour than the object code level. An example for this are spilled registers. At object code level it is possible that the analysis cannot determine whether other memory accesses are interfering with the content of a spilled register.

To overcome these problems, one needs a mechanism that automatically transforms the additional flow facts required for the WCET analysis. A generic scheme for WCET analysis frameworks that are able to deal with different program representation levels is shown in Figure 1.1. The phase *extraction of flow facts* performs the calculation of the flow facts. To reduce the requirement for annotating the code manually with flow facts, it is a definite advantage to extract as many flow facts as possible, that are implicitly available in the source code. The execution time of single instructions or sequences of them is estimated within the *exec-time modelling* stage. The calculation within this stage depends on the target hardware for which the analysis has to be done. Before the WCET can be calculated, it is necessary to transform the flow facts in parallel to any code transformation. Such code transformations are typically done by an optimising compiler. Without knowing what code transformations have been performed, it is generally not possible to map flow facts from the source code to the object code level. Therefore, a mechanism to update the flow facts is needed, which we call *transformation of flow facts*.

The topic of this thesis is to develop a safe and precise concept for the *transformation of flow facts*. The supported flow facts have to be flexible enough to allow for the calculation of tight bounds for the WCET. At the same time, the interface must be applicable for manual code annotations as well as flow facts automatically derived by semantic code analysis. The challenges for the development of such a flow facts transformation framework are advanced code optimisations that change the control flow of a

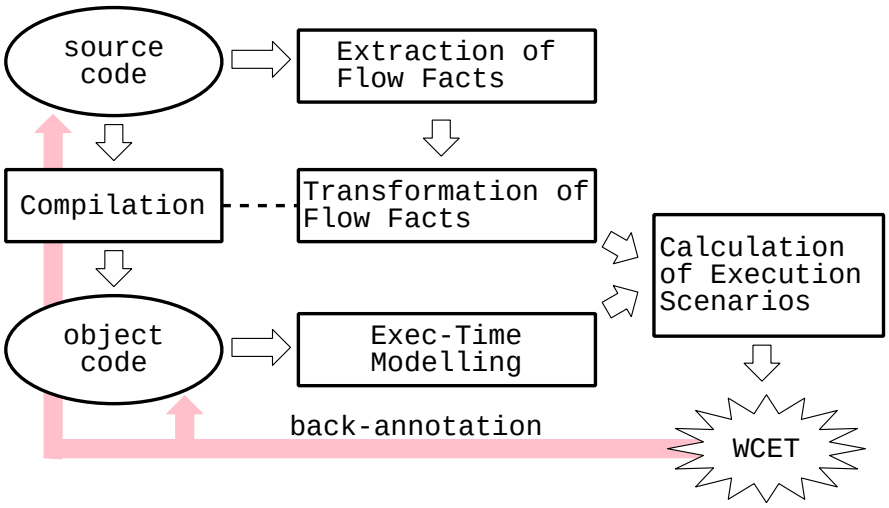


Figure 1.1: Generic WCET Analysis Framework

program dramatically. The approach has to provide a mechanism that can be integrated into a compiler with low implementation effort. It also has to be flexible enough to allow the later expansion for the support of new code optimisations.

Beside correctness, precision is an important quality criterion of a flow facts transformation framework. Within the context of flow facts transformation, two different aspects of precision can be identified:

- Inherent precision limitations due to the fact that flow facts are orthogonal information to the program semantics. This precision limitation can be minimised by selecting a type of flow facts that allows a flexible description of the possible program control flow.
- Precision limitations due to the fact that a flow facts transformation rule of an optimising compiler does not describe the required flow facts update precisely. To minimise this precision limitation, it is required for the construction of the flow facts transformation rule to exploit all information available about a code transformation.

The flow facts transformation framework we present within this thesis is hardware independent. This makes the approach universally applicable to various WCET analysis frameworks.

Contributions of the Thesis

This thesis presents a safe and precise *flow facts transformation* framework that is capable to transform flow facts correctly and accurately for any semantically correct code transformation. Towards the development of the flow facts transformation framework we provide the following main contributions over the state of the art in WCET analysis:

- During the initial stages of our work we had to define the role of flow facts within the process of WCET analysis. We divided the analysis process into independent problem categories. During this investigation we encountered problems with the existing terminology. Therefore, before starting with the development of the transformation framework, it was necessary to make a profound definition of independent problem categories. On this basis we developed a generic and implementation independent WCET analysis framework.

- The main contribution of the thesis is the development of a flow facts transformation framework that can correctly update the flow facts for any code transformation performed by the compiler. Within this framework, the flow facts transformation rule for each code optimisation consists of a set of basic transitions. There is only a small set of different basic transitions required, which are well-suited to describe the effects of low-level code transformations. More complex code transformations are simply handled by grouping a set of basic transitions together.

- We present a method based on the developed transformation framework to construct safe and precise flow facts transformations. This method allows the tool developer to systematically construct an appropriate flow facts update function for a given code transformation. The semantic control-flow information known by the compiler for a given code optimisation is used for that.

1.2 Structure of the Thesis

The content of this thesis is structured as follows:

Chapter 2 presents a detailed introduction to the research field of WCET analysis. The first part describes different aspects that are important for performing WCET analysis, including a discussion whether to use an approach based on static analysis or runtime measurements. Afterwards, we present a motivation for the selection of our concrete flow facts that are supported by our transformation approach. We also sketch a WCET calculation method based on these flow facts.

The related work for the context of this thesis is given in Chapter 3. Besides the issue of flow facts transformation we also present relevant work in the fields of WCET analysis in general, compilation for real-time systems, and symbolic debugging.

The basic foundations for the construction of our transformation framework are given in Chapter 4. The theory of *abstract interpretation* is described in more detail, because it forms the base for our construction of safe flow facts transformation rules.

Chapter 5 discusses the required update of flow facts in case of code transformations performed by an optimising compiler.

Chapter 6 describes the theoretical principles behind the construction of our safe flow facts transformation. It also sketches the operations required to update the flow facts in parallel to code transformations.

Based on these theoretic principles, Chapter 7 presents a concrete flow facts transformation framework. A description of the required operations and data structures is given.

Chapter 8 describes a scheme for using the transformation framework to develop concrete flow facts transformation rules that are safe and precise. This is supported by providing several examples for concrete transformation rules.

Chapter 9 gives an assessment of the approach presented within this thesis. Theoretical discussions show the potential of the approach. This is followed by a practical assessment based on a comparison of the results of the static WCET analysis with those obtained from runtime measurements.

Finally, Chapter 10 reviews the work done, points out strengths and weaknesses of our approach, and concludes the thesis giving directions for future research in this field.

'Is there any other point to which you should draw my attention?'
'To the curious incident of the dog in the night-time.'
'The dog did nothing in the night-time.'
'That was the curious incident,' remarked Sherlock Holmes

Sir Arthur C. Doyle, *Memoirs of Sherlock Holmes* (1894)

Chapter 2

Worst-Case Execution Time Analysis

This chapter describes the context of this thesis, the static WCET analysis. After an introduction about WCET analysis techniques, a generic WCET analysis framework is developed. Afterwards, the type of flow information and the corresponding WCET calculation method we use in this thesis are described.

2.1 Introduction to WCET Analysis

Computer systems often have to interact with their physical environments. As the environment has its typical temporal behaviour, these computing systems have to be designed to fulfil certain temporal requirements. Such computer systems are called *real-time systems*. The specification of tasks for real-time systems includes deadlines for the results they had to calculate. Depending on whether the miss of such a deadline is considered as a critical failure or not, we can distinguish between *hard* and *soft real-time systems*. Soft real-time systems follow a best effort strategy where deadline misses are in general treated as a question of quality of service.

An example for a real-time system is given in Figure 2.1. $\hat{t}_{observation}$ describes a request time which can have a static trigger or can occur dynamically. $\hat{t}_{reaction}$ is the time when the real-time system finishes processing the event and setting the resulting output signals. The difference $(\hat{t}_{reaction} - \hat{t}_{observation})$ is the *response time* $t_{response}$. The internal architecture of a real-time computing system can have various structures. Typically, it has a real-time scheduler that triggers the system tasks depending on their deadlines and worst-case execution times (WCET). The knowledge about the temporal behaviour and requirements of each task (here described by its WCET and deadline) is mandatory for the design of hard real-time systems. In the following we use the term *program* as a synonym for a simple task or the whole software running on a computer system since in this thesis we do not deal with the internals of software architectures.

Typical terms used to describe the execution time of a program are shown in Fig-

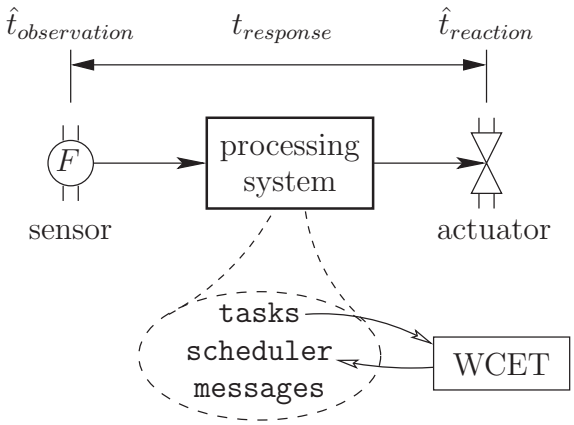


Figure 2.1: Schematic View of Real-Time System

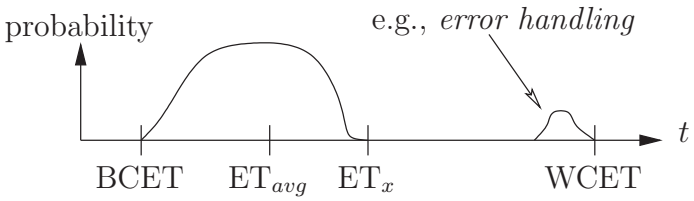


Figure 2.2: Distribution of Execution Time

ure 2.2: WCET is the worst-case execution time and analogous, BCET is the best-case execution time. ET_{avg} denotes the average execution time. To visualise the requirement for a systematic WCET approach we have drawn the execution time probability as a graph with two separate areas. The left area represents the normal program execution with a variance typically caused by variable input data and variable initial processor states (the influence of processor states is discussed in Section 2.1.1) having an upper execution time limit ET_x. However, due to the small second peak on the right end of the figure, the real WCET is much higher than ET_x. A reasonable example is handling of input data being outside specification (e.g., by an exception handler or saturation function).

A company reported to us that they had sporadically significantly higher results on runtime measurements of software in production. First, they thought it was a failure in their system but the reason was the rare case that a certain part of the software was sporadically executed. As this result was later directly shown by our prototype WCET analysis tool, the benefit of using analytical methods for WCET analysis has been demonstrated.

2.1.1 Hardware Characteristics

A processor has to maintain an internal state machine to model the semantics of program code. In the simplest case the state consists only of several registers and flags, which can

be modelled easily. Modern complex processors have a larger internal state to improve peak performance. The execution time of instructions depends on this larger state which is set by the previous instruction stream. In WCET analysis, approximations to reduce the set of modelled processor states normally result in impreciseness of the WCET analysis.

The following list shows performance increasing hardware mechanisms that increase the internal state space of a processor:

Memory hierarchies are used for a compromise between hardware costs and access times. Important characteristics of caches are associativity and replacement policy. Separate data and instruction caches have a less efficient memory usage than unified caches but their temporal behaviour can be predicted more precisely.

Parallel instruction processing; *overlapped instruction execution* or *pipelining* are used to increase throughput. Pipelining splits the instruction processing into several stages like FETCH, DECODE, EXECUTE and WRITEBACK. If the sequence of previous instructions cannot be modelled unambiguously, we get pessimism in the pipeline analysis.

Parallel execution units are an extension of the simple pipeline concept. The pipeline is splitted after a certain stage into multiple parallel pipelines, that form a so-called *scalar pipeline*. A further extension with duplicated execution units and dynamic instruction scheduling is called *superscalar pipeline*. The instruction scheduling for a VLIW (very long instruction word) processor is done statically by the compiler.

Out-of-order execution is used to improve parallelism of *superscalar* processors as it reduces the penalty of data hazards. For example, if the processing of an instruction stalls due to a data hazard, a subsequent instruction may overtake this instruction.

Out-of-order execution can result in "unexpected" timing behaviour, so-called *timing anomalies*. A timing anomaly is given if the execution time of a single instruction I embedded in an instruction sequence S is increased by a latency of d_I cycles, and the change d_S for the whole sequence S does *not* fulfil $0 \leq d_S \leq d_I$. Timing anomalies caused by a combination of out-of-order execution and data caches have been studied in the literature [LS99b].

Continuity of instruction stream within the processor pipeline can be improved using *branch prediction*, which guesses the control flow of conditional jumps. Dynamic branch prediction is based on the execution history. Branch prediction can be extended by a *trace cache* which is a special cache that stores sequences of instructions including information about which branch history has constructed these instruction sequences.

The above list of performance increasing hardware mechanisms with non-local timings effects gives an impression why it can be very complex to make a safe and precise

prediction about the execution time of code executing on modern processors because most effects of these mechanisms are interfering with each other.

Timing anomalies or unbounded timing effects of pipelines can arise. Another problem for WCET analysis, beside the inherent complexity of hardware modelling is, that it is usually hard to get a detailed documentation (if it exists) about the temporal behaviour of hardware.

The design of the whole system can increase complexity by involving for example DRAM refreshes or task preemptions.

Another barrier for developing a universal WCET analysis method are customisable processors where the manufacturer does not have the control over the final instruction timing. An example is the "BIOS update feature" introduced by Intel with the *P6* processor family to handle hardware bugs like the infamous Pentium FDIV floating-point division bug. Another example are *configurable* or *extensible processors* that have parts of its functionality - traditionally realised by register-transfer-level hardware - replaced by firmware program control. Application developers can write their own firmware targeted to a specific application domain.

2.1.2 Measurement vs. Static WCET Analysis

A static WCET analysis tool calculates the WCET based on a timing model of the target hardware. To overcome the problem of undecidability it is in general required to annotate the code with control flow information. Static WCET analysis tools are usually designed to calculate a safe upper bound of the WCET. The design and development of a safe and still precise WCET analysis tool for modern processors is a quite complex task.

A different approach with a longer tradition is to derive execution time bounds by measurements. The problem of simple measurements is that it can become infeasible to cover all execution scenarios (combinations of control flow paths and memory access locations). A first improvement is to select representative execution scenarios for testing the critical execution scenarios. By analysing the input-dependent statements of the source code one can try to extract execution scenarios that potentially result in the worst-case execution trace. However, in the general case it is not possible to ensure that the right scenarios have been selected. Typical problems are the manifold internal states of modern processors. Another problem of measurements is the *probe effect* (e.g., by code instrumentation to set output lines at certain control points). In the following we discuss why and how measurement could be still an option to derive the WCET for a piece of code/program.

Execution Time Measurements

As already discussed in the introduction, measurements were used to assess WCET in the beginning of building real-time systems due to the lack of theoretic foundations for static WCET analysis. One can argue that the current trend in processor development

has thrown static WCET analysis back to its roots and therefore execution time measurement has again become an alternative. In the following we discuss more technical aspects to question the applicability of measurements.

WCET estimation means by default to get a safe upper bound of the real WCET. Using execution time measurements has the following difficulties to achieve this:

1. The initial processor state (target hardware) influences the measured execution time.

2. Control transfer instructions (CTI) depending on input data induce multiple execution traces.

3. Memory references with addresses depending on input data induce different processor states.

In general, the above factors yield an exponentially high number of measurement test cases. For example, measuring the execution time for a real-size program with all possible combinations of input values can easily take more time than the known age of the universe[1]. Therefore, WCET measurement in general cannot test all cases and can just provide a reasonable unsafe lower bound for the real WCET. There may also be correlations of overspending with special cases of execution scenarios, e.g., calling an exception handler in case of a numeric overflow. Techniques like genetic programming are used to bring systematics into the search for the worst-case input data. However, such methods only increase the probability that the deviation of the measured execution time to the real WCET is within a certain bound, they cannot guarantee that the measured value will be equal to the real WCET. The arguments for favouring the concept of *probabilistic WCET* are often that by using statistical methods it would be possible to reduce the risk of a WCET underestimation to the same magnitude or even lower than the failure rate of system components. Anyhow, these techniques do not provide absolute error boundaries.

Advantages of Measurement-Based Techniques

Beside all the problems listed up to here, using measurement to get the WCET has its obvious benefits:

- Measurement equipment is almost independent of the target hardware.

[1] We assume a test function that takes an array as input data where n is the length of the array and k is the bit-width of the array elements. The age of our universe is denoted as T, the time needed for one measurement as t_m. To finish all measurements within the known age of our universe, we have to fulfil $2^{k \cdot n} \leq \frac{T}{t_m}$, which can be rewritten as $n \leq \frac{log(T/t_m)}{log(2^k)}$. Some scientists believe in an age of our universe of about $T \approx 15 \cdot 10^9$ years [Haw01]. Further, we assume that $t_m = 1$ ms and $k = 32$ bit. Then, it follows that the maximum feasible array length n for the input parameter of the test function is only $n \leq 2$!

- Detailed knowledge of the underlying hardware is also required to design good series of test data for the measurements, but not to the detail needed for static analysis.

Safe Upper Bounds of WCET by Measurement?

To obtain safe WCET bounds by measurement, the initial state scenario that generates the WCET has to be determined. For common processors it is not possible to set the internal processor state to a custom configuration. Instead, they support instructions to set parts of the internal state to a fixed value (e.g., invalidating all cache lines, enforcing all pending memory instructions to be finished, etc.). Even using measurements, it is therefore still important to carefully examine the behaviour of the processor to ensure that the initialised processor state represents a worst-case configuration. Timing anomalies demonstrate that this can be quite tricky.

Assuming that a safe worst-case initial state can be set, execution time measurements are useful to get a safe WCET bound under the following conditions:

small input space: If the value range of the input parameters is very small (e.g., bit flags), exhaustive search over the input space may be feasible.

sequential code: If the CTIs in the program do not depend on the input data, only one measurement is required. Programming paradigms like the single-path approach can be used to generate sequential code [PB02]. Writing WCET-oriented programs in general means to reduce the number of input-data-dependent CTIs in a program [Pus03].

Comparison Summary

Safe upper WCET bounds only can be derived by measurement from code that has few variations in its control flow paths. Static WCET analysis tools require much more effort to model the internal architecture of the target hardware.

In the following, both static WCET analysis and measurements are described in more detail.

2.1.3 Static WCET Analysis

Static WCET analysis is used to obtain safe upper bounds of the WCET. Static program analysis has its theoretical limitations due to undecidability (e.g., the *Halting Problem* [Man74, Lew85]). As a consequence, safe approximations about the program behaviour have to be used. Not all required information can be extracted by semantic analysis of source code. Depending on the analysis method, there remains some basic information that has to be provided separately, for example by code annotations. Such basic information could, for example, be the provision of information about the value range of instances of input variables or the direct specification of loop bounds.

The challenge for the design of a WCET analysis tool is the calculation of *safe* and *precise upper bounds* of the WCET with a *minimum* set of program annotations. As shown in Figure 2.3 the functionality of WCET analysis tools can be classified into three orthogonal aspects which are called *flow facts handling*, *representation level* and *exec-time modelling*. They are described in the following subsections.

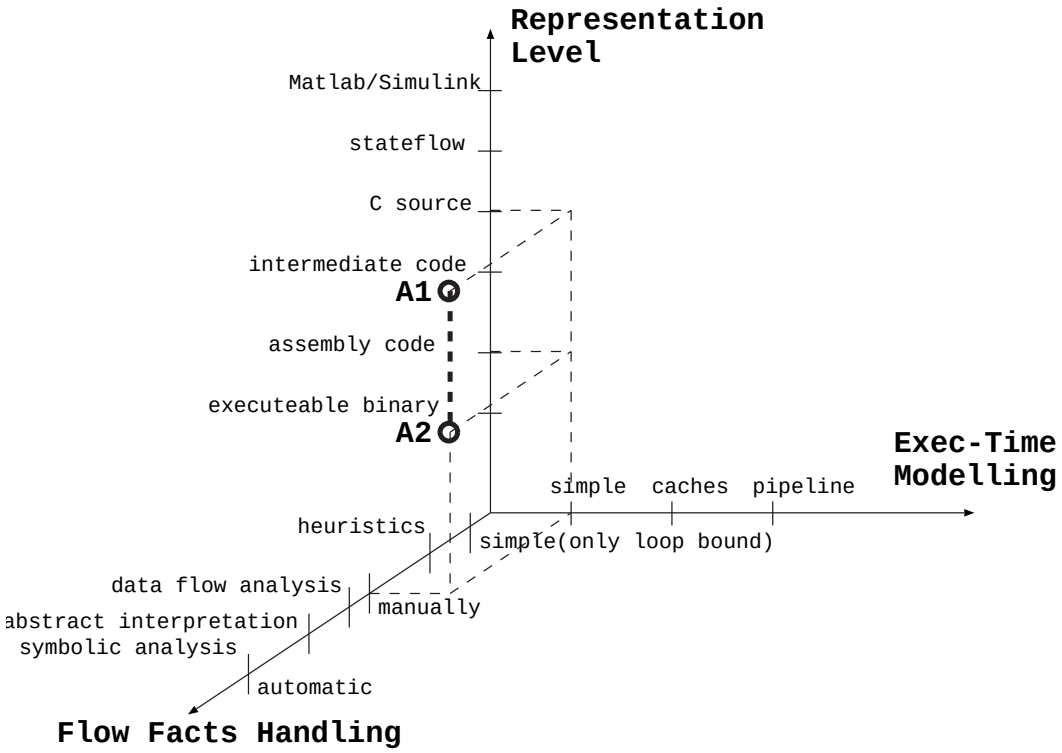


Figure 2.3: Orthogonal Aspects of WCET Analysis

a) Flow Facts Handling

Flow facts *ff* are used to describe the possible control flow paths *CFP* of a program $\mathcal{P}$. The definitions for the terms *ff* and *CFP* (as well as $CFP_{opt}(\mathcal{P})$ and $CFP_{ff}(\mathcal{P})$) are given by Definition 2.1.3 and Definition 2.1.1. The possible *CFP* of a program $\mathcal{P}$ described by the flow facts *ff* is denoted as $CFP_{ff}(\mathcal{P})$ where the tightest closure of the possible *CFP* is denoted as $CFP_{opt}(\mathcal{P})$. In general, the $CFP_{opt}(\mathcal{P})$ cannot be determined due to undecidability. For correctness it is required that $CFP_{ff}(\mathcal{P})$ is a superset of $CFP_{opt}(\mathcal{P})$. The longest execution trace within $CFP_{opt}(\mathcal{P})$ that yields the WCET is denoted by $CFP_{WCET,opt}(\mathcal{P})$ (see Definition 2.1.2).

Definition 2.1.1 Control flow paths *$CFP(\mathcal{P})$ specify the set of possible execution paths of program $\mathcal{P}$ with applied execution constraints. Such constraints are for example ranges for the value instantiation of input parameters. $CFP(\mathcal{P})$ is a description of the*

set of different possible execution traces that occur on execution of program $\mathcal{P}$. In case of unbounded loops this set is unbounded. We distinguish the following two types of CFP:

- *$CFP_{opt}(\mathcal{P})$ ... Control flow paths for a program $\mathcal{P}$ as seen by the omniscient observer. $CFP_{opt}(\mathcal{P})$ is an abstract and exact description of all possible program execution traces. $CFP_{opt}(\mathcal{P})$ includes only these execution traces that can really occur on program execution. One can reduce $CFP_{opt}(\mathcal{P})$ by specialisation of the program execution (e.g., by assuming input parameters of restricted shape or value range).*
- *$CFP_{ff}(\mathcal{P})$ ... Control flow paths of a program $\mathcal{P}$ described by flow facts ff. $CFP_{ff}(\mathcal{P})$ is an approximation of $CFP_{opt}(\mathcal{P})$ by considering a set of flow facts ff: $CFP_{opt}(\mathcal{P}) \subseteq CFP_{ff}(\mathcal{P})$. $CFP_{ff}(\mathcal{P})$ can be described as the by ff spawn closure of execution paths for a program $\mathcal{P}$.*

Definition 2.1.2 *$CFP_{WCET,opt}(\mathcal{P})$ is an execution trace that yields to the optimal solution of the WCET (denoted as $WCET_{opt}$) for a program $\mathcal{P}$. $CFP_{WCET,opt}(\mathcal{P})$ is derived from $CFP_{opt}(\mathcal{P})$. Its corresponding WCET value is therefore an optimal solution. $CFP_{WCET,ff}(\mathcal{P})$ is the analogous execution trace for the calculated $WCET_{calc}$, depending on ff.*

Definition 2.1.3 Flow facts *ff give hints about the possible CFP of a program $\mathcal{P}$. The resulting CFP over ff is called $CFP_{ff}(\mathcal{P})$. Flow facts can be expressed implicitely by the structure of the program itself as also by additional information provided by the user.*

To make WCET analysis feasible there must be enough flow facts to limit the execution count of every statement. Flow facts that can be extracted from the structure and semantics of the program are denoted by ff_{impl} (Definition 2.1.4). The extractable flow facts ff_{impl} are in general not enough to calculate the WCET since the execution behaviour of the program may also depend on external data. Additional flow facts ff_a (Definition 2.1.5) have to be specified. ff_a are typically given as annotations inside the source code, by separate data files, or interactively. The specification of ff_a inside the source code is preferable for the user of a WCET analysis tool since it frees the user from manually mapping or translating the flow facts and the flow facts only have to be specified once for the same revision of the source code.

Definition 2.1.4 Implicit flow facts *ff_{impl} are ff that are given implicitely by the program structure and semantics. If the $CFP_{opt}(\mathcal{P})$ of a program $\mathcal{P}$ does not depend on input variables or external events, it can be completely described by ff_{impl}.*

Definition 2.1.5 Flow facts by annotations *ff_a are ff that are given explicitely by code annotations. ff_a are used to simplify WCET tool implementation (avoiding complicated code analysis) or to bring in additional information to make WCET analysis feasible and tight.*

Methods for characterisation of flow facts can have different level of automatism. In the optimal way *ff* can be completely extracted automatically from the program's structure and semantics. Since not all information about the possible *CFP* is given implicitely by the program code, additional flow facts are needed, that are given manually (ff_a). At least, the determination of *ff* that are input data dependent requires the provision of additional ff_a. For simplicity reasons regarding the WCET tool implementation, also some classes of *ff* that could be extracted from the program code (ff_{impl}) are often specified redundantly by ff_a.

Definition 2.1.6 (In)Feasible Paths: *A control flow path p of a program $\mathcal{P}$ is called* feasible*, if it potentially can be taken during program execution: $p \in CFP_{opt}(\mathcal{P})$. Analogously, a control flow path p of a program $\mathcal{P}$ is called* infeasible*, if it cannot be taken during program execution: $p \notin CFP_{opt}(\mathcal{P})$. Infeasible paths typically occur in program analysis when not the complete semantics of a program is considered.*

For a *CFP* that contains cycles (generally called loops), the knowledge of the maximum execution number of the backward edges (loop bounds) is mandatory for WCET analysis. To obtain tighter time bounds, additional knowledge about infeasible paths (see Definition 2.1.6) has to be considered. The following is a list of examples for different levels of detail of execution frequency constraints (given as ff_a):

- loop bounds (exact versus safe overestimating bounds)
- loop bounds + additional constraints (e.g., loop sequences [Pus88], maximum execution count of a certain operation within a specific scope)
- arbitrary constraints on execution count of blocks
- having only one type of constraints (only conjunctive) versus supporting also disjunctive sets of constraints (necessary for completeness of constraints that model the algorithmic program behaviour)

Another type of ff_a are constraints that describe the execution order of operations. An example for this can be found in [EE00] where the authors describe execution frequency constraints that also refer to certain iteration ranges within a loop. Other types of ff_a are constraints on the ranges of value instantiation of input parameters [Gus00]. Such ff_a require advanced analysis techniques to transform them into information that can be used by the underlying WCET calculation method. For domain-specific solutions the calculation of the *CFP* can be simplified by using a quite restrictive syntax for the programming language. Some flow facts of ff_a can also be given redundantly to check for consistency with timing requirements. In [KHR+96] they use time annotations (concrete execution times) for certain code fragments and check them against the calculated WCET. They also use some other type of ff_a for the WCET calculation itself.

The automatism of WCET analysis depends on the implemented techniques in the analysis to extract flow facts. The following are typical techniques to extract and collect flow facts ff_{impl} and ff_a:

Abstract interpretation: if the tool performs some kind of *abstract interpretation* [CC77] it may be sufficient to specify as ff_a only the possible values for instances of input data. Gustafsson describes a WCET tool that extracts ff by using abstract interpretation [Gus00].

Simplified methods: they require the user to specify program annotations at a more abstract level, for example *loop bound* annotations. Using such annotations it may suffice to extract only the syntactic program structure as further flow facts.

Symbolic compution: this approach has a computational complexity between the two above extremes. The *CFP* is here computed by solving algebraic equations. Blieberger describes such a framework in [Bli02].

Depending on the expressiveness of the source language to be analysed it can be complex to develop a precise method based on abstract interpretation. Depending on the structure of the analysed code, such methods tend to have a high computation effort.

Simplified methods are not required to analyse the semantics of the code. Additional annotations about infeasible execution paths can enhance the precision of the result. For example, WCETC is a programming language where it is possible to specify ff_a directly inside the source code [Kir02].

A crucial drawback of symbolic computation is that it is limited to a restricted shape of expressions and constructs.

b) Representation Level

The coding of a program and the applied WCET analysis may be done at different *representation levels*. To obtain accurate and tight time bounds, WCET analysis is typically performed at assembly/object code level.

Programming in assembly code should only be done where it is strictly necessary in case of resource limitations, e.g., strict computation time or memory limitations. Typical representations for program development are *third generation languages* (3GL). Actually, research was carried out on WCET interfaces for 3GL like Euclid [KS86], Modula2 [Vrc94a], Java [BBW00], C [Kir00, Par93], etc. Furthermore, tools that model an application by its algorithm are for example MATLAB/Simulink[2] or the Statemate Statechart system [HN96].

As discussed above, it is required to use flow facts ff about the possible *CFP*. The most practical and intuitive approach to provide flow facts ff_a is to place them directly inside the source code at the location where they affect the CFP. If the representation level of the programming language is different to the one where the WCET analysis is done, compilers have to transform the ff to the level of analysis (we use the term compiler here for all kind of program transformations). A generic WCET analysis framework using ff transformation is shown in Figure 1.1. Compilers typically provide powerful optimisations that change the structure of the program during transformation down to

[2] http://www.mathworks.com/

assembly level dramatically. In this case, the transformation of flow facts cannot be done without compiler support since the structural matching between source and transformed code can become ambiguous while applying code optimisation techniques.

This leads to the challenge to transform the ff from the programming language down to assembly level (specifying the ff_a at a different representation level than the programming language is not feasible in practice because this only means that the user has to do this transformation manually). Therefore, methods are required to keep the ff consistent and useful even in presence of such optimisations. Figure 2.3 on page 13 shows an example, where the program is coded in C (marked as `A1`) and the analysis is done at assembly level (marked as `A2`).

c) Exec-Time Modelling

Static WCET analysis requires information about the execution time of each code statement, which we call *exec-time* information. For simple processors this is just a constant value for each statement and it may be of parametric form in case of more complex timing dependencies.

The potential complexity behind exec-time modelling has been described in Section 2.1.1. Attempts to model all timing features of modern processors at once have been shown to be too complex in development and computation. Approaches have been developed that split exec-time modelling into separate phases that can be computed sequentially. Such approaches can induce additional overestimation by approximation. But the benefit is to have simpler components that can be also replaced with less effort, which could be the base for a retargetable static WCET analysis tool.

For modern processors, i.e. processors optimised for peak-performance, exec-time modelling would require the knowledge of specific flow facts (e.g., the set of all possible previous instruction sequences) to build a precise context-dependent timing model. Exec-time modelling for modern processors is currently an active research topic. The motivation comes from the fact that modern processors are increasingly being used in time-critical embedded systems.

The opposite way of tackling the problem of exec-time modelling is to search for more predictable hardware mechanisms, which is currently less attractive since it does not target an already available mass market.

2.1.4 Calculating the WCET

The calculation of the WCET is done by searching the longest execution trace through the CFP. Currently, three techniques have been published to find this longest execution trace (it is also possible to use hybrid solutions of these WCET calculation methods):

- **Tree-based calculations** (timing schema) [Sha89, CP00, PK89] calculate the WCET hierarchically. Such methods are quite simple and fast but very restricted in the specification of (in)feasible paths.

- **Path-based calculations** [HAM+99, SA00] subdivide the program into several scopes. Typically, each loop is treated as a scope. The WCET calculation works hierarchically on the scopes by using exhaustive search.

- **Implicit path enumeration technique** (IPET) **based calculation** [PS97, LM95, LMW95a, LMW96] can be applied on the whole program code at once and allows to consider global flow facts. A program is translated systematically into a set of IPET constraints. The maximum solution of the goal function is the desired WCET. This method can be also used hierarchically to reduce computing resources. A typical technique to resolve the IPET constraints is integer linear programming (ILP). The IPET based WCET calculation is described in more detail in Section 2.3.

Due to its elegant modelling of flow facts and simple implementation by using standard constraint solvers, IPET based WCET calculation has evolved to be the most often used WCET calculation method. The calculation principle of using IPET for WCET analysis is presented in Section 2.3.2 on page 28.

Limitations of Static WCET Analysis

Static WCET analysis in general always requires additional flow facts which can be reduced to a theoretically minimal set by semantic code analysis. Concrete implementations typically require more annotations to reduce the complexity of the tool.

Depending on the desired target architecture and requirements on the preciseness of the results, static WCET analysis is ready for industrial use. Current research challenges are *non-local timing effects* (e.g., caches, pipelines, etc.) of modern processors and the development of modular and retargetable analysis frameworks that provide precise and safe results.

2.1.5 Measurement of WCET

The direct way to measure the execution time of a program run is to use a setup based on the real target hardware. Triggers of start and stop events can be generated by instrumenting the code or placing a test wrapper around it. The measurement can be done by means of a logic analyser where the execution time is given as an interval of physical time. By using a counter device, the time base of the external processor clock of the target hardware can be used for measurements. This allows a simple cycle-accurate execution time measurement.

It is also possible to use a "cycle-accurate" simulator of the target processor which has the great advantage of requiring no external hardware components and providing typically more flexibility in setting and monitoring the processor state. The performance of cycle-accurate simulation can typically be even faster but also orders of magnitude slower (possibly more than 1000 times) than execution on the real target.

Considering the problems discussed in Section 2.1.1 on page 8 for modelling the dynamic behaviour of the target processor, "cycle-accurate" simulation has similar drawbacks as exec-time modelling in static WCET analysis has.

Hybrid Measurement Techniques

Hybrid methods are an interesting alternative approach to deal with the inherently high effort of testing representative execution scenarios: combining static analysis techniques with measurement by grouping the CFP into blocks. To reduce pessimism, the granularity of the blocks should be coarser than just basic blocks. After measurement of all blocks, the overall WCET is calculated by a timing composition algorithm, i.e., putting together the measured execution times of all blocks.

Such hybrid methods can be used to determine safe upper bounds of the WCET as well as potentially more precise but unsafe probabilistic WCET values. Research has already been published for both paradigms. When using the probabilistic approach, the statistic distribution of measurement results is used to estimate the WCET for each block [PF99]. A possible method to determine safe upper bounds of the WCET is to measure the execution time of each basic block for all different control flow scenarios [Eng02].

An advantage of such hybrid methods compared to pure analytical methods is that it is relatively easy to retarget hybrid methods to a new processor platform.

Discussion of Usability

Execution time measurements have their clear benefits but determining a safe WCET bound for generic program structures by simple measurements is not feasible for programs with much input-data-dependent control flow due to the potential exponentially growing set of CFPs. Thus, pure runtime measurements themselves can only provide a lower bound of the real WCET.

Hybrid methods or software programming paradigms that enforce a simple program structure demonstrate that measurement still can be an adequate technique to determine safe WCET bounds. Hybrid methods can be also used to provide probabilistic WCET values.

2.1.6 Visualisation of WCET Results

Besides system scheduling, the knowledge of the WCET is also interesting as feedback for tuning programs. Thus, it is useful to have the WCET result available at a fine granularity and at different program representation levels. A prototype environment to propagate the WCET values back to individual instructions of the C source code and assembly code is described in [KHR+96]. Further research in mapping the WCET results in case of code optimisations is required to avoid confusing the user due to unexpected execution time distributions at source-code level. Such an unexpected execution time

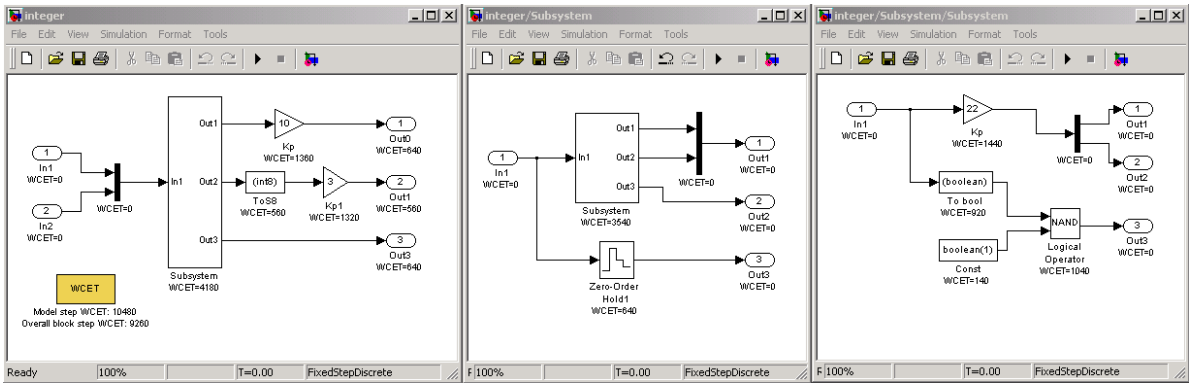


Figure 2.4: WCET results in MATLAB/Simulink

distribution can occur when, for example, the compiler reuses code or deletes useless code. For processors having pipelines or caches it is also interesting to map the WCET result down to the hardware states to reason about the efficiency of these mechanisms.

A continuous WCET analysis chain for MATLAB/Simulink realised in course of a research project demonstrates a smooth integration of WCET analysis into existing engineering tools [KLFP02]. The code generator of MATLAB/Simulink was modified to generate additional flow facts. As shown in Figure 2.4 the result of WCET analysis is propagated back to each block of the MATLAB/Simulink model. A nice feature of this approach is that MATLAB/Simulink can use the WCET result to simulate the temporal behaviour of a distributed system application.

2.1.7 The Current State of the Art

The industrial awareness for the requirement to perform WCET analysis is currently not very high. Heuristic measurements are common practice to get a rough overview about the timing behaviour of a computer system. Small pieces of code like interrupt service routines are typically analysed by counting the execution time of each instruction manually. Both practices are not really satisfying since it is required for hard real-time systems to get safe upper WCET bounds without the potentially high risk of having too much error-prone user interaction involved.

Academic research on methods for static WCET analysis has already provided prototype tools that work very well for relatively simple processors. The sufficient support of more complex processor architectures is still under research. There are also commercial tools available, but they are limited to specific usability. For example, some of the available tools model the target hardware only partially (e.g., modelling only one pipeline out of multiple processor pipelines) or they calculate an upper bound for the WCET that is too pessimistic to be directly used in an industrial production process.

Retargetability is still a challenge since it currently requires high effort to deal with problems like changing processor masks and the frequent emergence of new processor architectures.

Due to the increasing complexity of modern processors, research emerges that recycles the measurement approach to deal with such processors in a relatively simple way. Alternative research is starting in the area of predictable software and hardware. Favouring the WCET, for example the single path approach [PB02] promotes measurement as an adequate method to get a safe WCET value that is even precise.

To present the WCET results to the developer in a concise form, additional research would be required in the area of visualisation of WCET results.

2.2 The Process of Static WCET Analysis

The previous section gave an overview of existing WCET analysis techniques. In this section we introduce the fundamental components of a WCET analysis framework and analyse their functionality. The "feature space" of WCET analysis is shown in Figure 2.3 on page 13. This section also introduces the operations that provide these features.

2.2.1 A Generic WCET Analysis Framework

To show how the topic of this thesis is used within a WCET analysis framework, we introduce an abstract WCET analysis framework. This framework is kept quite generic so that it can be applied to various existing WCET analysis frameworks.

The main components of this framework are shown in Figure 1.1 on page 3. The input is the source representation of the program. The representation level of the input program and the required level where analysis has to be performed together determine what is required from the compilation process. As indicated by a dotted line, the transformation of *ff* is directly coupled with the steps performed by the compilation process. This thesis addresses this *ff* transformation in a correct and precise way.

Some existing frameworks directly read the compiled object code as input and request the user interactively to provide the required flow information [LMW95a]. In their case the *extraction of flow facts* and *transformation of flow facts* is completely left to the intellectual power of the user.

There have been previous attempts to develop a generic WCET analysis framework. A drawback of previous attempts is that there are mixtures between concepts and concrete implementation issues. An example for this are the often used names *high-level analysis* and *low-level analysis*. These names are intended for a certain type of WCET framework, but they are in general not appropriate. Using our categorisation, they can be understood as *flow facts handling* and *exec-time modelling*, which only describes their functionality without assuming a specific structure of the underlying WCET analysis framework.

2.2.2 Formal Definitions

We use the operator $\diamondsuit$ from *modal logic* to model the relation *"it can be"* ($\diamondsuit$ means that the given expression can be true under at least one interpretation or variable instantiation).

Definition 2.2.1 Intermediate Representations *The WCET analysis can be divided into several phases. The following operations and intermediate results are considered:*

- P_{src} *... source representation of the program*
- $P_{obj} = c(P_{src})$ *... object code of the program; obtained by compiling the source code.*
- $ff = e(P_{src})$ *... flow facts that gave hints about the possible execution scenarios; both,* ff_{impl} *and* ff_a *are extracted:* $ff_{impl} \cup ff_a = ff = e(P_{src})$
- $ff_{obj} = \tilde{c}(ff)$ *... transformed flow facts (from source to object code, including symbolic or numeric calculations);* $\tilde{c}(ff)$ *has to consider the operations performed by the code compilation* $c(P_{src})$
- $m_t = t_M(P_{obj})$ *... concrete hardware timing model; used to specify the execution time of a given code sequence for a specific target hardware*
- $WCET_{calc} = \omega(ff_{obj}, m_t)$ *... calculated WCET*
- $SC_\omega = \beta_s(WCET_{calc}, P_{src})$ *... WCET, back-annotated to source level*
- $OC_\omega = \beta_o(WCET_{calc}, P_{obj})$ *... WCET, back-annotated to object level*

The meaning of the symbols defined in Definition 2.2.1 is described in the following subsections.

2.2.3 Extraction of Flow Facts

Calculating the WCET by only using ff_{impl} (see Definition 2.1.4) is in general impossible since this task can be reduced to the well-known *Halting Problem* (described in [Man74, Lew85]). To make WCET analysis feasible, the use of additional ff_a (see Definition 2.1.5) is required:

$$ff = ff_{impl} \cup ff_a \tag{2.1}$$

As shown in Equation 2.2, redundant ff_a are often used to simplify the extraction of ff. The drawback is that specifying ff_a explicitely could be a source for errors if it is done manually by the user.

$$\diamondsuit(ff_{impl} \cap ff_a \neq \{\}) \tag{2.2}$$

The Dualism between ff and $CFP_{\mathit{ff}}(\mathcal{P})$

To perform WCET analysis for a program $\mathcal{P}$, the minimum required set of ff has to contain the syntactic structure and bounds for all loops. We call this minimal set of flow facts $\mathit{ff}_{syntax,lb}$. $\mathit{ff}_{syntax,lb}$ spawns the maximal set of execution traces $CFP_{syntax,lb}(\mathcal{P})$. The abstract flow facts that would be required to build $CFP_{opt}(\mathcal{P})$ (Definition 2.1.3) are called ff_{opt}. All WCET analysis frameworks support some kind of ff within these two extrema. It requires precise and flexible ff to minimise the potential cause for WCET overestimation - the infeasible paths: $CFP_{\mathit{ff}}(\mathcal{P}) - CFP_{opt}(\mathcal{P})$.

The $CFP_{\mathit{ff}_x}(\mathcal{P})$ for different ff_x are in partial order. From the structure of the partial order we can construct a lattice $L\langle M, \cap, \cup, \bot, \top\rangle$ where $M = \bigcup_{x\in X} CFP_{\mathit{ff}_x}(\mathcal{P})$, $\bot = CFP_{opt}(\mathcal{P})$ and $\top = CFP_{syntax,lb}(\mathcal{P})$. This formalism intuitively shows the effect of enriching ff_x. Figure 2.5 shows an example for such a lattice of CFPs by means of a *Hasse Diagram*.

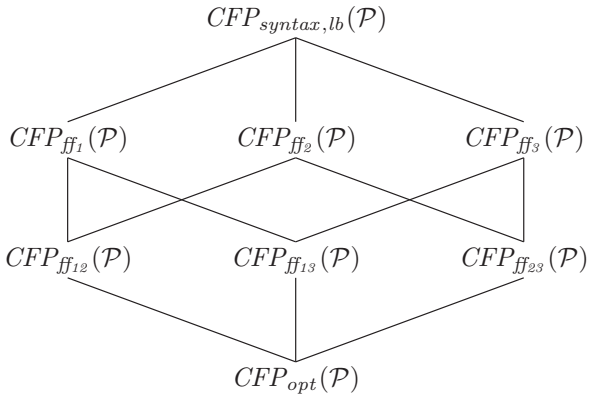


Figure 2.5: Partial Order of $CFP_{\mathit{ff}}(\mathcal{P})$ and $CFP_{opt}(\mathcal{P})$ for a Sample Program $\mathcal{P}$

Assuming that all ff_x are in normalised form without redundancy, we can show the dualism between changes in $CFP_{\mathit{ff}}(\mathcal{P})$ and ff. ff are in normalised format, when certain constraints on the CFP are always represented in the same format (e.g., transforming ranges for the value instantiation of input parameters into execution frequency constraints). This normalisation is an abstract model as it is in general not trivial to define a normalised form for all kinds of ff because it would also depend on the underlying method for calculating the execution scenarios. From that we get the definition of the lattice $L_D\langle M_D, \cap, \cup, \bot, \top\rangle$ where $M_D = \bigcup_{x\in X} \mathit{ff}_x$, $\bot = \mathit{ff}_{syntax,lb}$ and $\top = \mathit{ff}_{opt}$.

$$\begin{aligned} \mathit{ff}_z = \mathit{ff}_x \cup \mathit{ff}_y &\iff CFP_{\mathit{ff}_z}(\mathcal{P}) = CFP_{\mathit{ff}_x}(\mathcal{P}) \cap CFP_{\mathit{ff}_y}(\mathcal{P}) \\ \mathit{ff}_z = \mathit{ff}_x \cap \mathit{ff}_y &\iff CFP_{\mathit{ff}_z}(\mathcal{P}) = CFP_{\mathit{ff}_x}(\mathcal{P}) \cup CFP_{\mathit{ff}_y}(\mathcal{P}) \end{aligned} \tag{2.3}$$

The dualism of L_D and L in modifying ff and $CFP_{\mathit{ff}}(\mathcal{P})$ is shown in Equation 2.3. It demonstrates how enriching ff will bring $CFP_{\mathit{ff}}(\mathcal{P})$ closer to $CFP_{opt}(\mathcal{P})$. Additionally, we can derive the following rules from Figure 2.5:

- If ($CFP_{ff_x}(\mathcal{P}) = CFP_{opt}(\mathcal{P})$) holds, then ff_x is an optimal path description.
- If ($CFP_{ff_x}(\mathcal{P}) \cap CFP_{opt}(\mathcal{P}) \subset CFP_{opt}(\mathcal{P})$) holds, then ff_x is an invalid path description and can cause an underestimation of the WCET.

Methods

As already mentioned in Section 2.1.3, methods for characterisation of ff can have different levels of automatism. To bring the ff into a format useful for the WCET calculation method, the flow facts extraction pass $e(P_{src})$ has to evaluate and convert them. $e(P_{src})$ provides implicit as well as explicit ff available at P_{src}-level: $ff_{impl} \cup ff_a = ff = e(P_{src})$.

The syntactic structure can be extracted easily. The same is true for ff_a that act as simple structure information (e.g., loop bounds). ff_a that describe the program behaviour indirectly, are called *indirect* ff_a. An example for *indirect* ff_a are symbolic expressions that describe a loop bound in an algorithmic way similar to the program code.

For all kinds of *indirect* ff_{impl} and ff_a it is required to perform semantic analysis during extraction of ff. Examples for semantic analysis techniques are given in Section 2.1.3.

2.2.4 Compilation

To analyse a program for its WCET it has to be transformed from its source representation (P_{src}) to the representation where the analysis is done (P_{obj}). This transformation $P_{obj} = c(P_{src})$ (as defined in Definition 2.2.1 on page 22) is usually the program compilation. This compilation is a surjective projection from P_{src} to P_{obj}. The projection is defined by the compiler version and the activated compiler switches. Therefore, it is not possible to match the control structures from P_{obj} directly with that from P_{src} for all kinds of code optimisations of the compiler.

Obviously, in case P_{src} and P_{obj} are of the same representation level, no transformation is required. A typical example of this is writing and analysing a program at assembly level. Another case is described in [PS91], where WCET analysis is done directly at the C language level (actually on a small subset of C). However, the authors do virtual code compilation by assuming that the compiler is called with disabled code optimisations and therefore allows a direct assembly code prediction for each source statement.

2.2.5 Transformation of Flow Facts

As motivated in Section 2.2.4, it is required to transform the flow facts ff of the program in accordance with the program compilation. The operation for transforming ff has been defined in Definition 2.2.1 as $ff_{obj} = \tilde{c}(ff)$.

$\tilde{c}(ff)$ must work in close connection with the compilation process $c(P_{src})$. Using the debug information of the compiler can sometimes work as a simple mapping solution. In case of strong code optimisations it is required to get additional support by the compiler.

The simplest approach would be to transform ff manually from P_{src} down to P_{obj} [LM95]. This technique is simple to implement but hard to maintain in case of program updates and is potentially error-prone.

Another approach would be to let the compiler generate a code optimisation and transformation trace [EEA98]. This approach is easier to implement for simple optimisations. However, to support all types of optimisations, it is too complex [Eng97].

An alternative approach is to let the compiler do the mapping and generation of the correct ff_{obj} [KP01]. This approach requires more compiler modifications but provides the most flexible support for code optimisations.

2.2.6 Exec-Time Modelling

The subject of *exec-time modelling* is described in Section 2. To perform WCET analysis, it is required to derive a concrete time model m_t for the program P_{obj}. As described in Section 2.2.4, P_{obj} does not necessarily mean object code.

The operation to derive m_t is $t_M(P_{obj})$ (defined by Definition 2.2.1 on page 22). It is required that the semantics of m_t is compatible with the method for execution scenario calculation. The construction of an accurate m_t together with the search for a minimal $CFP_{ff}(\mathcal{P})$ are most challenging to minimise the overestimation of the WCET. To model some hardware features, it may be also useful to perform $t_M(P_{obj})$ at inter-procedural level, especially for recursive or short callee functions.

It is important to note that different *exec-time modelling* can yield a different $CFP_{WCET,ff}(\mathcal{P})$ for the same ff. Furthermore, Atanassov et al. reported in [AP01] the effect that adding the effects of DRAM refreshes changed the calculated $CFP_{WCET,ff}(\mathcal{P})$.

2.2.7 Calculation of Execution Scenarios

As shown by the functions used in Equation 2.4, the calculation of $WCET_{calc}$ depends on the sequence of previous operations.

$$WCET_{calc} = \omega(\tilde{c}(e(P_{src})), t_M(c(P_{src}))) \tag{2.4}$$

The Worst-Case Execution Trace

In Section 2.2.3 we have seen that all $CFP_{ff_x}(\mathcal{P})$ are in partial order and converge to $CFP_{opt}(\mathcal{P})$ by enriching ff_x (see Figure 2.5 on page 23). This is because ff_x always have to describe a safe approximation of the possible control flow $CFP_{opt}(\mathcal{P})$.

The $WCET_{calc}$ calculated by $\omega(ff_{obj}, m_t)$ has a corresponding execution trace

$CFP_{WCET,ff_{obj}}(\mathcal{P})$. The actual execution time of $CFP_{WCET,ff_{obj}}(\mathcal{P})$ is at most $WCET_{calc}$. There is still a partial order between $CFP_{ff}(\mathcal{P})$ and $CFP_{WCET,ff}(\mathcal{P})$ as shown in Equation 2.5. The same is true for $CFP_{WCET,opt}(\mathcal{P})$. It is also interesting to note that if $(CFP_{WCET,opt}(\mathcal{P}) = CFP_{opt}(\mathcal{P}))$ then the program has a *single-path* structure, i.e. sequential code. As discussed in Section 2.1.2, pure runtime measurements are an adequate method to obtain the WCET of programs having a single-path structure.

$$\begin{aligned} CFP_{WCET,opt}(\mathcal{P}) &\subseteq CFP_{opt}(\mathcal{P}) \\ CFP_{WCET,ff}(\mathcal{P}) &\subseteq CFP_{ff}(\mathcal{P}) \end{aligned} \tag{2.5}$$

Comparing $CFP_{WCET,ff}(\mathcal{P})$ and $CFP_{WCET,opt}(\mathcal{P})$ we can see some interesting properties of the control flow of programs. As shown in Figure 2.6, the various $CFP_{WCET,ff_x}(\mathcal{P})$ do not converge into $CFP_{WCET,opt}(\mathcal{P})$. For example, $CFP_{ff_1}(\mathcal{P})$ and $CFP_{ff_{12}}(\mathcal{P})$ yield at least partially different $CFP_{WCET,ff_x}(\mathcal{P})$. Furthermore, $CFP_{ff_{23}}(\mathcal{P})$ shows that we can get different $CFP_{WCET,ff}(\mathcal{P})$ for the same ff. The reason for this is due to the fact that *exec-time modelling* also influences $WCET_{calc}$ (as defined in Definition 2.2.1 on page 22).

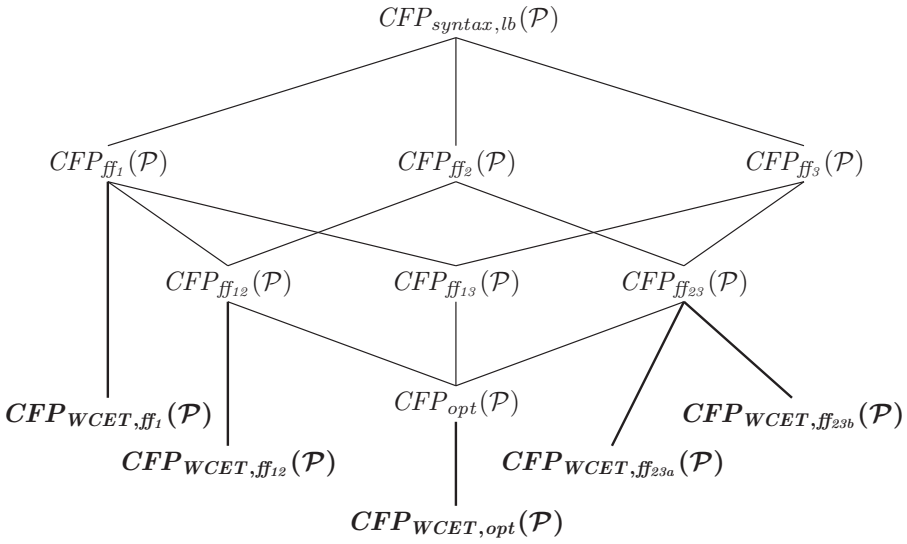


Figure 2.6: Partial Order of the Sets of Execution Traces $CFP_{ff}(\mathcal{P})$ and the Execution Trace $CFP_{WCET,ff}(\mathcal{P})$ that Yields to the WCET for a Sample Program $\mathcal{P}$

The fact, that the calculated $CFP_{WCET,ff}(\mathcal{P})$ can be different from the execution trace $CFP_{opt}(\mathcal{P})$ of the optimal WCET solution is formulated in Equation 2.6. For the other case $(CFP_{WCET,ff}(\mathcal{P}) = CFP_{WCET,opt}(\mathcal{P}))$ we would have found the *optimal path solution*. But more important, Equation 2.7 states that if we have not found the *optimal path solution*, we have found a $CFP_{WCET,ff}(\mathcal{P})$, outside of in reality possible execution paths. Beside incomplete exec-time modelling this is a major reason for *overestimating* the WCET.

$$\Diamond(CFP_{WCET,ff}(\mathcal{P}) \neq CFP_{WCET,opt}(\mathcal{P})) \quad (2.6)$$

$$(CFP_{WCET,ff}(\mathcal{P}) \neq CFP_{WCET,opt}(\mathcal{P})) \longrightarrow (CFP_{WCET,ff}(\mathcal{P}) \notin CFP_{opt}(\mathcal{P})) \quad (2.7)$$

Theorem 2.2.2 *It is* not safe *to extrapolate from a calculated* $WCET_{calc}$ *value to take the effects of additional hardware properties into account. Applying a different timing model can result into a different* $CFP_{WCET,ff}(\mathcal{P})$.

The main result of investigating $CFP_{WCET,ff}(\mathcal{P})$ is given in Theorem 2.2.2. To demonstrate this, a practical example for the effect of DRAM refreshes is given in [AP01].

Calculation Methods

Typical calculation methods for $WCET_{calc} = \omega(ff_{obj}, m_t)$ are discussed in Section 2.1.4. The result of the *execution scenario calculation* is the $WCET_{calc}$, but some approaches also deliver additional information like $CFP_{WCET,ff}(\mathcal{P})$ or the execution frequency of each statement.

For example, IPET-based WCET analysis (described in Section 2.3) provides information about the execution frequency of each statement but not the concrete $CFP_{WCET,ff}(\mathcal{P})$.

Back-Annotation of Results

To examine the timing behaviour of a program $\mathcal{P}$ and looking for best places to optimise them for lower WCET, it is required to split the $WCET_{calc}$ to its contribution to blocks of certain granularity. It is desired to know the WCET contribution for each single statement. Depending on the target hardware it may be also interesting to know more about $CFP_{WCET,ff}(\mathcal{P})$.

Back-annotation can be done on several representation levels within P_{src} ($SC_\omega = \beta_s(WCET_{calc}, P_{src})$) and P_{obj} ($OC_\omega = \beta_o(WCET_{calc}, P_{obj})$). It depends on the method used for *execution scenario calculation*, how much information is available. More aspects of back-annotation are described in Section 2.1.6 on page 19.

2.3 Static WCET Analysis Using Implicit Path Enumeration

This section gives an introduction to WCET calculation using implicit path enumeration technique (IPET). First, a description of integer linear programming (ILP) to be used as a constraint solver for IPET is given. Afterwards, it is shown how IPET can be applied to calculate the longest execution trace and what kind of flow facts can be used to describe (in)feasible paths.

2.3.1 Integer Linear Programming

Integer linear programming (ILP) is a method used in the area of operations research [Bur72]. ILP is a special form of linear programming where it is possible to require that certain variables in the solution are integer numbers. A typical mathematical problem for the use of ILP is the distribution of limited resources to concurrent processes. The goal in this case is to maximise the overall productivity of all processes.

An ILP problem consists of the following components:

- `n` parameters $c_1, c_2, c_3, \ldots, c_n$ of the system
- `n` decision variables $x_1, x_2, x_3, \ldots, x_n$
- the target function $Z = \sum_{i=1}^{n} c_i x_i$ ... is to be maximised
- `m` constraints like $\sum_{j=0}^{n} a_{ij} x_j \leq b_i \qquad \forall i \in [1, m]$
- `n` nonnegativity conditions $x_j \geq 0 \qquad \forall j \in [1, n]$
- specification of integer variables $x_j \in int \qquad \forall i \in [1, n]$

In the context of the resource distribution problem, the components of an ILP problem can be interpreted as follows:

x_j	...	amount produced of product j
c_j	...	profit per piece of product j
b_i	...	capacity of resource i
a_{ij}	...	input factors (used amount of resource i for the production of one piece of product j)

For the purpose of WCET calculation all variables must have integer solutions, because the decision variables in this case represent the execution counts of the control flow edges of the program to be analysed. It is therefore assumed throughout this work that every ILP variable is an integer variable.

2.3.2 Timing Analysis based on IPET

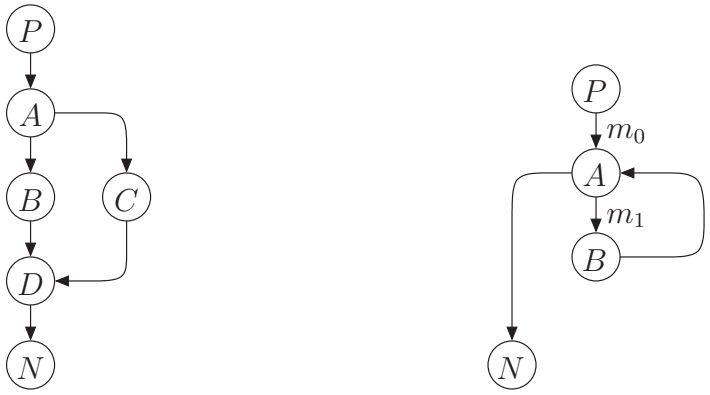
The WCET calculation method used with the proposed framework is based on implicit path enumeration technique (IPET) [PS97, LM95]. This method is quite flexible and allows to model the whole program or function globally as one IPET problem. One advantage of this method is the ability to specify *ff* that describe global dependencies between the execution frequency of certain execution paths. All kind of possible IPET constraints can be given as annotated *ff* in the source code and are translated safely in parallel to code transformations.

To demonstrate the expressiveness of the *ff* supported by our transformation framework, we will give a short introduction to IPET-based WCET calculation.

Assume the structure of a program is given by its control flow graph (CFG). The edges of the CFG are denoted by $\langle N_i, N_j, t\rangle$ which means that there exists a control flow edge from node N_i to node N_j of type $t \in \{s, b\}$. An edge of type s denotes a sequential control flow where a type of b denotes a branching control flow. $x_{N_iN_j[t]}$ represents the execution time of a single node N_i in the case it takes the control flow edge $\langle N_i, N_j, t\rangle$. $N_iN_j[t]$ represents a variable that counts the execution frequency of a CFG edge $\langle N_i, N_j, t\rangle$ during the execution of the program. Using this notation, the WCET can be expressed by the following IPET target function:

$$WCET = max \sum_{\langle N_i,N_j,t\rangle \in CFG} x_{N_iN_j[t]} \cdot N_iN_j[t]$$

The structure of the CFG is modelled as a sequence of constraints over the execution counting variables $N_iN_j[t]$. Furthermore, it is assumed that the minimum value of each such counter variable is nonnegative. Putting all this together, the solution of the resulting IPET problem represents the WCET of the program. This solution can be calculated by using integer linear programming. The structure of the resulting IPET constraints are explained by showing two examples:



(a) Modelling Alternatives (b) Modelling Loops

Figure 2.7: Modelling Program Structures

Figure 2.7(a) gives an example for alternatively executable CFG edges. There is no additional *ff* required to model the control flow of this structure. The possible control flow can be expressed by the following IPET constraints:

$$\begin{aligned} PA[s] &= AB[s] + AC[b] \\ AB[s] &= BD[s] \\ AC[b] &= CD[s] \\ BD[s] + CD[s] &= DN[s] \end{aligned} \tag{2.9}$$

An example for a loop is given in Figure 2.7(b). A loop requires at least additional *ff* to describe its loop bound LB. The syntactic control flow of the loop is modelled by the following constraints:

$$\begin{aligned} PA[s] + BA[b] &= AB[s] + AN[s] \\ AB[s] &= BA[b] \end{aligned}$$

The loop bound is modelled by applying the additional *ff*:

$$\begin{aligned} m_0 &= PA[s] \\ m_1 &= AB[s] \\ m_1 &\leq LB \cdot m_0 \end{aligned}$$

In addition to loop bounds, *ff* can also be used to express further knowledge about the runtime behaviour of the code. Such *ff* can be derived by semantic code analysis or just stated by having additional knowledge about the possible values of the input parameters.

2.3.3 Flow Information

The WCET calculation method based on IPET allows the specification of additional flow facts to limit the *CFP* in a quite flexible way. It is possible to use arbitrary constraints over the execution frequency of several *CFG* edges. The use of this flow information to obtain a more precise WCET value for IPET based WCET calculation is described in Section 2.3.2. These flow facts basically consist of the following components:

Markers to label control flow edges from the *CFG* of the program. It is also possible to assign multiple markers to the same flow edge.

Restrictions to limit the possible *CFP* by linear constraints over the execution frequency of *CFG* edges. The markers are used for reference to the *CFG* edges.

Loop bounds to limit the iteration count of a loop. For each loop the loop bound specifies the interval of possible iterations counts. Loop bounds can be directly translated into a combination of markers and restrictions. But during the flow facts transformation phase in parallel to code optimisations by the compiler they are kept as distinct values. This allows to transform flow facts more precisely. Loop bounds are mandatory flow facts to perform WCET analysis.

Such flow facts can be directly integrated into the source code to enable an intuitive interface for developers. An example for such an integration is WCETC [Kir02] which is a programming language derived from ANSI C. In WCETC it is possible to specify additional flow facts by source code annotations. The usage of WCETC in a compiler within a complete WCET calculation framework is described in [Kir00].

One of the initial questions for the start of our work was the selection of the type of flow facts to be supported by our flow facts transformation framework. As discussed in Section 2.1.4, IPET allows to consider the most general form of flow facts among the discussed WCET calculation methods. IPET based WCET calculation has also evolved to be the most often used WCET calculation method. Therefore, IPET was selected as the supported WCET calculation method. However, as other WCET calculation methods will typically exploit only a subset of the flow facts available for IPET, they will be supported by our method as well. As mentioned above, flow facts for IPET based WCET calculation allow the specification of arbitrary constraints over the execution frequency of *CFG* edges. There exists also a proposal given by Engblom et al. to extend the flow facts for IPET to support also constraints over on execution sequence of *CFG* edges by addressing subsets of a loop's iteration instances [EE00]. Providing flow facts at such a granularity becomes awkward when doing it manually. On the other side, current flow facts extraction techniques also do not support flow facts at this granularity. As a consequence, we decided not to support flow facts at such a granularity. But principally, our framework to construct safe flow facts transformation rules could be also applied to such flow facts.

2.4 Chapter Summary

This chapter described WCET analysis techniques, including a discussion about whether to use static WCET analysis or runtime measurements. A generic static WCET analysis framework has been presented to clarify the context of this thesis. Furthermore, the type of flow information that is used by our flow-facts transformation approach has been described in Section 2.3. This flow information allows to specify arbitrary constraints over the iteration frequency of control flow edges.

There exists a great chasm between those, on one side,
who relate everything to a single central vision...
and, on the other side, those, who pursue many ends,
often unrelated and even contradictory...
The first kind of intellectual and artistic personality
belongs to the hedgehogs, the second to the foxes.

Sir Isaiah Berlin, *Hedgehog and Fox* (1953)

Chapter 3

Related Work

This chapter gives an overview about related work about WCET analysis and other related topics. The basic tasks of a generic WCET analysis framework are shown in Figure 1.1 on page 3. The related work about WCET analysis is structured according to the different tasks of the generic WCET analysis framework shown in Figure 1.1.

3.1 Extraction of Flow Facts

Code annotations can be used to express additional flow facts (*ff*) for a program to enable the calculation of a (tight) WCET bound. Such code annotations can be implemented in various was, for example by programming language extensions, special compiler pragmas, separate description files, or as interactive user input. In the following we describe existing methods to annotate code with additional *ff* and to extract flow facts from the program code.

3.1.1 Manual Code Annotations

This subsection lists various types of *ff* that are proposed to support WCET analysis. For assessing the concepts, is also important to note that the *ff* transformation framework proposed in this thesis is not limited to a certain type of *ff*. This is because before *ff* are transformed by our method, they may be converted to an adequate format by the *flow facts handling* features of the WCET analysis framework.

Klingerman and Stoyenko describe the language Real-Time-Euclid [KS86], which was especially designed to write programs for hard real-time systems. This language disallows the use of recursion or `goto` statements. Loops are restricted to statically bounded `for` loops for which the number of iterations can be bounded easily and time-bounded generic loops. Real-Time-Euclid does not provide further mechanisms to describe infeasible paths.

Park and Shaw describe *timing schema* to calculate the WCET [Sha89, PS91, Par93]. The authors defined a regular-expression-based path language to express possible ex-

ecution paths. This path language may be expressive enough to exactly specify the possible *CFP*, but does not represent an adequate user interface for code annotations. Therefore, the authors developed the *information description language* (IDL) to express (in)feasible paths. IDL allows to express the exclusive, combined, etc. execution of program parts, but it is in general not flexible enough to describe a program's *CFP* precisely. The integration of a subset of IDL into a commercial compiler has been described by Börjesson [Bör95].

Puschner and Koza propose the language MARS-C to be used in combination with a tree-based WCET calculation method [PK89]. Beside bounded loops, they use new constructs like *scopes*, *markers*, and *loop sequences* to describe infeasible paths. Markers are used to restrict the execution count of program positions relative to its surrounding scope. The language also allows to bound loops by specifying a time bound, which has to be checked at runtime. However, the tree-based WCET calculation is not an adequate method to handle the flexible type of *ff* used in MARS-C. Therefore, the authors restricted the locations where markers can be used inside the code.

With the development of implicit path enumeration (IPET) based WCET calculation methods [PS97, LM95] it was possible to process more complex *ff*. Thus, also code annotation techniques have been developed to explicitely express flow information to be used for methods like IPET. Vrchoticky presented the programming language Modula/R [Vrc92, Vrc94b] that includes a generalisation of the marker/scope concept introduced in [PK89]. Modula/R was a language targeted to the MARS system [KFG+93] with additional annotations like provision of time bounds. To start with our research of *ff* transformation in parallel to code optimisations, we developed a new language, called WCETC [Kir00, Kir02]. WCETC is derived from ANSI C and contains *loop bound*, *marker*, *scope*, and *restriction* constructs as *ff* to describe the possible *CFP* of a program. The *ff* of WCETC can be directly used for IPET based WCET calculation methods. Engblom and Ermedahl described a language for IPET based calculation methods that also allows to address by *ff* subsets of a loop's iteration instances. No code example has been given by the authors, that exploits such an *ff* extension.

Blieberger has constructed so-called *discrete loops* to simplify the computation of upper loop iteration bounds [Bli94]. Discrete loop constructs demand from the programmer to describe, in which way each loop-variant variable changes from one iteration to the next iteration. This information is then analysed to compute a safe upper iteration bound for the loop. Discrete loops are limited to certain types of loops. For example, it is required that all loop-variant variables change their values monotonically. Annotated discrete loops are a compromise between simple manual code annotations without semantic code analysis and complex semantic code analysis methods that are able derive loop bounds and feasible paths automatically.

A WCET analysis framework that uses just basic program annotations like the value range of input parameters is described by Gustafsson [Gus00, Gus02]. The semantic code analysis technique of this WCET analysis framework is discussed in the following section.

3.1.2 Semantic Code Analysis Techniques

Before a WCET analysis framework is able start the calculation of a WCET bound, it has to extract the required flow facts from the input data. Input data are typically the program code and additional code annotations as describted above. The *extraction of flow facts* phase has to calculate from the annotated code the *ff* in a format that is suitable for the incorporated WCET calculation method. The complexity of the flow facts extraction depends on the type of provided code annotations and the possible complexity of the analysed code. For example, *ff* provided in a language like WCETC [Kir02] do not require complex calculations in the extraction phase. Analogous, when analysing only simple code structures (e.g., straight-line code without loops, as described by Stappert and Altenbernd [SA00]), the flow facts extraction phase can be kept quite simple. In the following we describe some approaches that spend more attention to the flow facts extraction phase.

Chapman described WCET analysis for SPARK Ada, a subset of Ada93 including annotations for static analysis [CBW94, CBW96]. Iteration bounds for loops are described in SPARK Ada by annotations about input data values or the "mode" of the program. The framework calculates loop bounds and infeasible paths (which are called *dead paths*). The approach uses symbolic execution and graph rewriting to calculate the WCET.

Altenbernd describes a WCET calculation method for real-time programs that uses only loop iteration bounds and function recursion bounds as *ff* annotations [Alt96a, Alt96b]. The described WCET analysis method is used within the CHaRy (C-Lab Hard Real-Time) system [Alt97]. The presented approach can identify certain infeasible paths automatically by using symbolic execution. The algorithm initially assumes all variable values to be *undefined* and updates them based on assignments or conditional tests. The accuracy of the method is restricted due to the fact that it does not consider annotations for input variables and uses a coarse value domain based on simple relations from variables to constant values.

Healy et al. describes a flow facts extraction framework to bound the number of loop iterations [HSW98, HW99, Hea99, HSR+00]. The authors present methods to bound the number of iterations for three special types of loops: loops with multiple exits, iteration bound depending on unknown variable values (requires annotations about value bounds of these variables), and inner loops that depend on counter variables of outer level loops. The analysis method is based on a technique to detect value-dependent constraints. The proposed analysis methods are relatively efficient for larger programs compared to methods that can analyse generic loop structures, but the proposed methods are limited to certain types of loops.

Gustafsson presents a WCET analysis framework for programs written in RealTimeTalk (RTT) which requires only basic program annotations like the value range of input parameters as *ff* [GE98, Gus00, Gus02]. The *ff* extraction is done by abstract interpretation. This method can find infeasible paths automatically. Compared to other approaches, the flow facts extraction phase of this approach takes more computation time, but automatically calculates precise (in)feasible paths based on simple program

annotations. It would be also interesting to see how this approach can be applied to analyse more complex data structures like lists or arrays.

3.2 Transformation of Flow Facts

While the research community has spent intensive effort to develop WCET calculation methods and modelling the target hardware, the implication of using optimising compilers for WCET analysis has received less attention. Since they radically change the code structure, the problem of mapping the structure of the source code and additional flow facts to the object code in case of optimising code transformations by a compiler has not been tackled sufficiently. The complexity arises when code optimisations radically change the structure of the generated code.

The group of Mok et al. uses special event markers to keep a mapping between the C source and the assembly code [MACT89]. Event markers describe the begin and end of program sections. These event markers are inserted automatically as annotations into the source code by a tool. A modified compiler [Ame88] has been used to transform the annotations to assembly code and to generate a timing analysis language (TAL) script. The TAL [Che87] script is interpreted to calculate the WCET. The authors do not describe whether they allow code optimisations during the compilation phase. A drawback of the approach is that the compiler only transforms the event markers from source to assembly code but no *ff* that are required for the calculation of the WCET. The user has to edit the generated TAL script to specify the loop iteration bounds or to express infeasible paths. The granularity of the timing blocks can be refined by inserting special "split points" into the annotated assembly code. Even if this approach would support the transformation of event markers in case of code optimisations, the task to correctly update the TAL scripts has to be done manually after the compilation phase.

Park et al. modified the GNU C compiler to perform WCET analysis of programs written in a subset of the C language and translated for the M68010 processor [PS91, Par93]. The *ff* are expressed by *information description language* (IDL) programs. The IDL statements are mapped with labels to the source statements. The authors perform WCET analysis at source code with predicting the code, generated by the compiler with default optimisations. This WCET analysis approach is not appropriate to handle code optimisations performed by a compiler.

The use of *debug information* is a simple technique to obtain a mapping between source and object code. This mapping becomes less precise in case of code optimisations and sometimes leads to surprising results. In case of radical code transformations, specifying *ff* for the source code and mapping them to object code cannot be done unambiguously [Exl99]. The timing tool described in [LMW96] does not derive *ff* from the source code. Instead, it interactively requests the specification of loop bounds from the user. The framework described in [FHL+01] allows to specify *ff* like loop bounds or recursion bounds at the source code and performs an external mapping without compiler support. Such an approach cannot support generic code optimisations.

A first improvement is to *modify the compiler* to output information, for example, about control flow or memory access addresses. [LBJ+95, HAM+99]. The described approaches mostly focus on providing information to model the target hardware, whereas they do not address the handling of *ff* in case of structure changing code optimisations.

Vrchoticky has taken the approach to completely integrate WCET analysis into the Modula/R compiler [Vrc94a]. This compiler does not perform any structure changing code optimisations. Therefore, the underlying *ff* transformation was not designed to support techniques like loop optimisations.
Beside the transformation of *ff*, Vrchoticky also discussed the aspect of presenting the timing effects of code transformations intuitively inside the source code. The reported overall WCET for a program or function must be always correct. A further refinement is to report also the relative execution times of single source statements in an intuitive way. For example, when performing *common subexpression elimination* (CSE, discussed in Section 5.4), it may be more intuitive when the execution time is distributed evenly to both source statements. Such a simple technique can be only applied when performing CSE on a single basic block. But more complex code transformations and sequential applications of them make it impossible to present the execution time of single source statements in an intuitive way.

Lim et al. let the compiler generate additional optimisation information [LKM98]. Their WCET calculation method is based on *extended timing schema*. As this calculation is performed hierarchically, they only need to consider loop bounds as *ff*. No other types of *ff* are supported. By using labels and transformation rules, their approach is powerful enough to model, for example, the construction of a new loop from two loops in the original code. But they do not calculate the new loop bound automatically, it has to be translated manually by the user.

Engblom et al. describe a more advanced approach for *compiler generated optimisation traces*, called *co-transformation* in [EEA98]. They designed an *optimisation description language* (ODL) to reflect the code optimisations performed by the compiler. As this approach is currently the most advanced published approach to deal with *ff* updates in case of code transformations, we will in the following discuss its limitations in more detail. A more detailed discussion about the capabilities of ODL is given in [Eng97]. During the code optimisations the compiler has to generate an ODL trace which is used by an external tool to perform the required *ff* update. It is desirable to improve four important aspects of this approach:

1. The co-transformer requires knowledge about how code transformations are performed by the compiler. An update of the optimisation code in the compiler raises the problem of maintaining the co-transformer itself. It is desireable to have a solution without the burden of updating two separate tools in parallel.

2. The WCET calculation method is based on the implicit path enumeration technique (IPET) which directly allows to consider *ff* like global constraints on the execution frequency between certain execution paths. But the data structure maintained by the co-transformer consists only of loop scopes with maximum loop

bounds for each loop and a maximum execution count relative to its surrounding loop scope for each basic block. These *ff* cannot describe more advanced relations like the execution count of triangle loops. Some code transformations, such as *loop unrolling* cannot be handled precisely due to this limited flexibility in representing *ff*. It is important to represent *ff* in a more flexible way to *support more code optimisations in a precise way.*

3. As stated by Engblom in [Eng97] (p. 56) it is still an open question to find a representation of infeasible paths in a transformable way. A flexible *ff* representation has to be found that supports also the transformation of information about infeasible paths.

4. The co-transformer maintains a control flow graph (CFG) in addition to the current *ff*. The matching of ODL statements with the CFG is done via the syntactical structure of its nodes. Edges of the CFG cannot be directly addressed. As a result, even common optimisations like *branch optimisation* are not supported. The introduction of new loops is not supported. The general graph structure in ODL is flat. It would be preferable to have a hierarchical representation that allows to match more generic code transformations.

In the *ff* transformation framework presented in this thesis, we addressed all these improvement aspects mentioned above. The aspect of Item 1 is addressed by a small set of transition rules to describe the *ff* transformation, which are described in Section 7.2. The flexible *ff* representation as mentioned in Item 2 and Item 3 has been addressed by representing the *ff* in a format close to that at which the *ff* are finally used to calculate the WCET. The structure of the data tuples is described in Section 7.1. The argument of Item 4 is supported by our method since we map all *ff* that describe infeasible paths directly to the edges of a program's CFG. Therefore, our approach supports also low-level code transformations that directly change control-flow edges of the CFG. Concrete *ff* transformation rules for representative code transformations are given in Chapter 8.

3.3 Exec-Time Modelling

Exec-time modelling deals with the construction of a hardware model that is suitable to reason about the execution time of program statements. Over the last years, the research community has spent high attention on performing exec-time modelling for modern processors that have performance-enhancing features like caches, pipelines, branch prediction, etc. This subsection gives an overview of this area by describing some representative contributions.

Lim et al. present an extension of the *timing schema* approach to handle pipeline and caches [LBJ+95]. Pipelines are modelled by using reservation tables. For cache analysis the authors assume some kind of cache partitioning to prevent tasks from disturbing each other's cache behaviour. The described method works for direct-mapped and set-associative instruction caches. Data caches are handled in a rather simple way. Lim

et al. have extended the approach in [LHKM98] to in-order superscalar processors, by maintaining instruction dependence and latency graphs instead of reservation tables.

Li et al. [LMW95b, LMW95a] modelled instruction caches (direct mapped as well as set associative) and data caches. They constructed a timing model by generating constraints, which represent some kind of flow facts. The cache analysis is directly integrated into IPET-based WCET calculation by using cache conflict graphs. The usage of this approach is limited, because the resulting complexity of the *calculation of execution scenarios* becomes infeasible complex for real-world programs.

Arnold et al. introduces static cache simulation for direct-mapped instruction caches [AMWH94]. The analysis uses the categorisations "always hit", "always miss", "first hit", and "first miss". Mueller has extended this static cache analysis method to set-associative instruction caches [Mue97]. Healy et al. describe this instruction cache analysis together in combination with a pipeline analysis using reservation tables. This approach is also able to capture pipeline effects that affect more than just neighboring basic blocks [HAM+99]. White et al. extend the WCET analysis framework by a data cache analysis [WMH+97, WMH+99]. Mueller describes in [Mue00] a framework to model instruction caches with arbitrary levels of associativity.

Colin and Puaut [CP00, CP01] modelled the branch-prediction behaviour of the Intel Pentium processor. They modelled the instruction cache, the branch prediction mechanism and the pipeline. They reconstructed the pipeline reservation table of each instruction by using the tool Salto[BS96]. Mitra et al. model the effect of advanced branch predictors with global histories using linear constraints [MR01, MRL02].

Ferdinand et al. describe in [FHL+01] a WCET analysis framework of the USES[1] group. This framework uses abstract interpretation to model caches (by *must analysis* [AFMW96]) and pipelines [SF99]. They reported troubles on modelling the unified instruction/data cache of the MCF 5307 (Motorola ColdFire) processor since this cache used a so-called "pseudo round robin" cache replacement policy, that is quite difficult to model [FHL+01].

Atanassov et al. have done exec-time modelling for the processor C167 from Infineon [AKP01]. The timing modell was derived and refined by doing measurements on the real hardware. The authors report the experience that the timing data given in the processor's manual are not sufficient to construct a safe timing model.

Engblom et al. describe a method to perform WCET analysis for processors with pipelines [EE99, EJ02, Eng02]. The exec-time modelling is done by using a standard trace-driven processor simulator. The authors simulate small code sequences, up to a length where the pipeline effect disappears.

Lundqvist and Stenström present an approach based on instruction-level architecture simulation for the PowerPC processor [LS98, LS99a, Lun02]. The method uses symbolic execution to exclude infeasible paths without having to use manual code annotations. The analysis is based on a relative simple value domain. Therefore, the path analysis is less precise than those described in [Gus00].

[1] USES ... University of the Saarland Embedded Systems

Petters and Färber describe a combination of static analysis and measurements [PF99, Pet00, Pet02]. A reduced control graph is generated for the program to be analysed, which limits the number of paths that have to be analysed. The execution times of all paths in the reduced graph are then measured on real hardware. The authors use additional instrumentation code to enforce all path combinations. The execution times of partitioned blocks are merged using extreme value statics. This approach provides a probabilistic WCET bound and not a safe upper bound. The retargeting of this approach to another target hardware is relatively easy, because exec-time modelling is done by measurements. Bernat et al. [BCP02] describe another probabilistic WCET analysis method based on measurements that does not rely on code instrumentation. However, this method relies on the provision of representative input test data to measure the execution profiles. The WCET is calculated by using a probabilistic variant of the *timing schema*.

3.4 Calculation of Execution Scenarios

There have been three basic techniques published for the calculation of the WCET by searching the longest execution trace: tree-based calculations, path-based calculations, and implicit path enumeration techniques. These techniques are briefly described in Section 2.1.4 on page 17.

The implicit path enumeration techniques is used within this thesis to calculate the WCET after correct transformation of the flow facts. Therefore, this technique is discussed in more detail in Section 2.3 on page 27.

3.5 Other Related Work

In this section we describe relevant work for this thesis that does not deal with WCET analysis issues. Instead, the following reseach topics deal with the code transformations performed by a compiler.

3.5.1 Code Optimisation for Real-Time Software

The scope of this thesis is not to consider the selection criteria for appropriate code optimisations on real-time programs. Instead, we enable the analysis of their impact on the WCET. It is a complementing research domain to analyse the effect of certain code transformations to the real-time behaviour, especially the WCET.

Marlowe et al. have shown that the application of several code transformations to reduce the average execution time can have serious impacts by causing deadline misses [MM92]. They argue that code optimisations for hard real-time programs must be safe in the sense that they will never cause a deadline to be missed in any execution of the program. The authors identified the following five categories of standard code

Optimisation	*Safety class*	*Notes*
DAG optimisation	dependent	Memory copy vs. immediate initialisation time.
common subexpression elimination	dangerous	Requires inserting a statement; may delay events.
copy propagation	safe	
constant propagation	dependent	Memory copy vs. immediate initialisation time.
dead code elimination	safe	
code hoisting	unsafe	Code may be hoisted past event; reach definitions analysis needed to determine if temporary variable must be added.
reduction in strength	dependent	Safety depends on machine architecture.
invariant code motion	fixable	Fix by unrolling to cover added initialisation time.
vectorisation	dangerous	Safety depends on vector length and relative execution times of vector and scalar instructions, if the loop includes events.
loop concurrentisation	fixable	Unroll to cover fork/join time.
loop fusion	dangerous	May delay events in second loop; safety depends on ability to concurrentise later.
index splitting	dependent	May be safe if vectorisation is possible later.

Table 3.1: Summary of Optimisations and their Safety [MM92]

transformations:

1. transformations which are always safe ("safe" transformations).

2. transformations which may not be safe but which have safe variants in many cases ("fixable" transformations).

3. transformations whose safety depends on memory and instruction cost information ("dependent" transformations).

4. transformation which are in general unsafe, whose safety depends upon the placement of events in the program being optimised ("dangerous" transformations).

5. transformations which provably cannot be applied to at least some real-time programs, and whose safety cannot be determined from local analysis of the transformed code ("unsafe" transformations).

A categorisation summary of concrete code transformations given in Table 3.1 has been extracted from [MM92].

Younis et al. investigated the effect of optimising a single process to other processes by introducing contention for shared resources [YMTS96]. An algorithm to safely apply machine-independent compiler optimisations to distributed real-time systems. The algorithm is based on resources' busy-idle profiles to investigate effects of optimising one process on other processes, whereby a restricted form of resource contention is assumed to simplify the analysis.

3.5.2 Source-Level Debugging of Optimised Code

A similar problem to WCET analysis of optimised code is source-level debugging of optimised code (SDOC). For precise WCET analysis of source programs at object code level it is necessary to update flow facts that describe the possible *CFP* of a program. To support source-level debugging of optimised code, a mapping has to be established between the statements of the source and the object code for each execution instance. Such a mapping can be complex since it has to handle the following two problems that arise for SDOC:

Code location problem; is determining which source statement corresponds to a certain instruction in the object code, and vice versa.

Data value problems; concerns the question where the value of a certain variable can be found for a given breakpoint of the program and whether this value is up-to-date.

The code location problem is also relevant for WCET analysis since the calculated timing information has to be reported back to the source code level. In the following we give a brief description of some work in the field of SDOC.

The definitions of several basic terms used in SDOC are given in [Hen82].

Copperman uses data flow analysis to deal with the data value problem [Cop94]. He performs the analysis on a single graph, which represents both the unoptimised and the optimised programs, to determine whether variables are current, noncurrent, or endangered (is noncurrent for certain control flow) at breakpoints. Thus, data value problems are partially handled. The described approach cannot handle all code transformations, for example, loop interchange is not handled. The author gives examples how to handle basic graph transformations (e.g., introducing a block, deleting a block or edge, coalescing two blocks, inlining a subroutine, unrolling a loop).

Wismüller describes a method to handle the data value problem in loops [Wis94a, Wis94b]. He maintains a copy of both the flow graph for the source and the object code of a program and preserves a relation on them, that is called CODE. An example for the creation of the mapping by CODE is given in [Wis94a]. For the static analysis, loops are unrolled in the unoptimised and optimised CFGs to distinguish among different instances of definitions. Then, data flow analysis is performed on the unrolled unoptimised and optimised program CFGs to determine whether variables are current or noncurrent at

breakpoints. The data value problem is therefore only partially handled. This approach works only with static information.

Adl-Tabatabai and Gross present an approach to handle the data value problem [ATG96]. In contrast to the approach from Wismüller described above, the method of Adl-Tabatabai and Gross uses analyses that are very similar to other analyses that are done by the compiler and can thus take advantage of an infrastructure that is already present. The work has resulted in a prototype implementation as part of the `cmcc` optimising C compiler. This approach is also based on only static information and the data value problem is handled only partially.

Jaramillo et al. describes an approach called FULLDOC, a solution for SDOC that can report every value that should be reportable at a breakpoint and is actually computed [JGS98, Jar00]. FULLDOC is based on static and dynamic information to handle the data value problem. Jaramillo et al. propose a code mapping that reflects the effects of code transformations. The mappings are established by analysing how the position, number, and order of instances of a statement can change in a particular context when transformations are applied. To report precariously placed values, FULLDOC also gathers dynamic information during execution. FULLDOC saves values before they are overwritten and deletes them as soon as they are no longer needed for reporting. FULLDOC also prematurely executes the optimised program until it can report a value while saving the values overwritten by the roll ahead execution, so that they can be reported at subsequent breakpoints. The only limitation of FULLDOC is that it cannot report values if their calculation has been omitted due to code optimisations. But as stated in [Jar00], existing techniques can be incorporated that recover some of these problems.

To conclude, the common task to support WCET analysis or to support source-level debugging is to transform meta-information in parallel to the code transformations. For WCET analysis, this meta-information describes the possible *CFP* of a program, where for source-level debugging, the meta-information forms a mapping of code locations and data values between source and object code. The meta-information for WCET analysis has a completely different semantics than the meta-information for SDOC. Therefore, new transformation methods have to be developed to support WCET analysis of optimised code. The context of WCET analysis and source-level debugging for optimised code is also discussed in Section 6.1 on page 99.

3.6 Chapter Summary

This chapter has presented scientific work related to the context of our thesis. Numerous work has been published on the issues of WCET calculation methods. Over the last years there has been a strong research focus on *extraction of flow facts* and *exec-time modelling*. Currently, there has been less attention on the issue of performing WCET analysis of optimised code.

There has been no work published on flow facts transformation methods that support arbitrary code transformations performed by the compiler. The topic of this thesis

extends the state of the art in WCET analysis by providing such a method.

Never explain – your friends do not need it
and your enemies will not believe you anyway.

Elbert Hubbard, *Motto Book* (1907)

Chapter 4

Foundations of Program Transformation and Abstract Interpretation

The formalisation of program transformations requires solid theoretical foundations. This chapter describes an abstract program representation based on the control flow graph. Formal semantics is introduced to describe the meaning of a program. Based on this semantics, abstract interpretation is described as a method to safely construct an approximation of a program execution. Further, techniques to combine abstract interpretations are discussed.

4.1 Program Flow Representation

In this thesis, the abstract program representation to describe code transformations is the control flow graph, which is described as follows:

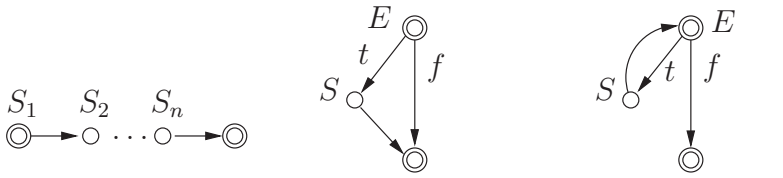
4.1.1 Control Flow Graph

The control flow graph (*CFG*) describes the possible control flow through the program. Formally, a *CFG* is a possibly cyclic, directed Graph given by the quadruple $G = \langle N, E, s, t \rangle$, where N is the set of nodes, E is the set of directed edges, s is a unique start node of in-degree 0 and t is a unique exit node of out-degree 0. For regular cases, all nodes $n \in N$ are reachable from s and t is reachable from all nodes n.

Series-Parallel Flowgraphs

Most popular definitions of *structured programming* assert that a program is structured if it is 'built-up' using only a small number of allowed 'constructs', which are *sequence*, *selection* and *iteration* [Fen91]. Boehm and Jacopini showed in 1966 the classical result

that every algorithm may be implemented using just *sequence*, *selection* and *iteration* [BJ66]. The corresponding flowgraph of these constructs is shown in Figure 4.1. All programs that are only built of these three constructs have a so-called 'D-structured' (or just structured) *CFG*.



a) Sequence: $S_1; S_2; \ldots; S_n$

b) Selection: `if` E `then` S

c) Iteration: `while` E `do` S

Figure 4.1: Basic Constructs for Structured Code

Each structured *CFG* is automatically also a series-parallel graph which follows from the hierarchical composition. A series-parallel graph is constructed starting from one start and one termination node by only applying series and parallel replacement rules [Miz02] as shown in Figure 4.2. Series-parallel graphs (as also structured graphs) are always *reducible* due to their construction rules.

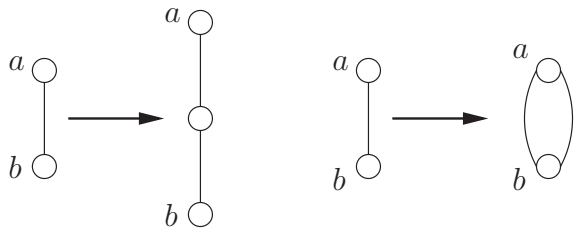


Figure 4.2: Series and Parallel Graph Replacement Rules

It is obvious that to guarantee series-parallel *CFG*s, program statements like `continue`, `break` or `goto` of ANSI C are not allowed. But also `switch/case` is not allowed since it violates the series-parallel structure if the `break` statement at the end of a `case` part (except the last one) is not given.

Reducible Flow Graphs

Series-parallel flowgraphs as described above represent structured programs. However, quite few common languages follow this strictly structured program syntax. Even constructs like *early-loop-exit* or *skip-to-next-loop-iteration* do violate the series-parallel flowgraph structure.

In practice, program analysis algorithms that are not applicable for generic program structures require a *reducible flow graph* [ASU97]. A *flow graph* G is *reducible* if and only

if its edges can be partitioned into two disjoint groups, the *forward edges* and *backward edges* with the following two properties:

1. The *forward edges* form an acyclic graph in which every node can be reached from the initial node of G.
2. The *back edges* consist only of nodes whose heads dominate their tails.

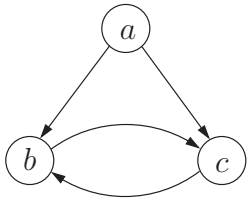


Figure 4.3: Irreducible Flow Graph

The flow graph given in Figure 4.3 is obviously *irreducible* since neither b nor c do dominate each other, but according to the above definition one of them has to be classified as a *back edge*.

In practice, programs rarely have *irreducible flow graphs*. *Irreducible flow graphs* can be typically constructed by generic jump statements (like `goto` in ANSI C). Program transformations like *loop unrolling* (described in Section 5.3.5) cannot be applied to a loop with an *irreducible* structure.

Control Flow Path

Normally, the *CFG* is directly derived from the syntax tree of the parsed program. Considering the semantics of the program statements and maybe also some a-priori knowledge about possible values of the input data, it can happen that not all execution traces represented by the *CFG* potentially can be taken during execution. The resulting set of possible execution traces is called *control flow paths* (*CFP*). A definition of *CFP* is given in Definition 2.1.1 on page 14. The concrete type of *CFP* to be considered for a certain static analysis technique depends on the needs of the analysis algorithms and the required precision. A technique to gradually improve efficiency of program analysis is to rewrite the *CFP* during analysis after preliminary results are available.

4.1.2 Call Graph

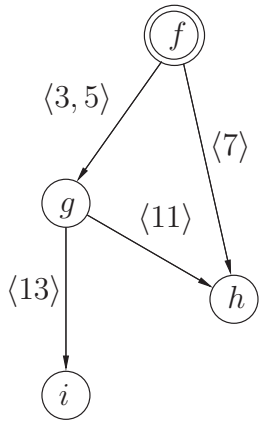
Problems that address any kind of interprocedural control-flow analysis are based on the *call graph* of the program. The call graph G is a directed *multigraph*[1] [Jun90] and can formally be defined as $G = \langle N, S, E, s \rangle$, where N is the set of nodes, S are the call

[1] multigraph . . . multiple edges allowed between two nodes

site labels, $E \subseteq N \times S \times N$ is the set of directed edges and s is the distinguished start node. The call graph is a graph where each node represents a function and each edge is a function call. Recursive function calls are marked by cycles in the graph. To be potentially executed, a function $n \in N$ must be reachable from s.

```
01 function f()
02   begin
03    call g();
04    if () then
05     call g()
06    else
07     call h();
08   end
09 function g()
10   begin
11    call h();
12    if () then
13     call i();
14   end
15 function h()
16   begin
17   end
18 function i()
19   begin
20   end
```

(a) Program Code



(b) Call Graph

Figure 4.4: Example for a Call Graph

The corresponding *call graph* for the sample program given in Figure 4.4(a) is shown in Figure 4.4(b). The source line numbers are used as call site labels to mark different multiple calls to a callee function within the same caller.

4.1.3 Global Control Flow Graph

CFGs are typically constructed at function level. Applications that depend on the syntactical structure of programs only have to construct the CFG at function level. This is typically the case for traditional software development tool chains where the compiler generates object files that are linked together afterwards by the linker. It is obvious that the restriction to a single function can limit the effectiveness of CFG based program analysis applications.

Each time a callee function is called within any caller function, it represents a differ-

ent calling instance of the callee functions. Constructing the global control flow graph (*GCFG*) allows to pass specific information to and from the calling instance. Several WCET analysis processing steps would benefit from this technique.

The *GCFG* is built by combining the *call graph* (Section 4.1.2) [Muc97] with the set of CFGs. Inlining of function calls is a simple *GCFG* construction method applicable for small programs with an acyclic call graph. A more flexible approach that also deals with cyclic graphs is *virtual inlining*.

An example for such a *GCFG* is presented in [AMW95] which is called *extended super graph*. It can be described as a graph with virtually inlined function calls. This virtual inlining works by duplicating the data fields for each CFG node for different caller functions. The data fields are the analysis-specific information in each node. Cyclic call graphs are resolved by allocating a set of data fields with the size limited to a constant value k. Additionally to the *GCFG* structure a set of mapping functions that connect the data index of the caller instance with the data index of the callee instance is needed. The *call string* approach [SP81] is used to distinguish different caller instances.

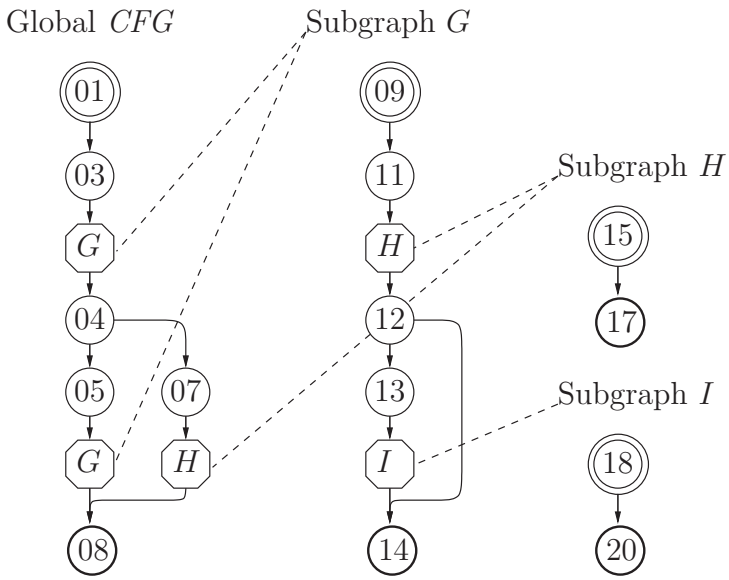


Figure 4.5: Global Flow Graph

The concrete modelling of a *GCFG* depends on the specific features of the programming languages. A *GCFG* having unbounded recursive function calls has to be modelled descriptively and calculated on demand. A compact *GCFG* for the sample program given in Figure 4.4(a) by defining subgraphs is given in Figure 4.5.

4.2 Semantics

The *semantics* describes the interpretation (i.e., the meaning) of a set of commands or statements. The semantics defines the resulting output and may also be defined on

other input parameters than just the commands. In this work, we do not limit the term semantics to interpretation of computer programs only. It is needed for different kinds of executing commands. Consequently, Definition 4.2.1 defines the relation between execution and semantics generically. The term *interpreter* is used to name the (abstract or concrete) machine or algorithm that is considered to execute the given commands.

Definition 4.2.1 (Execution) *is the processing of several commands under the control of a certain semantics* S*.*

In the following we will sketch three typical ways to describe the semantics of commands in computer programs [Feh89]. Examples for the definitions of semantics are given for the example language $\overline{\text{WHILE}}$ (see Annex A).

4.2.1 Operational Semantics

The operational semantics $\mathcal{S}^o$ describes the meaning of a program as a stepwise update of the state of the *interpreter*. To focus on relevant properties, the modelled states are usually an abstraction of the concrete states of the *interpreter*. The operational semantics is suitable for describing the meaning of programming languages in view of their implementation.

A *configuration* c is a pair $\langle \xi, \varepsilon \rangle$ where ξ is a *program control point* referring to the next statement to be executed and ε is the *environment*. An *environment* ε is a tuple $\langle \sigma, \eta, \theta \rangle$ where σ is the local store (VAR→(NUM ∪ {free})), η is the input file and θ is the output file. Based on the above terms and the syntax of $\overline{\text{WHILE}}$ (Annex A) we define the following tokens for the definition of the operational semantics of $\overline{\text{WHILE}}$:

Token		**Description**
INPUT:	(NUM ∪ BOOL)∗	the set of *input files* η
OUTPUT:	(NUM ∪ BOOL)∗	the set of *output files* θ
STORE:	VAR→(NUM ∪ {free})	the set of local variable bindings σ
ENV:	STORE×INPUT×OUTPUT	the set of *environments* ε
PREF		the set of program control points ξ (reference point marked with ↓)
CONF:	PREF×ENV	the set of *configurations* c

The operational semantics of $\overline{\text{WHILE}}$ is defined by four semantic functions:

$$\begin{array}{lll} \mathcal{S}^o_A[\![aexp]\!]\varepsilon: & (\text{AEXP}\times\text{ENV}) & \rightarrow (\text{NUM}\times\text{ENV}) \\ \mathcal{S}^o_B[\![bexp]\!]\varepsilon: & (\text{BEXP}\times\text{ENV}) & \rightarrow (\text{BOOL}\times\text{ENV}) \\ \mathcal{S}^o_S[\![\xi]\!]\varepsilon: & \text{CONF} & \rightarrow \text{CONF} \\ \mathcal{S}^o_P[\![\downarrow prog]\!]\varepsilon: & \text{CONF} & \rightarrow \text{CONF} \end{array}$$

The semantic function $\mathcal{S}^o_A[\![aexp]\!]\varepsilon$ defines the semantics of a numeric expression *aexp*, $\mathcal{S}^o_B[\![bexp]\!]\varepsilon$ the semantics of a boolean expression *bexp*, $\mathcal{S}^o_S[\![\xi]\!]\varepsilon$ the semantics of a single

program statement at control point ξ, and $\mathcal{S}_P^o[\![\downarrow prog]\!]\varepsilon$ defines the semantics for the whole program *prog*. The symbol $\downarrow$ denotes the current program control point. The following shows some fractions of the definition of these semantic functions to give an example of how they have to be defined (a more complete description of operational semantic functions is given in [Feh89]):

Arithmetic Operation:
$\mathcal{S}_A^o[\![a_1 + a_2]\!]\varepsilon$
$\Longrightarrow (n_1 + n_2, \varepsilon'')$ where $(n_1, \varepsilon') = \mathcal{S}_A^o[\![a_1]\!]\varepsilon$ and $(n_2, \varepsilon'') = \mathcal{S}_A^o[\![a_2]\!]\varepsilon'$
Assignment:
$\mathcal{S}_S^o[\![\downarrow V := a; S_2]\!]\varepsilon$
$\Longrightarrow \langle (V := a; \downarrow S_2), \varepsilon'' \rangle$ where $(n, \langle \sigma', \eta', \theta' \rangle) = \mathcal{S}_A^o[\![a]\!]\varepsilon$ and
$\varepsilon'' = \langle \sigma'[V \mapsto n], \eta', \theta' \rangle$

For modelling the occurrence of errors during program execution, the semantic functions have to be extended to deal with error values in their value domain. Based on the above semantic functions we can define the operational semantics by a mathematical machine $M = \langle C, \rightarrow, C_0, C_T \rangle$, where $C \subseteq$ Conf is the set of possible states (configuration), C_0 is the set of initial states, C_T is the set of termination states and $\rightarrow$ is the transition function as defined in Definition 4.2.2.

Definition 4.2.2 (Transition Function $\rightarrow$) *The transition function* Conf$\rightarrow$Conf *is defined as* $\langle \xi, \varepsilon \rangle \mapsto \langle \xi', \varepsilon' \rangle$ *where* $\langle \xi', \varepsilon' \rangle = \mathcal{S}_S^o[\![\xi]\!]\varepsilon$. *This transition function is also called* small step program transition function.

4.2.2 Denotational Semantics

The denotational semantics $\mathcal{S}^\natural$ is a mathematic formalism suitable to describe the meaning of a programming language in an abstract, short, concise and complete way. The denotational semantics abstracts from the stepwise state update of an *interpreter* and statically assigns to each program a function that maps from the program input to the output. The denotational semantics is also called *standard semantics* as it describes the "meaning" of single commands in a program. To derive the semantic functions for denotational semantics, the principle of the *curry-isomorphism* is used:

$$\begin{array}{lll} A: & (\text{Aexp}\times\text{Env})\rightarrow\text{Aexp} & \\ B: & (\text{Bexp}\times\text{Env})\rightarrow\text{Bexp} & \end{array} \quad \begin{array}{c} curry \\ \Longleftrightarrow \end{array} \quad \begin{array}{lll} A: & \text{Aexp}\rightarrow(\text{Env}\rightarrow\text{Aexp}) & \\ B: & \text{Bexp}\rightarrow(\text{Env}\rightarrow\text{Bexp}) & \end{array}$$

Using this isomorphism, the denotational semantics of $\overline{\text{WHILE}}$ is defined by four semantic functions:

$$\begin{array}{lll} \mathcal{S}_A^\natural[\![aexp]\!]: & \text{Aexp} & \rightarrow (\text{Env}\rightarrow(\text{Num}\times\text{Env})) \\ \mathcal{S}_B^\natural[\![bexp]\!]: & \text{Bexp} & \rightarrow (\text{Env}\rightarrow(\text{Bool}\times\text{Env})) \\ \mathcal{S}_S^\natural[\![stmt]\!]: & \text{Pref} & \rightarrow (\text{Env}\rightarrow\text{Env}) \\ \mathcal{S}_P^\natural[\![prog]\!]: & \text{Pref} & \rightarrow (\text{Env}\rightarrow\text{Env}) \end{array}$$

Some fractions of the definition of denotational semantic functions as presented in the following (a more complete description can be found in [Feh89]):

Arithmetic Operation:
$\mathcal{S}_A^\natural[\![a_1 + a_2]\!]\varepsilon$
$\Longrightarrow (n_1 + n_2, \varepsilon'')$ where $(n_1, \varepsilon') = \mathcal{S}_A^\natural[\![a_1]\!]\varepsilon$ and $(n_2, \varepsilon'') = \mathcal{S}_A^\natural[\![a_2]\!]\varepsilon'$
Assignment:
$\mathcal{S}_S^\natural[\![V := a; S_2]\!]\varepsilon$
$\Longrightarrow \mathcal{S}_S^\natural[\![S_2]\!](\mathcal{S}_S^\natural[\![V := a]\!]\varepsilon)$
$\Longrightarrow \mathcal{S}_S^\natural[\![S_2]\!]\varepsilon''$ where $(n, \langle\sigma', \eta', \theta'\rangle) = \mathcal{S}_A^\natural[\![a]\!]\varepsilon$ and $\varepsilon'' = \langle\sigma'[n/V], \eta', \theta'\rangle$

To model arbitrary control flow, arising for example with `goto` statements, the above defined denotational semantics has to be extended with *continuation semantics* [Bru81]. As described in [Feh89], the continuation semantic function for statements is written as:

$$\mathcal{S}_S^\natural[\![stmt]\!]\varepsilon: \quad \textsc{Stmt} \quad \rightarrow \quad ((\textsc{Env}\rightarrow\textsc{Env})\rightarrow (\textsc{Env}\rightarrow\textsc{Env}))$$

4.2.3 Axiomatic Semantics

The axiomatic semantics [Feh89] provides further abstraction than the denotational semantics. The axiomatic semantics does not describe state transitions but provides logical propositions about states. The meaning of a program P is characterised by the *weakest precondition* (given by the logical formula Q) and the *strongest postcondition* (given by the logical formula R):

$$\{Q\}P\{R\}$$

The semantics of compound statements is derived by rules like

$$\frac{\{Q\}S_1\{R\}\ ,\ \{R\}S_2\{T\}}{\{Q\}S_1; S_2\{T\}}$$

An application of the axiomatic semantics is given in [HL99] for the concept of generic loop unrolling. This technique is only described by manual experiments and not implemented into an optimising compiler. Therefore, we have decided not to deal with this code optimisation in more detail within this thesis.

4.3 Abstract Interpretation

Abstract interpretation is a formalised interpretation method that supports the systematic construction of a safe and correct interpretation based on a given concrete interpretation.

As described in Section 4.2, an *interpreter* executes commands with respect to a certain semantics. We have described three ways how to specify the semantics, but so far we have not discussed the value domain on which the semantic functions operate. We implicitely assumed an ideal interpreter with standard semantics for mathematic

operations and without numeric overflow, even for the description of the *operational semantics*. Therefore, all these described types of semantics are abstract semantics. In fact, it is not possible to completely describe the concrete semantics for a specific interpreter since it would be too complex to describe and model all of its temporal properties. Here, we call *temporal properties* the steps performed to get the result and *static properties* the type and value of the generated results.

Applications like *program analysis* interpret programs with the goal to extract certain classes of information about the program behaviour. Such analysis is done is based on a semantics that is defined with an abstract value domain. The calculation of certain program properties is equivalent to the *Halting Problem* [Man74, Lew85]. Another problem for program analysis is *state explosion*.

Therefore, approximations that introduce new properties to guarantee termination and manageable calculation complexity have be taken into account. An example for such an approximation is the use of intervals instead of concrete values. *Abstract interpretation* is an interpretation, based on a formalism to guarantee the preservance of consistent abstract properties in relation to the concrete semantics.

Abstract interpretation has its historical roots in program analysis starting in the 1960s. A similar idea called *pseudo-evaluation* [Nau92] was used by Naur in an Algol compiler in 1965. Much of the early work in program analysis was rather ad-hoc and the algorithms did not always preserve the semantics of the analysed language. In 1977, P. Cousot and R. Cousot presented a formal basis of abstract interpretation [CC77].

4.3.1 Definition of the Abstract Interpretation

This section presents a formal, quite abstract definition of an abstract interpretation given in Definition 4.3.1 (taken from [CC77]; with generalisation in [CC02] by using posets). In classical abstract interpretation frameworks [CC77] the domain $\langle \mathfrak{D}_P, \sqsubseteq \rangle$ was defined to be a complete lattice $\langle \mathfrak{D}_P, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$.

Definition 4.3.1 (Abstract Interpretation) *An abstract interpretation of a program P is a tuple $\langle \mathfrak{D}_P, F_P \rangle$, where $\mathfrak{D}_P$ is a poset $\mathsf{po}\langle \mathfrak{D}_P, \sqsubseteq \rangle$ (the semantic domain) and $F_P : \mathfrak{D}_P \to \mathfrak{D}_P$ is a semantic transition function. F_P must be monotone.*

The following sections give a description of the basic terms like program semantics, semantic transition function F, domain for interpretation $\mathfrak{D}$, fixpoint solution, etc.

4.3.2 Basic Principles of Abstract Interpretation

Abstract interpretation of programs is an approximation of their semantics. The correctness proof of an abstract interpretation requires the existence of the *standard semantics* which describes the possible behaviour of programs during their execution. As stated in [CC92a], the abstract semantics focuses on classes of program properties which are usually defined by a *collecting semantics*. This *collecting semantics* can be an *instrumented*

version of the standard semantics or, alternatively, a reduced version of the standard semantics to the essentials in order to ignore "irrelevant" program execution details[2].

To construct a safe abstract interpretation framework, it is required to formalise the concrete semantic domain $\mathfrak{D}$ and the concrete semantic transition function $F \in \mathfrak{D} \rightarrowtail \mathfrak{D}$[3]. An element $d \in \mathfrak{D}$ could be for example a set of maximal execution traces, a function, an input-output relation, a set of states, etc. [CC92a]. For the construction of the abstract interpretation several basic choices have to be made:

1. design of the *abstract domain* $\widetilde{\mathfrak{D}}$.
2. definition of the correspondence between concrete and abstract properties. The meaning of the abstract properties can be described by means of a *correctness relation* $\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \rightarrow \{\text{true}, \text{false}\}$ where $d\,\mathcal{R}\,\widetilde{d} \mapsto \text{true}$ means that the concrete semantics d of the program has the abstract property $\widetilde{d}$.
3. design of the *abstract semantic transition function* $\widetilde{F} \in \widetilde{\mathfrak{D}} \rightarrowtail \widetilde{\mathfrak{D}}$.
4. selection of extrapolation operators for inducing the abstract semantics. If the correctness relation $\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \rightarrow \{\text{true}, \text{false}\}$ includes conditions like $\exists d \in \mathfrak{D}, \exists \widetilde{d}_1, \widetilde{d}_2 \in \widetilde{\mathfrak{D}} : (d\mathcal{R}\widetilde{d}_1) \wedge (d\mathcal{R}\widetilde{d}_1) \wedge (\widetilde{d}_1 \neq \widetilde{d}_2)$, then the concrete semantic transition function F may have many different abstract semantic transition functions $\widetilde{F}$. The choice of the abstract properties can be done using the extrapolation operators *widening*: $\widetilde{\nabla} \in \wp(\widetilde{\mathfrak{D}}) \rightarrowtail \widetilde{\mathfrak{D}}$ and *narrowing*: $\widetilde{\Delta} \in \wp(\widetilde{\mathfrak{D}}) \rightarrowtail \widetilde{\mathfrak{D}}$ [CC92a].
5. selection of a convergence acceleration method ensuring rapid termination of the abstract interpreter even for abstract domains $\widetilde{\mathfrak{D}}$ with infinite height (e.g. if $\widetilde{\mathfrak{D}}$ is a *poset* containing *chains* of infinite lengths). The convergence acceleration will also be achieved by using some *widening*: $\widetilde{\nabla} \in \widetilde{\mathfrak{D}} \times \widetilde{\mathfrak{D}} \rightarrowtail \widetilde{\mathfrak{D}}$ and *narrowing*: $\widetilde{\Delta} \in \widetilde{\mathfrak{D}} \times \widetilde{\mathfrak{D}} \rightarrowtail \widetilde{\mathfrak{D}}$ operators [CC77]. The issue of convergence acceleration is discussed in more detail in Section 4.3.10 on page 78.

The following subsections define the fundamental terms introduced above and show their construction in more detail.

4.3.3 Domain of the Interpretation

To describe the semantics of a program it is required to define the semantic domain $\mathfrak{D}$ and the semantic transition function $F : \mathfrak{D} \rightarrow \mathfrak{D}$.

The semantic domain $\mathfrak{D}$ is a structure that contains all data objects of a certain type. A transition function F of an interpretation has to operate on a well-defined

[2] for practical reasons, also the *concrete semantics* has to be reduced to certain properties that are considered relevant for correct program execution behaviour

[3] $\mathfrak{A} \rightarrowtail \mathfrak{B}$ denotes the set of all functions $F : \mathfrak{A}' \rightarrow \mathfrak{B}'$ with $\mathfrak{A}' \subseteq \mathfrak{A}$ and $\mathfrak{B}' \subseteq \mathfrak{B}$ ($\mathfrak{A} \rightarrowtail \mathfrak{B}$ is the set of all partial functions F from $\mathfrak{A}$ to $\mathfrak{B}$)

domain. The required structural properties on a semantic domain $\mathfrak{D}$ depend on the specific interpretation method. The interpretation method is derived from the type of semantics description (operational, denotational, ...) and the interested properties of program execution. These interested properties are application specific.

As stated in [CC02] the semantic domains used in abstract interpretation must be a poset (partial ordered set) $\langle \mathfrak{D}, \sqsubseteq \rangle$. However, semantic domains often also enjoy stronger properties. The following lists some typical structures of domains:

- **partial order** po $\langle \mathfrak{D}, \sqsubseteq \rangle$,
  a set $\mathfrak{D}$ of elements partially ordered by the relation $\sqsubseteq$ (poset).

- **complete partial order** cpo $\langle \mathfrak{D}, \sqsubseteq, \sqcap, \bot \rangle$,
  a partially ordered set with a least element $\bot$ and the greatest lower bound operator $\sqcap$.

- **dual complete partial order** (co-cpo) ccpo $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \top \rangle$,
  a partially ordered set with a top element $\top$ and the least upper bound operator $\sqcup$.

- **complete lattice**,
  $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ a partially ordered set with a bottom element $\bot$, a top element $\top$, a greatest lower bound operator $\sqcap$, and a least upper bound operator $\sqcup$.

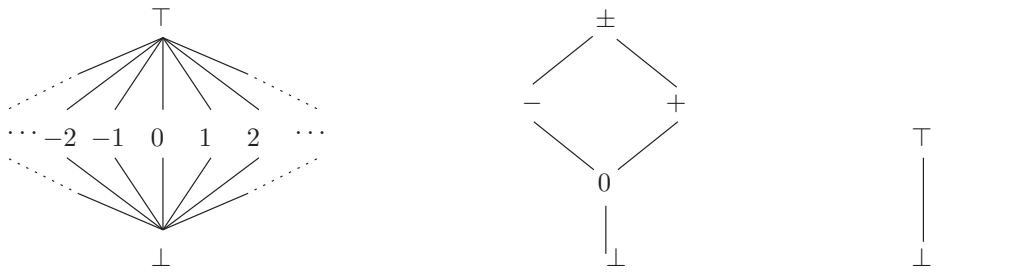
The concrete interpretation of a program $\mathcal{P}$ is defined using the concrete semantic domain. We denote the concrete domain with $\mathfrak{D}$. An abstract program interpretation is defined using an abstract domain which we denote as $\widetilde{\mathfrak{D}}$. The domain $\widetilde{\mathfrak{D}}$ will only describe execution properties instead of the concrete execution of a program $\mathcal{P}$.

The following describes examples of semantic domains:

- An example of interesting program properties is the set of possible values for a variable at a certain program point during execution. A typical program analysis based on this domain is *constant propagation* [Muc97]. The domain used for *constant propagation* is given in Figure 4.6(a), where $\top$ denotes that a variable could refer to more than one value during program execution.

- A domain suitable for *sign analysis* of variable values is given in Figure 4.6(b). The quite simple domain given in Figure 4.6(c) is expressive enough to perform *reachability analysis* for code blocks. The result of this analysis can be used for *unreachable code elimination* [ASU97].

- An abstract interpretation based on abstracting from sets of possible values of variables to a value interval can be done with the domain shown in Figure 4.7. Furthermore, this domain is an example of an abstract domain of infinite height. Therefore, special attention is needed to ensure termination of the interpretation.



(a) Constant Propagation (b) Sign Analysis (c) Reachability

Figure 4.6: Example Domains for Abstract Interpretation

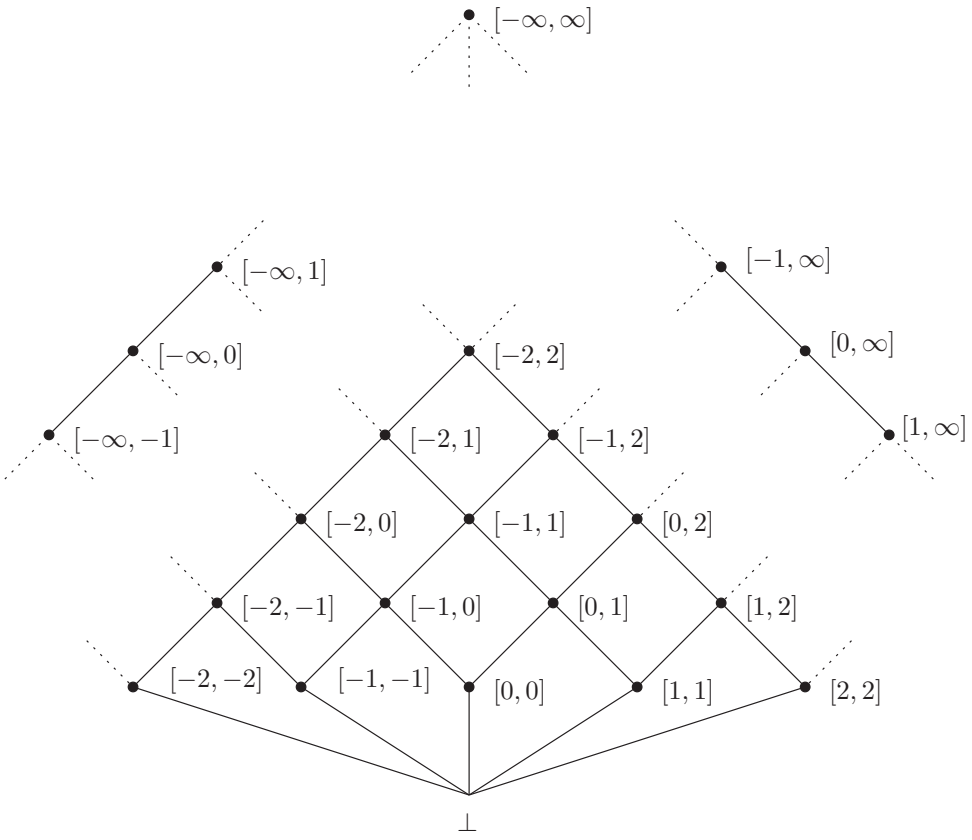


Figure 4.7: Domain for Value Interval Analysis

In this thesis we focus on correct program transformations to support correct and precise WCET calculation of programs. The concrete domain contains the code of a program that is to be transformed by the optimisation rules given by the semantic function F. The code is annotated with additional flow information given by a prior static analysis or by manual code annotations. The abstract domain $\mathfrak{D}$ represents the

possible control flow graph of this annotated program. The final result in the abstract domain will be used for timing analysis to calculate the best- and worst-case execution time of the concrete code (see Chapter 6 and Section 7.1).

4.3.4 Fixpoint Semantics for Abstract Interpretation

As shown in Section 4.2 the semantic transition function F of an abstract interpretation $\langle\langle\mathfrak{D},\sqsubseteq\rangle, F, \amalg, \bot\rangle$ is defined in general on single-program-statement level as a partial map $F \in \mathfrak{D} \rightarrowtail \mathfrak{D}$ (where $A \rightarrowtail B$ is the set of partial functions from the set A to the set B). The program iteration can be specified by transfinite recursion using a basis $\bot \in \mathfrak{D}$ (the initial environment) together with the semantic transition function F and an inductive join $\amalg \in \wp(\mathfrak{D}) \rightarrowtail \mathfrak{D}$ so that [CC92a]:

$$F^n = \begin{cases} \bot & \text{if } n = 0 \\ \amalg_{\beta<n} F^\beta & \text{if } n > 0 \text{ is a limit ordinal} \\ F(F^{n-1}) & \textit{otherwise} \end{cases} \tag{4.1}$$

Due to the fact that F and $\amalg$ are partial functions, we can distinguish between total (Definition 4.3.3) and partial (Definition 4.3.2) iterations. The existence of non-deterministic statements like ideal calculation of random numbers will lead to partial iteration.

Definition 4.3.2 (Partial iteration) *An iteration based on a partial semantic transition function F and a partial join $\amalg$ is called* partial iteration *if at least one of its iterates is not well-defined.*

Definition 4.3.3 (Total iteration) *An iteration based on a partial semantic transition function F and a partial join $\amalg$ is called* total iteration *if all its iterates are well-defined.*

The iteration is said to be convergent with limit F^ω whenever it is total and ultimatively stationary. From a convergent iteration it follows that $\exists\omega \in$ ordinal : such that $\forall n > \omega : \; F^n = F^\omega$.

The semantics $\mathcal{S}[\![\mathcal{P}]\!]$ of a program $\mathcal{P}$ can be expressed as [CC02][4]:

$$\mathcal{S}[\![\mathcal{P}]\!] = \mathrm{lfp}^{\sqsubseteq}_{\bot} F[\![\mathcal{P}]\!] \tag{4.2}$$

The symbol lfp itself denotes the least fixpoint of fixpoint iteration and is defined in Definition 4.3.4. The symbol $\mathrm{lfp}^{\sqsubseteq}_{\bot}$ represents a $\sqsubseteq$-*least fixpoint* which is defined in Definition 4.3.5 [Cou02].

Definition 4.3.4 (Least fixpoint) *The* least fixpoint lfp *of a monotone function $f : \mathfrak{D} \to \mathfrak{D}$ on a* poset $\langle\mathfrak{D},\sqsubseteq\rangle$ *is defined by* $\mathrm{lfp}\ f = f^\epsilon$ *where ϵ is the least ordinal such that $f(f^\epsilon) = f^\epsilon$.*

[4] The concrete notation depends on the type of semantics

Definition 4.3.5 (Restricted least/greatest fixpoint) $\mathrm{lfp}_{\ell}^{\sqsubseteq} F$ *is the* $\sqsubseteq$*-least fixpoint of* F *greater than or equal to* ℓ*, if it exists and dually,* $\mathrm{gfp}_{\ell}^{\sqsubseteq} F = \mathrm{lfp}_{\ell}^{\sqsupseteq} F$ *is the* $\sqsubseteq$*-greatest fixpoint of* F *less than or equal to* ℓ*, if it exists.*

The inductive join $\amalg$ can be defined as the least upper bound $\bigsqcup$ whereas $\bot$ may be then the infimum $\bot$. For the sequence definition given in Equation 4.1 the least upper bounds $\bigsqcup_{\beta<n} F^{\beta}$ are needed only for the iterations $F^n, n \leq \omega$ and not for all directed sets of $\mathfrak{D}$. This clearly shows that the domain $\langle \mathfrak{D}, \sqsubseteq \rangle$ does not necessarily need to be a cpo.

The iteration definition given by Equation 4.1 gives a sequence definition that leads to precise execution properties but at the same time to a potentially high computation effort. By merging sequences F^{β} more often (like for example $F^n = F(\amalg_{\beta<n} F^{\beta}) \amalg (\amalg_{\beta<n} F^{\beta})$ for all $n \in$ ordinal) the computation effort will be reduced and termination of the interpretation may be accelerated. But this comes at the cost of gaining less precise execution properties. An example for this is given in [Gus00] where the program analysis done by abstract interpretation is performed by merging the different execution traces after each loop iteration and for each loop exit over all iterations.

The following describes properties of semantics based on different common definitions of iteration sequences:

The Trace Semantics $\mathcal{S}_P^{\tau}$

As described in [Cou01] the *trace semantics* $\mathcal{S}_P^{\tau}$ is an execution of a program for a given initial state $c \in C_0$. The trace semantics is defined as sequences σ of states σ_i, observed at discrete intervals of time, starting from an initial state, then moving from one state to the next state by executing an atomic program step or transition and either ending in a final regular or erroneous state or nonterminating, in which the trace is infinite. The definition of different sequence types [Cou02] is given in Definition 4.3.6. An example of the execution sequences, computed for a set of initial states $\{a_1, a_2, \ldots, a6\}$ is given in Figure 4.8 (from [Cou01]).

Definition 4.3.6 (Sequences) *Based on the assumption that* A *is a nonempty alphabet we define the following types of sequences:*

- $A^{\vec{0}} \triangleq \{\vec{\epsilon}\}$ *where* $\vec{\epsilon}$ *is the empty sequence.*
- $A^{\vec{n}} | n > 0$ *is the set of finite sequences* $\sigma = \sigma_1, \ldots, \sigma_{n-1}$ *of length* $|\sigma| \triangleq n \in \mathbb{N}$ *over the alphabet* A*.*
- $A^{\vec{+}} \triangleq \bigcup_{n>0} A^{\vec{n}}$ *is the set of nonempty finite sequences over* A*.*
- $A^{\vec{*}} \triangleq A^{\vec{0}} \bigcup A^{\vec{+}}$ *is the set of finite sequences over* A*.*
- $A^{\vec{\omega}} \triangleq \mathbb{N} \mapsto A$ *is the set of infinite sequences* $\sigma_1, \ldots, \sigma_{n-1} \ldots$ *over* A*.*

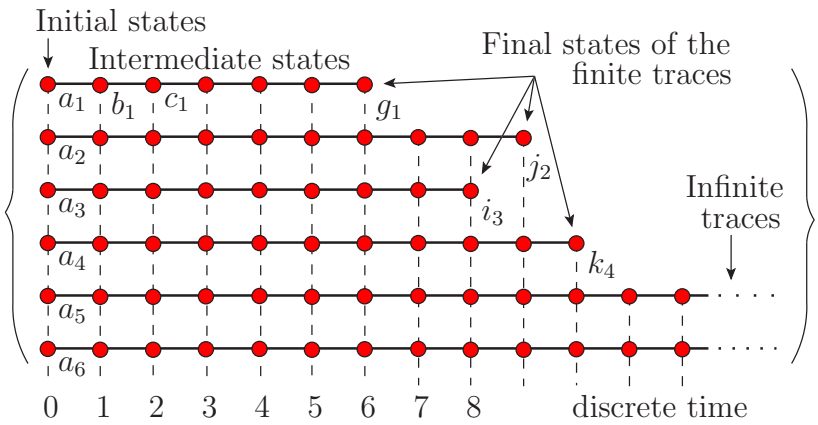


Figure 4.8: Computation Traces, recorded by the Trace Semantics $\mathcal{S}^{\tau}_{P}$

- $A^{\vec{\infty}} \triangleq A^{\vec{+}} \bigcup A^{\vec{\omega}}$ *is the set of all nonempty sequences over* A.
- $A^{\vec{\propto}} \triangleq A^{\vec{*}} \bigcup A^{\vec{\omega}}$ *is the set of all sequences over* A.

Based on the representation of the operational semantics $\mathcal{S}^{o}_{P}[\![]\!]$ as a mathematical machine $M = \langle C, \rightarrow, C_0, C_T \rangle$ (Section 4.2.1) we assume a transition system $\langle \Sigma, \tau \rangle$ where Σ is the set of possible states C and $\tau : \Sigma \times \Sigma \rightarrow \{\text{true}, \text{false}\}$ is the binary transition relation between a state and its possible successors. As in [Cou02] we formally define the maximal trace semantics $\mathcal{S}^{\tau}_{P} = \tau^{\vec{\infty}}$ of this transition system $\langle \Sigma, \tau \rangle$ by using the following trace semantics:

- $\tau^{\dot{\vec{n}}} \triangleq \{\sigma{=}\sigma_0 \ldots \sigma_{n-1} \in \Sigma^{\vec{n}} \mid \forall i < n-1 : \sigma_i \; \tau \; \sigma_{i+1}\}$ is the set of partial execution traces where σ_0 must be from the set of initial states C_0.
- $\tau^{\vec{n}} \triangleq \{\sigma{=}\sigma_0 \ldots \sigma_{n-1} \in \tau^{\dot{\vec{n}}} \mid \sigma_{n-1} \in C_T\}$ is the set of maximal/complete execution traces of length $n > 0$ terminating with a final/blocking state $\sigma_{n-1} \in C_T$.
- $\tau^{\vec{+}} \triangleq \bigcup\limits_{n>0} \tau^{\vec{n}}$ is the maximal finite trace semantics.
- $\tau^{\vec{\omega}} \triangleq \{\sigma \in \Sigma^{\vec{\omega}} \mid \forall i \in \mathbb{N} : \sigma_i \; \tau \; \sigma_{i+1}\}$ is the infinite trace semantics.
- $\tau^{\vec{\infty}} \triangleq \tau^{\vec{+}} \bigcup \tau^{\vec{\omega}}$ is the maximal trace semantics.

The transition function $\mathcal{S}^{o}_{S}[\![\xi]\!]\varepsilon$: Conf→Conf is defined for a concrete operational semantics (Section 4.2.1) and maps a configuration c to a new unique configuration c'. To also model the case where the result of a transition function is not simply a unique configuration $c = \langle \xi, \varepsilon \rangle$ we introduce a more generic transition function to define the maximal trace semantics $\mathcal{S}^{\tau}_{P} = \tau^{\vec{\infty}}$:

$$\mathcal{S}^{\tau}_{S}\text{: Conf}\rightarrow\wp(\text{Conf})$$

A transition function will result into multiple configurations for example in languages that have nondeterministic constructs. Another, for our context more interesting example, is the usage of an abstract semantics that uses a different definition for the content of a configuration. This is the case when using an interval of possible values instead of a concrete value in variable binding to reduce the resource requirement during program analysis by abstract interpretation. An analysis based on this semantic abstraction is used for example in [Gus00] to automatically derive the possible *control flow paths* through a given program.

The formal definition of the fixpoint trace semantics $\tau^{\vec{+}}$ for finite execution traces is given in Definition 4.3.7.

Definition 4.3.7 (Fixpoint finite trace semantics $\tau^{\vec{+}}$) *The fixpoint finite trace semantics $\tau^{\vec{+}}$ is defined as*

$$\tau^{\vec{+}} = lfp^{\subseteq}_{\emptyset} F^{\vec{+}} = gfp^{\subseteq}_{\Sigma^{\vec{+}}} F^{\vec{+}}$$

where $F^{\vec{+}}$: $\wp(\text{CONF}) \to \wp(\text{CONF})$ is defined as

$$F^{\vec{+}}[\![\tau]\!] \triangleq C_0 \cup \{\sigma s s' \mid \sigma s \in \tau \wedge s = \langle \xi, \varepsilon \rangle \wedge s' \in \mathcal{S}^{\tau}_{S}[\![\xi]\!]\varepsilon\} \quad .$$

Analogously, the formal definition of the fixpoint trace semantics $\tau^{\vec{\omega}}$ for infinite program execution is given in Definition 4.3.8.

Definition 4.3.8 (Fixpoint infinite trace semantics $\tau^{\vec{\omega}}$) *The fixpoint infinite trace semantics $\tau^{\vec{\omega}}$ is defined as*

$$\tau^{\vec{\omega}} = gfp^{\subseteq}_{\Sigma^{\vec{\infty}}} F^{\vec{\omega}} = lfp^{\subseteq}_{\Sigma^{\vec{\omega}}} F^{\vec{\omega}}$$

where $F^{\vec{\omega}}$: $\wp(\text{CONF}) \to \wp(\text{CONF})$ is defined as

$$F^{\vec{\omega}}[\![\tau]\!] \triangleq C_0 \cup \{\sigma s s' \mid \sigma s \in \tau \wedge s = \langle \xi, \varepsilon \rangle \wedge s' \in \mathcal{S}^{\tau}_{S}[\![\xi]\!]\varepsilon\} \quad .$$

Theorem 4.3.9 (Fixpoint fusion) *Let $\{D^+, D^\omega\}$ be a partition of D^∞ and $\langle \wp(D^+), \sqsubseteq^+ \rangle$ and $\langle \wp(D^\omega), \sqsubseteq^\omega \rangle$ be fixpoint semantics specifications. Using the following definitions:*

$$\begin{array}{rcl}
X^+ & \triangleq & X \cap D^+ \\
X^\omega & \triangleq & X \cap D^\omega \\
F^\infty(X) & \triangleq & F^+(X^+) \cap F^\omega(X^\omega) \\
X \sqsubseteq^\infty Y & \triangleq & X^+ \sqsubseteq^+ Y^+ \ \wedge \ X^\omega \sqsubseteq^\omega Y^\omega \\
\bot^\infty & \triangleq & \bot^+ \ \cap \ \bot^\omega \\
\top^\infty & \triangleq & \top^+ \ \cap \ \top^\omega \\
\sqcup^\infty_{i \in \Delta} X_i & \triangleq & \sqcup^+_{i \in \Delta} X_i^+ \ \cup \ \sqcup^\omega_{i \in \Delta} X_i^\omega \\
\sqcap^\infty_{i \in \Delta} X_i & \triangleq & \sqcap^+_{i \in \Delta} X_i^+ \ \cap \ \sqcap^\omega_{i \in \Delta} X_i^\omega
\end{array}$$

it follows:

- *if* $\langle \wp(D^+), \sqsubseteq^+ \rangle$ *and* $\langle \wp(D^\omega), \sqsubseteq^\omega \rangle$ *are* posets *(respectively complete lattices) then so is* $\langle \wp(D^\infty), \sqsubseteq^\infty \rangle$;
- *if* F^+ *and* F^ω *are monotone (resp. complete* $\sqcup$*-morphisms) then so is* F^∞;
- *if* $lfp^{\subseteq^+} F^+$ *and* $lfp^{\subseteq^\omega} F^\omega$ *are well-defined then* $lfp^{\subseteq^\infty} F^\infty = lfp^{\subseteq^+} F^+ \bigcup lfp^{\subseteq^\omega} F^\omega$;

Proof: given in annex C on page 189.

By using the results from Theorem 4.3.9 [Cou02], the formal definition of the fixpoint maximal trace semantics $\mathcal{S}^\tau_P = \tau^{\vec{\infty}}$ is given in Definition 4.3.10.

Definition 4.3.10 (Fixpoint maximal trace semantics $\tau^{\vec{\infty}}$) *The fixpoint maximal trace semantics* $\tau^{\vec{\infty}}$ *is defined as* $\tau^{\vec{\infty}} = lfpF^\infty = lfpF^+ \bigcup lfpF^\omega$.

The Collecting Semantics $\mathcal{S}^\mathcal{C}_P$

The *collecting semantics* $\mathcal{S}^\mathcal{C}_P$ is an abstraction of the *trace semantics* $\mathcal{S}^\tau_P$. The trace semantics collects the history of computation in an order-preserving manner. The collecting semantics is an abstraction by skipping the information of execution order. Instead, the collecting semantics collects all the configurations that can be reached during program execution.

An example of configurations, collected by the *collecting semantics* $\mathcal{S}^\mathcal{C}_P$ for a set of initial states $\{a_1, a_2, \ldots, a_6\}$ is given in Figure 4.9 (from [Cou01]).

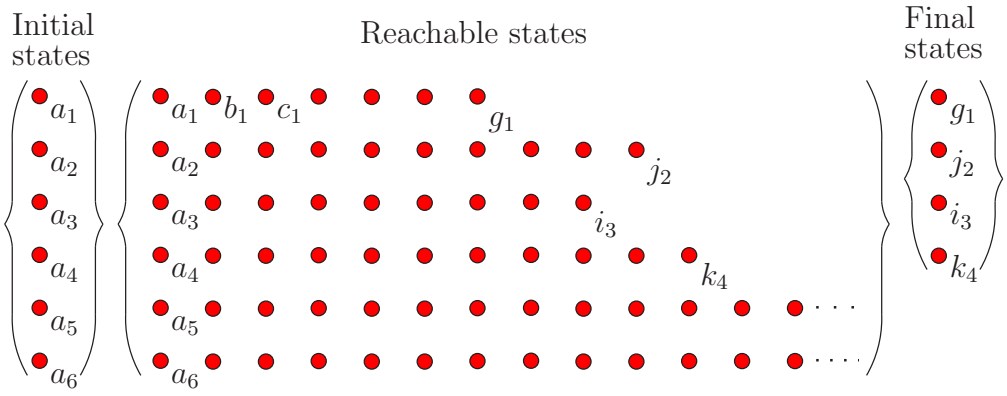


Figure 4.9: Set of Configurations, recorded by the Collecting Semantics $\mathcal{S}^\mathcal{C}_P$

A suitable domain for the calculation of the collecting semantics is the complete lattice $\langle \wp(\textsc{Conf}), \sqsubseteq \rangle$ with a set transfer function $F^\mathcal{C}$: $\wp(\textsc{Conf}) \to \wp(\textsc{Conf})$.

Similar to the transition function $\mathcal{S}^\tau_S$ of the trace semantics $\mathcal{S}^\tau_P$, the result of the transition function $\mathcal{S}^\mathcal{C}_S$ of the collecting semantics $\mathcal{S}^\mathcal{C}_P$ is not simply a single configuration: $\mathcal{S}^\mathcal{C}_S[\![\xi]\!]\varepsilon$: $\textsc{Conf} \to \wp(\textsc{Conf})$.

The formal definition of the collecting semantics $\mathcal{S}_P^{\mathcal{C}}$ is given in Definition 4.3.11.

Definition 4.3.11 (Collecting semantics $\mathcal{S}_P^{\mathcal{C}}$) *The collecting semantics $\mathcal{S}_P^{\mathcal{C}}[\![\xi_0]\!]\varepsilon_0$ for a program ξ and an initial configuration $c_0 = \langle \xi_0, \varepsilon_0 \rangle$ is defined as:*

$$\begin{aligned} \mathcal{S}_P^{\mathcal{C}}[\![\xi_0]\!]\varepsilon_0 &= \bigcup_{i \geq 0} C_i^{\mathcal{C}} \\ C_0^{\mathcal{C}} &= c_0 = \langle \xi_0, \varepsilon_0 \rangle \\ C_{i+1}^{\mathcal{C}} &= F^{\mathcal{C}}(C_i^{\mathcal{C}}) \\ F^{\mathcal{C}}(C) &= \{c \mid c' \in C \wedge c \in \mathcal{S}_S^{\mathcal{C}}\} \end{aligned}$$

where $\mathcal{S}_S^{\mathcal{C}}[\![\xi]\!]\varepsilon$ is the small step semantic function depending on the semantic computations.

Following from its formal definition, the collecting semantics can be directly derived from the trace semantics $\mathcal{S}_P^{\tau}$:

$$\mathcal{S}_P^{\mathcal{C}}[\![\xi]\!]\varepsilon = \bigcup_{\sigma \in \mathcal{S}_P^{\tau}[\![\xi]\!]\varepsilon} \left\{ \bigcup_{i=0}^{|\sigma|-1} \{\sigma_i \mid \sigma_i \in \sigma\} \right\} \tag{4.3}$$

The collecting semantics $\mathcal{S}_P^{\mathcal{C}}$ has the same calculation complexity as the trace semantics $\mathcal{S}_P^{\tau}$ and therefore cannot be calculated in general for programs using a semantic transfer function $\mathcal{S}_S^{\mathcal{C}}[\![\xi]\!]\varepsilon$ based on a concrete semantics.

The "Sticky" Collecting Semantics $\mathcal{S}_P^{\mathcal{SC}}$

The *collecting semantics* $\mathcal{S}_P^{\mathcal{C}}$ computes a set $C \subseteq \textsc{Conf}$ of potentially reachable configurations. An alternative representation is the "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$ (Definition 4.3.12)where each program point is associated with its possible set of environments.

Definition 4.3.12 ("Sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$) *The "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$ of a program ξ with an initial configuration ε is defined for each program point ξ_i as: $\mathcal{S}_P^{\mathcal{SC}}[\![\xi_i]\!]\varepsilon = \{\varepsilon' \mid \langle \xi_i, \varepsilon' \rangle \in \mathcal{S}_P^{\mathcal{C}}[\![\xi]\!]\varepsilon\}$*

A suitable domain for the calculation of the "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$ is the complete lattice $\langle \wp(\textsc{Pref} \times \wp(\textsc{Env})), \sqsubseteq \rangle$ with a set transfer function $F^{\mathcal{SC}}$: $\textsc{Pref} \times \wp(\textsc{Env}) \rightarrow \wp(\textsc{Pref} \times \wp(\textsc{Env}))$

The "sticky" collecting semantics can be calculated by solving a set of recursive equations that model the effect of all possible transitions to the environment at a certain program point. These equations given in Definition 4.4 are called *data flow equations*.

$$\mathcal{S}_P^{\mathcal{SC}}[\![\xi_i]\!] = f_i(\mathcal{S}_P^{\mathcal{SC}}[\![\xi_0]\!], \ldots, \mathcal{S}_P^{\mathcal{SC}}[\![\xi_n]\!]) \tag{4.4}$$

As shown in Definition 4.3.13 the *data flow equations* f_i can be derived from the semantic transition function $\mathcal{S}_S^{\mathcal{SC}}$ by collecting all environments that are the result of any transition to a program point ξ_i.

Definition 4.3.13 (Data flow equations f_i) *The* data flow equations *to solve the "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$ are defined as:*

$$\begin{aligned}\Sigma_i &= f_i(\Sigma_0, \ldots, \Sigma_n) \\ &= \bigcup_{j=0}^{n} \{\varepsilon' \mid \varepsilon \in \Sigma_i \wedge \langle \xi_i, \varepsilon' \rangle = \mathcal{S}_S^{\mathcal{SC}}[\![\xi_j]\!]\varepsilon\}\end{aligned}$$

where Σ_i denotes $\mathcal{S}_P^{\mathcal{SC}}[\![\xi_i]\!]$ and ξ_i, $0 \leq i \leq n$ are the discrete program points.

The set of recursive *data flow equations* can be solved with several methods. A quite efficient method is to solve them algebraically, but at the price of a high implementation effort.

An alternative are iterative solutions that are much more simple to implement. The simplest iterative solution method is the *Jacobi iteration*. Given the above *data flow equations*, the "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$ can be solved as *Jacobi iteration* using the following iterative definition (the index j denotes the iteration counter):

$$\begin{aligned}\mathcal{S}_P^{\mathcal{SC},0}[\![\xi_i]\!] &= \emptyset \\ \mathcal{S}_P^{\mathcal{SC},j+1}[\![\xi_i]\!] &= \bigcup_{0 \leq i \leq n} f_i(\mathcal{S}_P^{\mathcal{SC},j}[\![\xi_0]\!], \ldots, \mathcal{S}_P^{\mathcal{SC},j}[\![\xi_n]\!])\end{aligned} \tag{4.5}$$

The "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}[\![\xi_i]\!]$, $0 \leq i \leq n$ is obtained as the least upper bound (lub) of the following ascending chain:

$$\begin{array}{l}\mathcal{S}_P^{\mathcal{SC},0}[\![\xi_i]\!],\ \forall 0 \leq i \leq n \\ \mathcal{S}_P^{\mathcal{SC},1}[\![\xi_i]\!],\ \forall 0 \leq i \leq n \\ \mathcal{S}_P^{\mathcal{SC},2}[\![\xi_i]\!],\ \forall 0 \leq i \leq n \\ \vdots\end{array}$$

The *Jacobi iteration* does not exploit the structure of the program to be analysed and therefore usually provides poor performance. There are more efficient methods to calculate the fixpoint solution. One of them is *chaotic iteration* as described in [Nil92] (chapter 6.2).

Summary of the Fixpoint Semantics

A common property of the trace semantics $\mathcal{S}_P^{\mathcal{T}}$, the collecting semantics $\mathcal{S}_P^{\mathcal{C}}$ and the "sticky" collecting semantics $\mathcal{S}_P^{\mathcal{SC}}$ is that they are precise in the sense of collecting

all environments they encountered during interpretation. Therefore, they cannot be calculated for semantic domains of quite large or even infinite sizes. To overcome this limitation, one has to use an abstract interpretation method that uses a smaller domain at the cost of loosing precision due to data abstraction. As already mentioned in the beginning of this section, an alternative is to use more often the inductive join operator $\amalg$ during interpretation.

4.3.5 Approximate Abstract Interpretation

As discussed for fixpoint semantics (Section 4.3.4), the program interpretation based on a concrete semantics may not be computable in general. As proposed in [CC77, CC92b, CC92a], the solution is to approximate the interpretation based on the concrete domain $\langle \mathfrak{D}, \sqsubseteq \rangle$ by an interpretation based on a simpler domain $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$.

To bring both interpretation methods in context to each other, their domains $\mathfrak{D}$ and $\widetilde{\mathfrak{D}}$ are connected by the functions $\alpha : \mathfrak{D} \to \widetilde{\mathfrak{D}}$ and $\gamma : \widetilde{\mathfrak{D}} \to \mathfrak{D}$. α is called the *abstraction function* and γ is called the *concretization function*. This principle of connecting domains is shown in Figure 4.10.

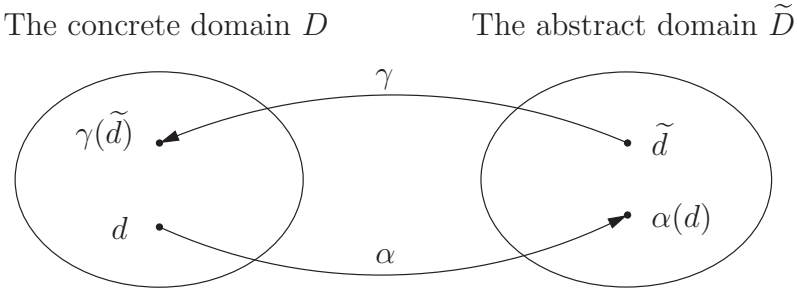


Figure 4.10: Connection of Domains

The concrete interpretation is denoted as $\langle \langle \mathfrak{D}, \sqsubseteq \rangle, F \rangle$. The abstract interpretation has to be designed following the steps 1 - 5 described in Section 4.3.2 with the requirement that the resulting abstract interpretation

***is safe*:** the calculated abstract results mapped back to the concrete domain always have to be a safe approximation of the concrete value.

***terminates*:** abstract interpretation used as program analysis should be able to provide information about the properties of the concrete interpretation, even in the case the concrete interpretation does not terminate.

Efficiency Aspects of Abstract Interpretation

Beside the functional requirements it is also important to consider low resource usage to make the abstract interpretation feasible, even for large programs. Therefore, the

approximate semantic function $\widetilde{F}$ and the mapping functions (α, β) between the concrete and the abstract domain have to be designed as compact as possible to achieve efficient calculation and increased termination speed.

Precision Aspects of Abstract Interpretation

Performing abstract interpretation usually requires to obtain very precise information about the execution behaviour of a program. This requires to design the abstract domain $\widetilde{\mathfrak{D}}$ to contain at least all the required information. The approximate semantic function $\widetilde{F}$ and the domain mapping functions α, β have to be designed to provide tight results.

The efficiency and precision aspects induce design requirements that are contradicting each other. Therefore, a trade-off often has to be made between efficiency and precision.

4.3.6 Correctness of Abstract Interpretation

The correctness of an abstract interpretation $\langle\langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle, \widetilde{F}\rangle$ of a program is measured against the concrete semantics of this program. We denote the concrete semantics here by its semantic transition function $F : \mathfrak{D} \to \mathfrak{D}$. We call elements $d \in \mathfrak{D}$ *values* of the program. Elements $\widetilde{d} \in \widetilde{\mathfrak{D}}$ are called *properties* of the program. The semantic transfer function

$$F : \mathfrak{D}_1 \to \mathfrak{D}_2; \qquad \mathfrak{D}_1, \mathfrak{D}_2 \subseteq \mathfrak{D}$$

calculates the program value $d_2 \in \mathfrak{D}_2$ based on an initial value $d_1 \in \mathfrak{D}_1$. In a similar way, program analysis based on the abstract semantic transfer function

$$\widetilde{F} : \widetilde{\mathfrak{D}}_1 \to \widetilde{\mathfrak{D}}_2; \qquad \widetilde{\mathfrak{D}}_1, \widetilde{\mathfrak{D}}_2 \subseteq \widetilde{\mathfrak{D}}$$

calculates how the program transforms a property $\widetilde{d}_1 \in \widetilde{\mathfrak{D}}_1$ into a property $\widetilde{d}_2 \in \widetilde{\mathfrak{D}}_2$.

For expressing correctness, we have to differ between the following two types of program analysis based on abstract interpretation:

***first-order analysis*:** program properties directly describe sets of program values. Examples are: *Constrant Propagation Analysis* or *Control Flow Analysis*.

***second-order analysis*:** program properties are related to relations between program values. An example of this is *Live Variable Analysis*.

In the following section we will describe correctness for both types of analyses.

Correctness for Single-Order Analysis

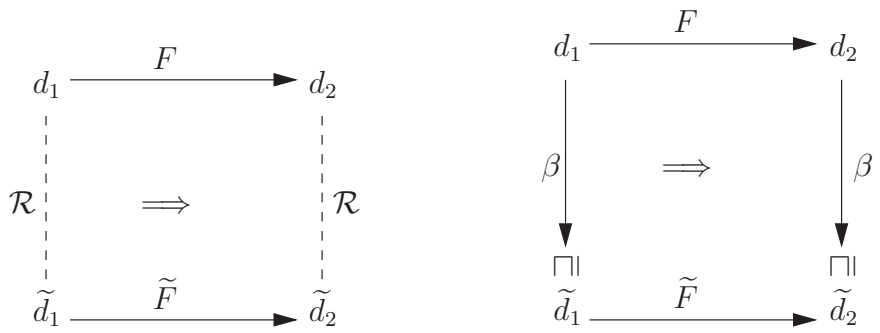
For single-order analysis, correctness is established by directly relating properties to program values using a *correctness relation*

$$\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \to \{\text{true}, \text{false}\}$$

It has to be proven that the correctness relation is preserved under computation: if the relation holds between the initial value and the initial property then it also holds between the final value and the final property. This can be formulated by the following implication:

$$d_1 \,\mathcal{R}\, \widetilde{d}_1 \;\wedge\; d_2 = F(d_1) \;\wedge\; \widetilde{d}_2 = \widetilde{F}(\widetilde{d}_1) \quad\Rightarrow\quad d_2 \,\mathcal{R}\, \widetilde{d}_2 \tag{4.6}$$

and is expressed by Figure 4.11(a).



(a) Correctness Relation (b) Representation Function

Figure 4.11: Correctness for Single-Order Analysis

To show correctness of an abstract interpretation we have to relate elements of the abstract domain to the correctness relation. For the most common scenario that $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ is a complete lattice $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}, \widetilde{\sqcup}, \widetilde{\sqcap}, \widetilde{\top}, \widetilde{\bot} \rangle$ we impose the following relationship between $\mathcal{R}$ and $\widetilde{\mathfrak{D}}$ [NNH99]:

$$d \,\mathcal{R}\, \widetilde{d} \;\wedge\; \widetilde{d} \,\widetilde{\sqsubseteq}\, \widetilde{d}' \;\Rightarrow\; d \,\mathcal{R}\, \widetilde{d}' \tag{4.7}$$

$$(\forall \widetilde{d} \in \widetilde{\mathfrak{D}}' \subseteq \widetilde{\mathfrak{D}} : \; d \,\mathcal{R}\, \widetilde{d}) \;\Rightarrow\; d \,\mathcal{R}\, (\widetilde{\sqcap} \widetilde{\mathfrak{D}}') \tag{4.8}$$

Equation 4.7 shows that the smaller a property is with respect to the partial ordering, the more precise it is. However, much data flow analysis algorithms in the literature denote a greater property to be more precise. This is no problem since the principle of duality from lattice theory shows that these representations can be mapped into each other by an *isomorphism* (see Definition B.2.12 on page 187).

An alternative approach to the use of the correctness relation $\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \rightarrow \{\text{true}, \text{false}\}$ is to use a *representation function*

$$\beta : \mathfrak{D} \rightarrow \widetilde{\mathfrak{D}}$$

that maps a value to the best program property describing it (as shown in Figure 4.11(b)). The correctness criterion for the analysis will then be formulated as given in

$$\beta(d_1) \mathbin{\tilde{\sqsubseteq}} \widetilde{d}_1 \ \wedge \ d_2 = F(d_1) \ \wedge \ \widetilde{d}_2 = \widetilde{F}(\widetilde{d}_1) \ \Rightarrow \ \beta(d_2) \mathbin{\tilde{\sqsubseteq}} \widetilde{d}_2 \tag{4.9}$$

To show the equivalence between the correctness formulations [NNH99] based on $\mathcal{R}$ and β, we show how to define a correctness relation $\mathcal{R}_\beta$ from a given representation function β:

$$d \, \mathcal{R}_\beta \, \widetilde{d} \quad \text{iff} \quad \beta(d) \mathbin{\tilde{\sqsubseteq}} \widetilde{d}$$

A representation function $\beta_\mathcal{R}$ can be defined from a correctness relation $\mathcal{R}$ as follows:

$$\beta_\mathcal{R}(d) \quad = \quad \sqcap\{\widetilde{d} \mid d \, \mathcal{R} \, \widetilde{d}\}$$

Lemma 4.3.14 (Equivalence of $\mathcal{R}$ and β)

(i) Given $\beta : \mathfrak{D} \rightarrow \widetilde{\mathfrak{D}}$, then the relation $\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \rightarrow \{true, false\}$ satisfies the conditions given in Equation 4.7 and Equation 4.8, and furthermore $\beta_{\mathcal{R}_\beta} = \beta$.

(ii) Given $\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \rightarrow \{true, false\}$ that satisfies Equation 4.7 and Equation 4.8, it follows that $\beta_\mathcal{R}$ is well-defined and $\mathcal{R}_{\beta_\mathcal{R}} = \mathcal{R}$.

Proof: given in annex C on page 189.

The generation of the correctness relation $\mathcal{R}_\beta$ from the representation function β is visualised in Figure 4.12.

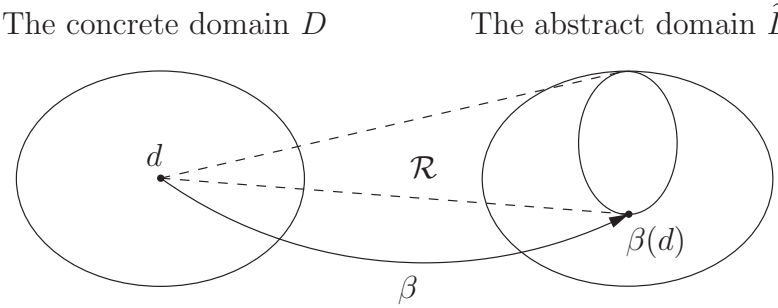


Figure 4.12: Correctness Relation $\mathcal{R}$ generated by Representation Function β

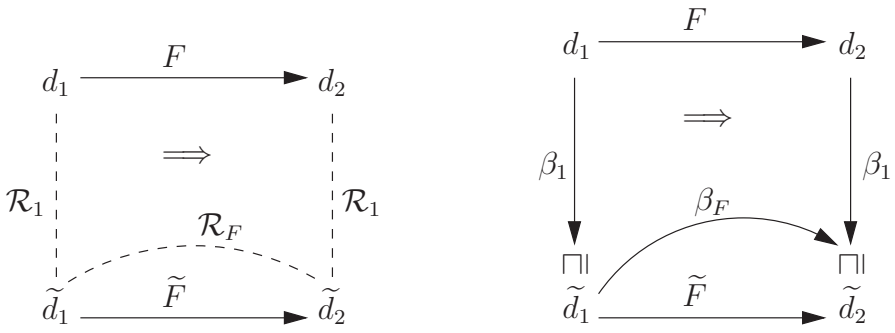
Correctness for Second-Order Analysis

For second-order analysis we have to use two different correctness relations, $\mathcal{R}_1$ for the initial program properties and $\mathcal{R}_2$ for the computed properties:

$$\begin{aligned} &\mathcal{R}_1 : \mathfrak{D}_1 \times \widetilde{\mathfrak{D}}_1 \to \{\text{true}, \text{false}\}, \qquad \text{generated by } \beta_1 : \mathfrak{D}_1 \to \widetilde{\mathfrak{D}}_1 \\ &\mathcal{R}_2 : \mathfrak{D}_2 \times \widetilde{\mathfrak{D}}_2 \to \{\text{true}, \text{false}\}, \qquad \text{generated by } \beta_2 : \mathfrak{D}_2 \to \widetilde{\mathfrak{D}}_2 \end{aligned}$$

The relations $\mathcal{R}_1$ and $\mathcal{R}_2$ are expressed in Figure 4.13(a). Based on them we define the correctness of the semantic function $\widetilde{F}$ as follows:

$$\widetilde{d_1}\, \mathcal{R}_F\, \widetilde{F}(\widetilde{d_1}) \quad \Leftrightarrow \quad \forall d_1, d_2 : d_1\, \mathcal{R}_1\, \widetilde{d_1} \ \wedge\ d_2 = F(d_1) \ \wedge\ d_2\, \mathcal{R}_2\, \widetilde{F}(\widetilde{d_1})$$



(a) Correctness Relations (b) Representation Functions

Figure 4.13: Correctness for Second-Order Analysis

The correctness relation $\mathcal{R}_F$ can be generated from β_F, which is defined as:

$$\beta_F(\widetilde{d_1}) \quad = \quad \forall d_1 : \sqcup\{\beta_2(d_2) \,|\, \beta_1(d_1) \mathrel{\tilde{\sqsubseteq}} \widetilde{d_1} \ \wedge\ d_2 = F(d_1)\}$$

The representation function β_F is expressed in Figure 4.13(b). Lemma 4.3.15 [NNH99] shows that $\mathcal{R}_F$ generated from β_F defines a correctness relation.

Lemma 4.3.15 ($\mathcal{R}_F$ is a correctness relation, generated from β_F) *If $\mathcal{R}_i$ is a correctness relation for $\mathfrak{D}_i$ and $\widetilde{\mathfrak{D}}_i$ that is generated by the representation function $\beta_i : \mathfrak{D}_i \to \widetilde{\mathfrak{D}}_i$, $i \in \{1, 2\}$ then $\mathcal{R}_F$ is a correctness relation and it is generated by the representation function β_F.*

Proof: given in annex C on page 189.

4.3.7 Galois Connection

The Galois connection framework provides constructive methods to design a correct abstract interpretation based on a concrete interpretation. The domain $\mathfrak{D}$ of the concrete interpretation $\langle\langle\mathfrak{D}, \sqsubseteq\rangle, F\rangle$ and the domain $\widetilde{\mathfrak{D}}$ of the abstract interpretation $\langle\langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle, \widetilde{F}\rangle$ are assumed to be *partial ordered sets* with having for every element $d \in \mathfrak{D}$ a best approximation in $\widetilde{\mathfrak{D}}$.

The correspondence between the concrete and the abstract domain is given by a Galois connection $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$ using the following functions:

- $\alpha : \mathfrak{D} \to \widetilde{\mathfrak{D}}$ is the abstraction function. Its purpose is to map elements from $\mathfrak{D}$ to approximated elements in $\widetilde{\mathfrak{D}}$ while keeping the partial order. Hence, if an abstract element $\alpha(d) \in \widetilde{\mathfrak{D}}$ is an approximation of $d \in \mathfrak{D}$ and $\alpha(d) \mathrel{\widetilde{\sqsubseteq}} \widetilde{d}$ then $\widetilde{d}$ is also a correct, but possibly less precise approximation of d. The fact that an element $\widetilde{d} \in \widetilde{\mathfrak{D}}$ is a valid approximation of $d \in \mathfrak{D}$ is expressed as $\alpha(d) \mathrel{\widetilde{\sqsubseteq}} \widetilde{d}$ (applications using greater values for more precise results can be directly applied by an *isomorphism* due to the principle of duality from lattice theory; as shown in Definition B.2.12 on page 187).
- $\gamma : \widetilde{\mathfrak{D}} \to \mathfrak{D}$ is the concretization function. Its purpose is to map elements from $\widetilde{\mathfrak{D}}$ to elements in $\mathfrak{D}$ while keeping the partial order. Assuming $\gamma(\widetilde{d})$ is a concretization of $\widetilde{d} \in \widetilde{\mathfrak{D}}$ and $d \sqsubseteq \gamma(\widetilde{d})$ then $\widetilde{d}$ is a correct approximation of the concrete value d, although d may give more precise information than $\gamma(\widetilde{d})$. The fact that an element $\widetilde{d} \in \widetilde{\mathfrak{D}}$ is a valid approximation of $d \in \mathfrak{D}$ is expressed as $d \sqsubseteq \gamma(\widetilde{d})$ (respective the opposite for the dual representation of a lattice, see the function α above).

The Galois connection does not lose safety by going back and forth between the two domains, although precision may be lost. The formal definition of a Galois connection is given in Definition 4.3.16.

Definition 4.3.16 (Galois connection) *A* Galois connection $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$ *between the two domains* $\langle\mathfrak{D}, \sqsubseteq\rangle$ *and* $\langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$ *is defined by two monotone functions* $\alpha \in \mathfrak{D} \rightarrowtail \widetilde{\mathfrak{D}}$ *and* $\gamma \in \widetilde{\mathfrak{D}} \rightarrowtail \mathfrak{D}$ *iff*

$$\forall d \in \mathfrak{D} \wedge \forall \widetilde{d} \in \widetilde{\mathfrak{D}} : \ \alpha(\gamma(\widetilde{d})) \mathrel{\widetilde{\sqsubseteq}} \widetilde{d} \ \wedge \ d \sqsubseteq \gamma(\alpha(d)) \tag{4.10}$$

The definition of the Galois connection is illustrated in Figure 4.14. Starting from the domain $\mathfrak{D}$, we have a class of elements $d \in \mathfrak{D}' \subseteq \mathfrak{D}$ that map to a unique element $\alpha(d)$. But going back again we get the element $\gamma(\alpha(d))$, which is an upper bound of all elements in $\mathfrak{D}'$: $\forall d, d' \in \mathfrak{D}' : d' \sqsubseteq \gamma(\alpha(d))$. The consequence is that we lose precision

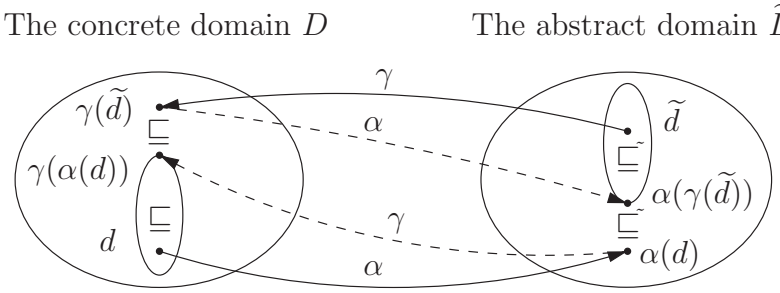


Figure 4.14: Galois Connection

by stepping into and back from $\widetilde{\mathfrak{D}}$. A dual relation happens when starting from the abstract domain. In this case $\alpha(\gamma(\widetilde{d}))$ will result into a lower bound of all elements $\widetilde{d} \in \widetilde{\mathfrak{D}}$ that map to the same element $\gamma(\widetilde{d})$. Figure 4.14 also shows that α and γ are monotone functions.

There is also an alternative formulation of the Galois connection $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ that is frequently easier to work with [NNH99]. $(\mathfrak{D} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \widetilde{\mathfrak{D}})$ is defined as an *adjunction* (Definition 4.11) between the domains $\langle \mathfrak{D}, \sqsubseteq \rangle$ and $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ iff the mapping functions α and γ are "inverse" to each other.

Definition 4.3.17 (Adjunction) *An* adjunction $(\mathfrak{D} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \widetilde{\mathfrak{D}})$ *is a pair of* total functions $\langle \alpha, \gamma \rangle$ *between the domains* $\langle \mathfrak{D}, \sqsubseteq \rangle$ *and* $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *iff*

$$\forall d \in \mathfrak{D} \wedge \forall \widetilde{d} \in \widetilde{\mathfrak{D}} : \ \alpha(d) \mathbin{\widetilde{\sqsubseteq}} \widetilde{d} \Longleftrightarrow d \sqsubseteq \gamma(\widetilde{d}) \tag{4.11}$$

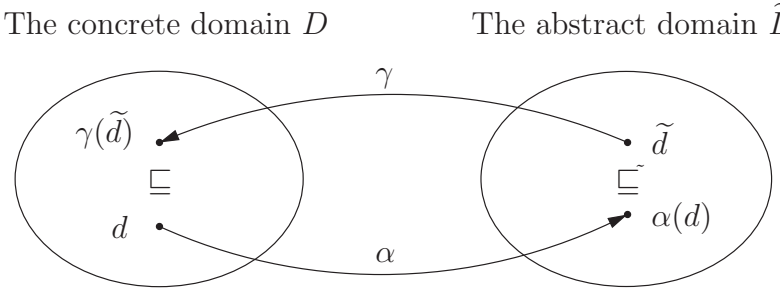


Figure 4.15: Adjoined Functions

The definition of the adjunction is illustrated in Figure 4.15. The equivalence between the Galois connection and the *adjunction* is given in Theorem 4.3.18.

Theorem 4.3.18 (Equivalence of adjunction and Galois connection) *An adjunction* $(\mathfrak{D} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \widetilde{\mathfrak{D}})$ *is also a Galois connection* $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *and vice versa.*

Proof: given in annex C on page 189.

Properties of Galois Connections

As given in [NNH99], a Galois connection provides additional properties, which are listed in the following.

Lemma 4.3.19 (Interrelation between α and γ) *If* $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *is a Galois connection then*

(i) α *uniquely determines* γ *by* $\gamma(\widetilde{d}) = \sqcup\{d \mid \alpha(d) \mathrel{\widetilde{\sqsubseteq}} \widetilde{d}\}$ *and* γ *uniquely determines* α *by* $\alpha(d) = \sqcap\{\widetilde{d} \mid d \sqsubseteq \gamma(\widetilde{d})\}$.

(ii) α *is completely additive and* γ *is completely multiplicative.*

Proof: given in annex C on page 190.

As shown by Lemma 4.3.20, it is sufficient to specify either a completely additive abstraction function α or a completely multiplicative concretization function γ in order to obtain a Galois connection $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$.

Lemma 4.3.20 (Existence of α respectively γ) *If* $\alpha : \mathfrak{D} \to \widetilde{\mathfrak{D}}$ *is a completely additive function then there exists a function* $\gamma : \widetilde{\mathfrak{D}} \to \mathfrak{D}$ *such that* $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *is a Galois connection. Dually, if* $\gamma : \widetilde{\mathfrak{D}} \to \mathfrak{D}$ *is completely multiplicative then there exists* $\alpha : \mathfrak{D} \to \widetilde{\mathfrak{D}}$ *such that* $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *is a Galois connection.*

Proof: given in annex C on page 190.

Lemma 4.3.21 shows that we do not lose or gain precision by iterating abstraction and concretisation.

Lemma 4.3.21 (Iteration invariance of the Galois connection) *If* $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *is a Galois connection, then*

$$\alpha \circ \gamma \circ \alpha = \alpha \text{ and } \gamma \circ \alpha \circ \gamma = \gamma \quad .$$

Proof: given in annex C on page 191.

Construction of a Galois Connection

In this section we describe how to construct a Galois connection by using a representation function β (see Section 4.3.6). A Galois connection $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ for a concrete domain $\mathfrak{D}$ can be constructed systematically following steps 1 - 5:

1. Start with the concrete domain $\mathfrak{D}$ having the partial order $\sqsubseteq$. $\mathfrak{D}$ and $\widetilde{\mathfrak{D}}$ must be posets.
2. Design the corresponding abstract domain to describe the desired properties of the concrete domain.
3. Specify a correctness relation $\mathcal{R} : \mathfrak{D} \times \widetilde{\mathfrak{D}} \to \{\text{true}, \text{false}\}$ to calculate the resulting representation function $\beta : \mathfrak{D} \to \widetilde{\mathfrak{D}}$.
4. If $\mathfrak{D}$ and $\widetilde{\mathfrak{D}}$ are required to be a complete lattice, check whether $\widetilde{\mathfrak{D}}$ and β are defined so that the greatest lower bound $\sqcap$ and the least upper bound $\sqcup$ exists for all subsets $\widetilde{\mathfrak{D}}' \subseteq \widetilde{\mathfrak{D}}$. This ensures that $\widetilde{\mathfrak{D}}$ is also a complete lattice.
   For simple posets it is sufficient to verify that $\forall d \in \mathfrak{D}, \forall \widetilde{d} \in \widetilde{\mathfrak{D}}, \exists \widetilde{d}_1 \in \widetilde{\mathfrak{D}} : \; d \mathcal{R} \widetilde{d}_1 \wedge (d \mathcal{R} \widetilde{d} \Rightarrow \widetilde{d}_1 \mathbin{\tilde{\sqsubseteq}} \widetilde{d})$.
5. Calculate the abstraction function α and the concretisation function γ from the function β according to Equation 4.12 and Equation 4.13 (the lub operator $\sqcup$ in Equation 4.13 is also guaranteed to be defined for simple posets due to the construction method used for $\mathcal{R}$).

$$\alpha(d) = \beta(d) \tag{4.12}$$

$$\gamma(\widetilde{d}) = \sqcup\{d \in \mathfrak{D} \mid \beta(d) \mathbin{\tilde{\sqsubseteq}} \widetilde{d}\} \tag{4.13}$$

After the calculation of the functions α and γ we have already constructed the Galois connection.

The representation function β in step 3) has to be defined according the structure of the concrete and abstract domain. For example, if we want to construct a Galois connection like $\langle \wp(\mathfrak{D}), \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ we can first define an auxiliary representation function $\beta' : \mathfrak{D} \to \widetilde{\mathfrak{D}}$. Then we can lift β' to the full powerset $\wp(\mathfrak{D})$ as given in Equation 4.14 to finally calculate the representation function $\beta_{\wp(\mathfrak{D})}$.

$$\beta_{\wp(\mathfrak{D})}(\mathfrak{D}') = \sqcup\{\beta'(d) \mid d \in \mathfrak{D}'\} \qquad \mathfrak{D}' \in \wp(\mathfrak{D}) \tag{4.14}$$

Another example would be, having a concrete domain $\mathfrak{D}$ that consists of several components $\mathfrak{D}_i$ that will be described by corresponding components $\widetilde{\mathfrak{D}}_i$ of the abstract domain $\widetilde{\mathfrak{D}}$. Then it is possible to construct a Galois connection for each component and finally combine them to an overall Galois connection (see Section 4.3.11 for more details).

Galois Insertions

Having a Galois connection $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$, there may exist several elements of $\widetilde{\mathfrak{D}}$ that map to the same element in $\mathfrak{D}$. To be more precise, the function γ may be not injective, i.e. there exist elements in $\widetilde{\mathfrak{D}}$ that are not relevant for the abstraction of $\mathfrak{D}$.

The framework of Galois insertion $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ is intended to rectify this situation. A Galois insertion is a Galois connection where α is a monotone surjection or as an equivalent requirement, γ is a monotone injection. The fact that γ is injective is denoted by the two-headed arrow $\twoheadleftarrow$. A formal definition of the Galois insertion is given in Definition 4.3.22, with an illustration of the principle shown in Figure 4.16.

Definition 4.3.22 (Galois insertion) *A* Galois insertion $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *between the two domains* $\langle \mathfrak{D}, \sqsubseteq \rangle$ *and* $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *is defined by two monotone functions* $\alpha \in \mathfrak{D} \mapsto \widetilde{\mathfrak{D}}$ *and* $\gamma \in \widetilde{\mathfrak{D}} \mapsto \mathfrak{D}$ *iff*

$$\forall d \in \mathfrak{D} \wedge \forall \widetilde{d} \in \widetilde{\mathfrak{D}} : \ \widetilde{d} = \alpha(\gamma(\widetilde{d})) \ \wedge \ d \sqsubseteq \gamma(\alpha(d)) \tag{4.15}$$

A Galois insertion is useful to approximate over the concrete domain $\mathfrak{D}$, since all elements $\widetilde{d} \in \widetilde{\mathfrak{D}}$ describe different elements $d \in \mathfrak{D}$. With a Galois insertion, the concretization function does not loose information, i.e., applying the abstraction function afterwards will result to the original element in $\widetilde{\mathfrak{D}}$.

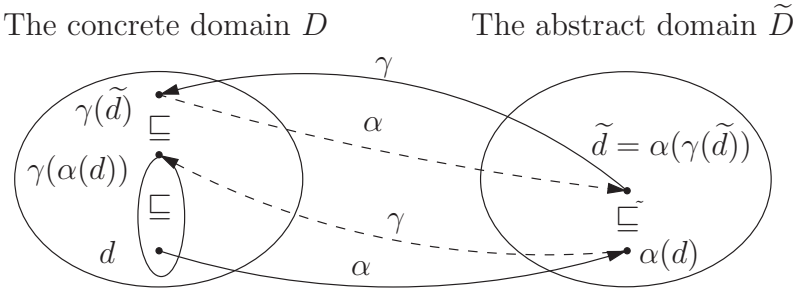


Figure 4.16: Galois Insertion

As shown in [NNH99], having a Galois connection, it is always possible to construct a Galois insertion by enforcing that the concretization function γ is injective. Following the definition of the reduction operator $\varsigma : \widetilde{\mathfrak{D}} \to \widetilde{\mathfrak{D}}$ in Proposition 4.3.23, this is basically done by removing all the superfluous elements from $\widetilde{\mathfrak{D}}$.

Proposition 4.3.23 *Having a Galois connection* $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq} \rangle$ *between the complete lattices* $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ *and* $\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}, \widetilde{\sqcup}, \widetilde{\sqcap}, \widetilde{\top}, \widetilde{\bot} \rangle$ *and a reduction operator*

$\varsigma : \widetilde{\mathfrak{D}} \to \widetilde{\mathfrak{D}}$ *which is defined as*

$$\varsigma(\widetilde{d}) = \sqcap\{\widetilde{d'} \mid \gamma(\widetilde{d}) = \gamma(\widetilde{d'})\}$$

then $\varsigma[\widetilde{\mathfrak{D}}] = \langle\{\varsigma(\widetilde{d}) \mid \widetilde{d} \in \widetilde{\mathfrak{D}}\}, \widetilde{\sqsubseteq}\rangle$ *is a complete lattice and* $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle\varsigma(\widetilde{\mathfrak{D}}), \widetilde{\sqsubseteq}\rangle$ *is a Galois insertion.*

Galois Isomorphism

The Galois insertion provides a framework with an efficient domain for representing the approximated values. However, for some applications it makes sense to have an abstract domain $\widetilde{\mathfrak{D}}$ representing elements of the concrete domain $\widetilde{\mathfrak{D}}$ without loosing precision. Such a framework will be called Galois isomorphism $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$. Its formal definition is given in Definition 4.3.24 and is illustrated in Figure 4.17.

Definition 4.3.24 (Galois isomorphism) *A* Galois isomorphism $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\leftrightarrows}} \langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$ *between the two domains* $\langle\mathfrak{D}, \sqsubseteq\rangle$ *and* $\langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$ *is defined by two monotone functions* $\alpha \in \mathfrak{D} \mapsto \widetilde{\mathfrak{D}}$ *and* $\gamma \in \widetilde{\mathfrak{D}} \mapsto \mathfrak{D}$ *iff*

$$\forall d \in \mathfrak{D} \wedge \forall \widetilde{d} \in \widetilde{\mathfrak{D}} : \ \widetilde{d} = \alpha(\gamma(\widetilde{d})) \ \wedge \ d = \gamma(\alpha(d)) \tag{4.16}$$

A typical application for using a Galois isomorphism framework is to apply the formalism of abstract interpretation where larger values mean less precision, to the category of classical data flow analysis frameworks, where smaller values represent less precision. For more details about the principle of lattice duality of lattice theory see Definition B.2.12 on page 187.

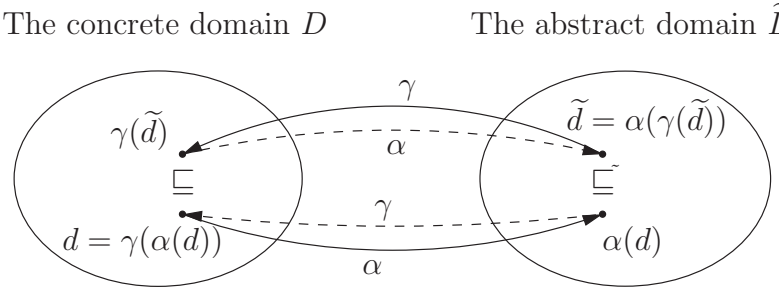


Figure 4.17: Galois Isomorphism

Another application of a Galois isomorphism would be a concrete domain consisting of combined subdomains where one component is directly mapped to the abstract domain without loosing precision. An example using this method would be an abstract interpretation based on additional program properties that have been derived by a prior static analysis.

4.3.8 The Safety of the Approximation

The term *correctness* introduced in Section 4.3.6 is used to specify whether an abstract interpretation models the semantics of a program correctly. We use the term *safety* when we compare two interpretations to show that the properties resulting from the one interpretation correctly describe the properties of the other. Therefore, *safety* is herewith used as a special case of correctness.

Using a Galois connection between two abstract interpretations we can say that one is an abstraction of the other, formally described in Definition 4.3.25.

Definition 4.3.25 ($\langle\alpha,\gamma\rangle$-abstraction) *Let $\langle\mathfrak{D},\sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle\widetilde{\mathfrak{D}},\widetilde{\sqsubseteq}\rangle$ be a Galois connection between the two abstract interpretations $\langle\langle\mathfrak{D},\sqsubseteq\rangle,F\rangle$ and $\langle\langle\widetilde{\mathfrak{D}},\widetilde{\sqsubseteq}\rangle,\widetilde{F}\rangle$. Then $\langle\langle\widetilde{\mathfrak{D}},\widetilde{\sqsubseteq}\rangle,\widetilde{F}\rangle$ is said to be an $\langle\alpha,\gamma\rangle$-abstraction of $\langle\langle\mathfrak{D},\sqsubseteq\rangle,F\rangle$.*

Having an interpretation $\langle\langle\mathfrak{D},\sqsubseteq\rangle,F\rangle$ the Galois connection framework can be used to construct $\langle\langle\widetilde{\mathfrak{D}},\widetilde{\sqsubseteq}\rangle,\widetilde{F}\rangle$, a safe approximation of $\langle\langle\mathfrak{D},\sqsubseteq\rangle,F\rangle$. The safety of the approximation is mandatory to be able to derive valid program properties. We therefore use the following definition to describe the safety of an approximation:

Definition 4.3.26 (Safe γ-approximation) *Let $\langle\langle\widetilde{\mathfrak{D}},\widetilde{\sqsubseteq}\rangle,\widetilde{F}\rangle$ be an $\langle\alpha,\gamma\rangle$-abstraction of $\langle\langle\mathfrak{D},\sqsubseteq\rangle,F\rangle$. $\langle\langle\widetilde{\mathfrak{D}},\widetilde{\sqsubseteq}\rangle,\widetilde{F}\rangle$ is said to be a safe γ-approximation of $\langle\langle\mathfrak{D},\sqsubseteq\rangle,F\rangle$ iff*

$$\forall\widetilde{d}\in\widetilde{\mathfrak{D}}:\ F(\gamma(\widetilde{d}))\sqsubseteq\gamma(\widetilde{F}(\widetilde{d}))$$

The safety of abstract interpretations is visualised in Figure 4.18.

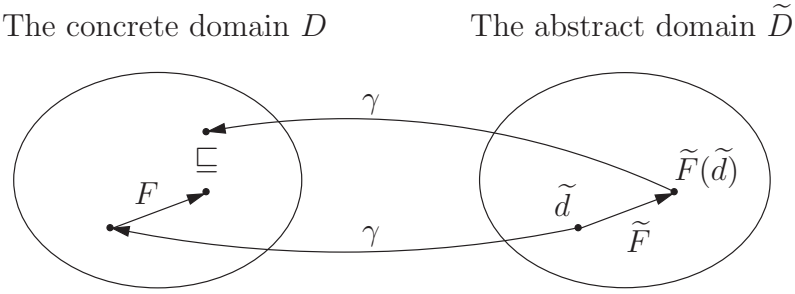


Figure 4.18: The Safety of the Approximation

Replacement of Abstract Interpretations

After studying safe γ-approximations based on a Galois connection we will now see a further motivation why it is useful to have a safe approximation of $\langle\langle\widetilde{\mathfrak{D}}_1,\widetilde{\sqsubseteq}_1\rangle,\widetilde{F}_1\rangle$ by $\langle\langle\widetilde{\mathfrak{D}}_2,\widetilde{\sqsubseteq}_2\rangle,\widetilde{F}_2\rangle$ based on a Galois connection $\langle\widetilde{\mathfrak{D}}_1,\widetilde{\sqsubseteq}_1\rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle\widetilde{\mathfrak{D}}_2,\widetilde{\sqsubseteq}_2\rangle$.

Consider the case we already have a correctness relation $\mathcal{R}_1$ between the program values $d \in \mathfrak{D}$ and the program properties $\widetilde{d}_1 \in \widetilde{\mathfrak{D}}_1$, which could also be represented by the representation function $\beta_1 : \mathfrak{D} \to \widetilde{\mathfrak{D}}_1$ (see Section 4.3.6).

Now we want to replace $\widetilde{\mathfrak{D}}_1$ by the poset $\widetilde{\mathfrak{D}}_2$. Therefore, we have to design a new correctness relation $\mathcal{R}_2 : \mathfrak{D} \times \widetilde{\mathfrak{D}}_2 \to \{\text{true}, \text{false}\}$. Due to the Galois connection between $\widetilde{\mathfrak{D}}_1$ and $\widetilde{\mathfrak{D}}_2$ it is natural to define $\mathcal{R}_2$ by

$$d \,\mathcal{R}_2\, \widetilde{d}_2 \text{ iff } d \,\mathcal{R}_1\, (\gamma(\widetilde{d}_2))$$

It can be shown that $\mathcal{R}_2$ fullfills Property 4.7 and Property 4.8 (given on page 66) and hence is a valid correctness relation.

Assuming that $\mathcal{R}_1$ was generated by the representation function $\beta_1 : \mathfrak{D} \to \widetilde{\mathfrak{D}}_1$, i.e., $d \,\mathcal{R}_1\, \widetilde{d}_1 \;\Leftrightarrow\; \beta_1(d) \,\widetilde{\sqsubseteq}_1\, \widetilde{d}_1$, it can be shown that $\beta_2 : \mathfrak{D} \to \widetilde{\mathfrak{D}}_2$ can be calculated as $\alpha \circ \beta_1 : \mathfrak{D} \to \widetilde{\mathfrak{D}}_2$ (for a formal proof refer to [NNH99]).

We can conclude that having $\langle\langle \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq}_2\rangle, \widetilde{F}_2\rangle$ as a safe γ-approximation of $\langle\langle \widetilde{\mathfrak{D}}_1, \widetilde{\sqsubseteq}_1\rangle, \widetilde{F}_1\rangle$ based on a Galois connection it becomes straight forward to safely exchange the abstract interpretation $\langle\langle \widetilde{\mathfrak{D}}_1, \widetilde{\sqsubseteq}_1\rangle, \widetilde{F}_1\rangle$ by $\langle\langle \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq}_2\rangle, \widetilde{F}_2\rangle$.

Combining Abstract Interpretations

As described in [Nil92], combining two abstract interpretations that represents safe approximations, their combined abstract interpretation is still safe. This is shown graphically in Figure 4.19 and proven by Theorem 4.3.27.

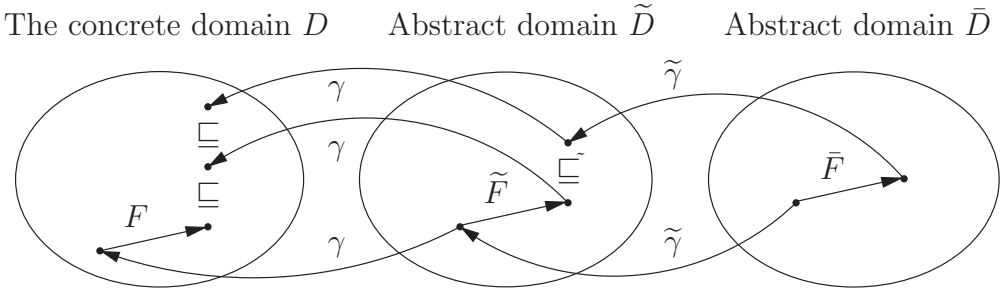


Figure 4.19: Combining Abstract Interpretations

Theorem 4.3.27 (Safety of combined abstract interpretations) *Assume that* $\langle\langle \bar{D}, \bar{\sqsubseteq}\rangle, \bar{F}\rangle$ *is a safe* $\widetilde{\gamma}$*-approximation of* $\langle\langle \widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle, \widetilde{F}\rangle$*, which is in turn a safe* γ*-approximation of* $\langle\langle \mathfrak{D}, \sqsubseteq\rangle, F\rangle$*. Then* $\langle\langle \bar{D}, \bar{\sqsubseteq}\rangle, \bar{F}\rangle$ *is a safe* $(\gamma \circ \widetilde{\gamma})$*-approximation of* $\langle\langle \mathfrak{D}, \sqsubseteq\rangle, F\rangle$*.*

Proof: given in annex C on page 191.

4.3.9 Induced Operators

To obtain a safely approximating abstract interpretation $\langle\langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle, \widetilde{F}\rangle$ for a concrete interpretation $\langle\langle\mathfrak{D}, \sqsubseteq\rangle, F\rangle$, it is required to design a safe abstract transition function $\widetilde{F} : \widetilde{\mathfrak{D}} \to \widetilde{\mathfrak{D}}$ based on a Galois connection $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$.

A simple solution for a safe approximation would be to design $\widetilde{F}(\widetilde{d})$ as $\forall \widetilde{d} \in \widetilde{\mathfrak{D}} : \widetilde{F}(\widetilde{d}) = \widetilde{\top}$, which provides an unusable imprecise interpretation. Therefore, the question arises whether there could be some hints how to design a more precise interpretation. The optimal design is in general application dependent. But it is possible to specify a generic method to construct the most precise transition function, the so-called *induced function* (defined in Definition 4.3.28).

Definition 4.3.28 (Induced function) *Let $\langle\langle\mathfrak{D}, \sqsubseteq\rangle, F\rangle$ be a concrete interpretation and $\langle\mathfrak{D}, \sqsubseteq\rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle\widetilde{\mathfrak{D}}, \widetilde{\sqsubseteq}\rangle$ a Galois connection between the domains $\mathfrak{D}$ and $\widetilde{\mathfrak{D}}$. The by F* induced function *$\widetilde{F}$ is defined as $\widetilde{F} = \alpha \circ F \circ \gamma$*

The construction of the induced function $\widetilde{F}$ is visualised in Figure 4.20.

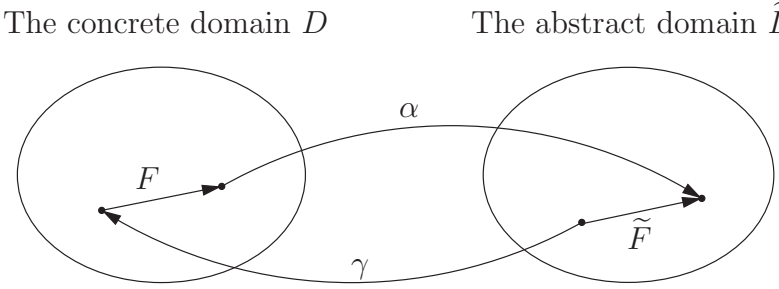


Figure 4.20: Induced Transition Function

Based on [JN94, Gus00] we show in the following some properties of the induced function.

Theorem 4.3.29 (The induced function is monotone) *The induced function $\widetilde{F} = \alpha \circ F \circ \gamma$ is monotone.*

Proof: given in annex C on page 191.

Theorem 4.3.30 (The induced function is safe) *The induced function $\widetilde{F} = \alpha \circ F \circ \gamma$ provides a safe approximation of the concrete function F.*

Proof: given in annex C on page 191.

Theorem 4.3.31 (The induced function is the most precise function on $\widetilde{\mathfrak{D}}$) *The induced function $\widetilde{F} = \alpha \circ F \circ \gamma$ is the most precise function on $\langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq} \rangle$ which is still safe, i.e., satisfying $\alpha((d)) \tilde{\sqsubseteq} \widetilde{F}(\alpha(d))$.*

Proof: given in annex C on page 191.

4.3.10 Termination of Abstract Interpretation

Abstract interpretation is used to calculate *properties* of a program with respect to the program semantics. The program semantics describes program *values* that will be obtained by program execution. It can happen that the execution of a program never terminates for a given initial environment. Nevertheless, program analysis done by abstract interpretation should provide program properties in finite calculation steps, even for nonterminating program execution.

The fixpoint semantics for an abstract interpretation $\langle \langle \mathfrak{D}, \sqsubseteq \rangle, F, \bot \rangle$ has been defined by the semantic transfer function given in Equation 4.1. Based on the result of applying it to the semantic function $F : \mathfrak{D} \to \mathfrak{D}$ an element $d \in \mathfrak{D}$ can be categorised into one of the following classes:

$$\begin{array}{lcll} \mathrm{RED}(F) & = & \{d \,|\, F(d) \sqsubseteq d\} & \ldots F(d) \text{ is reductive} \\ \mathrm{FIX}(F) & = & \{d \,|\, F(d) = d\} & \ldots F(d) \text{ is a fixpoint} \\ \mathrm{EXT}(F) & = & \{d \,|\, d \sqsubseteq F(d)\} & \ldots F(d) \text{ is extensive} \end{array}$$

The fixpoint semantics is given by recursive application of F on the initial configuration $\bot$: $\mathrm{lfp}^{\sqsubseteq} F(\bot)$. In case the height of the domain $\mathfrak{D}$ is finite (i.e., it fullfills the *ascending* and *descending chain condition*) it is guaranteed that the sequence $(F^n)_n$ stabilises by reaching a fixpoint.

If the height of $\mathfrak{D}$ is infinite, approximate interpretation based on a finite domain and/or *widening*/narrowing can be used to enforce termination or also to accelerate termination. In the following, both methods are briefly described.

Approximate Interpretation based on Finite Abstract Domain

A simple and efficient way to accelerate and enforce termination on a concrete interpretation $\langle \langle \mathfrak{D}, \sqsubseteq \rangle, F \rangle$ is to design a safe approximation $\langle \langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq} \rangle, \widetilde{F} \rangle$ based on a domain $\langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq} \rangle$ of finite height.

The drawback of this simple method is the loss of precision since the calculation is done on a coarser semantic domain.

Extrapolation Operators Widening and Narrowing

Having a domain $\langle \mathfrak{D}, \sqsubseteq \rangle$ of infinite height, the sequence $(F^n)_n$ may never stabilise. The use of a *widening* operator $\triangledown \in \mathfrak{D} \times \mathfrak{D} \rightarrowtail \mathfrak{D}$ [CC77] can help to approximate the least

fixpoint $\text{lfp}^{\sqsubseteq}F$. $\triangledown$ is an upper bound operator. The sequence $(F^n)_n$ to approximate $\text{lfp}^{\sqsubseteq}F$ will be defined as:

$$F^n_{\triangledown} = \begin{cases} \bot & \text{if } n = 0 \\ F^{n-1}_{\triangledown} & \text{if } n > 0 \ \wedge \ F(F^{n-1}_{\triangledown}) \sqsubseteq F^{n-1}_{\triangledown} \\ F^{n-1}_{\triangledown} \triangledown F(F^{n-1}_{\triangledown}) & \textit{otherwise} \end{cases} \tag{4.17}$$

The sequence $(F^n)_n$ may reach a value $F^m_{\triangledown} \in (\text{RED}(F) \cup \text{FIX}(F))$. $F^m_{\triangledown}$ will be a safe approximation of $\text{lfp}^{\sqsubseteq}F$. After reaching $F^m_{\triangledown}$ we can now define a further sequence starting from $F^m_{\triangledown}$ to refine the precision of the result by using the *narrowing* operator $\triangle \in \mathfrak{D} \times \mathfrak{D} \rightarrowtail \mathfrak{D}$. The sequence $(F^n_{\triangle})_n$ will be defined as:

$$F^n_{\triangle} = \begin{cases} F^m_{\triangledown} & \text{if } n = 0 \\ F^{n-1}_{\triangle} \triangle F(F^{n-1}_{\triangle}) & \textit{otherwise} \end{cases} \tag{4.18}$$

An essential part is the design of extrapolation operators $\triangledown$ and $\triangle$ that provide precise results.

A simple valid solution for $\triangledown$ and $\triangle$ are the following definitions:

$$\begin{aligned} d_1 \triangledown d_2 &= \begin{cases} d_1 & \text{if } d_2 \sqsubseteq d_1 \\ \top & \text{otherwise} \end{cases} \\ d_1 \triangle d_2 &= d_1 \end{aligned} \tag{4.19}$$

In fact, the definitions given in Equation 4.19 will only achieve the worst precision possible. To find better definitions we will first observe some properties that must hold for $\triangledown$ and $\triangle$:

$$\begin{aligned} \forall d_1, d_2 \in \mathfrak{D} : \quad & \sqcup\{d_1, d_2\} \sqsubseteq (d_1 \triangledown d_2) \sqsubseteq \top \\ \forall d_1, d_2 \in \mathfrak{D} : \quad & \sqcap\{d_1, d_2\} \sqsubseteq (d_1 \triangle d_2) \sqsubseteq d_1 \end{aligned}$$

It is interesting to note that $\triangledown$ and $\triangle$ do not have to be monotone, commutative, associative nor absorptive.

Based on [CC92b], the following methods can be used to improve the quality of $\triangledown$ and $\triangle$:

Extrapolation Threshold: Instead of applying the extrapolation operators $\triangledown$ and $\triangle$ on each element of the sequences $(F^n_{\triangledown})_n$ and $(F^n_{\triangle})_n$, they can be used after an extrapolation threshold of t iterations. For $n \leq t$, the least upper bound $\sqcup$ respective greatest lower bound $\sqcap$ is used. This could be done by simply extending values with an iteration counter and using modified extrapolation operators:

$$\begin{aligned} \langle d_1, i\rangle \ \bar{\triangledown} \ \langle d_2, i+1\rangle &= \begin{cases} \langle d_1, i+1\rangle & \text{if } d_2 \sqsubseteq d_1 \\ \langle \sqcup\{d_1, d_2\}, i+1\rangle & \text{if } d_1 \sqsubseteq d_2 \ \wedge \ i \leq t \\ \langle d_1 \triangledown d_2, i+1\rangle & \text{otherwise} \end{cases} \\ \langle d_1, i\rangle \ \bar{\triangle} \ \langle d_2, i+1\rangle &= \begin{cases} \langle \sqcap\{d_1, d_2\}, i+1\rangle & \text{if } i \leq t \\ \langle d_1 \triangle d_2, i+1\rangle & \text{otherwise} \end{cases} \end{aligned} \tag{4.20}$$

Stepwise Extrapolation: The first simple definition of ∇ and Δ in Equation 4.19 maps all values on a chain i $(\bot, \sigma_1^i, \sigma_2^i, \ldots, \top)$ using a single unconditional rule. An improvement could be to refine them with additional distributive limits $\bot \sqsubseteq \sigma_{L1}^i \sqsubseteq \ldots \sqsubseteq \sigma_{Ln}^i \sqsubseteq \top$:

$$
\begin{aligned}
d_1 \nabla d_2 &= \begin{cases} d_1 & \text{if } d_2 \sqsubseteq d_1 \\ \sigma_{L1}^i & \text{elseif } d_2 \sqsubseteq \sigma_{L1}^i \\ \vdots & \\ \sigma_{Ln}^i & \text{elseif } d_2 \sqsubseteq \sigma_{Ln}^i \\ \top & \text{otherwise} \end{cases} \\
d_1 \Delta d_2 &= \begin{cases} d_2 & \text{if } \left(\bigvee_{j=1}^{n} d_2 \sqsubseteq \sigma_{Lj}^i \sqsubseteq d_1 \right) \vee (d_1 = \top) \\ d_1 & \text{otherwise} \end{cases}
\end{aligned}
\tag{4.21}
$$

Another method to construct the upper bound operator ∇ is related to the construction of *induced operations*, described in Section 4.3.9. To enforce termination, the *widening* operator $\nabla : \mathfrak{D} \times \mathfrak{D} \to \mathfrak{D}$ can be build using a Galois connection $\langle \mathfrak{D}, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq} \rangle$ as follows:

$$
d_1 \nabla d_2 \;=\; \gamma(\alpha(d_1) \tilde{\nabla} \alpha(d_2)) \tag{4.22}
$$

The domain $\langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq} \rangle$ has to fulfil the *ascending chain condition* and $\tilde{\nabla} : \widetilde{\mathfrak{D}} \times \widetilde{\mathfrak{D}} \to \widetilde{\mathfrak{D}}$ has to be an upper bound operator.

4.3.11 Systematic Design of Galois Connections

As discussed in Section 4.3.8, the sequential combination of two abstract interpretations that are safe approximations gives raise to a new safe abstract interpretation. This rule is supported by the principle that the sequential combination of two Galois connections always results into a new Galois connection (and analogous for Galois insertions and Galois isomorphisms).

A similar rule can be found for the parallel combination of two abstract interpretations. Assuming two Galois connections $\langle \mathfrak{D}_1, \sqsubseteq_1 \rangle \underset{\gamma_1}{\overset{\alpha_1}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}_1, \tilde{\sqsubseteq}_1 \rangle$ and $\langle \mathfrak{D}_2, \sqsubseteq_2 \rangle \underset{\gamma_2}{\overset{\alpha_2}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}_2, \tilde{\sqsubseteq}_2 \rangle$ it may be desired to use their combined domains $\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq \rangle$ and $\langle \widetilde{\mathfrak{D}}_1 \times \widetilde{\mathfrak{D}}_2, \tilde{\sqsubseteq} \rangle$ where:

$$
\begin{aligned}
(d_1, d_2) \sqsubseteq (d_1', d_2') &\;\Leftrightarrow\; ((d_1 \sqsubseteq_1 d_1') \wedge (d_2 \sqsubseteq_2 d_2')) \\
(\tilde{d}_1, \tilde{d}_2) \tilde{\sqsubseteq} (\tilde{d}_1', \tilde{d}_2') &\;\Leftrightarrow\; ((\tilde{d}_1 \tilde{\sqsubseteq}_1 \tilde{d}_1') \wedge (\tilde{d}_2 \tilde{\sqsubseteq}_2 \tilde{d}_2'))
\end{aligned}
$$

Based on these domains the *independent attribute method* [NNH99] can be used to raise a new Galois connection

$$
\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}_1 \times \widetilde{\mathfrak{D}}_2, \tilde{\sqsubseteq} \rangle
$$

where:

$$\begin{aligned} \alpha(d_1, d_2) &= (\alpha_1(d_1), \alpha_2(d_2)) \\ \gamma(\widetilde{d}_1, \widetilde{d}_2) &= (\gamma_1(\widetilde{d}_1), \gamma_2(\widetilde{d}_2)) \end{aligned}$$

That this construction method indeed defines a new Galois connection can be shown by the following calculation

$$\begin{aligned} \alpha(d_1, d_2) \mathbin{\widetilde{\sqsubseteq}} (\widetilde{d}_1, \widetilde{d}_2) &\Leftrightarrow (\alpha_1(d_1), \alpha_2(d_2)) \mathbin{\widetilde{\sqsubseteq}} (\alpha_1(\widetilde{d}_1), \alpha_2(\widetilde{d}_2)) \\ &\Leftrightarrow (\alpha_1(d_1) \mathbin{\widetilde{\sqsubseteq}_1} \widetilde{d}_1) \wedge (\alpha_2(d_2) \mathbin{\widetilde{\sqsubseteq}_2} \widetilde{d}_2) \\ &\Leftrightarrow (d_1, d_2) \sqsubseteq (\gamma_1(\widetilde{d}_1), \gamma_2(\widetilde{d}_2)) \\ &\Leftrightarrow (d_1, d_2) \sqsubseteq \gamma(\widetilde{d}_1, \widetilde{d}_2) \end{aligned}$$

together with the application of Theorem 4.3.18.

Having that $\langle\langle \widetilde{\mathfrak{D}}_1, \widetilde{\sqsubseteq_1}\rangle, \widetilde{F}_1\rangle$ is a safe γ_1-approximation of $\langle\langle \mathfrak{D}_1, \sqsubseteq_1\rangle, F_1\rangle$ and $\langle\langle \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq_2}\rangle, \widetilde{F}_2\rangle$ is a safe γ_2-approximation of $\langle\langle \mathfrak{D}_2, \sqsubseteq_2\rangle, F_2\rangle$ then it follows that $\langle\langle \widetilde{\mathfrak{D}}_1 \times \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq}\rangle, \widetilde{F}_1 \times \widetilde{F}_2\rangle$ is a safe γ-approximation of $\langle\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq\rangle, F_1 \times F_2\rangle$. Therefore, the parallel combination of two abstract interpretations that are safe approximations gives raise to a new safe abstract interpretation.

There are similar results for parallel combination of safe approximations based on Galois insertions or Galois isomorphisms.

4.4 Chapter Summary

The correctness of the flow facts transformation framework presented in this thesis is based on the formalism of abstract interpretation.

This chapter presented the required theoretic foundations to construct the transformation framework. First, a description of various control flow graphs was given. Second, the meaning of semantics has been introduced to allow the formalisation of program executions. Afterwards, abstract interpretation as a tool for performing approximated calculations was described. Beside the properties of abstract interpretation, several methods to combine abstract interpretations were discussed. The concept of induced function allows the calculation of a safe abstract transition function based on a Galois connection.

To ask the hard question is simple

W. H. Auden, *Poems* (1930)

Chapter 5

Classification of Code Transformations

One aspect of WCET analysis is code representation, which belongs to compiler techniques. The calculation of a safe bound for the WCET uses flow facts *ff* as described in Section 2.3.3. This chapter discusses the impact of code optimisations performed by the compiler to WCET analysis. In particular, it analyses whether such flow information *ff* has to be transformed in parallel to certain code transformations. We divide the described code optimisations into two basic categories - in control flow changing and control flow preserving code transformations. For the *ff* transformation only the control-flow-changing code transformations have to be considered.

We have divided the control flow changing code transformations into further categories, depending on their impact on the control flow. For example, *low-level optimisations* only deal with merging and splitting of control flows without considering any loop structure. *Redundancy elimination* removes parts of the control flow graph. *Loop reordering transformations* influence only the loop control code but the iteration count of the loop body remain unchanged. *Other Loop Transformations* are more radical loop transformations that also change the iteration count of the loop body. *Procedure call transformations* cause changes in the *GCFG*. Concrete *ff* transformation rules for selected code transformations are described in Chapter 8, while *ff* transformation rules for further code transformations can be derived in an analogous way by looking at the concrete *ff* transformation rule of a similar code transformation.

5.1 Problem Statement

The following sections list typical code transformations and discuss their impact on WCET analysis. The impact of these code transformation depends on the type of flow facts being used. The details of code transformations are not discussed within this thesis. More accurate information about all described transformations itself is given by Bacon [BGS94] and Muchnik [Muc97]. Section 4.1.1 provides a definition of a control flow graph. The global control flow graph (GCFG) to represent also function calls is

described in Section 4.1.3 on page 48.

Flow facts with less expressiveness about (in)feasible paths are easier to transform precisely than more expressive flow facts. For the discussion of the transformations we assume flow facts as described in Section 2.3.3. However, the discussions may be also valid for WCET calculation methods using other types of flow facts.

5.2 Optimisations within a Basic Block

Code optimisations that only change the composition of a basic block by modifying its sequence of statement do not change the *CFP*. Flow facts are directly assigned to control flow edges of basic blocks. Code optimisations within a basic block therefore do not require the update of any flow facts. Thus we do not describe code transformations within basic blocks in further detail.

5.3 Optimisations Changing the Control Flow

The following describes code transformations that have a significant impact on the possible *CFP* of a program. After the application of certain code optimisations the flow facts also have to be transformed to avoid underestimation of the WCET. In general it is not possible to derive the new flow facts by comparing the code structure between the original and the transformed code. Adequate compiler support by providing information about the performed code transformations resolves this lack of information.

5.3.1 Low-Level Optimisations

This section discusses several code transformations to optimise the control flow within a procedure. They are discussed in further detail by Muchnik in [Muc97].

If Simplifications

If simplifications apply to conditional constructs where one or more of its branches are empty. Empty branches can be a result of previous transformations like code hoisting [Muc97]. Such empty branches are removed by *if simplification*.

Another class of *if simplification* where the compiler can check statically that a certain branch of an conditional construct cannot be reached is handled by unreachable code optimisation described on page 87.

Discussion: Similar to *useless code elimination*, *if simplifications* can remove potentially reachable parts of the *CFG*. Flow facts referring to the removed part have to be updated.

Straightening

Straightening is a code transformation to merge two basic blocks. Having two basic blocks A and B where A is the only predecessor of B and B is the only successor of A then these two blocks will be merged to a new single basic block.

Discussion: *Straightening* causes minimal modifications to the *CFG* resulting in small changes of the *CFP*. Any flow facts referring to the execution frequency of the control flow edge A-B from the above example have to be distributed to the predecessor respective the successor edges of the merged basic block.

Branch Optimisations

Branch optimisation (also called *jump optimisation*) is a transformation to optimise consecutive jump instructions. A typical example is a conditional jump to an unconditional jump. Branch optimisation will replace the jump target of the conditional jump by the jump target of the unconditional jump.

Discussion: *Branch optimisation* changes the execution frequency of basic blocks by redirection of jumps. Any flow facts referring to such basic blocks have to be updated.

Tail Merging

Tail merging, also called *cross jumping*, is a code transformation to reduce code size. Having two basic blocks where the last few instructions are identical and with an equal single successor node, tail merging will transform this code to share the identical instructions. The last few instructions will be put into a new basic block.

Discussion: *Tail merging* changes the *CFG* by creating a new basic block. Flow facts referring to control flow edges that will be changed, have to be updated correctly.

Tail Duplication

Tail duplication can be seen as the opposite transformation to *tail merging*. *Tail duplication* is a code transformation to reduce the number of branch statements by duplicating code for two control-flow branches. Another application of *tail duplication* is to prepare the construction of *superblocks*[1] or *hyperblocks*[2].

Discussion: *Tail duplication* changes the *CFG* by duplicating basic blocks and placing a copy to each incoming control flow edge. Flow facts referring to control flow edges that will be changed, have to be updated correctly.

[1] superblock... a trace with only one entry node but one or more exit nodes.

[2] hyperblock... a collection of connected basic blocks with only one entry node but one or more exit nodes.

Conditional Moves / If Conversion

Conditional moves are copy instructions controlled by a predicate. Depending on the value of the predicate the copy operation is performed or just ignored. Conditional move statements require hardware support which is currently provided by several modern processor architectures. A typical code optimisation that can create conditional moves is *if conversion*.

Conditional moves can increase the performance by avoiding conditional jumps. Another advantage of conditional move statements is that they can be used to make the program execution time more predictable [Pus02].

An example for if-conversion is given by Figure 5.1 and Figure 5.2. For this example we have extended the language $\overline{\text{WHILE}}$ by a predicated copy statement of the form dst:=(cond)src;

Listing 5.1: original code

```
1      if a>b then
2        goto L1;
3      else
4        c:=b;
5        goto L2;
6  L1: c:=a;
7  L2: ...
```

Listing 5.2: conditional move applied

```
1 t1:=(a>b);
2 c:=b;
3 c:=(t1)a;
```

Discussion: Transforming code to use conditional moves causes similar changes of the *CFG* than *if simplification*. The alternative branches if a conditional statement will be merged into a single branch by using conditional moves. Flow facts addressing the individual branches therefore have to be merged.

5.3.2 Partial Evaluation

Partial evaluation refers to the technique of performing parts of a computation at compile time. Typical examples for partial evaluation are all standard code optimisations based on data-flow analysis. The following transformation changes the *CFP*:

Function Cloning

Function cloning transforms a function call with some constants in the arguments to call a cloned function were the parameters are replaced by the constant values. An example for function cloning is given in Listing 5.3 and Listing 5.4.

Listing 5.3: original code

```
1 function add(a,b)
2   begin
3   return a+b;
4   end
5
6 c:=add(d,2);
```

Listing 5.4: after partial evaluation

```
1 function add_2(a)
2   begin
3   return a+2;
4   end
5
6 c:=add_2(d);
```

Discussion: Function cloning changes the *GCFG* dramatically by redirecting function calls to a new target. Flow facts related to the original function call have to be updated.

5.3.3 Redundancy Elimination

Redundancy elimination is a transformation to improve performance by identifying redundant calculations and removing them. The following will describe transformations for redundancy elimination that change the *CFG*.

Unreachable Code Elimination

Unreachable code is code that will never be executed, independent of the values of program parameters. Typical examples for unreachable code are a conditional where the test is based on a constant value or a loop that will not perform any iteration.

Discussion: *Unreachable code elimination* deletes only parts of the *CFG* that are not reachable. However, there may be flow facts that compare the execution frequency of several execution paths. Such flow facts have to be updated to reflect the removal of the unreachable part of the *CFG*.

Useless Code Elimination

Elimination of *useless code* is also called *dead code elimination* [Muc97]. A code is useless if the result of its computation is never used later. Dead variables for example are a special form of useless code. *Useless code* is removed by *useless code elimination*.

A simple example for useless code elimination is given in Listing 5.5 and Listing 5.6.

Listing 5.5: original code

```
1 function add(a,b)
2   begin
3   c:=a*b;
4   return a+b;
5   end
```

Listing 5.6: after useless code elimination

```
1 function add(a,b)
2   begin
3   return a+b;
4   end
```

Discussion: *Useless code elimination* removes code that is potentially reachable. If *useless code elimination* is implemented to also remove conditional control structures, the changes in the *CFG* can be dramatic. If flow facts refer to the execution frequency of such useless code, it can be quite difficult to perform an accurate update of the respective flow facts.

5.3.4 Loop Reordering Transformations

Loop reordering transformations change the order in which the iterations of a *perfect loop nest* are executed. Definition 5.3.1 gives an informal definition of a *perfect loop nest*. Several transformations change the *CFP* dramatically. Flow facts describing loop bounds as well as infeasible paths have to be updated.

Definition 5.3.1 (Perfect Loop Nest) *A loop nest is called* perfect*, if the body of every loop except the innermost loop consists only of the nested loop.*

The following will describe loop reordering transformations changing the control flow and discuss their effects on the *CFP*.

Loop Interchange

Loop interchange exchanges the position of two loops within a loop nest, typically by moving one of the outer loops to the innermost position. Typical applications of this transformation are the increase of data access locality or loop-invariant expressions of the inner loop. Another application domain is to enable vectorisation of the innermost loop on a vector architecture. An example of the effect of performing loop interchange is given in Listing 5.7 and Listing 5.8.

Listing 5.7: original code

```
1 for i:=1,m,1 do
2   for j:=1,n,1 do
3     a[i]:=a[i] + b[i,j];
```

Listing 5.8: interchanged loop nest

```
1 for j:=1,n,1 do
2   for i:=1,m,1 do
3     a[i]:=a[i] + b[i,j];
```

Discussion: *Loop interchanging* modifies the sequence of loop bounds within a loop nest as well as the execution frequency of several edges within the *CFG*. The general structure of the *CFG* remains unmodified and the execution frequency of the body of the innermost loop remains unchanged.

Loop Blocking

Loop blocking (also called *loop tiling*) is typically used to improve locality of data accesses. The locality is increased by iterating over sub-rectangles of the whole iteration space of

a loop nest. Thus, for example, cache lines can be used more efficiently. An example of the effect of performing *loop blocking* for a sub-rectangle of $k_1 \times k_2$ iterations is given in Listing 5.9 and Listing 5.10.

Listing 5.9: original code

```
1 for i:=1,m,1 do
2   for j:=1,n,1 do
3     a[i,j]:=b[i,j];
```

Listing 5.10: blocked loop

```
1 for ti:=1,n,k1 do
2   for tj:=1,m,k2 do
3     for i:=ti,min(ti+
4         k1-1,m),1 do
5       for j:=tj,min(tj+
6           k2-1,n),1 do
7         a[i,j]:=b[i,j];
```

Discussion: *Loop blocking* introduces new loops and changes the loop bounds of all involved loops. The execution frequency of the body of the innermost loop remains unchanged but the structure of the resulting *CFG* changes dramatically.

5.3.5 Other Loop Transformations

The following describes loop transformations that are not limited to *perfect loop nests*. These transformations also change the *CFP* dramatically. Flow facts describing loop bounds as well as infeasible paths have to be updated.

Loop Unrolling

Loop unrolling is performed to reduce the overhead of a loop. Another application for VLIW machines is to create a longer sequence of instructions inside the loop body. When a loop is unrolled by a factor of k, the body of the loop is duplicated to create k instances and the iteration count of the loop is decreased by the factor k. If the iteration count of the loop is not known statically to be a multiple of k, additional code for the remaining iteration has to be created. The principle of *loop unrolling* is shown in Listing 5.11 and Listing 5.12 using an unrolling factor of two. Since the loop bound is not defined, additional code to test for remaining iterations has been inserted.

Listing 5.11: original code

```
1 for i:=1,n,1 do
2   a[i]:=a[i]*b[i+1];
```

Listing 5.12: loop unrolled twice

```
1 for i:=1,n,2 do
2   a[i]:=a[i]*b[i-1];
3   a[i+1]:=a[i+1]*b[i];
4 if mod(n,2)=1 and
5    n>0 then
6   a[n]:=a[n]*b[n-1];
7 else
8   skip
```

Assume the iteration count of the loop is known as the factor n. When unrolling the loop n times it is said the loop is completely unrolled and the loop is removed.

Discussion: *Loop unrolling* by a factor k may introduce a new loop for the remainder of the iteration space with a loop bound $0 \ldots k$. The iteration bound of the original loop is distributed to the unrolled loop scaled by k and the remainder. The *CFG* will be changed dramatically.

Software Pipelining

Similar to hardware pipelining, *software pipelining* divides the execution of a loop body into several execution stages. The transformed loop body contains the reordered initial execution stages with ascending iteration instances. Additional prolog and epilog code is created to handle the start and termination of the loop. *Software pipelining* is used to enable instruction parallelism for VLIW architectures in case of dependencies between execution stages.

A simple example for *software pipelining* including prolog and epilog is shown in Listing 5.13 and Listing 5.14.

Listing 5.13: original code

```
1 for i:=1,n,1 do
2   a:=a+b[i];
3   c[i]:=a/2;
```

Listing 5.14: after software pipelining

```
1 a:=a+b[1];
2 for i:=2,n,1 do
3   c[i-1]:=a/2;
4   a:=a+b[i];
5 c[n]:=a/2;
```

Discussion: The iteration bound of the original loop is reduced by the number of different iteration instances already executed in the prolog (or will be executed afterwards in the epilog). If the involved iteration stages have more complex control flow like `if/then/else` constructs, it is also necessary to update the information assigned to such conditional control flow edges. *Software piplining* changes the *CFG* dramatically.

Perfect Pipelining

Perfect pipelining is a combination of *loop unrolling* and *software pipelining*. The consequences of applying this transformation is given separately by the description of *loop unrolling* and *software pipelining*

Loop Distribution

Loop Distribution (also called *loop fission* or *loop splitting*) is used to split a single loop into multiple loops with each loop containing only a subset of the original loop

body. *Loop distribution* can be used for example to create perfect loop nests, improve instruction cache locality or reduce memory consumption.

An example for *loop distribution* is given in Listing 5.15 and Listing 5.16.

Listing 5.15: original code

```
1 for i:=1,n,1 do
2   a:=a+b[i];
3   b[i]:=c[i];
```

Listing 5.16: distributed loop

```
1 for i:=1,n,1 do
2   a:=a+b[i];
3 for i:=1,n,1 do
4   b[i]:=c[i];
```

Discussion: *Loop distribution* creates new loops with equal iteration space. It is required to create flow facts for these new loops. *Loop distribution* changes the *CFG* significantly.

Loop Fusion

Loop fusion (also called *loop jamming*) is the inverse transformation to *loop distribution*. *Loop fusion* can be used, for example, to reduce loop overhead or increase instruction parallelism. The effects of performing *loop fusion* can be obtained from the description of *loop distribution*.

Loop Unswitching

Loop unswitching can be applied when a loop contains a conditional with a loop-invariant test condition. The loop is then replicated within each branch of the conditional to save the overhead of the conditional branch inside the loop. Other advantages are that the loop body is reduced and instruction level parallelism may be improved.

An example for *loop unswitching* is shown in Listing 5.17 and Listing 5.18.

Listing 5.17: original code

```
1 for i:=1,n,1 do
2   a[i]:=a[i]+b;
3   if (x>0) then
4     a[i]:=a[i]+2;
5   else
6     a[i]:=a[i]-2;
```

Listing 5.18: unswitched loop

```
1 if (x>0) then
2   for i:=1,n,1 do
3     a[i]:=a[i]+b;
4     a[i]:=a[i]+2;
5 else
6   for i:=1,n,1 do
7     a[i]:=a[i]+b;
8     a[i]:=a[i]-2;
```

Discussion: *Loop unswitching* changes the *CFP* radically. The loop itself is replicated with equal iteration spaces. Any information about (in)feasible paths to describe the *CFP* has to be updated carefully.

Loop Peeling

Loop peeling can be used to match the iteration control of adjacent loops and therefore enabling loop fusion. *Loop peeling* splits a loop into multiple parts without changing the iteration order. Therefore, this transformation can be applied to any loop.

An example for *loop peeling* is given in Listing 5.19 and Listing 5.20. The next step would be to apply *loop fusion* over the second and third loop of Listing 5.20.

Listing 5.19: original code

```
1 for  i:=1,n,1  do
2    a[i]:=a[i]+b;
3 for  i:=k,n,1  do
4    c:=c+a[i];
```

Listing 5.20: after loop peeling

```
1 for  i:=1,k-1,1 do
2    a[i]:=a[i]+b;
3 for  i:=k,n,1  do
4    a[i]:=a[i]+b;
5 for  i:=k,n,1  do
6    c:=c+a[i];
```

Discussion: *Loop peeling* creates a sequence of loops where the overall interaction count matches the iteration count of the original loop. The *CFG* is changed dramatically and information about loop iteration bounds and (in)feasible paths have to be updated.

Removing Empty Loops

Empty loops can be a result of previous code transformations. Deleting such a loop only requires to consider later references to the induction variable of that loop.

Discussion: Removing empty loops has a serious impact on the *CFG*. Flow facts about (in)feasible paths referring to control flow edges within these loops have to be updated.

5.3.6 Procedure Call Transformations

Procedure calls are a primitive mechanism for modularisation of software. The drawback is that a procedure call induces additional overhead to transfer program control. In this section we describe some code transformations on procedure calls that change the structure of the code dramatically. When WCET analysis is done at inter-procedural level it is necessary to update involved flow facts correctly.

To model the program control flow including function calls, the *CFG* is extended to a global control flow graph (*GCFG*). A description of the *GCFG* is given in Section 4.1.3 on page 48.

Procedure Inlining

Procedure inlining is a transformation that replaces a function call by the body of the

called function. This optimisation removes the control transfer functions at the cost of overall code size.

An example for *procedure inlining* is given in Listing 5.21 and Listing 5.22. The transformed code given in Listing 5.22 does not call the function add(a,b) anymore.

Listing 5.21: original code

```
1 function add(a,b)
2   begin
3   return a+b;
4   end
5
6 for 1,n,1 do
7   a[i] := add(b[i],c[i]);
```

Listing 5.22: after procedure inlining

```
1 for 1,n,1 do
2   a[i] := b[i]+c[i];
```

Discussion: *Procedure inlining* changes the *GCFG* dramatically. Flow facts referring to the removed function call have to be transformed.

Procedural Abstraction

Procedural abstraction [DEMS00] can be seen as the opposite code transformation to *procedure inlining*. *Procedural abstraction* is typically used for code compaction on embedded systems. Single-entry, single-exit code fragments are be moved into a function to reuse this code in other code locations.

An example to show the principle of *procedural abstraction* is given by Listing 5.23 and Listing 5.24. In practice, *procedural abstraction* does not create functions that pass and return operands as formal arguments. Instead, registers are simply used as global variables. Register renaming and additional copy statements may be necessary to match code fractions.

Listing 5.23: original code

```
1 tmp1 = (a + b)*2;
2 e    = tmp1 + 5;
3 ...
4 tmp2 = (c + d)*2;
5 f    = tmp2 + 5;
```

Listing 5.24: after procedural abstraction

```
1 function fn_abs(u,v)
2   begin
3   tmp = (u + v)*2;
4   return tmp + 5;
5   end
6
7 e = fn_abs(a,b);
8 ...
9 f = fn_abs(c,d);
```

Discussion: *Procedural abstraction* changes the *GCFG* dramatically. Flow facts referring to code that has been moved into the created function have to be transformed.

Function Memoization

Function memoization can be used for procedures free of side effects, i.e., procedures that change the state of a program only over their specified output interfaces. *Function memoization* is a code transformation that caches the result of previous function calls to avoid the overhead of calculating the same result more than once.

An example for *function memoization* is given in Listing 5.25 and Listing 5.26. Listing 5.25 shows a simple function call. The transformed code given in Listing 5.26 has additional code to cache the result of a function call.

Listing 5.25: original code

```
1 a := f(i);
```

Listing 5.26: after function memoization

```
1 var b_iscached[n];
2 var b_cache[n];
3
4 if b_iscached[i]≠1 then
5   b_cache[b]:=f(i);
6   b_cache[b]:=1;
7 else
8   skip
9 a := b_cache(i);
```

Discussion: *Function memoization* changes the structure of the *GCFG* significantly. Flow facts that refer to the execution frequency of the cached procedure have to be updated safely. To enable precise update of such flow facts it is required to have knowledge about the distribution of the values of the function arguments.

5.4 Control Flow Preserving Optimisations

Code transformations which do not change the *CFG* are not subject to any flow fact update. Such transformations only modify particular statements but do not directly change the structure of basic blocks. Typical operations are the move, duplication, deletion of single statements.

However, the application of such control flow preserving optimisations can enable further optimisations that may change the *CFG*. As a simple example, after moving all statements out of a block, the block may be removed from the *CFG*.

For a deeper understanding about which transformations are critical for the *CFP* calculation, the following list will show typical code transformations that do not change

the *CFG* of the code. All these transformations do not require to update flow facts describing the *CFP*. Further literature describing these transformations in more detail has been collected by Bacon [BGS94]. Muchnik explains most of these transformations in detail [Muc97].

5.4.1 Partial Evaluation

Partial evaluation refers to the technique of performing parts of a computation at compile time. Typical examples for partial evaluation are all standard code optimisations based on data-flow analysis. The following transformations do not change the *CFP*:

- Constant Propagation
- Constant Folding
- Copy Propagation
- Statement Substitution
- Algebraic Simplification
- Reassociation

5.4.2 Memory Access Transformations

Memory access transformations are used to optimise memory accesses by considering system configurations like memory page organisation and cache architectures. The following memory access transformations do not change the *CFP*:

- Memory Alignment
- Array Padding
- Code Co-location
- Displacement Minimisation
- Array Contraction
- Scalar Replacement

5.4.3 Redundancy Elimination

The following redundancy elimination transformations do not change the *CFP*:

- Dead Variable Elimination

- Common Subexpression Elimination
- Partial-Redundancy Elimination
- Short-Circuiting
- Code Hoisting

5.4.4 Loop Reordering Transformations

Loop reordering transformations change the order in which the iterations of a *perfect loop nest* are executed. The following loop reordering transformations do not change the iteration frequency within the *CFP*:

- Loop Skewing
- Loop Reversal

5.4.5 Other Loop Transformations

The following loop transformations do not change the *CFP*:

- Strength Reduction of Induction Variable Expressions
- Induction Variable Elimination
- Loop-invariant Code Motion
- Loop Normalisation

5.4.6 Procedure Call Transformations

Procedure call transformations work on the *GCFG*. The following transformations do not change the *CFP*:

- Leaf Procedure Optimisation
- Cross-call Register Allocation
- Parameter Promotion
- Frame Collapsing
- Tail Recursion Elimination

5.4.7 Other Transformations

Here we discuss transformations that are not covered by the above categories. The following transformations do not change the *CFP*:

- Strength Reductions
- Machine Idioms and Instruction Combining

5.5 Chapter Summary

Code optimisations performed by the compiler can change the control flow of a program dramatically. This is a challenging problem for WCET analysis since meta-information about the control flow is desired to be provided at source code level. It is the topic of this thesis to provide a mapping of this meta-information (flow facts) from the source code level to object code that can handle code optimisations.

This chapter gave an overview of typical code optimisations performed by a compiler. For each optimisation its impact on the control flow and the required update of flow facts was discussed. Various code transformations were listed which require compiler support to transform the flow facts precisely to reflect the control flow changes. Examples for that are several loop transformations. Code transformations that only change the composition of basic blocks without changing the control flow itself do not need an update of flow facts.

There is one thing stronger than all the armies in the world;
and this is an idea whose time has come.

ANONYMOUS, *Nation* (15. APRIL 1943)

Chapter 6

Timing Analysis of Optimised Code

This chapter describes an abstract flow facts transformation framework to handle code transformations performed by an optimising compiler. This abstract transformation framework is used in the next chapter to construct a safe and precise concrete flow facts transformation framework.

6.1 The Context of Code Transformations within WCET Analysis

There exist various techniques to transform code or meta information. Especially research in the domain of compiler technology has produced many of them. In this thesis we present a transformation technique to correctly transform flow facts describing the possible *CFP* of the program in parallel to code transformations performed by the compiler. One may ask whether it is necessary to develop a new transformation scheme from scratch. The most effective way would be to adopt an existing scheme for the specific requirement. However, in this section we give a short impression why the transformation of flow facts is quite specific.

A well known application domain for transforming meta information is symbolic debugging [JGS98] (see also Section 3.5.2). A requirement for symbolic debugging is to be able to find an appropriate object code breakpoint suitable for the desired breakpoint in the source code. At the breakpoint it should be possible to report the values of variables according to the semantics of the source language. The code location problem in symbolic debugging requires a two-way mapping between the source code and the object code. This mapping is denoted by map_i in Figure 6.1. The mapping map_i for symbolic debugging is a different meta information of a quite different nature than the mapping of program code to its possible *CFP*. In case of code transformations done by the compiler there are different problems to be solved. However, the code location mapping for symbolic debugging can be also used to propagate the result of the WCET calculation back to the source code.

To calculate the WCET of a program $\mathcal{P}$ it is generally necessary to have a mapping

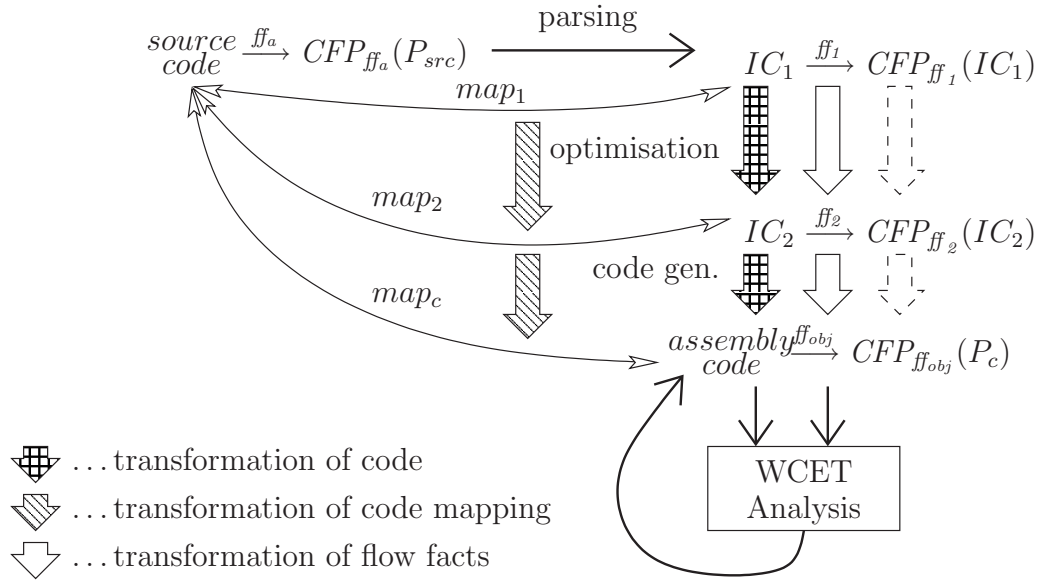


Figure 6.1: The Context of *ff* Transformations

between the program and its *CFP*. This mapping is done by the flow facts *ff* described in Section 2.3.3 which provides together with the program code $\mathcal{P}$ a safe upper closure $CFP_{ff}(\mathcal{P})$ for $CFP_{opt}(\mathcal{P})$. As shown in Figure 6.1, the flow facts to describe the *CFP* are denoted by ff_i. The extraction of flow facts (see Figure 2.2.3) itself by parsing ff_a annotations and analysing the structure and semantics of the code is not the scope of this thesis. During the transformation from intermediate code IC_1 to IC_2 the compiler can perform drastic changes of the code structure.

This work addresses the safe and precise parallel update of the flow facts from ff_1 to ff_2 so that the resulting $CFP_2(IC_2)$ is still a safe upper closure for $CFP_{opt}(IC_2)$ of the transformed program IC_2. Finally, the generated ff_{obj} together with the assembly code forms the basis for calculating the WCET. The concrete execution scenario calculation method to get a safe upper bound for the real WCET is not part of this thesis. However, Section 2.3.2 gives a short introduction of the WCET calculation method based on the implicit path enumeration technique (IPET). The mechanism to update the flow facts has to fulfil the following requirements:

- it must be hardware-independent to use the approach for any hardware platform (this is no challenge, as the type of flow facts we support is hardware-independent itself).
- it must provide a flow-facts interface applicable to manual code annotations as well as automatic flow facts extraction tools (this is achieved by supporting flow facts for IPET-based WCET calculation).
- it must support any compiler optimisation (this is achieved by developing basic update functions that can be grouped to support more complex code transformations).

- the flow facts update rules must be safe (this is achieved by developing a formalism to induce safe (correct) flow facts update rules).
- the flow facts update rules should be precise (this is achieved by designing basic operations allowing flexible modifications of flow facts).
- the integration of the flow facts transformation into a compiler should be possible with reasonable implementation effort (this is achieved by the modular construction of update rules).

6.2 Dependable Flow Facts Transformation

Our transformation method uses abstract interpretation to induce the correct flow facts transformation function. Abstract interpretation is a formalism supporting the systematic construction of a safe and correct interpretation, based on a given concrete interpretation [CC77]. The basic principles of abstract interpretation are described in Section 4.3. The resulting properties of having a Galois connection between the concrete and the abstract domain are described in Section 4.3.7.

6.2.1 The Correctness of the Transformation

$\mathcal{P} \in \mathbb{P}$ represents the program to be transformed by the transformation function $F_{t1} : \mathbb{P} \rightarrow \mathbb{P}$. To enable the calculation of a WCET bound for $\mathcal{P}$, additional flow facts $ff \in \mathbb{F}$ are assigned to $\mathcal{P}$. The flow facts $\mathbb{F}$ form a domain $\langle \mathbb{F}, \sqsubseteq_2 \rangle$ where $\sqsubseteq_2$ is defined as $ff_1 \sqsubseteq_2 ff_2 \Leftrightarrow (ff_2$ is a less restrictive subset of $ff_1)$. The exact definition of $\sqsubseteq_2$ depends on the concrete type of supported flow facts. Intuitively spoken, for each element $f_2 \in ff_2$ there exists an element $f_1 \in ff_1$ where f_2 is less restrictive than f_1.

To describe the correct transformation $\mathbb{F}$, $\mathbb{P}$ and $\mathbb{F}$ are grouped together to form the domain $\langle \mathfrak{D}, \sqsubseteq \rangle$ with $\mathfrak{D} : \mathbb{P} \times \mathbb{F}$, having a combined transformation function $F_t = F_{t1} \times F_{t2}$. The relation $\sqsubseteq$ is defined as

$$\forall \langle \mathcal{P}_1, ff_1 \rangle, \langle \mathcal{P}_2, ff_2 \rangle \in \mathfrak{D} : \ \langle \mathcal{P}_1, ff_1 \rangle \sqsubseteq \langle \mathcal{P}_2, ff_2 \rangle \Leftrightarrow (\mathcal{P}_1 = \mathcal{P}_2) \wedge (ff_1 \sqsubseteq_2 ff_2)$$

It remains to construct a correct $\mathbb{F}$ transformation function $F_{t2} : \mathbb{F} \rightarrow \mathbb{F}$ to complete the definition of $F_t : \mathfrak{D} \rightarrow \mathfrak{D}$.

The correctness of the transformation is proven by showing observational equivalence [CC02]: an abstraction function α_o is used to extract the relevant properties for correctness. An example prepared for our needs is given in Figure 6.2 for the transformation of flow facts in parallel to the code transformation. As already mentioned, the calculation of flow facts cannot be complete. Therefore, certain flow facts ff_a are given manually by the user (denoted by the operation a). Further flow information ff_{impl} is extracted by semantic code analysis denoted by the operation F_s. The resulting flow information is denoted $ff = ff_a \cup ff_{impl}$. Finally, the operation

$$F_t = F_{t1} \times F_{t2}$$

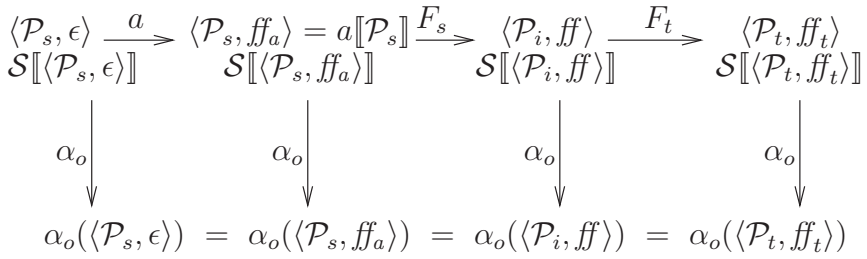


Figure 6.2: Observational Correctness of Transformation

represents the code optimisation performed by the compiler and the flow facts transformation performed in parallel. The correctness condition shown in Figure 6.2 requires that the observational abstraction α_o has an unchanged semantics $\mathcal{S}[\![\langle \mathcal{P}, ff \rangle]\!]$ for both code annotation and transformation (Definition 6.2.1).

$\mathcal{P}_i$ is the program which has been annotated with flow facts. $\mathcal{P}_i$ is transformed by the compiler to $\mathcal{P}_t$. ff has to be transformed to ff_t in parallel with the transformation of $\mathcal{P}_i$ to $\mathcal{P}_t$. Conventional WCET analysis tools will use $\mathcal{P}_t$ and ff_t as input to calculate the WCET.

Definition 6.2.1 (Extended Program Semantics $\mathcal{S}[\![\langle \mathcal{P}, ff \rangle]\!]$) *represents the semantics of program $\mathcal{P}$ under consideration of the flow facts ff. The CFP described by ff for a program $\mathcal{P}$ is denoted as $CFP_{ff}(\mathcal{P})$. $\mathcal{S}[\![\langle \mathcal{P}, ff \rangle]\!]$ is the standard program semantics $\mathcal{S}[\![\mathcal{P}]\!]$ with the additional constraint that the possible $CFP_{opt}(\mathcal{P})$ of $\mathcal{P}$ is a subset of $CFP_{ff}(\mathcal{P})$. If the $CFP_{opt}(\mathcal{P})$ of $\mathcal{P}$ is not a subset of $CFP_{ff}(\mathcal{P})$ then the extended program $\langle \mathcal{P}, ff \rangle$ is invalid.*

Definition 6.2.2 (Observational Correctness of $F : \mathbb{P}\times\mathbb{F} \to \mathbb{P}\times\mathbb{F}$) *A transformation $F : \mathbb{P}\times\mathbb{F} \to \mathbb{P}\times\mathbb{F}$ is defined to be correct for a given input tuple $\langle \mathcal{P}, ff \rangle \in \mathbb{P}\times\mathbb{F}$, iff the standard program semantics $\mathcal{S}[\![\mathcal{P}]\!]$ is not changed by the transformation F and $\langle \mathcal{P}, ff \rangle$ as well as $F(\langle \mathcal{P}, ff \rangle)$ are valid programs with respect to their* extended program semantics *(Definition 6.2.1).*
A formal definition of observational correctness for F is:

$$\begin{array}{rcl} \langle \mathcal{P}_2, ff_2 \rangle = F(\langle \mathcal{P}, ff \rangle) & \wedge & CFP_{opt}(\mathcal{P}) \subseteq CFP_{ff}(\mathcal{P}) \quad \wedge \\ \mathcal{S}[\![\mathcal{P}]\!] = \mathcal{S}[\![\mathcal{P}_2]\!] & \wedge & CFP_{opt}(\mathcal{P}_2) \subseteq CFP_{ff_2}(\mathcal{P}_2) \end{array}$$

To conclude, the correctness of the example given in Figure 6.2 requires that the observational correctness (Definition 6.2.2) holds for the transformations a, F_s, and F_t. The transformation $F_{t2} : \mathbb{F} \to \mathbb{F}$ has to be defined correctly so that the observational correctness of $F_t = F_{t1} \times F_{t2}$ is guaranteed.

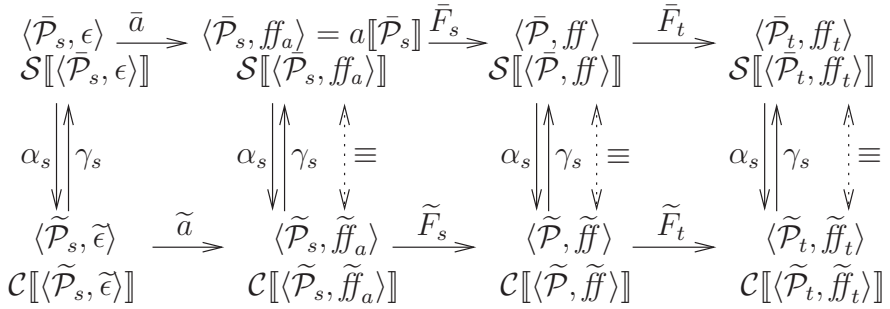


Figure 6.3: Transformation of Flow Facts

6.2.2 Transformation of Flow Facts

Based on the code annotation and transformation shown in Figure 6.2 we perform an abstract interpretation with *CFP* abstraction to correctly transform the flow facts in parallel to the code transformation F_{t1}. The extraction of flow facts ff_{impl} from the source code is not topic of our work. Work like [Gus00] is tackling this problem.

The concept of our method based on the theory of abstract interpretation to construct a correct ff transformation function $F_{t2} : \mathbb{F} \rightarrow \mathbb{F}$ is shown in Figure 6.3. The flow facts ff describe a closure for the possible *CFP* of a program $\mathcal{P}$. An abstract interpretation that operates on the structure of a program $\mathcal{P}$ is appropriate to induce a correct update function of the flow facts. The meaning of $\mathcal{S}[\![\langle \bar{\mathcal{P}}, ff \rangle]\!]$ and $\mathcal{C}[\![\langle \widetilde{\mathcal{P}}, \widetilde{ff} \rangle]\!]$ is explained after the construction of the concrete and abstract domains.

Construction of Concrete and Abstract Domains

The abstract interpretation operating on the program structure requires to abstract from the concrete program transformation. The abstraction can be done independently for the $\mathbb{P}$ and $\mathbb{F}$ attributes. We therefore use the *independent attribute method* described in Section 4.3.11 on page 80 to construct a Galois connection out of two separate Galois connections.

The construction of the Galois connection for the flow facts is trivial. Since the flow facts are already at a representation level that describes the control flow of a program, their abstraction can be constructed by a Galois isomorphism (see Section 4.3.7): $\langle\langle \mathbb{F}, \sqsubseteq_2 \rangle, \alpha_{\equiv}, \gamma_{\equiv}, \langle \widetilde{\mathbb{F}}, \tilde{\sqsubseteq}_2 \rangle\rangle$. The advantage of a Galois isomorphism compared to a Galois connection is that data are converted between abstract and concrete domain without loss of information.

The construction of the Galois connection to abstract the program representation $\mathbb{P}$ requires more considerations. The five steps described in Section 4.3.7 to construct a Galois connection are:

1. *Construction of a concrete domain* $\langle \bar{\mathbb{P}}, \bar{\sqsubseteq}_1 \rangle$: It is intended to use a program abstraction based on the structure of a program $\mathcal{P} \in \mathbb{P}$. The program structure will contain information like control-flow and loop scopes. Constructing an ap-

propriate partial order for a simple concrete domain like $\langle \mathbb{P}, \sqsubseteq_1 \rangle$ is not possible because the concretisation from a code structure cannot be mapped to a single program $\mathcal{P} \in \mathbb{F}$ due to information loss by the abstraction. The solution is to lift the programs $\mathcal{P} \in \mathbb{P}$ to sets of programs $\bar{\mathcal{P}} \in \bar{\mathbb{P}}$ with $\bar{\mathbb{P}} : \wp(\mathbb{P})$ and the additional restriction that all programs $\mathcal{P} \in \bar{\mathcal{P}}$ have the same code structure, denoted by $\forall \mathcal{P}_1 \in \bar{\mathcal{P}}_1, \forall \mathcal{P}_2 \in \bar{\mathcal{P}}_2 : \ (\bar{\mathcal{P}}_1 = \bar{\mathcal{P}}_2) \rightarrow (struct(\mathcal{P}_1) = struct(\mathcal{P}_2))$.
The partial order $\langle \bar{\mathbb{P}}, \bar{\sqsubseteq}_1 \rangle$ can be now defined as:

$$\forall \bar{\mathcal{P}}_1, \bar{\mathcal{P}}_2 \in \bar{\mathbb{P}} : \ \bar{\mathcal{P}}_1 \ \bar{\sqsubseteq}_1 \ \bar{\mathcal{P}}_2 \Leftrightarrow \bar{\mathcal{P}}_1 \subseteq \bar{\mathcal{P}}_2$$

2. *Construction of the corresponding abstract domain* $\langle \widetilde{\mathbb{P}}, \tilde{\sqsubseteq}_1 \rangle$: The abstract program domain $\langle \widetilde{\mathbb{P}}, \tilde{\sqsubseteq}_1 \rangle$ is designed to represent the unique code structure of a program set $\bar{\mathcal{P}} \in \bar{\mathbb{P}}$ which is calculated by the function $struct : \mathbb{P} \rightarrow \widetilde{\mathbb{P}}$. The domain $\langle \widetilde{\mathbb{P}}, \tilde{\sqsubseteq}_1 \rangle$ is a "flat poset": $\forall \widetilde{\mathcal{P}}_1, \widetilde{\mathcal{P}}_2 \in \widetilde{\mathbb{P}} : \ \widetilde{\mathcal{P}}_1 \ \tilde{\sqsubseteq}_1 \ \widetilde{\mathcal{P}}_2 \Leftrightarrow \widetilde{\mathcal{P}}_1 = \widetilde{\mathcal{P}}_2$.

3. *Correctness relation* $\mathcal{R}_s$: The correctness relation $\mathcal{R}_s : \bar{\mathbb{P}} \times \widetilde{\mathbb{P}} \rightarrow \{true, false\}$ is defined as $\bar{\mathcal{P}} \mathcal{R}_s \widetilde{\mathcal{P}} \Leftrightarrow (\forall \mathcal{P} \in \bar{\mathcal{P}} : \ struct(\mathcal{P}) \ \tilde{\sqsubseteq}_1 \ \widetilde{\mathcal{P}})$. Since each $\mathcal{P} \in \bar{\mathcal{P}}$ has the same program structure $struct(\mathcal{P})$, the resulting representation function $\beta_s : \bar{\mathbb{P}} \rightarrow \widetilde{\mathbb{P}}$ is calculated as follows: $\forall \bar{\mathcal{P}} \in \bar{\mathbb{P}} : \ (\mathcal{P} \in \bar{\mathcal{P}}) \Rightarrow (\beta_s(\bar{\mathcal{P}}) = struct(\mathcal{P}))$.

4. *Check for the existence of a best approximation*: Because the domain $\langle \widetilde{\mathbb{P}}, \tilde{\sqsubseteq}_1 \rangle$ is designed as a "flat poset" (there exists a unique abstract property that represents a concrete property), it directly follows that $\forall \bar{\mathcal{P}} \in \bar{\mathbb{P}}, \forall \widetilde{\mathcal{P}} \in \widetilde{\mathbb{P}}, \exists \widetilde{\mathcal{P}}_1 \in \widetilde{\mathbb{P}} : \ \bar{\mathcal{P}} \mathcal{R}_s \widetilde{\mathcal{P}}_1 \wedge (\bar{\mathcal{P}} \mathcal{R}_s \widetilde{\mathcal{P}} \Rightarrow \widetilde{\mathcal{P}}_1 \ \tilde{\sqsubseteq} \ \widetilde{\mathcal{P}})$.

5. *Calculation of the abstraction function* α_s *and the concretisation function* γ_s: The abstraction function $\alpha_s : \bar{\mathbb{P}} \rightarrow \widetilde{\mathbb{P}}$ is calculated as follows: $\forall \bar{\mathcal{P}} \in \bar{\mathbb{P}} : \ \alpha_s(\bar{\mathcal{P}}) = \beta_s(\bar{\mathcal{P}})$. The concretisation function $\gamma_s : \widetilde{\mathbb{P}} \rightarrow \bar{\mathbb{P}}$ calculates the set of all programs that match the given program structure: $\gamma_s(\widetilde{\mathcal{P}}) = \sqcup \{\bar{\mathcal{P}} \in \bar{\mathbb{P}} \,|\, \beta(\bar{\mathcal{P}}) \ \tilde{\sqsubseteq}_1 \ \widetilde{\mathcal{P}}\}$. It is important to note that $\gamma_s(\widetilde{\mathcal{P}})$ cannot be calculated in practice, since it results in a set of infinite programs. However, the calculation of $\gamma_s(\widetilde{\mathcal{P}})$ is not required as we use *abstract co-interpretation* (see Definition 6.2.3 on page 106) to induce $\bar{F}_{t2}$.

It is interesting to note that the above defined Galois connection $\langle \langle \bar{\mathbb{P}}, \bar{\sqsubseteq}_1 \rangle, \alpha_s, \gamma_s, \langle \widetilde{\mathbb{P}}, \tilde{\sqsubseteq}_1 \rangle \rangle$ also forms a Galois insertion (see Section 4.3.7).

The Galois insertion $\langle \langle \bar{\mathbb{P}}, \bar{\sqsubseteq}_1 \rangle, \alpha_s, \gamma_s, \langle \widetilde{\mathbb{P}}, \tilde{\sqsubseteq}_1 \rangle \rangle$ and the Galois isomorphism $\langle \langle \mathbb{F}, \sqsubseteq_2 \rangle, \alpha_{\equiv}, \gamma_{\equiv}, \langle \widetilde{\mathbb{F}}, \tilde{\sqsubseteq}_2 \rangle \rangle$ are combined using the *independent attribute method* (Section 4.3.11) to construct the new Galois insertion $\langle \langle \bar{\mathbb{P}} \times \mathbb{F}, \bar{\sqsubseteq} \rangle, \alpha_s \times \alpha_{\equiv}, \gamma_s \times \equiv, \langle \widetilde{\mathbb{P}} \times \widetilde{\mathbb{F}}, \tilde{\sqsubseteq} \rangle \rangle$.

The Semantics of the Concrete and Abstract Domains

In Figure 6.3, the semantics of the concrete domain $\langle \bar{\mathbb{P}} \times \mathbb{F}, \bar{\sqsubseteq} \rangle$ and the abstract domain $\langle \widetilde{\mathbb{P}} \times \widetilde{\mathbb{F}}, \tilde{\sqsubseteq} \rangle$ are denoted by $\mathcal{S}[\![\langle \bar{\mathcal{P}}, ff \rangle]\!]$ and $\mathcal{C}[\![\langle \widetilde{\mathcal{P}}, \widetilde{ff} \rangle]\!]$.

$\mathcal{S}[\![\langle\bar{\mathcal{P}}, \bar{ff}\rangle]\!]$ represents the *extended program semantics* (Definition 6.2.1) for all programs $\mathcal{P} \in \bar{\mathcal{P}}$. Since we use a special interpretation – which we call *abstract co-transformation* – we do not need to calculate the concretisation function $\gamma_s : \widetilde{\mathbb{P}} \rightarrow \bar{\mathbb{P}}$. As a consequence, the program set $\bar{\mathcal{P}}$ of $\langle\bar{\mathcal{P}}, \bar{ff}\rangle$ constains only a single program which has to be valid in terms of the *extended program semantics*.

The abstract semantics $\mathcal{C}[\![\langle\widetilde{\mathcal{P}}, \widetilde{ff}\rangle]\!]$ describes $CFP_{\bar{ff}}(\mathcal{P})$, a closure for the possible control flow paths $CFP_{opt}(\mathcal{P})$ during the execution of a program $\mathcal{P} \in \bar{\mathcal{P}}$. The code structure information of $\langle\widetilde{\mathcal{P}}, \widetilde{ff}\rangle$ may contain for example the control-flow graph (CFG) and information about loop scopes.

Construction of a Safe Approximation to Calculate F_t

Based on the concrete domain $\langle\bar{\mathbb{P}}, \bar{\sqsubseteq_1}\rangle$ and the program transformation function $F_{t1} : \mathbb{P} \rightarrow \mathbb{P}$ we construct an interpretation $\langle\langle\bar{\mathbb{P}}, \bar{\sqsubseteq_1}\rangle, \bar{F}_{t1}\rangle$ using the following transition function:

$$\forall\bar{\mathcal{P}} \in \bar{\mathbb{P}} : \ \bar{F}_{t1}(\bar{\mathcal{P}}) = \{F_{t1}(\mathcal{P}) \mid (\mathcal{P}{\in}\bar{\mathcal{P}}) \wedge defined(F_{t1}(\mathcal{P}))\}$$

The constraint $defined(F_{t1}(\mathcal{P}))$ is required since F_{t1} is not a total function over all programs $\mathcal{P}$ of the set $\bar{\mathcal{P}}$ of programs with the same code structure. The use of $defined()$ is only necessary for formal completeness since we never use the concretisation function γ_s for the calculation of F_t.

The concrete interpretation $\langle\langle\bar{\mathbb{P}}{\times}\mathbb{F}, \bar{\sqsubseteq}\rangle, \bar{F}_t\rangle$ of the concrete transformation of programs with attached flow facts has the following transition function $\bar{F}_t : \bar{\mathbb{P}}{\times}\mathbb{F} \rightarrow \bar{\mathbb{P}}{\times}\mathbb{F}$:

$$\bar{F}_t = \bar{F}_{t1}{\times}F_{t2}$$

To calculate F_{t2} we construct $\langle\langle\widetilde{\mathbb{P}}{\times}\widetilde{\mathbb{F}}, \tilde{\sqsubseteq}\rangle, \widetilde{F}_t\rangle$ with the transition function $\widetilde{F}_t : \widetilde{\mathbb{P}}{\times}\widetilde{\mathbb{F}} \rightarrow \widetilde{\mathbb{P}}{\times}\widetilde{\mathbb{F}}$:

$$\widetilde{F}_t = \widetilde{F}_{t1}{\times}\widetilde{F}_{t2}$$

as a safe $\gamma_s{\times}\gamma_{\equiv} - approximation$ of $\langle\langle\bar{\mathbb{P}}{\times}\mathbb{F}, \bar{\sqsubseteq}\rangle, \bar{F}_t\rangle$. The construction of a sound operation $\widetilde{F}_t$ is done by fulfilling Equation 6.1. $\widetilde{F}_{t1}$ is the abstraction of $\bar{F}_{t1}$ by transforming a program's code structure.

$$\forall\langle\widetilde{\mathcal{P}}, \widetilde{ff}\rangle \in \widetilde{\mathbb{P}}{\times}\widetilde{\mathbb{F}} : \ \bar{F}_{t1}(\gamma_s(\widetilde{\mathcal{P}})) \sqsubseteq_1 \ \gamma_s(\widetilde{F}_{t1}(\widetilde{\mathcal{P}})) \ \wedge \ F_{t2}(\gamma_{\equiv}(\widetilde{ff})) \sqsubseteq_2 \ \gamma_{\equiv}(\widetilde{F}_{t2}(\widetilde{ff})) \qquad (6.1)$$

The flow facts transformation function $\widetilde{F}_{t2}$ can be directly calculated from $\widetilde{F}_{t1}$. $\widetilde{F}_{t1}$ describes the structural program transformation including semantic information about the transformation describing the update of the program's control flow. An example for such a control-flow update information is the information known by the compiler for the update of the iteration bound of the modified loop when performing the code transformation *loop unrolling* (as described in Section 5.3.5). The information about the structural program transformation of $\widetilde{F}_{t1}$ is sufficient to describe the transformation of the flow facts done by $\widetilde{F}_{t2}$.

Definition 6.2.3 (Abstract co-interpretation)
Assumptions:

- $\langle\langle \mathfrak{D}_1, \sqsubseteq_1\rangle, \alpha_1, \gamma_1, \langle \widetilde{\mathfrak{D}}_1, \widetilde{\sqsubseteq_1}\rangle\rangle$ *and* $\langle\langle \mathfrak{D}_2, \sqsubseteq_2\rangle, \alpha_2, \gamma_2, \langle \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq_2}\rangle\rangle$ *are two Galois connections with independent attributes and the Galois connection* $\langle\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq\rangle, \alpha_1 \times \alpha_2, \gamma_1 \times \gamma_2, \langle \widetilde{\mathfrak{D}}_1 \times \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq}\rangle\rangle$ *has been constructed based on the* independent attribute method *(see Section 4.3.11).*

- $\langle\langle \widetilde{\mathfrak{D}}_1, \widetilde{\sqsubseteq_1}\rangle, \widetilde{F}_1\rangle$ *is a safe* γ_1 *– approximation of* $\langle\langle \mathfrak{D}_1, \sqsubseteq_1\rangle, F_1\rangle$ *and* $\langle\langle \widetilde{\mathfrak{D}}_1 \times \widetilde{\mathfrak{D}}_2, \widetilde{\sqsubseteq}\rangle, \widetilde{F}_1 \times \widetilde{F}_2\rangle$ *is a safe* $\gamma_1 \times \gamma_2$ *– approximation of* $\langle\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq\rangle, F_1 \times F_2\rangle$.

A definition of a function $\widetilde{F}_2$ *that can be implied from the transformation performed by* $\widetilde{F}_1$ *is denoted as* $\widetilde{F}_2 = impl(\widetilde{F}_2/\widetilde{F}_1)$. *If the implied function* $\widetilde{F}_2 = impl(\widetilde{F}_2/\widetilde{F}_1)$ *fulfils the following condition*

$$\forall \langle \widetilde{d}_1, \widetilde{d}_2\rangle \in \widetilde{\mathfrak{D}}_1 \times \widetilde{\mathfrak{D}}_2 : \; F_2(\gamma_2(\widetilde{d}_2)) \sqsubseteq_2 \; \gamma_2(\widetilde{F}_2(\widetilde{d}_2))$$

then $\langle\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq\rangle, F_1 \times \gamma_2 \circ impl(\widetilde{F}_2/\widetilde{F}_1) \circ \alpha_2\rangle$ *is a safe approximation of* $\langle\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq\rangle, F_1 \times F_2\rangle$.
The approximation $\langle\langle \mathfrak{D}_1 \times \mathfrak{D}_2, \sqsubseteq\rangle, F_1 \times \gamma_2 \circ impl(\widetilde{F}_2/\widetilde{F}_1) \circ \alpha_2\rangle$ *is called an* abstract co-interpretation.

Since $\langle\langle \mathbb{F}, \sqsubseteq_2\rangle, \alpha_{\equiv}, \gamma_{\equiv}, \langle \widetilde{\mathbb{F}}, \widetilde{\sqsubseteq_2}\rangle\rangle$ is a Galois isomorphism (i.e., $ff = \gamma_{\equiv}(\alpha_{\equiv}(ff))$ and $\widetilde{ff} = \alpha_{\equiv}(\gamma_{\equiv}(\widetilde{ff}))$) we can use *abstract co-interpretation* (as defined in Definition 6.2.3) to construct $\langle\langle \bar{\mathbb{P}} \times \mathbb{F}, \bar{\sqsubseteq}\rangle, \bar{F}_{t1} \times \gamma_{\equiv} \circ impl(\widetilde{F}_{t2}/\widetilde{F}_{t1}) \circ \alpha_{\equiv}\rangle$ as an approximation of $\langle\langle \bar{\mathbb{P}} \times \mathbb{F}, \bar{\sqsubseteq}\rangle, \bar{F}_t\rangle$. This approximation is safe, because Equation 6.2 follows from the definition of the *abstract co-interpretation*. Further, as $\langle\langle \mathbb{F}, \sqsubseteq_2\rangle, \alpha_{\equiv}, \gamma_{\equiv}, \langle \widetilde{\mathbb{F}}, \widetilde{\sqsubseteq_2}\rangle\rangle$ is a Galois isomorphism it follows that even Equation 6.3 holds and therefore, it follows that this approximation is also precise: $F_{t2} = \gamma_{\equiv} \circ impl(\widetilde{F}_{t2}/\widetilde{F}_{t1}) \circ \alpha_{\equiv}$.

$$\forall \langle \bar{\mathcal{P}}, ff\rangle \in \bar{\mathbb{P}} \times \mathbb{F} : \; \langle \bar{F}_{t1}(\bar{\mathcal{P}}), F_{t2}(ff)\rangle \quad \sqsubseteq \quad \langle \bar{F}_{t1}(\bar{\mathcal{P}}), \gamma_{\equiv}(\widetilde{F}_{t2}(\alpha_{\equiv}(ff)))\rangle \tag{6.2}$$

$$\forall \langle \bar{\mathcal{P}}, ff\rangle \in \bar{\mathbb{P}} \times \mathbb{F} : \; \langle \bar{F}_{t1}(\bar{\mathcal{P}}), F_{t2}(ff)\rangle \quad = \quad \langle \bar{F}_{t1}(\bar{\mathcal{P}}), \gamma_{\equiv}(\widetilde{F}_{t2}(\alpha_{\equiv}(ff)))\rangle \tag{6.3}$$

In this subsection we have described the theory to construct a correct flow facts transformation function $F_{t2} : \mathbb{F} \to \mathbb{F}$. The transformations that have to be performed by F_{t2} are discussed in the following subsection.

6.3 Flow Facts for WCET Calculation

Flow facts ff are hints describing constraints on the possible *control flow paths* (*CFP*). Possible sources for ff are syntax and semantics of the program code or additional annotations (ff_a).

Our WCET calculation is based on the implicit path enumeration technique (IPET), see Section 2.3.2. This calculation method transforms the structure of a program into a

set of flow constraints and allows to incorporate arbitrary flow facts describing iteration counts. Other methods, like tree-based [CP00, PK89] or path-based [HAM+99] WCET calculation, are in contrast limited to certain classes of structured flow facts.

The power of the IPET-based WCET calculation can be fully exploited by describing ff with

- markers,
- restrictions, and
- loop bounds.

This flow information to express (in)feasible paths is described in more detail in Section 2.3.3. A sample code demonstrating the usage of these code annotations is given in Figure 6.4. The code is written in WCETC, a language derived from ANSI C with grammar extensions to express ff_a inside the source code [Kir02]. Each loop is assigned with a lower and upper iteration bound. The safe modelling of certain code transformations requires both the lower and the upper iteration bound. This is true for best-case execution time (BCET) calculation as well as WCET calculation. *Markers* are used to label execution paths of the code. The *restrictions* are used to set the execution counts of several markers in relation to each other. In a restriction, numeric factors without a marker represent execution counts relative to the execution count of the surrounding *scope*. Restrictions have to be valid under all considered possible execution scenarios, e.g., using specific knowledge about the possible input data of a program can result in restrictions that describe the possible *CFP* more precisely.

In the sample code of Figure 6.4 we used the implicit semantic information of the code to specify two restrictions for the lower and upper execution bound of the conditional branch labelled by marker `m2`. For tight WCET results it is necessary to specify both upper and lower bounds since we do not know which branch will contribute more to the execution time. The analogous argument is valid for the calculation of the BCET.

6.3.1 Required Transformation of Flow Facts

As described in Section 6.2.2, the update of $\widetilde{ff}$ is induced by the *CFP* transformations done by $\widetilde{F}_{t1}$. In the following we assume the flow facts $\widetilde{ff}$ to be a tuple $\langle \Xi, \Gamma, \ell \rangle$ where Ξ is the set marker bindings, Γ the set of restrictions, and ℓ are the set of loop frames. Loop frames ℓ contain explicit information about loops, for example iteration bounds.

Typical compiler optimisations consist of a program analysis phase and a resulting program transformation phase which can be performed interleaved. By using the abstracted program transformation function $\widetilde{F}_{t1}$ it will be obvious that different code optimisations fall into the same class of abstract *CFP* transformations. This fact simplifies the design of a transformation function $\widetilde{F}_{t2}$ that is complete and correct.

Analysing the actions performed by $\widetilde{F}_{t1}$, we can identify the following operations performed at instruction level:

```
scope
{
  for (i=0,i<=m,i++)
     range 4...10 iterations
  {
     marker m1;
     if (i%2 == 0)
     {
        marker m2;
        arr[i] = d;
     }
     else
        arr[i] = i;
  }
  /* min.  exec.  count of m2 */
  restriction m1 <= 2*m2;
  /* max.  exec.  count of m2 */
  restriction 2*m2 <= m1+1;
}
```

Figure 6.4: Sample Code, annotated with Flow Facts

- insert
- move
- copy
- delete
- replace

Changing only the statements within a single basic block does not require to update Ξ, Γ or ℓ. But it becomes more complex when $\widetilde{F}_{t1}$ also includes structural changes of the *CFP*. Facing the operations of $\widetilde{F}_{t1}$ at single instruction level does not allow to induce precise operations to be performed by $\widetilde{F}_{t2}$. Depending on the context of the operation done by $\widetilde{F}_{t1}$ it could be required to

- duplicate involved $m \in \Xi$ and $r \in \Gamma$,
- duplicate involved $m \in \Xi$ and update $r \in \Gamma$ by the sum of original and new markers,
- duplicate involved $m \in \Xi$ and create new restrictions using the old and new markers,
- update the multiplication factor of certain terms in $r \in \Gamma$,
- delete involved $m \in \Xi$ and maybe also $r \in \Gamma$, or
- no update of $m \in \Xi$ or $r \in \Gamma$ required.

The transformations done by $\widetilde{F}_{t1}$ can also address more drastic *CFP* updates modifying loop bounds or loop scopes (information about loop nesting). In this case, in addition to the update of Ξ and Γ, it can be required to

- copy $l \in \ell$ when duplicating a loop,
- delete $l \in \ell$ when deleting a loop,
- modify $l \in \ell$ of the involved loop, or
- modify $l \in \ell$ of involved loops and create new $m \in \Xi$ and $r \in \Gamma$ to express dependencies between the iteration space of involved loops.

Without the knowledge of the overall structure update of $\widetilde{\mathcal{P}}$ it is not possible to decide which of the above $\widetilde{ff}$ updates would be required to maintain semantic correctness of the flow facts. As a consequence, we have to group these atomic operations done by $\widetilde{F}_{t1}$ into operations of coarser granularity and use the semantic context of the operations done by $\widetilde{F}_{t1}$ to induce the $\widetilde{ff}$ update.

The challenge for designing the $\widetilde{ff}$ update function is that there are numerous different code optimisations and each compiler may even handle them slightly different. To overcome this infeasible complexity, we systematically abstract the impact of each code optimisation to the changes of the *CFP*. As a result we get generic *CFP* update patterns for which we can induce the required $\widetilde{ff}$ update:

- SPLIT EXECUTION PATHS: A branch in block b_x in the *CFP* which leads to a block b_y is changed leading to another block b_z. An example for this transformation pattern are *jump optimisations*.

  Transformation: After setting the branch target in b_x to b_z, the execution count of b_y is decreased by the branching count of b_x. All restrictions using markers of the original path of b_y have to be updated with an additional marker assigned to the branching edge of b_x.

- MERGE EXECUTION PATHS: The branches of a conditional block are merged together. A typical example for such a transformation pattern is the introduction of *conditional moves*.

  Transformation: The execution count of the merged edge can be used as an upper bound for each of the conditional branches to update involved restrictions.

- DELETE EXECUTION PATHS: One or more blocks of code are deleted. Typical examples for such a transformation pattern are *dead code, unreachable code or common subexpression elimination*.

  Transformation: For all markers in the blocks to be deleted we have to update in a safe way all restrictions that use them. In the case we do not know anything about the execution count of the blocks to be deleted, a safe approximation has to be used. For example, if it is known at compile time that the code is unreachable (unreachable code can also be produced by prior code optimisations), the transformation is precise by just removing in all restrictions globally the terms using markers that are defined inside the blocks to be deleted.

Possible structural changes on the *CFP* involving iterations (i.e., cyclic subgraphs of the *CFG*) are:

- SPLIT ITERATION SPACE OF LOOP: The loop body gets to be executed outside the original loop. Examples of this are *loop unrolling, loop peeling.*

  Transformation: The original loop and any potentially new created loop have to get an updated loop bound. Additionally, a restriction for limiting the execution count of the original loop and the copies to the original loop bound is emitted.

- CHANGING LOOP SCOPE OF CODE: Blocks are moved from a certain loop scope to another loop scope. Examples of this are *loop unswitching, loop-invariant code motion.*

  Transformation: The restriction multiplication factor for all markers within the blocks have to be updated by the iteration bound of the new loop scope.

- CHANGE OF ITERATION COUNT: The control code of a loop is updated to perform a new number of iterations. Examples of this are *loop interchange, loop coalescing or loop vectorisation.*

  Transformation: The loop bound of the loop has to be updated. If the overall iteration count of the loop body is also changed, then all restrictions using markers from within the loop body have to be corrected. The iteration of the body may not be changed if this transformation pattern is performed on nested loops.

Using these abstractions, it is possible to define universal $\widetilde{ff}$ update functions. $\widetilde{F}_{t2}$ has to compose simple $\widetilde{ff}$ updates to perform the induced operations. Using these generic patterns simplifies the correctness proof of the induced function $\widetilde{F}_{t2}$ in relation to the abstract interpretation of the code transformation.

6.4 Chapter Summary

For correct and precise static WCET analysis, code transformations performed by an optimising compiler make it necessary to update flow facts given at source code level.

In this chapter we constructed an abstract framework to transform flow facts correctly and precisely in parallel to code transformations. This framework is based on the theory of abstract interpretation. The term *correctness* was defined by means of an *observational correctness* condition. Using this formalism it is possible to construct a correct flow facts transformation in a modular way, even for such generic and complex flow facts as described in Section 2.3.3.

Where is the life we have lost in living?
Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?

T. S. Eliot, *The Rock* (1934)

Chapter 7

Handling Flow Facts during Code Optimisation

The previous chapter describes design rules to construct a safe *ff* transformation framework. This chapter defines the fundamental sets of data tuples and operations that are necessary to build such a framework. The described operations can be combined to describe the correct *ff* update for even complex code optimisations performed by a compiler.

7.1 Data Tuples to Handle Flow Information

The representation of $\widetilde{ff}$ has to be simple but powerful enough to support correct $\widetilde{ff}$ updates during code optimisation. As described in Section 6.2.2, the construction of the $\widetilde{ff}$ transformation function $\widetilde{F}_{t2}$ is directly induced from the *CFP* update function $\widetilde{F}_{t1}$ by using the semantic code information required to allow the safe application of $\widetilde{F}_{t1}$.

Listing 7.1: example code including loop and conditional

```
1   for i:=1,n,1 do
2     if even(i)=1 then
3       a[i]:=a[i]+1;
4     else
5       skip
```

In this section we describe concrete data tuples for $\widetilde{\mathcal{P}}$ and $\widetilde{ff}$ to be used by the update function $\widetilde{F}_t = \widetilde{F}_{t1} \times \widetilde{F}_{t1}$. To demonstrate the application of these data structures we use the small sample code given in Listing 7.1. This code contains a loop and a simple conditional statement. For the loop it is assumed that semantic code analysis has found the iteration bound to be $[3 \ldots 7]$. The function even(n)$\mapsto \{0, 1\}$ returns 1 iff the argument n is a multiple of 2. The corresponding *CFG* for the code is given in Figure 7.1.

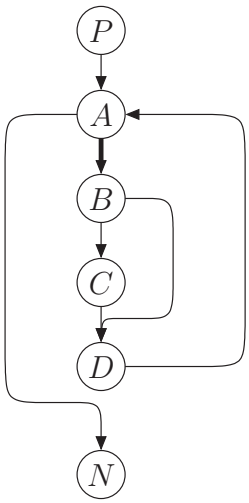


Figure 7.1: Syntactic Structure

7.1.1 The Abstract Program Representation

The program $\mathcal{P}$ is transformed by function F_{t1}. The abstraction $\widetilde{F}_{t1}$ of F_{t1} works on the abstract program representation $\widetilde{\mathcal{P}}$. To describe the operation of $\widetilde{F}_{t1}$ in more detail we present a data tuple representation for $\widetilde{\mathcal{P}}$. These data tuples are called CFPS (*CFP* structure) and are described in Table 7.1. These data tuples are suitable for the processing by the function $\widetilde{F}_{t1}$. CFPS represents the *CFP* of $\mathcal{P}$ that can be derived from the syntactic structure of the program $\mathcal{P}$ together with loop scope information. $\widetilde{\mathcal{P}}$ alone without $\widetilde{ff}$ is simply the control flow graph (CFG) of $\mathcal{P}$ extended with loop scope information LoopScope. The CFG of $\mathcal{P}$ is represented by the structure StatConn. The CFG nodes P of StatConn refer to single basic blocks in $\mathcal{P}$. Each control flow edge of StatConn is labelled with a flow type FType. The values "seq" and "bra" of FType are used to indicate sequential or branching control flow. Alternatively, the edges can be labelled by a numeric index Num to support generic CFGs. The loop scope information LoopScope is required for correct *ff* updates. LoopScope is a tree structure that represents the nesting levels of loops. A unique identifier Lid is assigned to each loop scope.

It is important to note that $\widetilde{\mathcal{P}}$ does not have to be calculated explicitely since in most compiler architectures it is implicitely represented by $\mathcal{P}$. Only the loop scopes may be an additional set of data tuples that has to be maintained.

Example: To give an example for the application of these data tuples, the program code given in Listing 7.1 is used. Its corresponding *CFG* is shown in Figure 7.1. The syntactic structure StatConn contains the following data:

$$\begin{aligned}\wp(\text{StatConn}) = \{\quad & \langle P, A, seq\rangle,\ \langle A, B, seq\rangle,\ \langle A, N, bra\rangle,\\ & \langle B, C, seq\rangle,\ \langle B, D, bra\rangle,\ \langle C, D, seq\rangle,\\ & \langle D, A, bra\rangle \qquad\qquad\qquad\qquad \}\end{aligned}$$

CFPS:	$\wp(\text{StatConn}) \times \wp(\text{LoopScope})$
StatConn:	P×P×FType
FType:	Num ∪ {seq, bra}
LoopScope:	Lid×Lid×P×P
P:	*... reference to basic block*
Lid:	*... identifier for loop scope*

Table 7.1: Data Tuples of *CFP* ($\widetilde{\mathcal{P}}$)

The additional information about loop scopes is given by the data tuples LoopScope (the identifier of the surrounding loop scope is written as "_"):

$$\wp(\text{LoopScope}) = \{ \quad \langle \_, L_1, A, D \rangle \quad \}$$

The data tuples CFPS contains only information that can be directly extracted from the program code. StatConn can be constructed while parsing the code. LoopScope can be constructed from the syntactic code structure or otherwise calculated by domination relations.

7.1.2 Representation of Flow Facts

The *CFP* of a program $\mathcal{P}$ is described by the syntactic structure $\widetilde{\mathcal{P}}$ of $\mathcal{P}$ together with the flow facts $\widetilde{ff}$. We define sets of data tuples for $\widetilde{ff}$ that store all the information to support a correct update of $\widetilde{ff}$ by the induced function $\widetilde{F}_{t2}$. These data tuples for $\widetilde{ff}$ are given in Table 7.2.

FF (which represents $\widetilde{ff}$) consists of a set of marker bindings for control-flow edges (MB), a set of restrictions (Restr) and a set of additional loop information (FFLF). The set of restrictions is the same for calculating the WCET and the BCET. Information like the loop bounds given in FFLF could be expressed directly by restrictions but is treated separately to have more semantic information available when modelling the *ff* transformations. The explicit transformation of loop bounds allows improved precision of the transformed *ff* in case of certain code optimisations. Loop bounds are later translated into a set of marker bindings and restrictions.

The structural changes resulting from several code optimisations make it necessary to keep the loop bounds as explicit values. As already mentioned, we maintain an upper and a lower loop bound value. For the final calculation of the WCET only the upper loop bound and for the BCET calculation only the lower loop bound is required. But for the safe *ff* update in case of certain loop transformations (e.g., loop unrolling), the lower and the upper iteration bound of loops are required for both calculations. A loop has also assigned two markers to express the execution frequency of entering the loop and executing the loop body.

The data tuple set FF described above is flexible enough to allow a safe update of $\widetilde{ff}$ during transformations of the program $\mathcal{P}$.

FF:	$\wp$(MB)×$\wp$(Restr)×$\wp$(FFLF)
MB:	Marker×StatConn
Restr:	$\wp$(Term)×Rel×$\wp$(Term)
Term:	Num×Marker
Rel:	$\{=,<,\leq\}$
FFLF:	Lid×Bound×LoopMarker
Bound:	Num×Num
LoopMarker:	Marker×Marker
Marker:	*... reference to a marker name*

Table 7.2: Data Tuples of Flow Facts ($\widetilde{ff}$)

Example: To give an example for the application of these data tuples, the program code given in Listing 7.1 is used. Using the *CFG* from Figure 7.1, the following marker bindings are used for modelling the flow facts:

$$\wp(\text{MB}) = \{ \quad \langle M_2, \langle P, A, seq\rangle\rangle,\ \langle M_3, \langle B, C, seq\rangle\rangle, \\ \langle M_4, \langle B, D, bra\rangle\rangle,\ \langle M_5, \langle A, B, seq\rangle\rangle \quad \}$$

Analysing the code, it is possible to derive additional information about (in)feasible control flow paths. Using the notation described in Section 2.3.2, the following linear constraints can be derived: $2 \cdot BC[s] \leq AB[s]$ and $2 \cdot BD[s] \leq AB[s] + PA[s]$. The data tuple set Restr contains the following constraints:

$$\wp(\text{Restr}) = \{ \qquad \langle\{\langle 2, M_3\rangle\},\ \leq,\ \{\langle 1, M_5\rangle\}\rangle, \\ \langle\{\langle 2, M_4\rangle\},\ \leq,\ \{\langle 1, M_5\rangle, \langle 1, M_2\rangle\}\rangle \quad \}$$

The loop scope information together with the loop iteration bound of $[3 \ldots 7]$ is modelled by the data tuples FFLF:

$$\wp(\text{FFLF}) = \{ \quad \langle L_1, \langle 3, 7\rangle, \langle M_0, M_1\rangle\rangle \quad \}$$

The data tuple set FF described above contains the additional $\widetilde{ff}$ required to calculate the possible *CFP* of the code. The markers $\langle M_0, M_1\rangle$ are special markers assigned to the loop scope L_1. M_0 refers to the iteration count of the loop entry in L_1 and M_1 refers to the iteration count of the loop body in L_1. The markers $\langle M_0, M_1\rangle$ are used to express iteration bounds for the loop scope L_1. Further, these references $\langle M_0, M_1\rangle$ to the iteration count of loop scope L_1 allows to create additional constraints – beside the loop bounds – that restrict the iteration count of a loop scope.

7.1.3 Transformation of Flow Facts

After the definition of the set of data tuples for $\widetilde{ff}$ we now mention the required $\widetilde{ff}$ transformations. The update of $\widetilde{ff}$ is induced by the *CFP* transformations done by $\widetilde{F}_{t1}$. We use the data symbols $\widetilde{ff} = \langle \Xi, \Gamma, \ell\rangle \in \text{FF}$ where Ξ is the set of marker bindings, Γ the set of restrictions, and ℓ is the set of loop frames.

Typical compiler optimisations consist of a program analysis phase and a resulting program transformation phase which can also be performed interleaved. By using the abstract program transformation function $\widetilde{F}_{t1}$ it becomes obvious that different code optimisations fall into the same class of abstract *CFP* transformations. This fact simplifies the design of a transformation function $\widetilde{F}_{t2}$ that is both complete and correct.

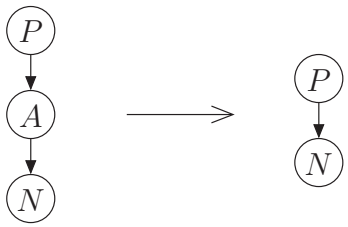


Figure 7.2: Deleting a Basic Block

Code transformations at the instruction level do not require an update of Ξ, Γ or ℓ. Only transformations that change the data tuples of *CFP* have to be considered. To demonstrate the required $\widetilde{ff}$ transformation, we use the simple example in Figure 7.2 where a basic block with only one successor and predecessor node is deleted. This operation requires the following *CFP* update done by $\widetilde{F}_{t1}$:

$$\textsc{StatConn}' = (\textsc{StatConn} \;/\; (\langle P, A, seq\rangle \cup \langle A, N, seq\rangle)) \cup \langle P, N, seq\rangle$$

By $mN_iN_j[t]$ we denote the reference "*name*" for the marker $\langle name, \langle N_i, N_j, t\rangle\rangle \in$ MB. For $t \in$ FType with index "s" we denote a sequential and with "b" a branching control flow type. Using this notation, the above code transformation induces the following $\widetilde{ff}$ update performed by $\widetilde{F}_{t2}$:

$$\text{MB}' = \text{MB} \;/\; (mPA[s] \cup mAN[s]) \cup mPN[s]$$

$$\begin{aligned}
\forall t_1 \in \textsc{Term} : ((\exists \langle t_1, r, t_2\rangle \in \textsc{Restr}) \wedge (t_1 = \langle n, mPA[s]\rangle)) \longrightarrow \\
(\textsc{Restr}' = (\textsc{Restr}/t_1) \cup \langle n, mPN[s]\rangle) \\
\forall t_1 \in \textsc{Term} : ((\exists \langle t_1, r, t_2\rangle \in \textsc{Restr}) \wedge (t_1 = \langle n, mAN[s]\rangle)) \longrightarrow \\
(\textsc{Restr}' = (\textsc{Restr}/t_1) \cup \langle n, mPN[s]\rangle) \\
\forall t_2 \in \textsc{Term} : ((\exists \langle t_1, r, t_2\rangle \in \textsc{Restr}) \wedge (t_2 = \langle n, mPA[s]\rangle)) \longrightarrow \\
(\textsc{Restr}' = (\textsc{Restr}/t_2) \cup \langle n, mPN[s]\rangle) \\
\forall t_2 \in \textsc{Term} : ((\exists \langle t_1, r, t_2\rangle \in \textsc{Restr}) \wedge (t_2 = \langle n, mAN[s]\rangle)) \longrightarrow \\
(\textsc{Restr}' = (\textsc{Restr}/t_2) \cup \langle n, mPN[s]\rangle)
\end{aligned}$$

As one can see, even formalising such a simple *CFP* and $\widetilde{ff}$ update leads to a quite long result. Therefore, we developed a compact generic transformation description based on graph transformations.

IndTrans:	⟨Opt,TransCFPS,TransFF⟩
TransCFPS:	$\wp$(StatConn) $\longrightarrow$ $\wp$(StatConn)
TransFF:	TransMB×TransRestr×TransFFLF
TransMB:	MB $\longrightarrow$ $\wp$(MB)
TransRestr:	Term $\longrightarrow$ $\wp$(Term)
TransFFLF:	FFLF $\longrightarrow$ $\wp$(FFLF)
Opt:	*. . . identification of optimisation type*

Table 7.3: Framework for Induced Flow Fact Update

7.2 Developing a Transformation Framework

For the specification of the induced $\widetilde{ff}$ updates we have developed a graph transformation framework. Graph transformation frameworks for specification purposes are described in [AEH+99] and for hierarchic graphs in [DHP02].

To describe generic code transformations we had to develop our own framework that supports graph hierarchies with "boundary-crossing" edges [DHP02]. The basic components of the transformation framework IndTrans = ⟨Opt,TransCFPS,TransFF⟩ are given in Table 7.3. Opt is simply a symbolic reference to label each code transformation. TransCFPS represents the *CFP* update done by $\widetilde{F}_{t1}$. The *ff* update is done by TransFF. TransFF is the induced *ff* by considering the semantic information of the code transformation. In the following, each of these basic components of IndTrans is described.

7.2.1 Specification of *CFP* Transformation

The specification TransCFPS of the performed *CFP* transformation is given by a graph representation supporting hierarchic transformations. Its modules are shown in Figure 7.3.

The specification of a basic block X_i with *arbitrary predecessor* or *successor edges* that are not modified during the transformation is shown with the symbols of Figure 7.3a and Figure 7.3c. If these edges have to be modified, we can also specify a generic name for the predecessor nodes P_i and the successor nodes N_i (Figure 7.3b and Figure 7.3d). A node having arbitrary predecessor and successor nodes is shown by Figure 7.3e. The number of such arbitrary edges is not specified and may even be zero. A *single edge* between two involved nodes is shown in Figure 7.3f. It is important to note that each node has only these edges explicitely described by single or arbitrary edges.

The specification of *hierarchies by composed blocks* is given in Figure 7.3g. Such blocks can consist of an arbitrary subgraph of composed or basic blocks. Composed blocks use the same notation for arbitrary edges as basic blocks (Figure 7.3h).

A loop scope with loop identifier L_x and lower/upper iteration bound is given in Figure 7.3i. The direct nesting of such loop scopes is shown in Figure 7.3j. Arbitrary nesting levels of loop scopes are denoted by the symbol given in Figure 7.3k. The

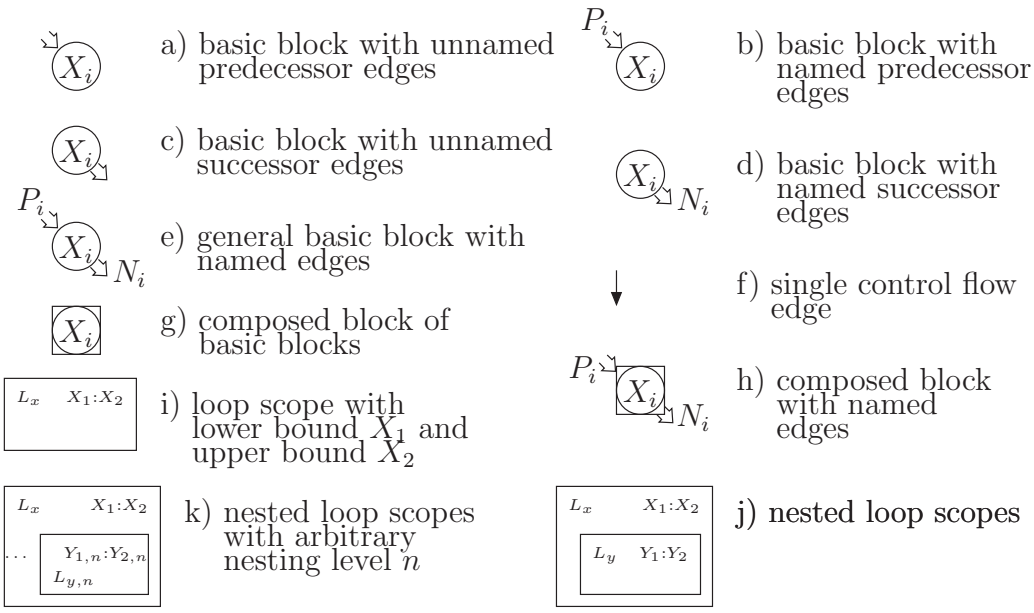


Figure 7.3: CFG Representation Symbols

individual nested loop scopes $L_{y,i}$ are marked by an additional index i with $1 \leq i \leq n$. For example, if we have a loop scope $L_{y,n}$ with loop nesting level n relative to a loop scope L_x then there exists a chain $\{L_{y,i} | 1 \leq i \leq n-1\}$ of loop nestings between them. The iteration count of the body of loop $L_{y,n}$ is therefore a multiple within the interval $[\prod_{i=1}^{n} Y_{1,i} \ldots \prod_{i=1}^{n} Y_{2,i}]$ of the iteration bound of the body of loop L_x. These arbitrary loop nesting levels are used to graphically describe code transformations that work on loops with arbitrary nesting levels without modifying the nested loops between them.

a) *control flow edge* that is not subject to *ff* updates or structural changes.

b) *control flow edge* that is subject to a change of its possible iteration count range or is created/deleted by the code transformation with *ff* assigned to it. The first case requires at least the update of involved restrictions and the second case the update of marker bindings to reflect the new CFG structure.

c) *control flow edge* that is created by the code transformation and has currently no *ff* assigned to it.

Figure 7.4: Special Highlighting for Single Control Flow Edges of the CFG

The structural description of code transformations by the graphical notation given in Figure 7.3 is only used to reflect the syntactic *CFG* changes. Additional semantic information about the possible execution frequency changes of control flow edges is given implicitely by the type of the performed code optimisation. This semantic information is available by the compiler and will be used to induce the *ff* update function $\widetilde{F}_{t2}$. To

support the development of $\widetilde{F}_{t2}$ we defined a notation to specify different types of control flow edges involved in the described code transformation. The different types of control flow edges and their meanings are summarised in Figure 7.4.

7.2.2 Specification of Induced *ff* Transformation

In the following we describe the three components of the *ff* update TransFF = TransMB × TransRestr × TransFFLF.

Update of Marker Bindings

The induced update of marker bindings is given by a transition sequence of the following form:

$$mN_iN_j[t] \xrightarrow{M} \{mN_kN_l[t_1],\ mN_mN_n[t_2],\ \ldots\}$$

The semantics is to remove the marker binding $mN_iN_j[t]$ and instead create the marker bindings $\{mN_kN_l[t_1],\ mN_mN_n[t_2],\ \ldots\}$. If the marker binding $mN_iN_j[t]$ *does not exist*, such a transition has no effect. If we want to delete the marker binding, we write

$$mN_iN_j[t] \xrightarrow{M} \emptyset$$

Specifying more than one transition with the same marker binding on the left has the same meaning as using one transition with a merged list of the right sides. All transitions are applied simultaneously.

Possible Optimisation for Implementation: if a target marker binding $mN_iN_j[t]$ from a marker binding transition does not occur in any restriction term after applying the restriction transition set of the current code transformation, then this marker binding can be deleted by using the transition $mN_iN_j[t] \xrightarrow{M} \emptyset$.

Handling Composed Blocks: The predicate $M_i(B)$ addresses the marker bindings for a *composed block* B. We can move the marker bindings from B to a new composed block C or just delete them. Moving the marker bindings from B to C does only work if both blocks have the same syntactical structure (e.g., a duplicated loop body after loop unrolling), because they will be attached to the equivalent position as they were in the original block. The transition of marker bindings from the composed block B to C is therefore written as:

$$M_i(B) \xrightarrow{M} M_i(C)$$

Handling Loop Markers: The two special markers assigned to each loop scope are referenced by special predicates. $LME(L_x)$ refers to the marker that labels the execution frequency of the entry of loop L_x and analogously $LMB(L_x)$ refers to the marker for the body of loop L_x. The transition of a marker binding from entry of loop L_x to entry of loop L_y is therefore written as:

$$LME(L_x) \xrightarrow{M} LME(L_y)$$

Update of Restrictions

The specification of TransRestr resembles the syntax of TransMB:

$$\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle n_1 \cdot mN_kN_l[t_1] \rangle,\ \langle n_2 \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$$

The semantics of this transition is to replace the term $\langle n \cdot mN_iN_j[t] \rangle$ in the left and right side of all restrictions by the list of terms $\{\langle n_1 \cdot mN_kN_l[t_1] \rangle, \langle n_2 \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$. If the term $\langle n_0 \cdot mN_iN_j[t] \rangle$ does not occur in any restriction, such a transition has no effect.

If there is more than one transition with the same term on the left side, the semantics is to *create copies* of the restriction so that all term updates are visible. This semantics differs from the semantics of TransMB. Again, all transitions are applied simultaneously. If we want to delete a restriction term, we write

$$\langle n \cdot mN_iN_j[t] \rangle \xrightarrow{R} \emptyset$$

If any of the two term sets of a restriction is empty, it is implicitely replaced by the constant "0".

Scaling Restriction Terms by an Interval: If the scaling value of a restriction term transition is a single value, it can be directly applied to a restriction term by only changing its multiplication value.

For some code transformations it could be the case that the relative change of the iteration count cannot be bound by a single value. Instead, it is expressed by an interval giving the lower and upper bound for the relative change. For example, when moving a block out of a loop scope and the lower and upper iteration bound of the involved loop are not equal, the relative change of the iteration count becomes an interval.

The resulting transition has the following form:

$$\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle [n_{11} \ldots n_{12}] \cdot mN_kN_l[t_1] \rangle,\ \langle [n_{21} \ldots n_{22}] \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$$

The semantics of this transition depends on the type of relation used by the restriction and the position of the term inside the restriction. If the relation of the restriction is "$<$" or "$\leq$" then for each interval from the left side of the transition we use the lower bound of the interval in the new term and analogous the upper interval bound for the right side. If the restriction where the term $\langle n \cdot mN_iN_j[t] \rangle$ occurs uses the relation "$=$" then we have to replace the whole restriction by two restrictions having an "$\leq$" relation to consider both the lower and the upper interval bound.

To give an example, for the transition $\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle [n_{11} \ldots n_{12}] \cdot mN_kN_l[t_1] \rangle\}$ the formal update of restrictions that contain marker bindings from the left side of the transition is as follows:

$$
\begin{array}{rcl}
\forall \textsc{Rel} \in \{\leq, <\}: & & \\
\langle TL_1 \cup \{\langle n_0, mN_iN_j[t]\rangle\},\ \textsc{Rel},\ TL_2\rangle & \longrightarrow & \langle TL_1 \cup \{\langle n_{11}, mN_kN_l[t_1]\rangle\},\ \textsc{Rel},\ TL_2\rangle \\
\langle TL_1,\ \textsc{Rel},\ TL_2 \cup \{\langle n_0, mN_iN_j[t]\rangle\}\rangle & \longrightarrow & \langle TL_1,\ \textsc{Rel},\ TL_2 \cup \{\langle n_{12}, mN_kN_l[t_1]\rangle\}\rangle \\
\forall \textsc{Rel} \in \{=\}: & & \\
\langle TL_1 \cup \{\langle n_0, mN_iN_j[t]\rangle\},\ \textsc{Rel},\ TL_2\rangle & \longrightarrow & \langle TL_1 \cup \{\langle n_{11}, mN_kN_l[t_1]\rangle\},\ \leq,\ TL_2\rangle, \\
 & & \langle TL_2,\ \leq,\ TL_1 \cup \{\langle n_{12}, mN_kN_l[t_1]\rangle\}\rangle \\
\langle TL_1,\ \textsc{Rel},\ TL_2 \cup \{\langle n_0, mN_iN_j[t]\rangle\}\rangle & \longrightarrow & \langle TL_2 \cup \{\langle n_{11}, mN_kN_l[t_1]\rangle\},\ \leq,\ TL_1\rangle, \\
 & & \langle TL_1,\ \leq,\ TL_2 \cup \{\langle n_{12}, mN_kN_l[t_1]\rangle\}\rangle
\end{array}
$$

This transition rule makes the involved restrictions less effective but still safe. The generalisation of this operation to the generic form of the transition given above is straightforward by using the whole term list from the right side of the transition with the lower respective upper bound of their scaling interval.

It is important to note that the above transition rule is designed for *normalised restrictions*. A restriction is normalised if the multiplication value n of each term $\langle n \cdot mN_iN_j[t]\rangle$ is positive. If this is not the case the restriction has to be normalised before applying the rule. This normalisation is done by moving the term from the current to the other term list and multiplying the multiplication value of the term by -1 to make it positive.

Handling Loop Markers: The predicates $LME(L_x)$ and $LMB(L_x)$ refer to the two special markers of a loop scope L_x. For example, the update of a term using the body marker of loop L_x is written as:

$$\langle n_0 \cdot LMB(L_x)\rangle \xrightarrow{R} \langle n_1 \cdot LMB(L_x)\rangle$$

Handling Composed Blocks: The predicate $M_i(B)$ represents all terms in restrictions referring to a marker binding in a *composed block* B. We can replace all terms using markers from block B by other terms using corresponding markers from block C or just delete them.

Replacing the terms using marker bindings in B with terms using marker bindings in C only works if both blocks have the same syntactical structure (e.g., a duplicated loop body after loop unrolling) because this transition also requires that the marker binding itself can be moved from B to C. The transition of restriction terms using marker bindings from the composed block B to terms using corresponding marker bindings in C is therefore written as:

$$\langle n_0 \cdot M_i(B)\rangle \xrightarrow{R} \langle n_1 \cdot M_i(C)\rangle$$

Update of Flow Facts for Loops

A loop frame is denoted as $L_x\langle l, u\rangle$. L_x is the loop identifier and $\langle l, u\rangle$ $(l \leq u)$ is the interval for the iteration bound of this loop.

The induced update of loop flow facts is given by a transition sequence of the following form:

$$L_x\langle l_0, u_0\rangle \xrightarrow{L} \{L_y\langle l_1, u_1\rangle, L_z\langle l_2, u_2\rangle, \ \ldots\}$$

The semantics is to remove the old loop information $L_x\langle l_0, u_0\rangle$ and instead create the new loop information $\{L_y\langle l_1, u_1\rangle, L_z\langle l_2, u_2\rangle, \ \ldots\}$.

If no loop information with the key L_x exists, such a transition has no effect. If the loop information has simply to be deleted, we write:

$$L_x\langle l_0, u_0\rangle \xrightarrow{L} \emptyset$$

If a new loop L_x having iteration bounds $\langle l, u\rangle$ is introduced (created) without modifying any existing loop, it is written as:

$$\emptyset \xrightarrow{L} L_x\langle l, u\rangle$$

7.2.3 Grouping *ff* Transitions for a Single Code Optimisation

The basic operations defined above are used to compose the *ff* update function TransFF = TransMB × TransRestr × TransFFLF. Simple code transformations require only a few of these basic operations to correctly update the *ff*. Code optimisations with more complex code transformations require longer transition sequences.

All transitions belonging to the *ff* update of a certain code optimisation have to be executed together. The compiler has to generate the *ff* transition sequences and group them for each code optimisation.

If the *ff* transition sequences are not grouped for each optimisation and executed simultaneously for each optimisation, the result would be a wrong *ff* update. To give an example, assume that the following two restriction term transitions belong to the same code optimisation:

$$\begin{aligned}\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot mBC[s]\rangle \\ \langle n \cdot mBC[s]\rangle &\xrightarrow{R} \langle n \cdot mBC[s] + n \cdot mCD[s]\rangle\end{aligned}$$

Executing these two transitions in sequence yields an illegal scaling of the restriction terms.

7.3 Properties of the Transformation Framework

This section shows the completeness and further properties of the presented *ff* update framework TransFF.

7.3.1 The Completeness of the Approach

After the definition of this transformation framework there is the question whether a correct *ff* transformation can be induced for every type of code transformation.

In this section we show that it is always possible to find a correct *ff* transformation. This method is quite simple and does not consider the semantic information known by the compiler about the performed code transformation. Therefore, depending on the structure of the transformed code, the resulting *ff* transformation function may be not very accurate, but still safe. At this point it is also important to remember that the simplest safe *ff* transformation function is to throw away all information about (in)feasible paths and transform only loop bound information. However, we show a way to induce a more accurate *ff* transformation function.

For showing the completeness of the *ff* transformation framework we have to distinguish between two types of *ff*. The first one is flow facts for loops (lower and upper iteration bounds) and the second one is additional information about (in)feasible paths. We do not have to show the completeness for the update of flow facts for loops, because the loop bound transformation is *always given directly* by the type of performed code transformation. For example, when we perform loop unrolling with an unrolling factor k, then we have to use this information (the unrolling factor k) to update the iteration bound of the loop. As another example, for loop duplication we can directly take the loop bound from the original loop and assign it to the copied loop. What we have to show is that there is also a way to transform the information about infeasible paths. In our framework, the information about infeasible paths is represented by the restriction terms. Therefore, we have to show the existence of a safe restriction term transformation function TransRestr which is described in Section 7.2.2.

First, we have to introduce several formal definitions that are required to show the completeness of the transformation framework regarding coverage of code transformations. To show this, we represent the program to be transformed as a control flow graph $G = \langle N, E, S, s, t\rangle$ where N is the set of nodes and $E : N \times N \times S$ is the set of different control flow edges. S is the set of control flow types which are used to select a specific control flow edge between two nodes. s denotes a unique start node and t a unique exit node of the program. Referring to the data tuples defined in Table 7.1, we can represent the basic blocks P by the set of nodes N and the control flow edges StatConn by the set E.

Based on the control flow graph $G = \langle N, E, S, s, t\rangle$ for the whole program, we use a so-called *modification graph* G_M that represents the subgraph of G that is affected by a given code transformation. The definition of G_M is given in Definition 7.3.1.

Definition 7.3.1 (Modification graph G_M) *A* modification graph G_M *is a subgraph of a given control flow graph* $G = \langle N, E, S, s, t\rangle$ *which represents the part of* G *that will be affected by a given code transformation.* G_M *is defined as*

$$G_M = \langle N_M, E_M, N_P, E_P, N_N, E_N\rangle$$

where $N_M \in N$ *and* $E_M \in E$ *are the nodes and edges that are changed by the given code*

transformations. $E_P \in E$ *is the set of input edges into* G_M *that link the subgraph* G_M *with the rest of the graph* G*. The start nodes of all these input edges are collected in* N_P*.* $E_N \in E$ *is the set of output edges from* G_M *that link the subgraph* G_M *with the rest of the graph* G*. The end nodes of all these output edges are collected in* N_N*. Based on* N_M *and* E_M *the modification graph* G_M *can be formally defined as:*

$$\begin{aligned} E_P &= \{e \mid e = \langle n, n', t\rangle \wedge (n \notin N_M) \wedge (n' \in N_M)\} \\ N_P &= \{n \mid \langle n, n', t\rangle \in E_P\} \\ E_N &= \{e \mid e = \langle n', n, t\rangle \wedge (n' \in N_M) \wedge (n \notin N_M)\} \\ N_N &= \{n \mid \langle n', n, t\rangle \in E_N\} \end{aligned}$$

Definition 7.3.2 (Domain of loop scopes $\langle \mathfrak{L}, \sqsubseteq, \sqcap, \bot\rangle$) *The set of loop scopes can be partially ordered regarding their nesting hierarchy. We define* L_0 *as the initial loop scope for the whole program, respective function. The function* $\{L_0, \ldots, L\} = nesting(L)$ *is used to calculate the set of all surrounding loop scopes for a given loop scope* L*. Based on the above assumptions, the domain* $\langle \mathfrak{L}, \sqsubseteq, \sqcap, \bot\rangle$ *is defined as a semi-lattice as follows:*

$$\begin{aligned} \forall L_1, L_2 \in \mathfrak{L}: \; L_1 \sqsubseteq L_2 \iff & \; nesting(L_1) \subseteq nesting(L_2) \\ \forall L_1, L_2 \in \mathfrak{L}: \; \sqcap(L_1, L_2) = L_x \iff & \; (L_x \sqsubseteq L_1) \wedge (L_x \sqsubseteq L_2) \wedge \\ & \; (\forall L_y \in \mathfrak{L}: \; ((L_y \sqsubseteq L_1) \wedge (L_y \sqsubseteq L_2) \wedge (L_x \sqsubseteq L_y)) \\ & \quad \rightarrow (L_x = L_y)) \end{aligned}$$

All control flow information about infeasible paths in our *ff* transformation framework is attached only to control flow edges. To perform an adequate scaling of restriction terms in case that flow facts are mapped from one control flow edge to another one within a different loop nesting level, it is required to define the loop scope of a control flow edge.

First, we denote by $L = scope(n)$ the loop scope of each node $n \in N$. The loop scope of $scope(n)$ for each node n is derived from the data tuple specification LoopScope given in Table 7.1. To calculate the loop nesting difference between two edges, we need a definition of a loop scope function $L = scope(e)$ for every edge $e \in E$. To specify the loop scope calculation function $scope(e)$ also for edges we use the domain $\langle \mathfrak{L}, \sqsubseteq, \sqcap, \bot\rangle$ over loop scopes as described in Definition 7.3.2. Based on this domain, the loop scope calculation function $scope(e)$ for edges $e = \langle n_1, n_2, t\rangle$ can be defined as

$$scope(\langle n_1, n_2, t\rangle) = \sqcap(scope(n_1), scope(n_2))$$

To calculate the required scaling of restriction terms in case that flow facts are mapped from one control flow edge to another, we have to calculate their common and their different set of surrounding loop scopes. These sets are calculated by the functions defined in Definition 7.3.3 and Definition 7.3.4.

Definition 7.3.3 (Common surrounding loop scopes: $CS(e_1, e_2)$) *The common subset of the surrounding loop scopes of two control flow edges* e_1 *and* e_2 *is calculated by the function* $CS(e_1, e_2)$*, which is defined as follows:*

$$\forall e_1, e_2 \in E: \; CS(e_1, e_2) = \{L \mid (L \sqsubseteq scope(e_1)) \wedge (L \sqsubseteq scope(e_2))\}$$

Definition 7.3.4 (Unique surrounding loop scopes: $US(e_1, e_2)$) *The unique surrounding loop scopes of an control flow edge e_1 compared to another control flow edge e_2 is calculated by the function $US(e_1, e_2)$, which is defined as follows:*

$$\forall e_1, e_2 \in E: \; US(e_1, e_2) \;=\; \{L \mid (L \sqsubseteq scope(e_1)) \wedge (L \notin CS(e_1, e_2))\}$$

Further, we assume that the lower iteration bound of a loop scope L is given by $LLB(L)$ and the upper iteration bound is given by $ULB(L)$. Based on the above definitions and the modification graph $G_M = \langle N_M, E_M, N_P, E_P, N_N, E_N \rangle$ we define the simple safe restriction-term-transformation function $\text{TRANSRESTR}_{simple}$ as follows:

$$\langle n \cdot mN_iN_j[t] \rangle$$

$$\xrightarrow{R}$$

$$\langle \sum_{\langle N_x, N_y, t' \rangle \in E_P} \left(\left[0 \ldots \frac{\prod_{L_a \in US(\langle N_i, N_j, t \rangle, \langle N_x, N_y, t' \rangle)} ULB(L_a)}{\prod_{L_b \in US(\langle N_x, N_y, t' \rangle, \langle N_i, N_j, t \rangle)} LLB(L_b)} \right] \cdot mN_xN_y[t'] \right) \rangle \tag{7.1}$$

The restriction term transition given in Equation 7.1 is safe since it maps the iteration count of every control flow edge $e \in E_M$ of the modification graph G_M to the iteration counts of all input edges E_P of G_M with proper scaling by multiplication factors resulting from different loop scopes of the edge e and each of the edges E_P. The scaling vector starting with zero has to be applied since the exact correlation between the execution counts of the edges E_P and the edge e is not considered.

In this section we have shown a simple way to transform *ff* without considering the implicit control flow information known by the compiler about a given code transformation. This transformation can be applied to every code transformation, but the resulting accuracy may be poor since the information represented by *ff* can be blurred depending on the type of code transformation.

7.3.2 Refinement of the Transformations

The previous section has shown that it is possible to map all flow information about infeasible paths to the border of the modification graph. It is important to note that this method does not exploit control flow information known by the compiler about the given type of code transformation. In this section we present a stepwise refinement towards an accurate *ff* transformation for all types of code transformation.

Refinement by Control-Flow Analysis; The simple transformation shown in Section 7.3.1 is used to show how the transformation of information about infeasible paths can be done automatically without using additional control flow information of the applied code transformation.

An obvious refinement is to map a restriction term from the current control-flow edge only to those input edges of the modification graph from which this control flow edge is reachable. Standard backward control-flow analysis can be used to test this reachability. However, as this refinement still does not consider semantic information of the applied code transformation, the obtained accuracy of the transformed flow information can still be poor depending on the structure of the code.

Modelling Simple Transformations; The next refinement is to exploit semantic information of the applied code transformation. Two simple control flow modification patterns are described to demonstrate the application of semantic information known by the compiler about the concrete type of code transformation.

- **Changing the iteration bound of a surrounding loop scope:** As already discussed in Section 7.3.1, the required update of flow facts for loops (e.g., the upper and upper iteration bound of a loop) can be alway extracted from the semantics of the applied code transformation.
  However, if the iteration count of a loop is changed, it is also required to update the control flow edges within this loop. The semantic control flow information known by the compiler is the relative change of the iteration count of the loop. Using this information, the loop flow facts FFLF can be scaled by this value.

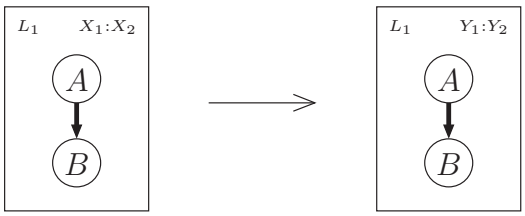


Figure 7.5: Changing the Iteration Bound of a Surrounding Loop Scope

The lower and upper iteration bound from both, the old and new loop flow facts FFLF can be used to update the information about infeasible path from edges within the loop. The code transformation shown in Figure 7.5 changes the iteration bound of the loop from $X_1 : X_2$ to $Y_1 : Y_2$. In the general case, it can happen that the resulting relative change of the loop execution count is not a constant value. In this case, it is required to scale the restriction term for the control flow edge $mAB[s]$ by a vector that safely approximates the execution count change of this edge:

$$\langle n \cdot mAB[s] \rangle \xrightarrow{R} \langle \left[n \cdot \frac{X_1}{Y_2} \ldots n \cdot \frac{X_2}{Y_1} \right] \cdot mAB[s] \rangle$$

In the other case, if the semantics of the applied code transformation guarantees that the iteration-bound change of the surrounding loop is given by

a constant value, then the induced flow facts update can be performed precisely. Assuming that the new execution count is k times higher than the old one, then the following precise transformation can be used instead:

$$\langle n \cdot mAB[s] \rangle \xrightarrow{R} \langle n{\cdot}k \cdot mAB[s] \rangle$$

Similar rules can be applied in case of introducing or deleting surrounding loops of an control flow edge.

- **Merging or splitting the control flow:**
  The transformation given in Figure 7.6 can be applied in two directions. The first is to merge/delete a control flow (a) and the second is to split a control flow (b). We will discuss in the following both types of transformations:

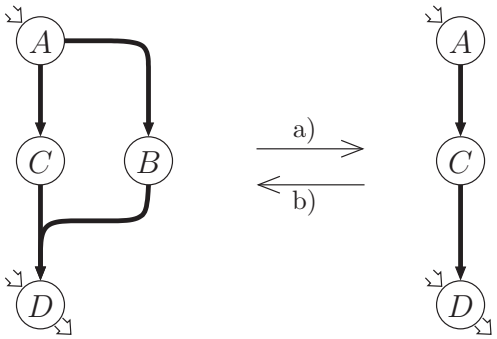


Figure 7.6: Merging/Deleting (a) or Splitting of Control Flow Paths (b)

a) Merging/deleting a control flow; For the transformation a) in Figure 7.6, the structural change alone does not describe the performed code transformation. Additional information about the semantics of the applied code transformation is required to transform the flow facts accurately.
The first possibility is that the control flow edges $mAC[s]$ and $mAB[b]$ have been merged together. In this case, the resulting flow facts transition rules becomes (with analogous transitions for $mCD[s]$ and $mBD[s]$):

$$\begin{aligned} mAB[b] &\xrightarrow{M} mAC[s] \\ \langle n \cdot mAC[s] \rangle &\xrightarrow{R} \langle [0 \ldots n] \cdot mAC[s] \rangle \\ \langle n \cdot mAB[b] \rangle &\xrightarrow{R} \langle [0 \ldots n] \cdot mAC[s] \rangle \end{aligned}$$

The second possibility is that it is statically known that the branch B will be never taken. The branch with B can be simply deleted and the other branch remains unchanged. In this case, the resulting flow facts transition rules becomes (with analogous transitions for $mCD[s]$ and $mBD[s]$):

$$\begin{aligned} mAB[b] &\xrightarrow{M} \emptyset \\ \langle n \cdot mAB[b] \rangle &\xrightarrow{R} \emptyset \end{aligned}$$

b) Splitting a control flow; The splitting of control flow can be handled easily with the presented *ff* transformation framework. It only requires to replace a restriction term for the original control flow edge by a set of restriction terms. For the transformation pattern b) in Figure 7.6 the following *ff* transition rules are induced (with analogous transitions for $mCD[s]$ and $mBD[s]$):

$$\begin{aligned} mAC[s] &\xrightarrow{M} mAC[s],\ mAB[b] \\ \langle n \cdot mAC[s] \rangle &\xrightarrow{R} \langle n \cdot mAC[s] + n \cdot mAB[b] \rangle \end{aligned}$$

These two examples have shown how it is possible to precisely model simple modification patterns. One important aspect was the use of semantic information known by the compiler about the code transformation to construct more accurate transition rules.

Generic Distribution of Control-Flow Information; In the previous refinement of the *ff* transformation we have shown how to model simple code modification patterns. Here we finally explain why this *ff* transformation framework is powerful enough to accurately handle every type of code transformation.

The key operation to describe the update of information about infeasible paths is the transition $\xrightarrow{R}$. Its general syntax is

$$\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle n_1 \cdot mN_kN_l[t_1] \rangle,\ \langle n_2 \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$$

By this operation it is possible to distribute information about the control flow of a specific control flow edge to an arbitrary set of control flow edges, each edge scaled by an individual scaling factor. With the above form it is possible to model each code transformation where the relative execution frequency change of all control flow edges can be expressed by a constant value.

By using intervals as scaling values, $\xrightarrow{R}$ can be also used to model the *ff* update for the transformation of code with fuzzy execution counts. Examples for code with fuzzy execution counts are all conditional constructs like variable loops or conditional statements. For the construction of safe and accurate *ff* transitions in case of fuzzy execution counts of control flow edges, the following extended format of $\xrightarrow{R}$ is used:

$$\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle [n_{11} \ldots n_{12}] \cdot mN_kN_l[t_1] \rangle,\ \langle [n_{21} \ldots n_{22}] \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$$

The transition $\xrightarrow{R}$ is powerful enough to model arbitrary control flow transformations of the code. The interesting question is how it is possible to get information

about which control flow transformation is performed. As already described, this information is available to the compiler. The following two types of semantic information about the code transformation can be distinguished:

- Information about the structure of the code. For several optimisations a certain structure is required to ensure the correct application of a given code transformation.
- Semantic information about the concrete code transformation. The structural update that is visible after the transformation is finished does not always allow to reason about the exact control flow modification that has been performed. This additional information is determined by the type of performed code transformation.

The flexible applicability of the restriction term transition $\xrightarrow{R}$ together with the knowledge of semantic information about the applied code transformation allows to handle arbitrary code transformations.

Concrete examples for the exploitation of semantic information about a given code transformation are given in Chapter 8.

7.3.3 Modelling Basic Operations of $\widetilde{F}_{t2}$

An abstract description of a *ff* transformation framework is given in Section 6.3.1. The following basic operations to update *ff* at instruction level are listed there:

- insert
- move
- copy
- delete
- replace

These operations are listed there in order to describe what type of *ff* updates can be induced from the abstract program transformation function $\widetilde{F}_{t1}$.

The concrete *ff* update function TransFF = TransMB × TransRestr × TransFFLF has been developed to efficiently handle the type of *ff* supported by our transformation framework. As a consequence, the granularity of TransFF is at a different level than the above basic *ff* update operations. The transitions of TransFF can perform several basic operations simultaneous. Further, depending on the context of a transition they may represent a different type of basic operation to update *ff*.

The following examples will demonstrate how these basic operations are modelled by the concrete *ff* transition rules of TransFF:

insert: An example for the *insert* operation is the creation of new restrictions. A marker binding transition like $mN_iN_j[t_1] \xrightarrow{M} \{mN_iN_j[t_1],\ mN_kN_l[t_2]\}$ can be seen as a combination of a *copy* and an *insert* operation.

move: Flow facts can be simply moved from a control-flow edge $mN_iN_j[t_1]$ to another control-flow edge $mN_kN_l[t_2]$ by using the transitions $mN_iN_j[t_1] \xrightarrow{M} mN_kN_l[t_2]$ and $\langle n_0 \cdot mN_iN_j[t_1]\rangle \xrightarrow{R} \langle n_0 \cdot mN_kN_l[t_2]\rangle$.

copy: A typical example for the *copy* operation is the duplication of loop iteration bounds by using the transition $L_x\langle l_0, u_0\rangle \xrightarrow{L} \{L_x\langle l_0, u_0\rangle, L_y\langle l_0, u_0\rangle\}$.

delete: Transitions like $mN_iN_j[t] \xrightarrow{M} \emptyset$, $\langle n \cdot mN_iN_j[t]\rangle \xrightarrow{R} \emptyset$, or $L_x\langle l_0, u_0\rangle \xrightarrow{M} \emptyset$ are used to *delete* certain flow facts.

replace: An example for the *replace* operation is a transition of a form like $\langle n_0 \cdot mN_iN_j[t]\rangle \xrightarrow{R} \langle n_1 \cdot mN_iN_j[t]\rangle$ or $\langle n_0 \cdot mN_iN_j[t]\rangle \xrightarrow{R} \langle [n_1 \ldots n_2] \cdot mN_iN_j[t]\rangle$. The semantics of such transitions is to reflect a change in the execution count of the control-flow edge $mN_iN_j[t]$.

More advanced *ff* updates that are composed of several basic operations can be modelled by a sequence of TransFF transitions.

7.4 Chapter Summary

To describe the correct and precise *ff* update in case of code optimisations it is necessary to specify an adequate representation for *ff* and define operations to transform the *ff*.

Within this chapter, two sets of data tuples have been defined; one to describe all important information for the abstract program representation and another one to describe related *ff* to limit the possible *CFP* of the program. It was shown that the use of quantified logic formulas is not an adequate formalism to describe the induced *ff* updates. Even for simple code transformations they result in a long list of formulas. Therefore, we developed a *ff* transformation framework with more powerful *ff* update operations. To obviously show the code transformations caused by code optimisations, we developed a special graph transformation framework. With this graph transformation framework, the abstract program transformation $\widetilde{F}_{t1}$ can be specified graphically. This representation simplifies the development of the correct *ff* update function by considering the control flow information known by the compiler for a concrete code transformation. *ff* update functions were developed that can be directly used to model the induced $\widetilde{F}_{t2}$ from the graphical description of the code transformation.

It is a capital mistake to theorise
before you have all the evidence.
It biases the judgement.

Sir Arthur C. Doyle, *Study in Scarlet* (1888)

Chapter 8

Developing Concrete Transformation Rules

In this chapter we describe the construction of concrete flow facts update rule for a given code transformation. These update rules use the flow facts transformation framework defined in Chapter 7. The semantic information known by the compiler about a code transformation is used to design precise flow facts update rules that are safe in the sense of the formal requirements described in Chapter 6.

8.1 General Considerations

The flow facts transformation rules described in this chapter are complete in the sense that they describe the update of flow facts attached to any modified control flow edge of the *CFG*. However, it is important to note that in practice only a few control flow edges will have flow facts attached to it. This reduces the flow facts transformation effort significantly.

For the calculation of iteration frequencies involving loop scopes we use the *loop bound reference* X_δ (defined in Definition 8.1.1) to mark potential variable iteration counts.

Definition 8.1.1 (Loop bound reference X_δ) *Assume, the iteration bound of a loop is described by the lower and upper bounds X_1 and X_2, i.e., all iteration counts of this loop are within the interval $[X_1, \ldots, X_2]$. To simplify the comparison of iteration bounds, X_δ is denoted as a generic reference to the iteration bound $[X_1, \ldots, X_2]$.*
For example, $mAB[s] = X_\delta \cdot mAB[b]$ denotes that the execution frequency of the control flow edge $mAB[s]$ is within the following range:

$$mAB[s] = X_\delta \cdot mAB[b] \quad \Leftrightarrow \quad X_1 \cdot mAB[b] \leq mAB[s] \leq X_2 \cdot mAB[b]$$

Using X_δ within a binary relation has the following meaning:

$$mAB[s] \leq X_\delta \cdot mAB[b] \quad \Leftrightarrow \quad mAB[s] \leq X_2 \cdot mAB[b]$$
$$X_\delta \cdot mAB[b] \leq mAB[s] \quad \Leftrightarrow \quad X_1 \cdot mAB[b] \leq mAB[s]$$

The construction of the flow facts transformation rules using the semantic information from a code transformation about iteration bounds is shown by two concrete examples. The relatively simple example by the *if simplification* given in Section 8.2.1 is intended to demonstrate basic concepts. A more sophisticated example is given by *loop blocking* in Section 8.3.1. This example shows the available semantic information about iteration bounds of a code optimisation in detail. This example for *loop blocking* also discusses the importance of choosing the right mapping of flow facts attached to control flow edges to obtain a precise transformation of the flow facts. For the other code optimisations only the final transformation rules are provided. These examples give an impression about how such flow facts transformation rules can be constructed for other code optimisations.

8.2 Low-Level Optimisations

This section describes the induced *ff* transformation function $\widetilde{F}_{t2}$ for low-level code optimisations. Such optimisations are typically simple but due to our expressive flow facts it is required to perform the update of several flow facts.

8.2.1 If Simplification

If simplification is applied to a conditional statement where it is known a priori which one of the possible branches will be always taken. In this case, the test code can be removed and the other conditional branches can be removed from the *CFG*. An example for the abstract code transformation TransCFPS code transformation in case of *if simplification* is given in Figure 8.1.

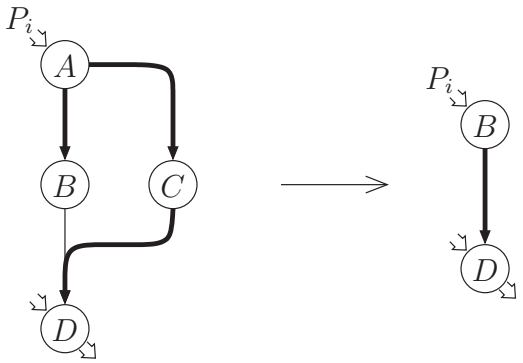


Figure 8.1: Example for If Simplification (else-branch from A never taken)

In addition to the structural *CFG* changes performed due to the code transformation the compiler has implicit knowledge about the relative iteration bounds before and after

CFG edge	*rel. bound*	*CFG edge*	*rel. bound*
$mAB[s]$	1	$mBD[s]$	1
$mAC[b]$	0	$mCD[s]$	0

Table 8.1: Implicit Control Flow Information on Original *CFG*

the transformation. The flow information known from the original code is given in Table 8.1. Using this flow information, the following safe *ff* update can be induced:

$$\begin{aligned}
mP_iA &\xrightarrow{M} mP_iB \\
mAB[s] &\xrightarrow{M} mBD[s] \\
mAC[b] &\xrightarrow{M} \emptyset \\
mCD[s] &\xrightarrow{M} \emptyset \\
\langle n \cdot mP_iA \rangle &\xrightarrow{R} \langle n \cdot mP_iB \rangle \\
\langle n \cdot mAB[s] \rangle &\xrightarrow{R} \langle n \cdot mBD[s] \rangle \\
\langle n \cdot mAC[b] \rangle &\xrightarrow{R} \emptyset \\
\langle n \cdot mCD[s] \rangle &\xrightarrow{R} \emptyset
\end{aligned}$$

These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened. The only *ff* update is to change existing control flow restrictions and update marker bindings.

8.2.2 Code Elimination

When updating flow facts for removing code it makes a difference whether this code is unreachable or not. The flow facts modification is simpler in case of deleting unreachable code because the knowledge about the absolute iteration bound (always zero) allows to safely remove the involved marker binding.

Delete Useless Basic Block

Useless code is potentially reachable code. Removing useless code therefore requires to map flow facts attached to a removed control flow edge to another control flow edge.

The resulting abstract code transformation TransCFPS for deleting a useless basic block is shown in Figure 8.2. The node A is to be deleted. As in this example the relative iteration bound for all control flow edges is the same, the induced *ff* update rules are:

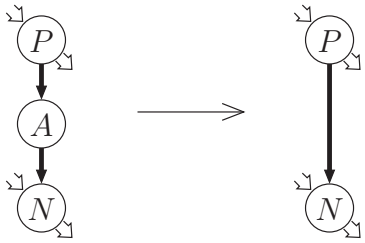


Figure 8.2: *CFG* Transformation on Deleting Useless Basic Block

$$
\begin{aligned}
mPA[s] &\xrightarrow{M} mPN[s] \\
mAN[s] &\xrightarrow{M} mPN[s] \\
\langle n \cdot mPA[s] \rangle &\xrightarrow{R} \langle n \cdot mPN[s] \rangle \\
\langle n \cdot mAN[s] \rangle &\xrightarrow{R} \langle n \cdot mPN[s] \rangle
\end{aligned}
$$

These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened. The only *ff* update is to change existing control flow restrictions and update marker bindings.

Delete Unreachable Basic Block

Due to the *CFG* structure or information obtained from dataflow analysis the compiler may be able to classify a code as unreachable. Deleting such unreachable code is a simple code transformation. Also the update of flow facts becomes trivial because the absolute iteration bound is known to be zero.

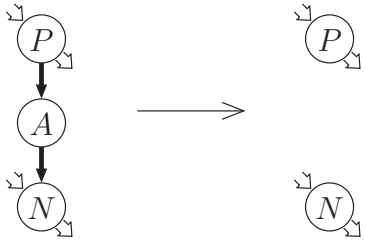


Figure 8.3: *CFG* Transformation on Deleting Unreachable Basic Block

The resulting abstract code transformation TransCFPS for deleting an unreachable basic block is shown in Figure 8.3. Safe *ff* transformation rules for deleting this unreachable basic block are induced as:

$$
\begin{aligned}
mPA[s] &\xrightarrow{M} \emptyset \\
mAN[s] &\xrightarrow{M} \emptyset \\
\langle n \cdot mPA[s] \rangle &\xrightarrow{R} \emptyset \\
\langle n \cdot mAN[s] \rangle &\xrightarrow{R} \emptyset
\end{aligned}
$$

These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened. The only *ff* update is to change existing control flow restrictions and update marker bindings.

8.2.3 Branch Optimisation

Branch optimisations are code transformations to avoid sequences of jumps under certain conditions. For example, a sequence of a conditional and an unconditional jump can be transformed to skip the unconditional jump. However, for the purpose of flow facts update it is not important which combination of jumps is optimised. The resulting abstract code transformation TransCFPS for the generic form of branch optimisation is given in Figure 8.4. This transformation scheme assumes a generic *CFG* with a conditional node B having m successor nodes.

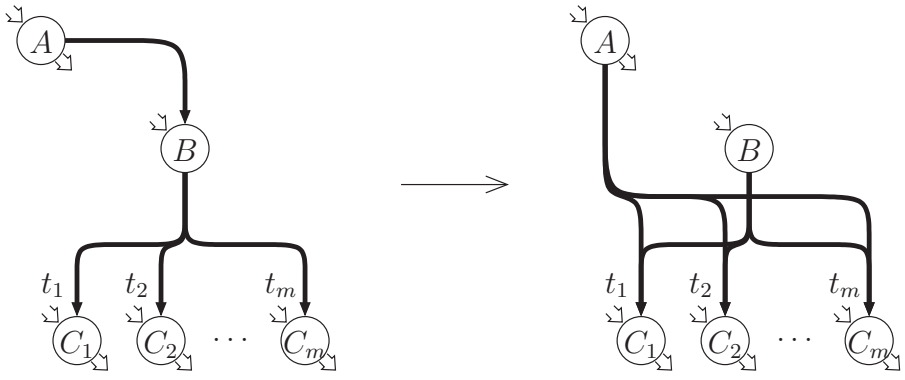


Figure 8.4: *CFG* Transformation on Branch Optimisation

The additional control flow information known by the compiler is that the individual iteration frequency of the nodes A, C_1, ..., C_m will not change. Only the iteration frequency bound of node A has to be subtracted from node B. Using this information, the following safe *ff* update rules can be induced:

$$
\begin{array}{rcl}
mBC_1[t_1] & \xrightarrow{M} & mBC_1[t_1],\ mAC_1[t_1] \\
\vdots & \vdots & \vdots \\
mBC_m[t_m] & \xrightarrow{M} & mBC_m[t_m],\ mAC_m[t_m] \\
mAB[b] & \xrightarrow{M} & mAC_1[t_1],\ mAC_2[t_2],\ \ldots,\ mAC_m[t_m] \\
\langle n \cdot mBC_1[t_1] \rangle & \xrightarrow{R} & \langle n \cdot mBC_1[t_1] - n \cdot mAC_1[t_1] \rangle \\
\langle n \cdot mBC_2[t_2] \rangle & \xrightarrow{R} & \langle n \cdot mBC_2[t_2] - n \cdot mAC_2[t_2] \rangle \\
\vdots & \vdots & \vdots \\
\langle n \cdot mBC_m[t_m] \rangle & \xrightarrow{R} & \langle n \cdot mBC_m[t_m] - n \cdot mAC_m[t_m] \rangle \\
\langle n \cdot mAB[b] \rangle & \xrightarrow{R} & \langle n \cdot mAC_1[t_1] + n \cdot mAC_2[t_2] + \ldots + n \cdot mAC_m[t_m] \rangle
\end{array}
$$

These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened. The only *ff* update is to change existing control flow restrictions and update marker bindings.

8.2.4 Conditional Moves

Conditional move statements are used to convert control flow into data flow. This improves the performance of modern processors with relatively long pipelines where a stall due to a conditional jump can reduce performance significantly. A code transformation called *if-conversion* is used to convert jumping code into a sequential code based on predicated execution. Conditional moves are therefore only a special form of predicated execution.

The resulting abstract code transformation TransCFPS for introducing a *conditional move* is shown in Figure 8.5.

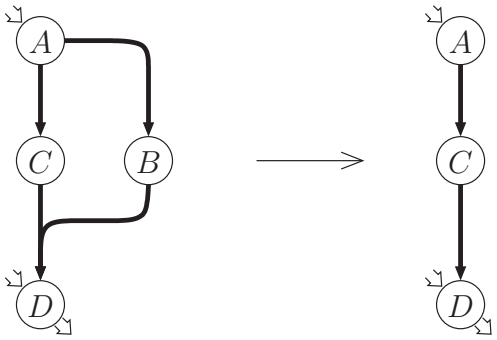


Figure 8.5: *CFG* Transformation to Insert Conditional Moves

The additional control flow information known by the compiler is that the sum of the iteration bounds of the two alternative branches in the original code exactly matches the

execution count of the single branch in the transformed code. Using this information, the following safe *ff* update can be induced:

$$\begin{aligned} mAB[b] &\xrightarrow{M} mAC[s] \\ mBD[s] &\xrightarrow{M} mCD[s] \\ \langle n \cdot mAC[s] \rangle &\xrightarrow{R} \langle [0 \dots n] \cdot mAC[s] \rangle \\ \langle n \cdot mAB[b] \rangle &\xrightarrow{R} \langle [0 \dots n] \cdot mAC[s] \rangle \\ \langle n \cdot mCD[s] \rangle &\xrightarrow{R} \langle [0 \dots n] \cdot mCD[s] \rangle \\ \langle n \cdot mBD[s] \rangle &\xrightarrow{R} \langle [0 \dots n] \cdot mCD[s] \rangle \end{aligned}$$

The only *ff* update is to change existing control flow restrictions and update marker bindings. It is important to note that these flow facts update rules are *not precise* in the sense that no relevant *ff* are lost or weakened. The reason is that the execution frequency of each of the two alternative branches cannot be matched exactly to the iteration bounds of a combination of control flow edges from the transformed code. The consequence is that the scaling factor for several restriction terms becomes a vector. It is explained in Section 7.2.2 that transitions with a vector as scaling value introduce pessimism.

8.3 Loop Optimisations

Precise and safe flow facts are very important for code fractions embedded into loops. Considering infeasible paths will be an impreciseness that is multiplied due to multiple executions of this code. Therefore it is very important to provide safe and precise *ff* transformation rules for loop optimisations.

8.3.1 Loop Blocking

Loop blocking is used as a detailed case study to explain the construction of safe *ff* update rules for loop optimisations. Loop blocking itself is a technique primarily designed to increase the locality of data accesses. Computer systems with data caches will benefit from this code transformation.

The resulting abstract code transformation TransCFPS for an example of the application of loop blocking is given by Figure 8.6. The original program consists of two nested loops. Loop blocking is applied by blocking the outer loop by a factor of k_1 and the inner loop by a factor of k_2.

To construct a precise and safe flow facts transformation function for this complex code transformation, the known information about the relative iteration bounds has to be considered. The iteration bounds relative to the loop scope L_1 for the original

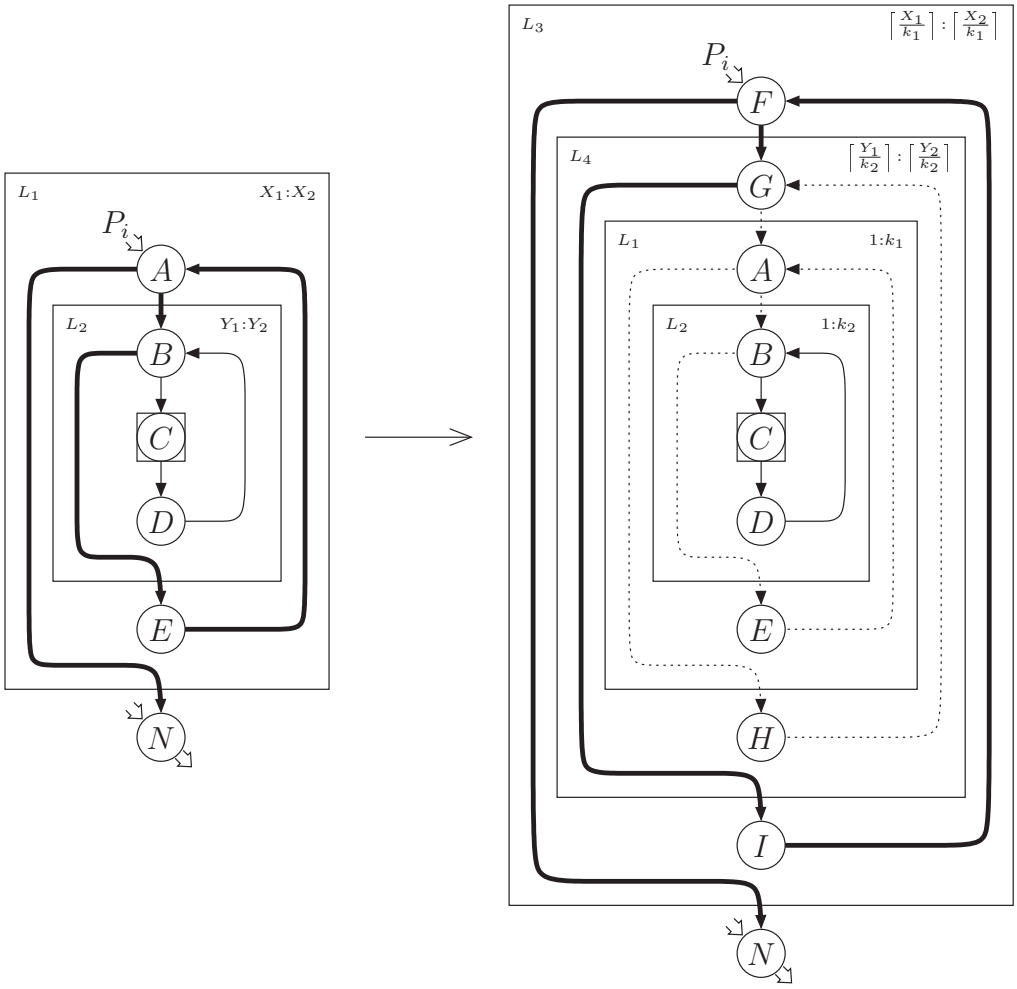


Figure 8.6: *CFG* Transformation on Loop Blocking

CFG edge	*rel. bound*	*CFG edge*	*rel. bound*
mP_iA	1	$mAN[b]$	1
$mAB[s]$	X_δ	$mBE[b]$	X_δ
$mEA[b]$	X_δ	$mBC[s]$	$X_\delta \cdot Y_\delta$
$mCD[s]$	$X_\delta \cdot Y_\delta$	$mDB[b]$	$X_\delta \cdot Y_\delta$

Table 8.2: Implicit Control Flow Information on Original *CFG*

CFG are given in Table 8.2. The iteration bounds relative to the loop scope L_3 for the transformed *CFG* are given in Table 8.3.

Based on the information about the relative iteration bounds of the *CFG* before and after the transformation, the first step is to decide how to map each marker binding of the original *CFG*. The calculation of the restriction term transitions is based on the transition of marker bindings. The calculation of safe and precise restriction term transitions is relatively straight-forward, as it is induced over the relative iteration

CFG edge	*rel. bound*	*CFG edge*	*rel. bound*
mP_iF	1	$mFN[b]$	1
$mFG[s]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil$	$mGI[b]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil$
$mIF[b]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil$	$mGA[s]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$
$mAH[b]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$	$mHG[b]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$
$mAB[s]$	$X_\delta \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$	$mBE[b]$	$X_\delta \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$
$mEA[b]$	$X_\delta \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$	$mBC[s]$	$X_\delta \cdot Y_\delta$
$mCD[s]$	$X_\delta \cdot Y_\delta$	$mDB[b]$	$X_\delta \cdot Y_\delta$

Table 8.3: Implicit Control Flow Information on Transformed *CFG*

bounds. The more critical phase for the overall precision is the design of the transitions for marker bindings.

For example, the marker binding $mAB[s]$ (relative bound X_δ) from the original *CFG* can be assigned either to $mAB[s]$ (relative bound $X_\delta \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil$) or $mFG[s]$ (relative bound $\left\lceil \frac{X_\delta}{k_1} \right\rceil$) of the transformed *CFG*. In the following, we discuss for both possibilities the impact to the precision of the *ff* update.

Attempt 1, mapping $mAB[s]$ to $mAB[s]$: The desired restriction term transformation will be of the form $\langle n \cdot mAB[s]_{src} \rangle \xrightarrow{R} \langle \vec{N} \cdot mAB[s]_{dst} \rangle$. Using the known semantic information given in Table 8.2 and Table 8.3 it follows

$$\frac{mAB[s]_{src}}{mAB[s]_{dst}} = \frac{X_\delta}{X_\delta \cdot \left\lceil \frac{Y_\delta}{k_2} \right\rceil}$$

which can be transformed to

$$mAB[s]_{src} = \frac{1}{\left\lceil \frac{Y_\delta}{k_2} \right\rceil} \cdot mAB[s]_{dst}$$

As the function $g(a) = \frac{1}{\left\lceil \frac{a}{k_2} \right\rceil}$ is an inverse monotonic function ($a_1 < a_2 \Rightarrow g(a_2) < g(a_1)$) it follows that $\vec{N} = \left[n\frac{1}{\left\lceil \frac{Y_2}{k_2} \right\rceil} \ldots n\frac{1}{\left\lceil \frac{Y_1}{k_2} \right\rceil} \right]$.

Discussion: The scaling value $(\vec{N}/n)$ in the transition of the restriction term is a vector having a length that depends on the loop bound Y_δ of the original inner loop. In case that Y_δ can be also zero it can be required to delete the involved restriction as it is not possible to scale by an unbounded value. But more critical

is the induced impreciseness by the possible variability of $\vec{N}$. For a minimum allowed loop bound $Y_1 \geq 1$ it follows that $\vec{N} \leq 1$. As the maximum loop bound Y_2 is approximately indirect proportional to the upper bound for $\vec{N}$ it follows that the mapping of the control flow edge $mAB[s]_{src}$ to $mAB[s]_{dst}$ is a still safe but not favourable choice.

Attempt 2, mapping $mAB[s]$ to $mFG[s]$: The desired restriction term transformation will be of the form $\langle n \cdot mAB[s]_{src} \rangle \xrightarrow{R} \langle \vec{N} \cdot mFG[s]_{dst} \rangle$. Using the known semantic information given in Table 8.2 and Table 8.3 it follows

$$\frac{mAB[s]_{src}}{mFG[s]_{dst}} = \frac{X_\delta}{\left\lceil \frac{X_\delta}{k_1} \right\rceil}$$

which can be transformed into

$$mAB[s]_{src} = \frac{X_\delta}{\left\lceil \frac{X_\delta}{k_1} \right\rceil} \cdot mFG[s]_{dst}$$

The function $g(X_\delta) = \frac{X_\delta}{\left\lceil \frac{X_\delta}{k_1} \right\rceil}$ is not monotonic but it can be bounded if the value range $X_1 \leq X_\delta \leq X_2$ of the argument X_δ is known. Since X_δ is an integer value it follows that $\left\lceil \frac{X_\delta}{k_1} \right\rceil \leq \frac{X_\delta + k_1 - 1}{k_1}$ and furthermore $\frac{X_\delta}{\frac{X_\delta + k_1 - 1}{k_1}} \leq \frac{X_\delta}{\left\lceil \frac{X_\delta}{k_1} \right\rceil}$ where $\frac{X_\delta \cdot k_1}{X_\delta + k_1 - 1}$ is a monotonic function under the constraint $((X_\delta + k_1) > 1)$. This constraint can be assumed to be fulfilled because for this code optimisation in the described shape it holds that $k_1 \geq 2$. Therefore, $\frac{X_1 \cdot k_1}{X_1 + k_1 - 1}$ is a safe lower bound for the function $g(X_\delta)$. To calculate an upper bound for $g(X_\delta)$ we start with $\frac{X_\delta}{k_1} \leq \left\lceil \frac{X_\delta}{k_1} \right\rceil$ and get $\frac{X_\delta}{\left\lceil \frac{X_\delta}{k_1} \right\rceil} \leq k_1$. As a result, it follows that a safe calculation of $\vec{N}$ is

$$\vec{N} = \left[n \frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1 \right]$$

Discussion: The scaling value $(\vec{N}/n)$ in the transition of the restriction term is a vector having a length that depends on the loop bound X_δ of the original outer loop. Depending on the value of X_1 the minimum value of $\vec{N}$ is determined by the relation $\left(n \leq n \frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \leq n \cdot k_1 \right)$.

The above discussions show that mapping $mAB[s]$ to $mFG[s]$ is better than mapping to $mAB[s]$ since in the first case the length of the loop bound interval X_δ has less influence than the length of the loop bound interval Y_δ in the second case. A useful mapping for all possible marker bindings of the original *CFG* is given in Table 8.4.

The construction of transitions for restriction terms using loop markers is analogous to that for markers directly attached to a control flow edge of the *CFG*. The marker should be replaced by another marker from the transformed *CFG* that has an equal

Original CFG		*Transformed CFG*	
CFG edge	*rel. bound*	*CFG edge*	*rel. bound*
mP_iA	1	mP_iF	1
$mAN[b]$	1	$mFN[b]$	1
$mAB[s]$	X_δ	$mFG[s]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil$
$mBE[b]$	X_δ	$mGI[b]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil$
$mEA[b]$	X_δ	$mIF[b]$	$\left\lceil \frac{X_\delta}{k_1} \right\rceil$

Table 8.4: Transformation of Iteration Counts by Loop Blocking

iteration bound. In this case, no scaling is required and therefore the transformation of the restriction term causes no loss of *ff* precision. For loop blocking, the restriction term $\langle n \cdot LME(L_1) \rangle$ is transformed to $\langle n \cdot LME(L_3) \rangle$ without any scaling operation. The restriction term $\langle n \cdot LMB(L_2) \rangle$ can be kept without any modification. The transformation of the restriction terms $\langle n \cdot LMB(L_1) \rangle$ and $\langle n \cdot LME(L_2) \rangle$ cannot be done without a loss in *ff* precision, because for the involved markers there is no control flow edge in the transformed *CFG* with a similar iteration bound. Therefore, $\langle n \cdot LMB(L_1) \rangle$ and $\langle n \cdot LME(L_2) \rangle$ have to be transformed in such a way that the scaling interval is minimised.

Using all this information and strategies, the following safe *ff* update can be induced:

$$
\begin{aligned}
mP_iA &\xrightarrow{M} mP_iF \\
mAB[s] &\xrightarrow{M} mFG[s] \\
mBE[b] &\xrightarrow{M} mGI[b] \\
mEA[b] &\xrightarrow{M} mIF[b] \\
mAN[b] &\xrightarrow{M} mFN[b] \\
L_1\langle X_1, X_2 \rangle &\xrightarrow{L} L_1\langle 1, k_1 \rangle,\ L_3\langle \left\lceil \frac{X_1}{k_1} \right\rceil, \left\lceil \frac{X_2}{k_1} \right\rceil \rangle \\
L_2\langle Y_1, Y_2 \rangle &\xrightarrow{L} L_2\langle 1, k_2 \rangle,\ L_4\langle \left\lceil \frac{Y_1}{k_2} \right\rceil, \left\lceil \frac{Y_2}{k_2} \right\rceil \rangle
\end{aligned}
$$

$$
\begin{aligned}
\langle n \cdot LME(L_1)\rangle &\xrightarrow{R} \langle n \cdot LME(L_3)\rangle \\
\langle n \cdot LMB(L_1)\rangle &\xrightarrow{R} \langle \left[n\frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1 \right] \cdot LMB(L_3)\rangle \\
\langle n \cdot LME(L_2)\rangle &\xrightarrow{R} \langle \left[n\frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1 \right] \cdot LME(L_4)\rangle \\
\langle n \cdot mP_iA\rangle &\xrightarrow{R} \langle n \cdot mP_iF\rangle \\
\langle n \cdot mAN[b]\rangle &\xrightarrow{R} \langle n \cdot mFN[b]\rangle \\
\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle \left[n\frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1 \right] \cdot mFG[s]\rangle \\
\langle n \cdot mBE[b]\rangle &\xrightarrow{R} \langle \left[n\frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1 \right] \cdot mGI[b]\rangle \\
\langle n \cdot mEA[b]\rangle &\xrightarrow{R} \langle \left[n\frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1 \right] \cdot mIF[b]\rangle
\end{aligned}
$$

The above rules will be used to update existing *ff* correctly when loop blocking is performed. However, for loop blocking it is also necessary to create new flow information to describe the iteration bound of nested loops more precisely. The following two restrictions have to be added to keep the absolute iteration bound for the body of the innermost loop precise:

$$
\begin{aligned}
\langle 1 \cdot LMB(L_2) &\leq X_2{\cdot}Y_2 \cdot LME(L_3)\rangle \\
\langle X_1{\cdot}Y_1 \cdot LME(L_3) &\leq 1 \cdot LMB(L_2)\rangle
\end{aligned}
$$

To conclude, loop blocking requires to change existing control flow restrictions, marker bindings and loop flow facts. It is also required to create new control flow restrictions. It is important to note that these flow facts update rules are *not precise* in the sense that no relevant *ff* are lost or weakened. The reason is that the execution frequency of the involved loops is not known as a constant value. The consequence is that the scaling factor for several restriction terms becomes a vector. It is explained in Section 7.2.2 that transitions with a vector as scaling value introduce pessimism. If the iteration count of the involved loops is known by the compiler to be constant, the *ff* update rules can be designed without any loss in precision.

8.3.2 Loop Inversion

Loop inversion is a standard code transformation to move the exit test from the top of the loop to the bottom. Using the notions from ANSI C, loop inversion converts a *while* loop into *do/while* loop.

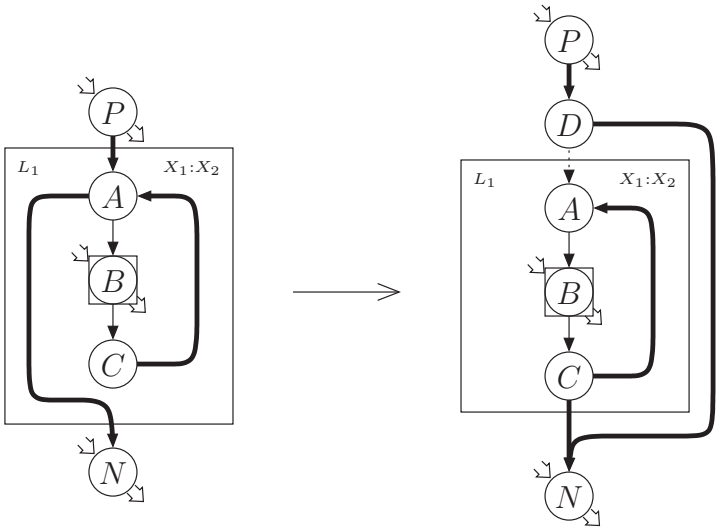


Figure 8.7: *CFG* Transformation on Loop Inversion

The resulting abstract code transformation TransCFPS for loop inversion is shown in Figure 8.7. Safe *ff* transformation rules for loop inversion are induced as:

$$
\begin{aligned}
mPA[s] &\xrightarrow{M} mPD[s] \\
mAN[b] &\xrightarrow{M} mCN[s],\ mDN[b] \\
mCA[b] &\xrightarrow{M} mCA[b],\ mCN[s] \\
\langle n \cdot mPA[s] \rangle &\xrightarrow{R} \langle n \cdot mPD[s] \rangle \\
\langle n \cdot mAN[b] \rangle &\xrightarrow{R} \langle n \cdot mCN[s] + n \cdot mDN[b] \rangle \\
\langle n \cdot mCA[b] \rangle &\xrightarrow{R} \langle n \cdot mCA[b] + n \cdot mCN[s] \rangle
\end{aligned}
$$

These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened. The only *ff* update is to change existing control flow restrictions and update marker bindings.

8.3.3 Loop Interchange

The application of this *ff* transformation framework is demonstrated by performing the code transformation of *loop interchange*. The resulting abstract code transformation TransCFPS is given in Figure 8.8.

The additional control flow information known by the compiler is that the iteration bounds of the loops are getting exchanged. Using this information, the following safe *ff* update can be induced:

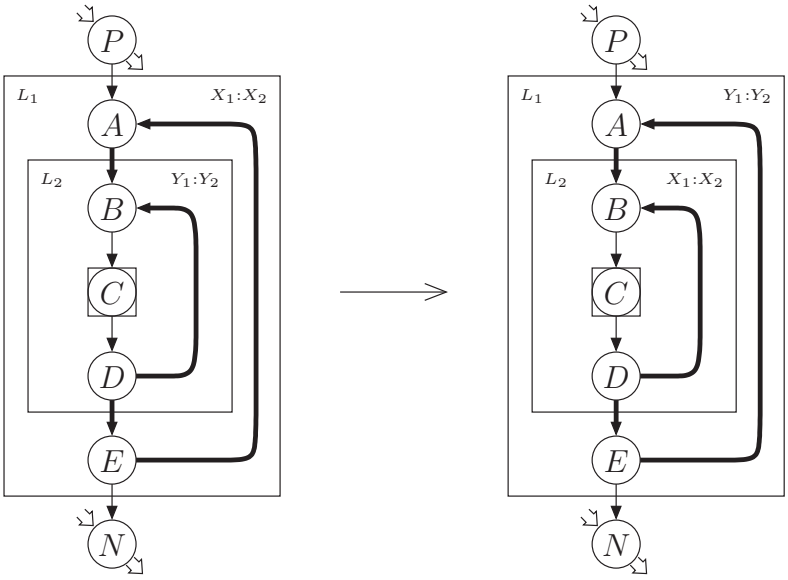


Figure 8.8: *CFG* Transformation on Loop Interchange

$$
\begin{aligned}
L_1\langle X_1, X_2\rangle &\xrightarrow{L} L_1\langle Y_1, Y_2\rangle \\
L_2\langle Y_1, Y_2\rangle &\xrightarrow{L} L_2\langle X_1, X_2\rangle \\
\langle n \cdot LMB(L_1)\rangle &\xrightarrow{R} \langle \left[n\frac{Y_1}{X_2} \ldots n\frac{Y_2}{X_1} \right] \cdot LMB(L_1)\rangle \\
\langle n \cdot LME(L_2)\rangle &\xrightarrow{R} \langle \left[n\frac{Y_1}{X_2} \ldots n\frac{Y_2}{X_1} \right] \cdot LME(L_2)\rangle \\
\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle \left[n\frac{Y_1}{X_2} \ldots n\frac{Y_2}{X_1} \right] \cdot mAB[s]\rangle \\
\langle n \cdot mDE[s]\rangle &\xrightarrow{R} \langle \left[n\frac{Y_1}{X_2} \ldots n\frac{Y_2}{X_1} \right] \cdot mDE[s]\rangle \\
\langle n \cdot mDB[b]\rangle &\xrightarrow{R} \langle \left[n\frac{Y_2(X_1-1)}{X_1(Y_2-1)} \ldots n\frac{Y_1(X_2-1)}{X_2(Y_1-1)} \right] \cdot mDB[b]\rangle \\
\langle n \cdot mEA[b]\rangle &\xrightarrow{R} \langle \left[n\frac{Y_1-1}{X_2-1} \ldots n\frac{Y_2-1}{X_1-1} \right] \cdot mEA[b]\rangle
\end{aligned}
$$

The TransRestr update for certain restriction terms is calculated with a fraction where the lower part of the fraction could probably become zero. The consequence in such a case is that we have to delete the involved restriction as it is numerically not possible to scale by an unbounded value. Since every restriction only reduces the possible *CFP*, removing a restriction is always a safe operation.

To conclude, *loop interchange* requires to change existing control flow restrictions, marker bindings and loop flow facts. It is important to note that these flow facts update

rules are *not precise* in the sense that no relevant *ff* are lost or weakened. The reason is that the execution frequency of the involved loops is not known as a constant value. The consequence is that the scaling factor for several restriction terms in TransRestr becomes a vector. It is explained in Section 7.2.2 that transitions with a vector as scaling value introduce pessimism. If the iteration count of the involved loops is known to the compiler to be constant, the TransRestr update rules can be designed without any loss of precision.

8.3.4 Loop Unrolling

This section describes the *ff* transformation for loop unrolling with an unroll factor of k. If the iteration bound N of the loop is constant and a multiple of k then the loop can be unrolled without keeping a copy of the rolled loop. Otherwise it is required to keep a copy of the rolled loop where the remaining loop iterations are executed that cannot be executed by the rolled loop. In the following, both variants are described. The first variant is described for a *do/while* loop and the second for a *while* loop.

Loop Unrolling without Copy of Rolled Loop

Performing loop unrolling without copy of rolled loop requires that the iteration bound N of the loop is known by the compiler to be constant. Furthermore, N must be a multiple of k.

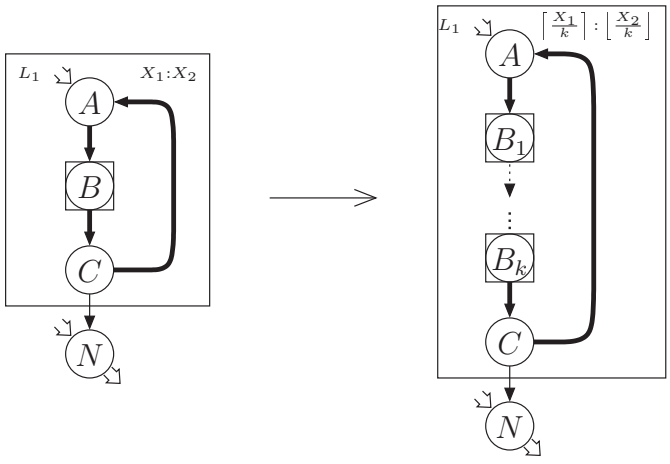


Figure 8.9: *CFG* Transformation on Loop Unrolling

The resulting abstract code transformation TransCFPS for loop unrolling is shown in Figure 8.9. The safe *ff* transformation rules for loop unrolling are induced as:

$$
\begin{aligned}
mAB[s] &\xrightarrow{M} mAB_1[s] \\
M_i(B) &\xrightarrow{M} M_i(B_1),\ M_i(B_2),\ \ldots, M_i(B_k) \\
mBC[s] &\xrightarrow{M} mB_kC[s] \\
L_1\langle X_1, X_2\rangle &\xrightarrow{L} L_1\langle \left\lceil \frac{X_1}{k} \right\rceil : \left\lfloor \frac{X_2}{k} \right\rfloor \rangle \\
\langle n \cdot LMB(L_1)\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot LMB(L_1)\rangle \\
\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot mAB_1[s]\rangle \\
\langle n \cdot M_i(B)\rangle &\xrightarrow{R} \langle n \cdot M_i(B_1) + \ldots + n \cdot M_i(B_k)\rangle \\
\langle n \cdot mBC[s]\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot mB_kC[s]\rangle \\
\langle n \cdot mCA[b]\rangle &\xrightarrow{R} \langle \left[n\frac{X_1 - 1}{\lceil \frac{X_1}{k} \rceil - 1} \ldots n\frac{X_2 - 1}{\lfloor \frac{X_2}{k} \rfloor - 1} \right] \cdot mCA[b]\rangle
\end{aligned}
$$

The restriction term transition for $\langle n \cdot mCA[b]\rangle$ uses a vector as a scaling value. This will introduce inaccuracy, depending on the length of this scaling vector. As described above, the iteration bound of the loop is known by the compiler to be constant. If the flow information models the loop bound with the same accuracy, all *ff* transformation rules will be precise. The only *ff* update is to change existing control flow restrictions, marker bindings, and loop flow facts.

It is interesting to note that a vector as scaling value will only occur when unrolling a *while* loop (having the exit test in the loop header). As shown below for *loop unrolling with keeping a copy of the rolled loop*, a vector as scaling value does not occur for a *do/while* loop structure.

Loop Unrolling with Copy of Rolled Loop

If the iteration bound of a loop is not known to be a constant, a copy of the rolled loop has to be kept when unrolling the loop.

The resulting abstract code transformation TransCFPS for *loop unrolling with keeping a copy of the rolled loop* is shown in Figure 8.10.

Using the implicitly known control flow information for this code transformation, safe *ff* transformation rules for loop inversion are induced as:

$$
\begin{aligned}
mAB[s] &\xrightarrow{M} mAB_1[s],\ mDB_{k+1}[s] \\
mAN[b] &\xrightarrow{M} mAD[b] \\
M_i(B) &\xrightarrow{M} M_i(B_1),\ M_i(B_2),\ \ldots, M_i(B_k),\ M_i(B_{k+1}) \\
mBC[s] &\xrightarrow{M} mB_kC[s],\ mB_{k+1}E[s] \\
mCA[b] &\xrightarrow{M} mCA[b],\ mED[b]
\end{aligned}
$$

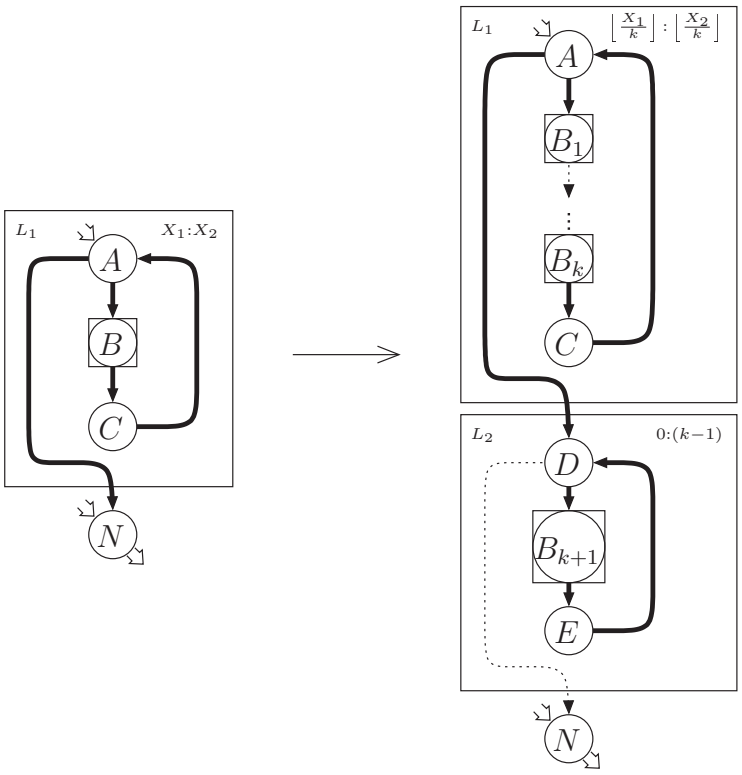


Figure 8.10: *CFG* Transformation on Loop Unrolling with Copy of Rolled Loop

$$
\begin{aligned}
L_1\langle X_1, X_2\rangle &\xrightarrow{L} L_1\langle \left\lfloor \frac{X_1}{k} \right\rfloor, \left\lfloor \frac{X_2}{k} \right\rfloor\rangle,\ L_2\langle 0{,}(k{-}1)\rangle \\
\langle n \cdot LMB(L_1)\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot LMB(L_1) + n \cdot LMB(L_2)\rangle \\
\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot mAB_1[s] + n \cdot mDB_{k+1}[s]\rangle \\
\langle n \cdot mAN[b]\rangle &\xrightarrow{R} \langle n \cdot mAD[b]\rangle \\
\langle n \cdot M_i(B)\rangle &\xrightarrow{R} \langle n \cdot M_i(B_1) + \ldots + n \cdot M_i(B_k) + n \cdot M_i(B_{k+1})\rangle \\
\langle n \cdot mBC[s]\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot mB_kC[s] + n \cdot mB_{k+1}E[s]\rangle \\
\langle n \cdot mCA[b]\rangle &\xrightarrow{R} \langle n{\cdot}k \cdot mCA[b] + n \cdot mED[b]\rangle
\end{aligned}
$$

The above rules will be used to update existing *ff* correctly when loop blocking is performed. However, for *loop unrolling with keeping a copy of the rolled loop* it is also necessary to create new flow information to describe the iteration bound of nested loops more precisely. The following two restrictions have to be added to keep the overall iteration bound for both loops precise:

$$\begin{aligned} \langle k \cdot LMB(L_1) + 1 \cdot LMB(L_2) &\leq X_2 \cdot LME(L_1) \rangle \\ \langle X_1 \cdot LME(L_1) &\leq k \cdot LMB(L_1) + 1 \cdot LMB(L_2) \rangle \end{aligned}$$

Loop unrolling with keeping a copy of the rolled loop requires to change existing control flow restrictions, marker bindings and loop flow facts. These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened.

8.3.5 Software Pipelining

Software pipelining is a loop transformation that changes the execution instance within a loop iteration. The resulting abstract code transformation TransCFPS for software pipelining by splitting the loop iteration into g stages is given by Figure 8.11.

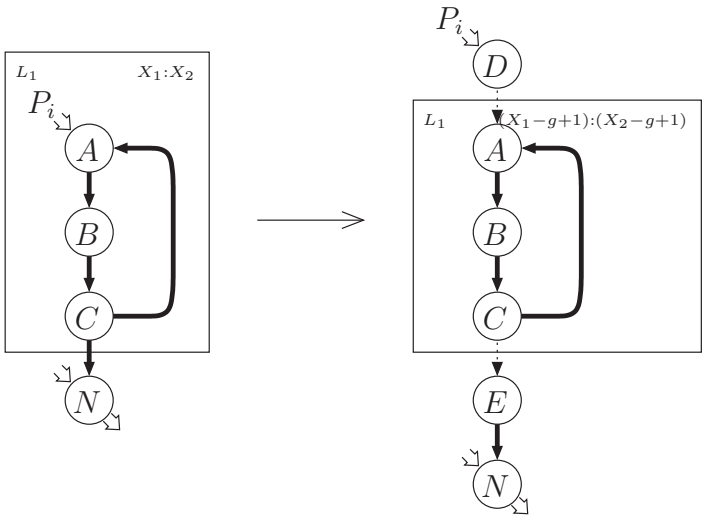


Figure 8.11: *CFG* Transformation on Software Pipelining

The additional control flow information known by the compiler is that the loop bound is reduced by the value g and that the execution count of the epilog and prolog created for the loop together with the loop itself have the original iteration bound. Using this information, the following safe *ff* update can be induced:

$$
\begin{aligned}
mP_iA &\xrightarrow{M} mP_iD \\
mAB[s] &\xrightarrow{M} mAB[s],\ mDA[s] \\
mBC[s] &\xrightarrow{M} mBC[s],\ mDA[s] \\
mCA[b] &\xrightarrow{M} mCA[b],\ mDA[s] \\
LMB(L_1) &\xrightarrow{M} LMB(L_1),\ mDA[s] \\
mCN[s] &\xrightarrow{M} mEN[s] \\
L_1\langle X_1, X_2\rangle &\xrightarrow{L} L_1\langle (X_1{-}g{+}1), (X_2{-}g{+}1)\rangle \\
\langle n \cdot LMB(L_1)\rangle &\xrightarrow{R} \langle n \cdot LMB(L_1) + (g{-}1) \cdot mDA[s]\rangle \\
\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle n \cdot mAB[s] + (g{-}1) \cdot mDA[s]\rangle \\
\langle n \cdot mBC[s]\rangle &\xrightarrow{R} \langle n \cdot mBC[s] + (g{-}1) \cdot mDA[s]\rangle \\
\langle n \cdot mCA[b]\rangle &\xrightarrow{R} \langle n \cdot mCA[b] + (g{-}1) \cdot mDA[s]\rangle \\
\langle n \cdot mCN[s]\rangle &\xrightarrow{R} \langle n \cdot mEN[s]\rangle \\
\langle n \cdot mP_iA[s]\rangle &\xrightarrow{R} \langle n \cdot mP_iD[s]\rangle
\end{aligned}
$$

Software pipelining requires to change existing control flow restrictions, marker bindings and loop flow facts. These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened.

8.3.6 Loop Unswitching

Loop unswitching is applied to a loop including a conditional statement where it is known by the compiler that the branch taken by the conditional statement will not change over loop iteration.

The resulting abstract code transformation TransCFPS for loop unswitching is given by Figure 8.11. The additional control flow information known by the compiler is that the iteration bound for the new loops is the same as the original loop and the iteration count of a block within the original loop is equal to the sum of corresponding blocks in the two loops of the transformed code. Using this information, the following safe *ff* update can be induced:

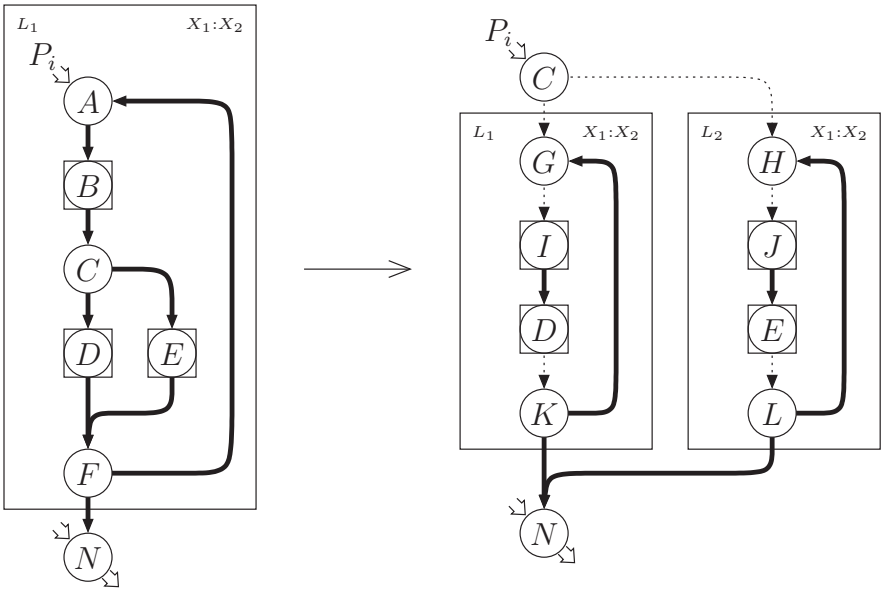


Figure 8.12: *CFG* Transformation on Loop Unswitching

$$
\begin{aligned}
mP_iA &\xrightarrow{M} mP_iC \\
M_i(B) &\xrightarrow{M} M_i(I),\ M_i(J) \\
mAB[s] &\xrightarrow{M} mID[s],\ mJE[s] \\
mBC[s] &\xrightarrow{M} mID[s],\ mJE[s] \\
mCD[s] &\xrightarrow{M} mID[s] \\
mCE[b] &\xrightarrow{M} mJE[s] \\
mDF[s] &\xrightarrow{M} mID[s] \\
mEF[s] &\xrightarrow{M} mJE[s] \\
mFA[b] &\xrightarrow{M} mKG[b],\ mLH[b] \\
mFN[s] &\xrightarrow{M} mKN[s],\ mLN[s]
\end{aligned}
$$

$$
\begin{aligned}
L_1\langle X_1, X_2\rangle &\xrightarrow{L} L_1\langle X_1, X_2\rangle,\ L_2\langle X_1, X_2\rangle \\
\langle n \cdot LME(L_1)\rangle &\xrightarrow{R} \langle n \cdot LME(L_1) + n \cdot LME(L_2)\rangle \\
\langle n \cdot LMB(L_1)\rangle &\xrightarrow{R} \langle n \cdot LMB(L_1) + n \cdot LMB(L_2)\rangle \\
\langle n \cdot mP_iA\rangle &\xrightarrow{R} \langle n \cdot mP_iC\rangle \\
\langle n \cdot M_i(B)\rangle &\xrightarrow{R} \langle n \cdot M_i(I) + n \cdot M_i(J)\rangle \\
\langle n \cdot mAB[s]\rangle &\xrightarrow{R} \langle n \cdot mID[s] + n \cdot mJE[s]\rangle \\
\langle n \cdot mBC[s]\rangle &\xrightarrow{R} \langle n \cdot mID[s] + n \cdot mJE[s]\rangle \\
\langle n \cdot mCD[s]\rangle &\xrightarrow{R} \langle n \cdot mID[s]\rangle \\
\langle n \cdot mCE[b]\rangle &\xrightarrow{R} \langle n \cdot mJE[s]\rangle \\
\langle n \cdot mDF[s]\rangle &\xrightarrow{R} \langle n \cdot mID[s]\rangle \\
\langle n \cdot mEF[s]\rangle &\xrightarrow{R} \langle n \cdot mJE[s]\rangle \\
\langle n \cdot mFA[b]\rangle &\xrightarrow{R} \langle n \cdot mKG[b] + n \cdot mLH[b]\rangle \\
\langle n \cdot mFN[s]\rangle &\xrightarrow{R} \langle n \cdot mKN[s] + n \cdot mLN[s]\rangle
\end{aligned}
$$

Analysing the above transitions for marker bindings one can see that markers are mapped to a certain control flow edge among all of them with the same iteration bound. This shows a useful optimisation technique to reduce the number of marker bindings. The result will be a speedup on *ff* transformations of further code optimisations.

Loop unswitching requires to change existing control flow restrictions, marker bindings and loop flow facts. These flow facts update rules are precise in the sense that no relevant *ff* are lost or weakened.

As explained in more detail in Section 8.3.1, the suitable design of transitions for marker bindings is important to avoid unnecessary loss in precision. For loop unswitching, if we had mapped the edge $mCD[s]$ to the structural similar edge $mCG[s]$ in the transformed code, the resulting transition rule would introduce a loss in precision:

$$\langle n \cdot mCD[s]\rangle \xrightarrow{R} \langle [n \cdot X_1 \ldots n \cdot X_2] \cdot mCG[s]\rangle$$

8.4 Chapter Summary

The transition rules of the *ff* transformation framework TransFF = TransMB × TransRestr × TransFFLF are used to describe a *ff* update function $\widetilde{F}_{t2}$ for a certain code transformation performed by the compiler.

In this chapter we presented the method to analyse and use the flow information given implicitely by the type of code optimisation to induce a correct and precise *ff* update

function by constructing a sequence of transition rules. Examples for *ff* transition rules have been given for several types of code transformations, including low-level optimisations and loop optimisations. It has been shown that the design of suitable marker binding transitions are an important step to avoid unnecessary pessimism in the induced *ff* transition rule. The number of *ff* transitions for a code transformation depends on the complexity of the involved *CFG*. A concrete *ff* transition rule only has to be considered in practice if flow facts exist, which are attached to the referenced control flow edge.

It is a good morning exercise for a research scientist
to discard a pet hypothesis every day before breakfast.
It keeps him young.

Konrad Lorenz, *Das sogenannte Böse* (1963)

Chapter 9

Assessment of the Approach

Chapter 7 presented the concept to develop safe and precise flow facts transformation rules by using the framework presented in Chapter 8. This chapter presents a theoretical as well as a practical assessment of this flow facts transformation framework.

9.1 Properties of the Flow Facts Transformation Framework

This section summarises properties of the presented *ff* transformation framework and discusses quality aspects regarding precision of *ff* transformation rules.

9.1.1 Flow Information described by Flow Facts

Flow facts are used to guide the WCET analysis tool to calculate the possible *CFP* of a program. The WCET calculation method we use is based on the implicit path enumeration technique (IPET) (described in Section 2.3). The advantage of IPET is that it allows to consider global flow facts. More information about the flow facts we use to calculate the *CFP* is given in Section 2.3.3.

To understand the limitations behind the *ff* transformation framework it is necessary to examine, what kind of information can be described by these flow facts and what is impossible to describe with them. The IPET based WCET calculation method calculates the execution frequency for each control flow edge of the program based on a set of constraints. Some of these constraints are derived from the syntactic structure of the code, the other are derived from the semantics or knowledge about the possible input parameters. The markers, restrictions, and loop bounds are used to express constraints regarding the relative execution frequency of different control flow edges. Considering all these constraints on the relative execution frequencies results in constraints for the absolute execution frequency of each control flow edge of the program.

In general, flow facts are only hints about the possible *CFP*. They cannot exactly

describe an execution trace as the concrete control flow path of an execution trace typically depends on the input data. This does not mean any impreciseness since we are interested in the longest execution trace which can be found using IPET. However, the approximation is caused by the fact that these flow facts only describe bounds for the execution frequency of each control flow edge. They cannot be used to specify the exact execution sequence order of several control flow edges. This makes no difference for simple processor architectures without performance enhancing features like pipelines or caches. But for exactly modelling pipelines or caches the execution order of instructions becomes significant.

This type of approximation becomes also important when we consider possible *ff* update rules in case of code optimisations.

9.1.2 The Meaning of Precision within this Context

As described above, the flow facts used for the WCET calculation are an approximation of the possible *CFP*. This is important to consider for defining the precision of the developed *ff* transformation framework. We can identify two different aspects of precision:

- inherent precision limitations due to the fact that flow facts are orthogonal information to the program semantics. For example, on loop interchanging the possible iteration bounds for both loops are given as interval, which makes it impossible to describe the change of the relative execution frequency for certain control flow edges.
- precision limitations due to the fact that the *ff* transformation rule does not describe the required *ff* update precisely. One possible reason for this would be that the framework is not able to perform the required *ff* update precisely. The other case would be that the concrete *ff* transformation rule for a certain code optimisation does not fully exploit the possibilities of the *ff* transformation framework.

This categorisation of precision is important to evaluate the quality of the *ff* transformation framework.

9.1.3 The Effect of Code Transformations

The operations of the induced *ff* transformation function $\widetilde{F}_{t2}$ are performed by sets of transition rules as described in Section 7.2.

In case of code transformation, the induced *ff* transition function may scale the numeric values of restriction terms or loop bounds according to the following scheme:

$$\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle n_1 \cdot mN_kN_l[t_1] \rangle,\ \langle n_2 \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$$

The above form of a restriction term transition is precise in the sense that it is scaled only by fixed numeric values. The following transition shows an imprecise update of *ff*:

$$\langle n_0 \cdot mN_iN_j[t] \rangle \xrightarrow{R} \{\langle [n_{11} \ldots n_{12}] \cdot mN_kN_l[t_1] \rangle,\ \langle [n_{21} \ldots n_{22}] \cdot mN_mN_n[t_2] \rangle,\ \ldots\}$$

As long as this transition was calculated by correctly using the implicit control flow information of the performed code transformation as described in Chapter 8, the resulting loss of precision is inherent because of the orthogonality between flow facts and program semantics.

When describing the impact of loop optimisations it also may be necessary to create new restrictions. This is the case when a transformed control flow edge is surrounded by at least two loops scaled by a vector, or when a control flow has been duplicated to at least two different loops generated from an original loop by "nonlinear" scalings.

9.1.4 Resulting Precision for Code Transformations

In Chapter 8 we presented *ff* transition rules for several code optimisations. As shown in several examples, by using the transformation framework, the development of concrete transition rules becomes quite simple. For several code optimisations (e.g., conditional moves, loop blocking, loop interchange, loop unrolling) the precision of the flow facts is weakened by using vectors as scaling factors for *ff* transitions. As discussed above, this loss of precision for the flow facts is not a limitation of the transformation framework, it is the result of inherent precision limitations due to the fact that flow facts are orthogonal information to the program semantics. For optimisations like loop blocking or loop unrolling it is also necessary to create new restrictions to reflect the original loop bounds.

Within the given examples, there were no precision limitations caused by weaknesses in the transformation framefork itself. The advantage of this *ff* transformation framework is that it consists of transition rules with small granularity which can directly be used for simple code transformations but also grouped together to form a *ff* transformation rule for more complex optimisations. As discussed in Chapter 8, to obtain precise *ff* transformations, it is important to select appropriate mappings for marker bindings.

The precision of the developed *ff* transformation framework is based on the following contributions regarding the state of the art in flow facts transformation:

- Control flow edges are first class entities (in contrast to the framework presented by Engblom in [Eng97, EEA98]). Considering control flow edges as first class entities for the *ff* transformation framework allows the design of flexible *ff* transition rules. For low-level code optimisations, these transition rules can be used locally without being bothered by the global structure of the flow information. If more complex code transformations are required, they can be constructed by composing multiple *ff* transitions (grouping them together).
- Loop iteration bounds X_δ are treated as flow information separated from conventional restrictions. They are converted on the fly into two markers and one

restriction for each loop when generating and solving the IPET constraints. On the other hand, each loop has to have two loop markers assigned from the beginning. Therefore, it is possible to mix them with conventional marker bindings in restrictions.

- The framework contains transitions to generally address and transform the flow information of a composed block without knowing the internal structure of such a block. This allows, for example, to transform all flow facts attached to the loop body in a precise way in case of loop unrolling (with keeping a copy of the rolled loop).

- Flow information attached to a certain control flow edge can be distributed to any set of control flow edges. A typical application for this is loop unrolling where the execution of the loop body is distributed to two loops. But this mechanism also allows the precise handling of more simple code optimisations such as branch optimisation.

- A graphical representation to describe the structural changes caused by a code transformation was developed. Using this graphical representation facilitates the extraction of the implicit flow information given by the type of code optimisation. It also allows us to control the existence of generic control flow edges within the subgraph involved in the code transformation.

- A potential cause for imprecision is the incomplete specification of *ff* transformation rules. However, completeness can be achieved easily by specifying a *ff* tansition for each control flow edge of the original *CFG* involved in the code transformation.

- All the induced *ff* transformations are composed of simple *ff* transition rules. This facilitates the integration of such rules into a compiler and the extension to support further code optimisations. To support a new code optimisation it is necessary to analyse its code and insert a few number of *ff* transition rules at the appropriate place. It is not required to implement a specific *ff* transformation rule for each new code optimisation.

9.2 Experiments

In this section we examine our approach by experiments. One motivation is to show that the flow facts transformation presented in this thesis allows us to calculate safe and precise bounds for the WCET. This is done by comparing the result of the analysis with real measurements on the target hardware. The other motivation is to keep in mind that the performance improvement of code by using code optimisations is important for high hardware utilisation. To take advantage of these performance improvements in the domain of hard real-time systems, it is necessary to support code optimisations performed by the compiler within the WCET analysis framework. First results for

a prototype implementation of the WCET analysis framework developed within the SETTA project[1] [SBV+02] are presented in this section.

9.2.1 The Target Hardware

The target hardware for all the experiments is the C167 16-bit microcontroller from Infineon [INF00]. The C167 processor uses a relatively simple four-stage pipeline and a jump cache mechanism that can cache one conditional jump instruction. This jump cache can speed up the execution of innermost loops. The timing analysis of the C167 includes additional complexities due to the programmable timing of external memory accesses. For our experiments we used a fixed timing setting, as given by the ACT setting in Table 9.1.

9.2.2 The Analysis Framework

The prototype implementation of the WCET analysis framework is based on a previous implementation [Kir00] that was missing a systematic support for code optimisations performed by the compiler. A structural overview of our WCET analysis framework is shown in Figure 9.1. This framework allows us to perform WCET analysis for programs written in WCETC. WCETC is a programming language derived from ANSI C allowing programmers to annotate the code by flow information [Kir02].

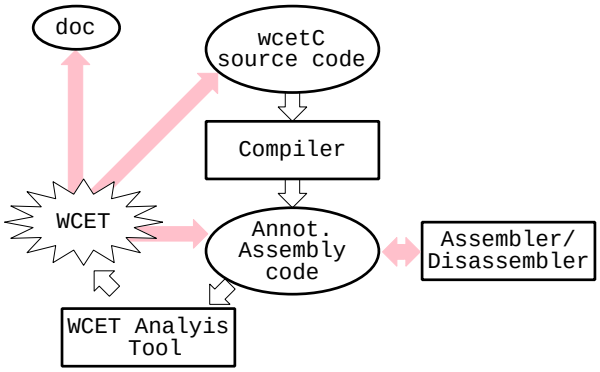


Figure 9.1: Structure of the WCET Analysis Framework

The compiler we modified to handle flow facts is based on a port of the GNU C compiler GCC 2.7.2 done by the company HighTec[2] to support the C167 processor. This relatively old compiler was selected because it is the only one available as source code that supports the C167. The concepts of the WCET calculation method implemented for our WCET analysis tool are described in [KP00]. They are based on the IPET approach developed by Puschner et al.[PS97]. The generic structure of WCET analysis

[1] IST research project "Systems Engineering for Time-Triggered Architectures (SETTA)" under contract IST-10043.

[2] http://www.hightec-rt.com/

Variable	*Value*	*Description*
EXEC_LOCATION	EXT	mem location of program code
READ_LOCATION	EXT	possible mem. locations for reading data
WRITE_LOCATION	EXT	possible mem. locations for writing data
BTYP	b_10	special function registers,
MCTC	b_1110	used to set the ACT
MTTC	b_1	(address latch enable cycle
ALECTL	b_0	time); current value is 3
MODEL_JUMP_CACHE	true	control flag, whether to model the jump cache
USE_DELTA_JUMP_CACHE	false	(not implemented yet)

Table 9.1: Configuration Settings for the Static WCET Analysis Tool

tools are given in Figure 1.1, which are described in Section 2.2.1. A central component of the analysis tool is the exec-time modelling. The development of an accurate exec-time model for the C167 processor was performed within the SETTA project. It has been shown, that due to lack of precise documentation, it requires much additional effort to develop an accurate exec-time model. The data given in the hardware manual were not sufficient [AKP01].

The hardware configuration of the WCET analysis tool is given in Table 9.1.

9.2.3 The Test Setup for Measurements

To compare the results from the static WCET analysis tools with real execution times, we performed measurements on a real hardware platform.

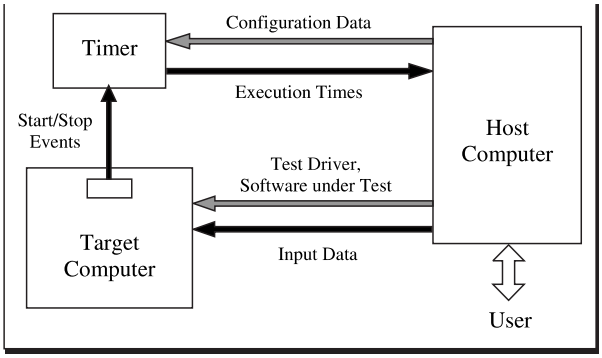


Figure 9.2: Test Environment for Runtime Measurements [AHP99]

The test setup used for the measurements was initially developed by Atanassov in order to refine the exec-time model of the C167 [AKP01]. A structural overview of the

Program	*Description*	*Properties*
Bubble Sort	Sorts an array with 25 elements in the range $[0, 124]$.	Contains a triangular loop nest with a significant number of infeasible paths.
Discrepancy	Code from the domain of railroad control. It detects discrepancies in the separate channels of a redundant system. It has 512 input parameters. 384 variables of them are in the range $[0, 255]$ and 112 variables are in the range $[0, 4095]$.	Interleaving of two nested loops with several conditional statements.
JPEG	Integer implementation of the forward DCT (discrete cosine transform).	Two sequential loops with relatively large bodies.
MatMul	Performs the multiplication of two 10×10 integer matrices with elements in the range $[0, 4095]$.	Three nested loops; without infeasible paths.
Integer	This code was generated automatically out of a MATLAB/Simulink model. It detects discrepancies in two channels of a redundant system.	Straight-line structure in the source code, but a sequence of conditional statements in the assembly code.

Table 9.2: Description of the Test Programs used for the Experiments

test setup is shown in Figure 9.2. The test method used a genetic algorithm to generate input data which promises to guide the tested task into longer execution paths than the ones in the preceding test runs. By using a timer as measurement device, this setup allows the user to measure execution times with the precision of a single clock tick. A more detailed description of the test setup can be found in [AHP99].

9.2.4 Example Programs

To perform the experiments we have chosen five sample programs, namely *Bubble Sort*, *Discrepancy*[3], *JPEG*[4], *MatMul*, and *Integer*[5]. A short description about the application domain and the code structure of these programs is given in Table 9.2.

[3]provided by the WCET research team at Daimler-Chrysler.

[4]taken from the benchmark suite of the Real-Time Research Group at the Seoul National University, available at http://www.c-lab.de/data/downloads/wcet/SNU.tgz.

[5]provided by the company DeComSys.

Optimisation Level	*Compiler Switches*
opt1	`-c -g -O0 -m7 -mcompact -mregparm -wcet`
opt2	`-c -g -O3 -m7 -mcompact -mregparm -wcet`
opt3	`-c -g -O3 -funroll-loops -m7 -mcompact -mregparm -wcet`

Table 9.3: Compiler Switches for the different Optimisation Levels

Program	*without Optimisation*	*with Optimisation*
Bubble Sort	opt1	opt2
Discrepancy	opt1	opt3
JPEG	opt1	opt2
MatMul	opt1	opt3
Integer	opt1	opt2

Table 9.4: Tested Optimisation Levels to compare Effect of Code Optimisations

The test code for the sample programs was generated by the compiler with various optimisation levels. The compiler flags to control the optimisation level of the compiler are abbreviated by the names given in Table 9.3. An exception is the compiler flag `-wcet`, which does not control the optimisation level. It is used to force the compiler into the mode to perform static WCET analysis of the code.

During the experiments we used two different optimisation levels to compare the effect of code optimisations done by the compiler for each test. The two different optimisation settings are denoted in the following subsection as "without optimisation" and "with optimisation". The assignment of this informal names to the concrete optimisation settings of the compiler is given in Table 9.4.

9.2.5 Performed Experiments

Representation of the Experiments

The results of the experiments are shown in tables with six columns. The first column, named *Method* denotes the two static analyses and the reference measurement. The first static analysis result, named *static,basic*, was done for code annotated with loop iteration bounds only. The second static analysis result, named *static,flow*, was done for code that was annotated with additional information about infeasible paths. The next four columns show the calculated/measured execution times (*cycles*) and the relative overestimation (*+%*) compared to the measurement for two different optimisation levels. The last column, named *improvement -%* shows the relative execution time reduction of the code by using code optimisations.

Method	*Code optimisations*				
	unoptimised		*optimised*		*improvement*
	cycles	+%	cycles	+%	-%
static,basic	198 741	91.08	57 816	89.65	70.91
static,flow	104 625	0.59	30 492	0.02	70.86
measurement	104 007	–	30 486	–	70.69

Table 9.5: Measured and Calculated WCET Results (in cycles) for *Bubble Sort*, With and Without Code Optimisations

Method	*Code optimisations*				
	unoptimised		*optimised*		*improvement*
	cycles	+%	cycles	+%	-%
static,basic	99 326	0.85	29 869	0.00	69.93
static,flow	99 326	0.85	29 869	0.00	69.93
measurement	98 493	–	29 869	–	69.67

Table 9.6: Measured and Calculated WCET Results (in cycles) for *Discrepancy*, With and Without Code Optimisations

Measurement Results

The program *Bubble Sort* shows how important it is to have further flow information than just loop bounds. The results for the experiments with *Bubble Sort* are summarised in Table 9.5. One result also shown by the experiments with other programs is that the relative execution time improvement when using code optimisations is similar for the static analysis methods and the reference measurement. This similarity gives the evidence that the control flow has been modelled accurately for performing WCET analysis. This result is also affirmed by the direct comparison of the calculated execution time with the measurement. While the overestimation for *static,basic* is more than 90 percent, it becomes less than one percent when additional flow facts about infeasible paths are applied.

The experiments with the programs *Discrepancy*, *JPEG*, and *MatMul* show similar results. For these programs the results for *static,basic* and *static,flow* are the same, as these programs do not show infeasible paths by program code analysis only. Again, the overall overestimation by the static WCET analysis compared to the reference measurement is always less than one percent. The difference between measurement and static analysis is a combination of both, the underestimation by the measurement and the overestimation by the static analysis. But as the difference is so small (less than one percent), it demonstrates that the WCET analysis tool uses an accurate exec-time model and a precise control flow analysis. The numeric results for these programs are given in Table 9.6, Table 9.7, and Table 9.8.

The experiments with the program *Integer* have been performed to show also results for simple straight-line code where no flow facts are required to perform static WCET

Method	*Code optimisations*				
	unoptimised		*optimised*		*improvement*
	cycles	+%	cycles	+%	-%
static,basic	29 367	0.15	10 031	0.00	65.84
static,flow	29 367	0.15	10 031	0.00	65.84
measurement	29 324	–	10 031	–	65.79

Table 9.7: Measured and Calculated WCET Results (in cycles) for *JPEG*, With and Without Code Optimisations

Method	*Code optimisations*				
	unoptimised		*optimised*		*improvement*
	cycles	+%	cycles	+%	-%
static,basic	359 925	0.07	75 915	0.04	78.91
static,flow	359 925	0.07	75 915	0.04	78.91
measurement	359 679	–	75 887	–	78.90

Table 9.8: Measured and Calculated WCET Results (in cycles) for *MatMul*, With and Without Code Optimisations

Method	*Code optimisations*				
	unoptimised		*optimised*		*improvement*
	cycles	+%	cycles	+%	-%
static,basic	648	3.02	589	0.86	9.10
static,flow	648	3.02	589	0.86	9.10
measurement	629	–	584	–	7.15

Table 9.9: Measured and Calculated WCET Results (in cycles) for *Integer*, With and Without Code Optimisations

analysis. It is also interesting to note that the compiler generated conditional data-driven control flow out of straight-line source code. Therefore, information about the possible values of input parameter could be used to generate more accurate results. Within our experiments we did not consider restrictions about the possible values of input parameters. The results for *Integer* are given in Table 9.9. Compared to the other sample programs with more complex control flow, this simple code structure yields to a higher execution time difference between measurement and static analysis. But this difference is still only 3 percent.

To summarise, the experiments showed a quite small difference between static WCET analysis and runtime measurements for different optimisation levels. From this small difference we know that the overestimation of the WCET analysis tool is the same or even smaller. Also for programs with infeasible and enabled code optimisations it was possible to calculate the WCET with an overestimation of less than one percent. The common performance improvement when using code optimisations with the concrete compiler was about 65%. Modern compilers with more powerful code optimisations may

show even more improvement. Therefore, the support of timing analysis of optimised code is an important cost factor in mass production.

9.3 Implementation Experience

The transformation of the *ff* is performed by the function TransFF. For each of its components TransFF = TransMB × TransRestr × TransFFLF a separate *ff* update function was implemented. Additionally, a function to create new restrictions was implemented. For each code optimisation, the locations of the compiler source code performing the code transformation is instrumented by these *ff* update functions.

As the compiler used for the prototype implementation does not support most of the more complex code transformations, we implemented a simplified version of the *ff* transformation framework. A typical code optimisation for which we had to use the full *ff* transformation framework was loop unrolling.

The *CFG* represents the syntactic description of the possible program control flow. The transformation function for this syntactic program description is TransCFPS. Some compilers explicitly maintain the *CFG* during code optimisations. In this case it is not necessary to implement TransCFPS as this is already integrated into the compiler.

For the chosen compiler, the *CFG* is not automatically updated during code transformations. Instead, the *CFG* is calculated dynamically for certain code optimisations. This requires to implement an additional handling for the update of the *CFG*. As an observation, the integration of the *ff* transformations into a compiler becomes simple, if the compiler itself maintains a *CFG* through all code transformations.

9.4 Chapter Summary

This chapter presented a theoretical and also a practical assessment of the flow-facts transformation framework.

The theoretical analysis discussed the precision of the flow-facts transformation rules developed in Chapter 8. The precision of the involved flow facts is weakenend if a *ff* transition uses a vector as a scaling value. This is the case with some control flow edges of certain *ff* transformation rules. However, these inherent precision limitations are due to the fact that flow facts are orthogonal information to the program semantics. The *ff* transformation framework itself has been shown to be flexible enough to avoid additional loss of precision.

The practical assessment was performed by comparing the results of a WCET analysis framework with that obtained by measurements on a real hardware platform. It was shown that the overall accuracy of the WCET analysis framework is very precise, even in case of code optimisations performed by the compiler.

Life is the art of drawing sufficient conclusions
from insufficient premises

Samuel Butler, Notebooks (1912)

Chapter 10

Conclusion

This work presented a novel approach to support worst-case execution time (WCET) analysis for optimised code. The challenges are code optimisations that dramatically change the control-flow structure of the program. As discussed within this thesis, WCET analysis techniques rely on the provision of additional control flow information (so called *flow facts*) to describe the program behaviour, and, in turn, to calculate a safe bound for the WCET. To ease the manual program annotation or semantic code analysis, it is necessary to provide these flow facts at the source code level. On the other hand, the calculation of a tight WCET bound has to consider any performed code optimisations and the interaction with the properties of the target hardware. To combine both aspects, it is required to transform the flow facts in parallel to code transformations. Existing flow facts transformation techniques were only capable to support simple flow facts (e.g., without support for the specification of infeasible paths) and the number of supported code optimisations was quite limited.

To improve this situation, we designed a flow facts transformation framework that is able to support any code transformation performed by an optimising compiler. The supported type of flow facts are expressive enough to fully support the advantages of IPET-based WCET calculation. Furthermore, the resulting flow-facts transformation rules are safe and precise.

10.1 Definition of the Role of Flow Facts for WCET Analysis

Before we started with the development of a flow-facts transformation framework we had to define the role of flow facts within the overall process of WCET analysis. First of all, we divided the WCET analysis process into independent problem categories. During this investigation we encountered problems with the existing terminology. Therefore, prior to the development of the transformation framework, a profound definition of independent problem categories was essential. We identified the following three orthogonal aspects of WCET analysis:

Flow facts handling: Due to undecidability, the calculation of the WCET in general requires the user to annotate the code with additional flow facts. Depending on capabilities in *flow facts handling* the flow facts have to be specified at different abstraction levels. Generally, the closer the properties of the flow facts are to the concrete semantics of the code, the more computation complexity is required to extract the flow facts. There are also differences in the quality of flow facts. In the simplest case, they must be sufficient to calculate iteration bounds for loops. More advanced flow facts allow the calculation of different types of (in)feasible paths.

Exec-time modelling: The WCET of a code is hardware-dependent. Thus, the WCET analysis tool has to model the behaviour of the target hardware, for example, basic instruction times, caches or pipelines. Exec-time modelling provides a description of the execution time for a given instruction sequence.

Representation level: The representation level where a program is coded and where WCET analysis is performed is often not the same. Programming languages like C or even more abstract development tools like MATLAB/Simulink are state of the art in software development. It is convenient for the developer to annotate the code with flow facts, required to perform WCET analysis, at the same level where the program is developed. On the other side, for maximum precision, the calculation of the WCET has to be performed at object code level. Thus, mechanisms are necessary to transform the flow facts to the representation level where WCET analysis is performed.

The relevant aspect for the application of our approach is the management of *representation levels* within a WCET analysis framework. Our framework allows to transform flow facts precisely from the coding level to the analysis level in parallel to code transformations.

In general, most WCET analysis frameworks with different program representation levels for the development and the analysis of a program benefit from our approach. Based on these three aspects we developed a generic and implementation independent WCET analysis framework.

10.2 Development of the Flow Facts Transformation Framework

First of all, we developed a formal transformation framework that guarantees the correctness of the transformations. The basic idea was to abstract the semantic of code transformations to their impact on the control-flow of the code. Based on that we identified the implicit control flow semantic of code optimisations known by the compiler. We developed a set of basic flow fact transitions expressive enough to deal with generic control-flow changes. We showed the completeness of this approach, i.e., that the flow

facts can be updated for every valid code transformation performed by an optimising compiler. Furthermore, it was described how the generic restriction term transition function can be used to perform accurate update of flow facts about (in)feasible paths.

The resulting *ff* transitions rules are accurate, since it is possible to distribute *ff* about infeasible paths from one control flow edge to any number of other control flow edges. The advantage of this concept is that the impact on the control flow of quite complex code optimisations can be described by simply grouping together a set of basic flow fact transitions. The semantics of the grouping operation is that the involved flow facts transitions are executed simultaneously. To summarise, the following list gives the basic functions developed for the flow facts transformation framework. These functions can be grouped together to update the flow facts in case of arbitrary complex code transformations:

Update of marker bindings ($\xrightarrow{M}$): Markers are attached to control flow edges to give them a label. With this function it is possible to distribute a marker binding from a control flow edge to any other control-flow edges.

Update of restrictions ($\xrightarrow{R}$): Restrictions are constraints to limit the possible control flow of a program. With this function it is possible to distribute each term of a restriction to any set of control-flow edges.

Update of Loop Flow Facts ($\xrightarrow{L}$): This function can create, modify, or delete iteration bounds for loops.

Creation of restrictions: For certain code transformations it is possible to increase the accuracy of the transformed flow facts by introducing new constraints.

Grouping of transition functions: The above described flow facts transition functions are applied in parallel for each code transformation. Therefore, a function is required to group these transition functions.

Based on these basic flow facts transitions we developed a method to systematically construct a safe and precise flow facts transformation rule for any code transformation performed by the compiler. We designed a hierarchical graph transformation framework that makes the development of these flow facts transformation rules more intuitive. Several examples for flow facts transformation rules demonstrated the use of this graphically supported development method.

10.3 Assessment of the Flow Facts Transformation Framework

The quality of the presented approach to transform flow facts precisely in parallel to code optimisations was assessed analytically and also practically. The analytical assessment showed that the only impreciseness introduced by the given flow facts transformations

are inherent precision limitations due to the fact that flow facts are information that is orthogonal to the program semantics. A simplified version of this framework was integrated into the GNU compiler GCC. Based on this implementation we performed experiments to compare the result of the static WCET analysis with real measurements on the target hardware. The experiments showed that the deviation of the obtained results for analysis and measurements was at most 3 percent and typically less than one percent, even when code optimisations were activated. These experiments gave an idea of the advantages of supporting a precise transformation of flow facts in case of code optimisations.

10.4 Outlook

The current implementation is a simplified version of the proposed framework. It would be of interest to demonstrate the potential of this concept by integrating it into a compiler that is capable of performing more complex code optimisations than the GNU compiler GCC 2.7.2 does.

The application context for our flow facts transformation framework described within this thesis was the support of code optimisations in a static WCET analysis framework. As with the current trend in processor development it will be increasingly difficult to analyse the behaviour of a given piece of code for a modern processor. Therefore, it would be interesting to consider the application of our flow facts transformation framework for hybrid WCET measurement methods.

For the future we plan to cooperate with compiler vendors to integrate our approach into industrial-strength compilers.

Bibliography

[AEH+99] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schrr, and G. Taentzer. Graph Transformation for Specification and Programming. *Science of Computer Programming*, 34(1):1–54, April 1999.

[AFMW96] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium (SAS), LNCS 1145*, pages 52–66. Springer, 1996.

[AHP99] Pavel Atanassov, Stefan Haberl, and Peter Puschner. Heuristic worst-case execution time analysis. In *Proc. 10th European Workshop on Dependable Computing*, pages 109–114. Austrian Computer Society (OCG), May 1999.

[AKP01] Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using real hardware to create an accurate timing model for execution-time analysis. In *International Workshop on Real-Time Embedded Systems RTES (in conjunction with 22nd IEEE RTSS 2001)*, London, UK, Dec. 2001.

[Alt96a] Peter Altenbernd. On the false problem in hard real-time programs. In *Proc. 8th Euromicro Workshop on Real Time Systems*, pages 102–107, L'Aquila, Italy, Jun. 1996.

[Alt96b] Peter Altenbernd. *Timing Analysis, Scheduling, and Allocation of Periodic Hard Real-Time Tasks*. PhD thesis, Department of Mathematics and Computer Science, University of Paderborn, Germany, Oct. 1996.

[Alt97] Peter Altenbernd. CHaRy: The C-LAB Hard Real-Time System to Support Mechatronical Design. In *Proc. IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS)*, pages 271–278, Monterey, 1997.

[Ame88] Prasanna Amerasinghe. An Extended C Compiler for Timing Analysis of Real-Time Software. Documentation, Dept. of Computer Science, University of Texas, 1988.

[AMW95] Martin Alt, Florian Martin, and Reinhard Wilhelm. Generating analyzers with PAG. Technical Report A10/95, Universität des Saarlandes, Germany, December 1995.

[AMWH94] Robert D. Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding Worst-Case Instruction Cache Performance. In *Proc. 15th Real-Time Systems Symposium (RTSS)*, pages 172–181, Brookline, Massachusetts, Dec. 1994.

[AP01] Pavel Atanassov and Peter Puschner. Impact of dram refresh on the execution time of real-time tasks. In *International Workshop on Application of Reliable Computing and Communication (WARCC)*, Dec. 2001.

[ASU97] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, June 1997. ISBN 0-201-10088-6.

[ATG96] Ali-Reza Adl-Tabatabai and Thomas Gross. Source-level debugging of scalar optimized code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 33–43, 1996.

[BBW00] Guillem Bernat, Alan Burns, and Andy Wellings. Portable Worst-Case Execution Time Analysis using Java Byte Code. In *Proc. 6th International EUROMICRO conference on Real-Time Systems*, Stockholm, June 2000.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *Proc. 23rd Real-Time Systems Symposium*, Austin, Texas, USA, Dec. 2002.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[BJ66] Corrado Boehm and Giuseppe Jacopini. Flow diagrams, turing machines and languages with only two formation rules. *CACM*, 9(5):366–371, 1966.

[Bli94] Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994.

[Bli02] Johann Blieberger. Data-flow frameworks for worst-case execution time analysis. *Real-Time Systems*, 22:183–227, 2002.

[Bör95] Hans Börjesson. Incorporating worst-case execution time in a commerical compiler. Docs msc thesis 95/69, Department of Computer Systems, Uppsala University, Sweden, Dec. 1995.

[Bru81] Arie De Bruin. Goto statements: semantics and deduction systems. *Acta Informatica*, 15:385–424, 1981.

[BS96] Ervin Rohou Francóis Bodin, , and André Seznec. Salto: System for assembly-language transformation and optimization. In *Proc. 6th Workshop on Compilers for Parallel Computers*, Dec. 1996.

[Bur72] R. E. Burkhard. *Methoden der ganzzahligen Optimierung*. Springer-Verlag, 1972.

[CBW94] Roderick Chapman, Alan Burns, and Andy Wellings. Integrated program proof and worst-case timing analysis of spark ada. In *Proc. ACM Workshop on Language, Compiler and Tool Support for Real-time Systems*, pages K1–K11, Jun. 1994.

[CBW96] Roderick Chapman, Alan Burns, and Andy Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, Aug. 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In *Proc. 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP'92), invited paper*, Leuven, Belgium, Aug. 1992.

[CC02] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Conference Record of the Twentyninth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 178–190, Portland, Oregon, Jan. 2002. ACM Press, New York.

[Che87] Moyer Chen. *A Timing Analysis Language - (TAL)*. Dept. of Computer Science, University of Texas, Austin, TX, USA, 1987. Programmer's Manual.

[Cop94] Max Copperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, May 1994.

[Cou01] Patrick Cousot. Abstract interpretation based formal methods and future challenges. *Informatics, 10 Years Back - 10 Years Ahead*, pages 138–156, 2001. Lecture Notes in Computer Science 2000.

[Cou02] Patrick Cousot. Constructive design of a hiertarchy of semantics of a transition system by abstract interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2):249–274, May 2000.

[CP01] Antoine Colin and Isabelle Puaut. A modular and retargetable framework for tree-based wcet analysis. In *Proc. 13th Euromicro Conference on Real-Time Systems*, pages 37–44, Delft, Netherland, Jun. 2001. Technical University of Delft.

[DEMS00] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, March 2000.

[DHP02] Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical Graph Transformation. *Journal of Computer and System Sciences*, 64:249–283, 2002. Short version in Proc. *FOSSACS 2000*, LNCS 1784.

[EE99] Jakob Engblom and Andreas Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*, Hong Kong, Dec. 1999.

[EE00] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proc. 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000.

[EEA98] Jakob Engblom, Andreas Ermedahl, and Peter Altenbernd. Facilitating Worst-Case Execution Time Analysis for Optimized Code. In *Proc. 10th Euromicro Real-Time Workshop*, Berlin, Germany, Jun. 1998.

[EJ02] Jakob Engblom and Bengt Jonsson. Processor pipelines and their properties for static wcet analysis. In *Proc. 2nd Embedded Software Conference*, Grenoble, France, Oct. 2002. LNCS 2491, Springer Verlag.

[Eng97] Jakob Engblom. Worst-case execution time analysis for optimized code. Master's thesis, Uppsala University, Uppsala, Sweden, September 1997.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Acta Universitatis Upsaliensis, Uppsala, Sweden, 2002.

[Exl99] Martin Exler. Propagierung von Pfadinformation für die Analyse von Programmlaufzeiten. Master's thesis, Technische Universität Wien, Vienna, Dec. 1999.

[Feh89] Elfriede Fehr. *Semantik von Programmiersprachen*. Springer, 1989. ISBN: 3-540-15163-X.

[Fen91] Norman Fenton, editor. *Software Metrics, A Rigorous Approach*. Chapman and Hall, first edition, 1991.

[FHL+01] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise wcet determination for a real-life processor. In *Proc. of the 1st International Workshop on Embedded Software (EMSOFT 2001)*, pages 469–485, Tahoe City, CA, USA, Oct. 2001.

[GE98] Jan Gustafsson and Andreas Ermedahl. Automatic derivation of path and loop annotations in object-oriented real-time programs. *Parallel and Distributed Computing Practices*, 1(2), Jun. 1998.

[Gus00] Jan Gustafsson. *Analysing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, Uppsala, Sweden, May 2000.

[Gus02] Jan Gustafsson. A prototype tool for flow analysis of object-oriented programs. In *Proc. 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Crystal City, VA, USA, Apr. 2002.

[HAM+99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David Whalley, and Marion G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[Haw01] Stephen Hawking. *The Universe in a Nutshell*. Hoffmann und Campe, Nov. 2001. ISBN: 3-553-80202-X.

[Hea99] Christopher A. Healy. *Automatic Utilization of Constraints for Timing Analysis*. PhD thesis, Florida State University, July 1999.

[Hen82] John Hennessy. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems*, 4(3):323–344, 1982.

[HL99] Jung-Chang Huang and Tau Leng. Generalized Loop Unrolling: a Method for Program Speed-Up. In *Proc. IEEE Symposium on Application-Specific System and Software Engineering Technology (ASSET99)*, pages 244–248, Mar. 1999.

[HN96] David Harel and Amnon Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4), October 1996.

[HSR+00] Christoper A. Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, pages 121–148, May 2000.

[HSW98] Christopher A. Healy, Mikael Sjödin, and David B. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proc. IEEE Real-Time Technology and Aplications Symposium*, pages 12–21, Jun. 1998.

[HW99] Christopher A. Healy and David B. Whalley. Tighter timing predictions by automatic detection and exploitation of value-dependent constraints. In *Proc. Real-Time Technology and Applications Symposium*, pages 79–88. IEEE, Jun. 1999.

[INF00] *C167CR Derivatives. 16-Bit Single-Chip Microcontroller. User's Manual.* Version 3.0. Infineon Technologies AG, Feb. 2000.

[Jar00] Clara I. Jaramillo. *Source Level Debugging Techniques and Tools for Optimized Code.* PhD thesis, University of Pittsburgh, Pittsburgh, Pennsylvania, USA, 2000.

[JGS98] Clara I. Jaramillo, Rajiv Gupta, and Mary L. Soffa. Capturing the Effects of Code Improving Transformations. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 118–123, Paris, France, Oct. 1998.

[JN94] Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. Technical report, University of Copenhagen and Computer Science Department, Aarhus University, Denmark, Jun. 1994.

[Jun90] Dieter Jungnickel, editor. *Graphen, Netzwerke und Algorithmen.* Wissenschaftsverlag, second edition, 1990.

[KFG+93] H. Kopetz, G. Fohler, G. Grünsteidl, H. Kantz, G. Pospischil, P. Puschner, J. Reisinger, R. Schlatterbeck, W. Schütz, A. Vrchoticky, and R. Zainlinger. Real-Time System Development: The Programming Model of MARS. In *Proc. of the International Symposium on Autonomous Decentralized Systems*, pages 290–299, 1993.

[KHR+96] Lo Ko, Christopher A. Healy, Emily Ratliff, Robert D. Arnold, David Whalley, and Marion Harmon. Supporting the specification and analysis of timing constraints. In *Proc. IEEE Real-Time Technology and Applicatons Symposium*, pages 170–178, Brookline, Massachusetts, Jun. 1996. IEEE press.

[Kir00] Raimund Kirner. Integration of Static Runtime Analysis and Program Compilation. Master's thesis, Technische Universität Wien, Vienna, Austria, May 2000.

[Kir02] Raimund Kirner. The Programming Language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[KLFP02] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully automatic worst-case execution time analysis for matlab/simulink models. In *Proc. 14th Euromicro Conference on Real-Time Systems*, pages 31–40, Vienna, Austria, Jun. 2002. Vienna University of Technology, IEEE.

[KP00] Raimund Kirner and Peter Puschner. Consideration of Optimizing Compilers in the Context of WCET Analysis. In *Informatiktage 2000, Fachwissenschaftlicher Informatik-Kongreß*, pages 123–126, Bad Schussenried, Germany, Oct. 2000. GI Gesellschaft für Informatik e.V.

[KP01] Raimund Kirner and Peter Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th IEEE Euromicro Conference on Real-Time Systems*, pages 29–36, Delft, The Netherlands, Jun. 2001. Technical University of Delft.

[KS86] Eugene Klingerman and Alexander D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-Time Systems. *IEEE Transactions on Software Engineering*, 12(9):941–989, Sep. 1986.

[LBJ+95] Sung-Soo Lim, Young H. Bae, Gyu T. Jang, Byung-Do Rhee, Sang L. Min, Chang Y. Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *Software Engineering*, 21(7):593–604, 1995.

[Lew85] Art Lew. *Computer Science: A Mathematical Introduction*. Prentice Hall, 1985.

[LHKM98] Sung-Soo Lim, Jun H. Han, Jihong Kim, and Sang L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proc. 19th Real-Time Systems Symposium (RTSS)*, Dec. 1998.

[LKM98] Sung-Soo Lim, Jihong Kim, and Sang L. Min. A worst case timing analysis technique for optimized programs. In *Proc. 5th International Conference on Real-Time Computing Systems and Applications (RTCSA)*, pages 151–157, Hiroshima, Japan, Oct. 1998.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proc. 32nd ACM/IEEE Design Automation Conference*, pages 456–461, Jun. 1995.

[LMW95a] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proc. IEEE Real-Time Systems Symposium*, pages 298–307, Dec. 1995.

[LMW95b] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance Estimation of Embedded Software with Instruction Cache Modeling. In *Proc. IEEE/ACM International Conference on Computer-Aided Design*, pages 380–387, Nov. 1995.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proc. 17th Real-Time Systems Symposium*, pages 254–263. IEEE, Dec. 1996.

[LS98] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 1–15, Jun. 1998.

[LS99a] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, Nov. 1999.

[LS99b] Thomas Lundqvist and Per Stenström. Timing analysis in dynamically scheduled mircoprocessors. In *Proc. 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 12–21, Dec. 1999.

[Lun02] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, Jun. 2002.

[MACT89] Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtorn Tantisirivat. Evaluating Tight Execution Time Bounds of Programs by Annotations. In *Proc. 6th IEEE Worksop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989.

[Man74] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[Miz02] Yoshihiro Mizoguchi. Shortest path length calculation using graph transformation. In *Proc. 6th Joint Conference on Imformation Science (JCIS)*, pages 358–361, North Carolina, 2002. AIM.

[MM92] Thomas J. Marlowe and Stephen P. Masticola. Safe optimization for hard real-time programming. In *Proc. 2nd International Conference on Systems Integration (ICSI)*, pages 436–445. IEEE, 1992.

[MR01] Tulika Mitra and Abhik Roychoudhury. A framework to model branch prediction for wcet analysis. Technical Report 11-01, National University of Singapore (NUS), Nov. 2001.

[MRL02] Tulika Mitra, Abhik Roychoudhury, and Xiafeng Li. Timing analysis of embedded software for speculative processors. In *Proc. 15th ACM International Symposium on System Synthesis*, pages 126–131, 2002.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.

[Mue97] Frank Mueller. Generalizing Timing Predictions to Set-Associative Caches. In *Workshop on Real-Time Systems*, pages 64–71, Jun. 1997.

[Mue00] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems Journal*, 18(2/3):209–239, May 2000.

[Nau92] Peter Naur. *Computing: A Human Activity – Selected Writings From 1951 To 1990*. ACM Press/Addison-Wesley, New York, 1992. ISBN: 0-201-58069-1.

[Nil92] Ulf Nilsson. *Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs*. PhD thesis, Department of Computer and Information Science, University of Linköping, Sweden, 1992.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN: 3-540-65410-0.

[Par93] Chang Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5(1):31–62, 1993.

[PB02] Peter Puschner and Alan Burns. Writing Temporally Predictable Code. In *Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 85–91, Jan. 2002.

[Pet00] Stefan M. Petters. Bounding the execution of real-time tasks on modern processors. In *Proc. 7th IEEE International Conference on Real-Time Computing Systems and Applications*, pages 12–14, Cheju Island, South Korea, Dec. 2000.

[Pet02] Stefan M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Institute for Real-Time Computer Systems, Technische Universit München, Germany, Sep. 2002.

[PF99] Stefan M. Petters and Georg Färber. Making worst case execution time analysis for hard real-time tasks on state of the art processors feasible. In *Proc. 6th Int. Conf. on Real-Time Computing Systems and Applications (RTCSA'99)*, Hongkong, ROC, Dec. 1999. IEEE Computer Society Press.

[PK89] Peter Puschner and Christian Koza. Calculating the Maximum Execution Time of Real-Time Programs. *The Journal of Real-Time Systems*, 1:159–176, 1989.

[PS91] Chang Y. Park and Alan C. Shaw. Experiments with a Program Timing Tool based on a Source-Level Timing Schema. *Computer*, 24(5):48–57, May 1991.

[PS97] Peter Puschner and Anton V. Schedl. Computing Maximum Task Execution Times – A Graph-Based Approach. *The Journal of Real-Time Systems*, 13:67–91, 1997.

[Pus88] Peter Puschner. Ermittlung der maximalen Abarbeitungszeit von Programmen. Master's thesis, Technische Universität Wien, Vienna, Sep. 1988.

[Pus02] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[Pus03] Peter Puschner. Algorithms for Dependable Hard Real-Time Systems. In *Proc. 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Jan. 2003.

[SA00] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.

[SBV+02] C. Scheidler, S. Boutin, U. Virnich, J. Rennhack, G. Grnsteidl, M. Pisecky, R. Lang, R. Kirner, and Y. Papadopoulos. Systems Engineering von zeitgesteuerten Systemen - die SETTA Methodik. In *VDI/VDE GMA Fachtagung, Steuerung und Regelung von Fahrzeugen und Motoren - AutoReg 2002*, pages 663–676, Mannheim, Germany, Apr. 2002.

[SF99] Jörn Schneider and Christian Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 35–44, 1999.

[Sha89] Alan C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, Jul. 1989.

[SP81] Micha Sharir and Amir Pnueli. *Program Flow Analysis: Theory and Application*, chapter 7, Two approaches to interprocedural data flow analysis, pages 189–233. Prentice Hall, 1981.

[Vrc92] Alexander Vrchoticky. Modula/R Language Definition. Technical report, Technische Universität Wien, Department of Realtime Systems, Vienna, Austria, Mar. 1992.

[Vrc94a] Alexander Vrchoticky. Compilation Support for Fine-Grained Execution Time Analysis. In *Proc. ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando FL, Jun. 1994.

[Vrc94b] Alexander Vrchoticky. *The Basis for Static Execution Time Prediction.* PhD thesis, Technische Universität Wien, Vienna, Austria, Apr. 1994.

[Wir71] Niklaus Wirth. The Programming Language Pascal. *Acta Informatica*, 1:35–63, Jun. 1971.

[Wis94a] Roland Wismüller. Debugging of globally optimized programs using data flow analysis. In *Proc. ACM/SIGPLAN Conference Programming Language Design and Implementation (PLDI'94)*, Orlando, FL, USA, Jun. 1994.

[Wis94b] Roland Wismüller. *Quellsprachorientiertes Debugging von optimierten Programmen*. PhD thesis, Fakultät für Informatik der Technischen Universität München, München, Germany, Oct. 1994.

[WMH+97] Randall T. White, Frank Mueller, Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Timing analysis for data caches and set-associative caches. In *Proc. Real-Time Technology and Applications Symposium*, pages 192–202, Jun. 1997.

[WMH+99] Randall T. White, Frank Mueller, Christopher Healy, David Whalley, and Marion G. Harmon. Timing analysis for data and wrap-around fill caches. *Real-Time Systems*, 17(2/3):209–233, Nov. 1999.

[YMTS96] Mohamed F. Younis, Thomas J. Marlowe, Grace Tsai, and Alexander D. Stoyenko. Toward compiler optimization of distributed real-time processes. In *Proc. 2nd International Conference on Engineering of Complex Computer Systems*, pages 35–42. IEEE, 1996.

Appendix A

Definition of $\overline{\text{WHILE}}$

The language $\overline{\text{WHILE}}$ is a simple programming language, designed to demonstrate concepts and algorithms. It is an extension of the language WHILE presented in [Feh89]. On the one hand, $\overline{\text{WHILE}}$ is used in Section 4.2 to demonstrate the meaning of program semantics. On the other hand, $\overline{\text{WHILE}}$ is used in Chapter 5 to describe code transformations performed by optimising compilers. To describe code optimisations compactly, $\overline{\text{WHILE}}$ has has been equipped with a special loop statement and constructs for low-level control flow.

A.1 The Syntax of $\overline{\text{WHILE}}$

The construction of syntactic correct $\overline{\text{WHILE}}$ programs is described in a BNF-like notation. The grammar is divided into the following nine categories:

1. the set Num of *numerals*, generated by the nonterminal Num
2. the set Bool of boolean values, generated by the nonterminal Bool
3. the set Var of variable names, generated by the nonterminal Var
4. the set Aop of operators (Aexp×Aexp)→Aexp, generated by the nonterminal Aop
5. the set Bop of operators (Aexp×Aexp)→Bexp, generated by the nonterminal Bop
6. the set Aexp of arithmetic expression, generated by the nonterminal Aexp
7. the set Bexp of boolean expression, generated by the nonterminal Bexp
8. the set Stmt of statements, generated by the nonterminal Stmt
9. the set Prog of programs, generated by the nonterminal Prog

A.1.1 Grammar Definition

The BNF-like grammar rules are given in the following. A speciality of $\overline{\text{WHILE}}$ is that keywords to specify block delimiters are often omitted to support writing compact code. Instead, grouping of statements is done only by indenting statements.

Num	⊨	*numeral*
Bool	⊨	**true** | **false**
Var	⊨	*identifier*
Fn	⊨	*identifier*
Aop	⊨	$+ \mid - \mid * \mid /$
Bop	⊨	$= \mid < \mid > \mid \leq \mid \geq \mid \neq$
Aexp	⊨	Num | Var | Var[LAexp] | Aexp_1 Aop Aexp_2
		| (Aexp) | Fn (LAexp) | **read**
Bexp	⊨	Bool | **not** Bexp | Aexp_1 Bop Aexp_2 | (Bexp)
Decl	⊨	Var | Var[LNum]
LNum	⊨	Num | LNum_1 , Num
LAexp	⊨	Aexp | LAexp_1 , Aexp
LDecl	⊨	Decl | LDecl_1 , Decl
Stmt	⊨	**skip**
		| **var** LDecl
		| Var **:=** Aexp
		| Stmt_1 **;** Stmt_2
		| **begin** Stmt **end**
		| **if** Bexp **then** Stmt_1 **else** Stmt_2
		| **while** Bexp **do** Stmt
		| **for** Var **:=** Aexp_1 **,** Aexp_2 **,** Aexp_3 **do** Stmt
		| **output** Aexp
		| **output** Bexp
		| *identifier* **:**
		| **goto** *identifier*
		| **return** Aexp
Function	⊨	**function** Fn (LDecl) **begin** Stmt **end**
Prog	⊨	Function Prog_1 | **begin** Stmt **end**

Preceding Rules

The given grammar rules itself will result into ambiguous derivations of certain constructs. It is therefore required to have additional precedence rules among certain grammar components:

- **if** and **while** have precedence over **;**
- precedence of operators is defined by their common mathematic priority

A.2 Comments on the Semantics

The semantics for all constructs of $\overline{\text{WHILE}}$ that are derived from the language WHILE is described (in)formally in [Feh89]. The extensions in $\overline{\text{WHILE}}$ to support functions or low-level control flow are similar to constructs in the programming language Pascal [Wir71]. The **for** loop of $\overline{\text{WHILE}}$ allows a compact description of a loop induction variable. For example, the following statement in $\overline{\text{WHILE}}$

for I:=a, b, c **do** output a[I];

has the same meaning as the source code given in Listing A.1 based on a **while** loop.

Listing A.1: transformation of a **for** loop into a **while** loop

```
1  var I;
2  I:=a;
3  while I ≤ b do
4    output a[I];
5    I:=I+c;
```

Appendix B

Foundations in Lattice Theory

B.1 Properties of Functions

Definition B.1.1 (Relation) *A* relation $\mathcal{R}$ *on the set* A *to a set* B *is a subset of the Cartesian product* $A \times B$*. The relation* $\mathcal{R}$ *has the* domain A *and the* range B*. A relation* $\mathcal{R} \subseteq A \times A$ *on the set* A *is called* reflexive*, iff* $\forall a \in A : a \mathcal{R} a$*. A relation* $\mathcal{R} \subseteq A \times A$ *is called* antisymmetric*, iff* $\forall a_1, a_2 \in A : (a_1 \mathcal{R} a_2 \wedge a_2 \mathcal{R} a_1) \Longrightarrow (a_1 = a_2)$*. A relation* $\mathcal{R} \subseteq A \times A$ *is called* transitive*, iff* $\forall a_1, a_2, a_3 \in A : (a_1 \mathcal{R} a_2 \wedge a_2 \mathcal{R} a_3) \Longrightarrow (a_1 \mathcal{R} a_3)$*. A function* $f : A \to B$ *implies a relation* $\mathcal{R}_f \subseteq A \times B$*.*

Definition B.1.2 (Reductive) *A function* $f : A \to A$ *on a domain* $\langle A, \sqsubseteq^A \rangle$ *is called* reductive *at* $a \in A$ *iff* $f(a) \sqsubseteq a$*.*

Definition B.1.3 (Extensive) *A function* $f : A \to A$ *on a domain* $\langle A, \sqsubseteq^A \rangle$ *is called* extensive *at* $a \in A$ *iff* $a \sqsubseteq f(a)$*.*

Definition B.1.4 (Monotonic function) *A function* $f : A \to B$ *from a domain* $\langle A, \sqsubseteq^A \rangle$ *to another domain* $\langle B, \sqsubseteq^B \rangle$ *is* monotonic *(also called* isotone *or* order-preserving *function) iff it preserves the partial order:* $\forall a_1, a_2 \in A : \left(a_1 \sqsubseteq^A a_2\right) \Longrightarrow \left(f(a_1) \sqsubseteq^B f(a_2)\right)$

Definition B.1.5 (Absorptive) *A function* $f : A \times A \to A$ *on a domain* $\langle A, \sqsubseteq^A \rangle$ *is called* absorptive *iff* $\forall a \in A : f(a, a) = a$*.*

Definition B.1.6 (Isotone function) *see* monotonic function*.*

Definition B.1.7 (Continuous function) *A function* $f : A \to B$ *from a domain* $\langle A, \sqsubseteq^A \rangle$ *to another domain* $\langle B, \sqsubseteq^B \rangle$ *is* continuous *iff it preserves the least upper bounds of* chains*:* $\forall A' \subseteq_{chain} A : f(\bigsqcup^A A') = \bigsqcup^B \{f(a) | a \in A'\}$
A continuous function *is also* monotonic*.*

Definition B.1.8 (Additive function) *A function $f : A \to B$ from a domain $\langle A, \sqsubseteq^A \rangle$ to another domain $\langle B, \sqsubseteq^B \rangle$ is* additive *(also called* join morphism *or* distributive *function) iff it preserves all least upper bounds:*
$\forall A' \subseteq A : \left(\exists a | a = \bigsqcup^A A'\right) \Longrightarrow \left(f(a) = \bigsqcup^B \{f(a) | a \in A'\}\right)$
An additive function *is also* continuous*.*

Definition B.1.9 (Multiplicative function) *A function $f : A \to B$ from a domain $\langle A, \sqsubseteq^A \rangle$ to another domain $\langle B, \sqsubseteq^B \rangle$ is* multiplicative *(also called* meet morphism*) iff it preserves all greatest lower bounds: $\forall A' \subseteq A : (\exists a | a = \bigsqcap^A A') \Longrightarrow (f(a) = \bigsqcap^B \{f(a) | a \in A'\})$*

Definition B.1.10 (Partial function) *A relation $f : A \to B$ from set A to set B is called a* partial function *iff for every $a \in A$ there is at most one $b \in B$ such that $\langle a, b \rangle \in f$:*
$\forall a \in A \wedge \forall b_1, b_2 \in B : (\langle a, b_1 \rangle \in f \wedge \langle a, b_2 \rangle \in f) \to (b_1 = b_2)$

Definition B.1.11 (Total function) *A relation $f : A \to B$ from set A to set B is called a* total function *iff f is a* partial function *and there is exactly* one *$b \in B$ for all $a \in A$ such that $\langle a, b \rangle \in f$:*
$\forall a \in A \wedge \exists b_1 \in B \wedge \forall b_2 \in B : \langle a, b_1 \rangle \in f \wedge (\langle a, b_2 \rangle \in f \to (b_1 = b_2))$

Definition B.1.12 (Fixpoint) *A value $x \in A$ is called a* fixpoint *for a function $f : A \to A$ iff $x = f(x)$.*

B.2 Sets and Algebraic Structures

Definition B.2.1 (poset, Partial ordered set) *For a relation $\sqsubseteq$ on a set $\mathfrak{D}$ the tuple $\langle \mathfrak{D}, \sqsubseteq \rangle$ is called a* partially ordered set *(poset), iff $\sqsubseteq$ is a* partial order *on $\mathfrak{D}$ ($\sqsubseteq$ is a partial order, iff it is* reflexive*,* antisymmetric *and* transitive*). A poset $\langle \mathfrak{D}, \sqsubseteq \rangle$ can be also expressed by $\mathsf{po}\langle \mathfrak{D}, \sqsubseteq \rangle$.*

Definition B.2.2 (Lower/Upper bound) *An element $d \in \mathfrak{D}$ of a* poset *$\langle \mathfrak{D}, \sqsubseteq \rangle$ is called a* lower *bound of $\mathfrak{D}' \subseteq \mathfrak{D}$ iff $\forall d' \in \mathfrak{D}' : d \sqsubseteq d'$. d is called an* upper *bound of $\mathfrak{D}'$ iff $\forall d' \in \mathfrak{D}' : d' \sqsubseteq d$.*

Definition B.2.3 (Greatest lower (glb)/Least upper (lub) bound) *An element $d \in \mathfrak{D}$ of a* poset *$\langle \mathfrak{D}, \sqsubseteq \rangle$ is a* greatest lower bound *(glb) of $\mathfrak{D}' \subseteq \mathfrak{D}$ (written as $d = \sqcap \mathfrak{D}'$) iff d is a lower bound of $\mathfrak{D}'$ and $(d' \sqsubseteq d)$ for all lower bounds d' of $\mathfrak{D}'$.*
An element $d \in \mathfrak{D}$ is a least upper bound *(lub) of $\mathfrak{D}' \subseteq \mathfrak{D}$ (written as $d = \sqcup \mathfrak{D}'$) iff d is an upper bound of $\mathfrak{D}'$ and $(d \sqsubseteq^{\mathfrak{D}} d')$ for all upper bounds d' of $\mathfrak{D}'$.*

Definition B.2.4 (Semi-lattice) *A* poset *$\langle \mathfrak{D}, \sqsubseteq \rangle$ is called a* join-semi-lattice *$\langle \mathfrak{D}, \sqsubseteq, \sqcup, \top \rangle$ iff for all finite subsets $\mathfrak{D}' \subseteq \mathfrak{D}$ the* lub *$\sqcup \mathfrak{D}'$ exists in $\mathfrak{D}$. $\top$ denotes $\sqcup \mathfrak{D}$. Analogous, a* poset *$\langle \mathfrak{D}, \sqsubseteq \rangle$ is called a* meet-semi-lattice *$\langle \mathfrak{D}, \sqsubseteq, \sqcap, \bot \rangle$ iff for all finite subsets $\mathfrak{D}' \subseteq \mathfrak{D}$ the* glb *$\sqcap \mathfrak{D}'$ exists in $\mathfrak{D}$. $\bot$ denotes $\sqcap \mathfrak{D}$.*

Definition B.2.5 (Total order) *A* poset $\langle \mathfrak{D}, \sqsubseteq \rangle$ *is called a* total order *iff* $\forall d_1, d_2 \in \mathfrak{D} : (d1 \sqsubseteq d2) \vee (d2 \sqsubseteq d1)$.

Definition B.2.6 (Chain) *A chain in a partially ordered set* $\langle \mathfrak{D}, \sqsubseteq \rangle$ *is a sequence* $(d_n)_n$ *of elements indexed by the natural numbers such that for all* $n \leq m$ *it follows* $d_n \sqsubseteq d_m$. *Every chain is also a* total order.

Definition B.2.7 (Ascending and descending chain condition) *A* poset *has finite height iff all chains are finite. It has finite height* h *iff all chains have at most* $h + 1$ *elements and there exists at least one chain with exactly* $h + 1$ *elements.*

A poset *satisfies the* ascending chain condition *iff all ascending chains eventually stabilise. Similarly, it fullfills the* descending chain condition *iff all descending chains eventually stabilise. A* poset *has finite height iff it satisfies both the* ascending *and* descending chain conditions.

Definition B.2.8 (cpo; Chain complete poset) *A* poset $\langle \mathfrak{D}, \sqsubseteq \rangle$ *is called a* chain complete poset *iff for all* $\mathfrak{D}' \subseteq \mathfrak{D}$ *the* glb $\sqcap \mathfrak{D}'$ *exists in* $\mathfrak{D}$ *and for all ascending chains* $\mathfrak{D}'' \subseteq \mathfrak{D}$ *the* lub $\sqcup \mathfrak{D}''$ *exists in* $\mathfrak{D}$. *Every* cpo *is also a meet-semi-lattice.*

Definition B.2.9 (Lattice) *A* poset $\langle \mathfrak{D}, \sqsubseteq \rangle$ *is called a* lattice $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ *iff for all finite subsets* $\mathfrak{D}' \subseteq \mathfrak{D}$ *the* glb $\sqcap \mathfrak{D}'$[1] *and* lub $\sqcup \mathfrak{D}'$ *exists in* $\mathfrak{D}$. $\bot$ *denotes* $\sqcap \mathfrak{D}$ *and* $\top$ *denotes* $\sqcup \mathfrak{D}$. *For all* $d_1, d_2, d_3 \in \mathfrak{D}$ *of a lattice* $\mathfrak{D}$ *it holds:* $\sqcup\{d_1, \sqcap\{d_2, d_3\}\} = \sqcap\{\sqcup\{d_1, d_2\}, \sqcup\{d_1, d_3\}\}$ *and* $\sqcap\{d_1, \sqcup\{d_2, d_3\}\} = \sqcup\{\sqcap\{d_1, d_2\}, \sqcap\{d_1, d_3\}\}$. *Every* lattice *is also a* chain complete poset.

Definition B.2.10 (Complete lattice) *A* lattice $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ *is called a* complete lattice *iff the* glb $\sqcap \mathfrak{D}'$ *and* lub $\sqcup \mathfrak{D}'$ *exists in* $\mathfrak{D}$ *also for infinite subsets* $\mathfrak{D}' \subseteq \mathfrak{D}$. *Every* finite lattice *is also a* complete lattice.

Definition B.2.11 (Moore family) *A* Moore family *is a subset* $\mathfrak{D}' \subseteq \mathfrak{D}$ *of a* complete lattice $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ *that is closed under greatest lower bounds:* $\forall \mathfrak{D}'' \subseteq \mathfrak{D}' : \sqcap \mathfrak{D}'' \in \mathfrak{D}'$.

Definition B.2.12 (Dual partial order) *For every* poset $\langle \mathfrak{D}, \sqsubseteq \rangle$ *we can obtain a dual partial ordering* $\langle \mathfrak{D}^d, \sqsubseteq^d \rangle$ *which is defined as* $d_1 \sqsubseteq^d d_2 \Leftrightarrow d_2 \sqsubseteq d_1$. *The dualism can be defined for any concept of partial order. E.g., the dual order of a lattice* $\langle \mathfrak{D}, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ *is the dual lattice* $\langle \mathfrak{D}, \sqsubseteq^d, \sqcup^d, \sqcap^d, \top^d, \bot^d \rangle$ *which is defined as*

$$\begin{array}{rrcl} \forall d_1, d_2 \in \mathfrak{D} : & d_1 \sqsubseteq^d d_2 & \Leftrightarrow & d_2 \sqsubseteq d_1 \\ \forall \mathfrak{D}' \subseteq \mathfrak{D} : & \sqcup^d \mathfrak{D}' & \Leftrightarrow & \sqcap \mathfrak{D}' \\ \forall \mathfrak{D}' \subseteq \mathfrak{D} : & \sqcap^d \mathfrak{D}' & \Leftrightarrow & \sqcup \mathfrak{D}' \\ & \bot^d & \Leftrightarrow & \top \\ & \top^d & \Leftrightarrow & \bot \end{array}$$

[1] in the literature some authors drop the requirement for greatest lower bounds

Appendix C

Mathematical Proofs

This chapter contains various mathematical proofs to clarify theoretical foundations of this thesis. Most of them are taken from other sources to assist the interested reader.

Proof 4.3.9 from page 61 (extracted from [Cou02],p.8)

The above result is known for the cartesian product $\wp(D^+) \times \wp(D^\omega)$ with componentwise ordering $\sqsubseteq^+ \times \sqsubseteq^\omega$. Since $\langle \wp(D^+) \times \wp(D^\omega), \sqsubseteq^+ \times \sqsubseteq^\omega \rangle \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \langle \wp(D^\infty), \sqsubseteq^\infty \rangle$ with $\alpha(\langle X, Y \rangle) = X \cup Y$ and $\gamma(X) = \langle X^+, X^\omega \rangle$ is a Galois isomorphism. As $\{D^+, D^\omega\}$ is assumed to be a partition of D^∞, it follows that $\{D^+, D^\omega\}$ is isomorph to $\langle \wp(D^+) \times \wp(D^\omega)$.

Proof 4.3.14 from page 67 (extracted from [NNH99],p.215)

To prove (i), Equation 4.7 follows directly by the transitivity of $\tilde{\sqsubseteq}$. Equation 4.8 immediately follows since when $\beta(d)$ is a lower bound for $\widetilde{\mathfrak{D}}'$ we get $\beta(d) \;\tilde{\sqsubseteq}\; (\tilde{\sqcap}\widetilde{D}')$. Finally, we obtain β from $\mathcal{R}_\beta$ by $\beta_{\mathcal{R}_\beta} = \sqcap\{\widetilde{d} \,|\, d \,\mathcal{R}_\beta\, \widetilde{d}\} = \sqcap\{\widetilde{d} \,|\, \beta(d) \;\sqsubseteq\; \widetilde{d}\} = \beta(d)$.

To prove (ii), from $(d \,\mathcal{R}\, \widetilde{d})$ it follows that $(\beta_{\mathcal{R}}(d) \;\tilde{\sqsubseteq}\; \widetilde{d})$ and hence $(d \,\mathcal{R}_{\beta_{\mathcal{R}}}\, \widetilde{d})$. On the other side, from $(d \,\mathcal{R}_{\beta_{\mathcal{R}}}\, \widetilde{d})$ we get $(\beta_{\mathcal{R}}(d) \;\tilde{\sqsubseteq}\; \widetilde{d})$. By writing $\widetilde{\mathfrak{D}}' = \{\widetilde{d} \,|\, d \,\mathcal{R}\, \widetilde{d}\}$ it is clear that Equation 4.8 gives $(d \,\mathcal{R}\, (\sqcap\widetilde{\mathfrak{D}}'))$ which amounts to $(d \,\mathcal{R}\, \beta_{\mathcal{R}}(d))$. And then by Equation 4.7 we get the desired $(d \,\mathcal{R}\, \widetilde{d})$.

Proof 4.3.15 from page 68 (extracted from [NNH99],p.218)

We shall prove $\widetilde{d}_1 \,\mathcal{R}_F\, \widetilde{F}(\widetilde{d}_1) \;\Leftrightarrow\; \beta_F(\widetilde{d1}) \;\tilde{\sqsubseteq}\; \widetilde{F}(\widetilde{d}_1)$. Thus, we calculate:

$$\begin{array}{lll}
\beta_F(\widetilde{d1}) \;\tilde{\sqsubseteq}\; \widetilde{F}(\widetilde{d}_1) & \Leftrightarrow & \forall d_1 : \sqcup\{\beta_2(d_2) \,|\, \beta_1(d_1) \;\tilde{\sqsubseteq}\; \widetilde{d}_1 \;\wedge\, d_2 = F(d_1)\} \;\tilde{\sqsubseteq}\; \widetilde{F}(\widetilde{d1}) \\
& \Leftrightarrow & \forall d_1, d_2 : \beta_1(d_1) \;\tilde{\sqsubseteq}\; \widetilde{d}_1 \;\wedge\; d_2 = F(d_1) \;\Rightarrow\; \beta_2(d_2) \;\tilde{\sqsubseteq}\; \widetilde{F}(\widetilde{d}_1) \\
& \Leftrightarrow & \forall d_1, d_2 : d_1 \,\mathcal{R}_1\, \widetilde{d}_1 \;\wedge\; d_2 = F(d_1) \;\Rightarrow\; d_2 \,\mathcal{R}_2\, \widetilde{F}(\widetilde{d}_1) \\
& \Leftrightarrow & \widetilde{d}_1 \,\mathcal{R}_F\, \widetilde{F}(\widetilde{d}_1)
\end{array}$$

Proof 4.3.18 from page 71 (extracted from [NNH99],p.234)

First, showing that a Galois connection is an *adjunction*: We assume $\alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}$ (from Equation 4.11) and since γ is monotone we get $\gamma(\alpha(d)) \mathrel{\tilde{\sqsubseteq}} \gamma(\widetilde{d})$; using the relation $d \sqsubseteq \gamma(\alpha(d))$ of Equation 4.10 we get $d \sqsubseteq \gamma(\alpha(d)) \sqsubseteq \gamma(\widetilde{d})$ that is $d \sqsubseteq \gamma(\widetilde{d})$. The proof in the other way (i.e. $(d \sqsubseteq \gamma(\widetilde{d})) \Rightarrow (\alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d})$) is analogous.

Second, showing that an *adjunction* is a Galois connection: From Equation 4.11 we get $(d \sqsubseteq \gamma(\widetilde{d})) \Rightarrow (\alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d})$, assuming that $d = \gamma(\widetilde{d})$ it follows $(\gamma(\widetilde{d}) \sqsubseteq \gamma(\widetilde{d})) \Rightarrow (\alpha(\gamma(\widetilde{d})) \mathrel{\tilde{\sqsubseteq}} \widetilde{d})$ which equals to $(\alpha(\gamma(\widetilde{d})) \mathrel{\tilde{\sqsubseteq}} \widetilde{d})$. Analogous, starting with $(\alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}) \Rightarrow (d \sqsubseteq \gamma(\widetilde{d}))$ and assuming that $\widetilde{d} = \alpha(d)$ it results $(d \sqsubseteq \gamma(\alpha(d)))$. It remains to show that α and γ are monotone. For α, supposing that $d_1 \sqsubseteq d_2$ and using the already proven relation $d \sqsubseteq \gamma(\alpha(d))$ we get $d_1 \sqsubseteq d_2 \sqsubseteq \gamma(\alpha(d_2))$; using $(d \sqsubseteq \gamma(\widetilde{d})) \Rightarrow (\alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d})$ it follows $\alpha(d_1) \mathrel{\tilde{\sqsubseteq}} \alpha(d_2)$. Starting with $\widetilde{d}_1 \mathrel{\tilde{\sqsubseteq}} \widetilde{d}_2$ and $\alpha(\gamma(\widetilde{d})) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}$ the monotony proof for γ is similar.

Proof 4.3.19 from page 71 (extracted from [NNH99],p.237)

To show (i) we first show that γ is determined by α. Since $(\mathfrak{D} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \widetilde{\mathfrak{D}})$ is an adjunction (Theorem 4.3.18), we get $\gamma(\widetilde{d}) = \sqcup\{d \mid d \sqsubseteq \gamma(\widetilde{d})\} = \sqcup\{d \mid \alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\}$. This shows that α uniquely determines γ: if both $\langle \mathfrak{D}, \sqsubseteq\rangle \underset{\gamma_1}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq}\rangle$ and $\langle \mathfrak{D}, \sqsubseteq\rangle \underset{\gamma_2}{\overset{\alpha}{\rightleftarrows}} \langle \widetilde{\mathfrak{D}}, \tilde{\sqsubseteq}\rangle$ are Galois connections, then $\gamma_1(\widetilde{d}) = \sqcup\{d \mid \alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\} = \gamma_2(\widetilde{d})$ for all $\widetilde{d} \in \widetilde{\mathfrak{D}}$ and hence $\gamma_1 = \gamma_2$. Similarly, we show that γ uniquely determines α by using $\alpha(d) = \sqcap\{\widetilde{d} \mid \alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\} = \sqcap\{\widetilde{d} \mid d \sqsubseteq \gamma(\widetilde{d})\}$.

To show (ii) we consider $\mathfrak{D}' \subseteq \mathfrak{D}$ and with use of Theorem 4.3.18 we calculate

$$\begin{aligned} \alpha(\sqcup\mathfrak{D}') \mathrel{\tilde{\sqsubseteq}} \widetilde{d} \quad &\Leftrightarrow \quad \sqcup\mathfrak{D}' \sqsubseteq \gamma(\widetilde{d}) \\ &\Leftrightarrow \quad \forall d \in \mathfrak{D}' : d \sqsubseteq \gamma(\widetilde{d}) \\ &\Leftrightarrow \quad \forall d \in \mathfrak{D}' : \alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d} \\ &\Leftrightarrow \quad \sqcup\{\alpha(d) \mid d \in \mathfrak{D}'\} \mathrel{\tilde{\sqsubseteq}} \widetilde{d} \end{aligned}$$

and it follows that $\alpha(\sqcup\mathfrak{D}') = \sqcup\{\alpha(d) \mid d \in \mathfrak{D}'\}$.

The proof that $\gamma(\sqcap\widetilde{\mathfrak{D}}') = \sqcap\{\gamma(\widetilde{d}) \mid \widetilde{d} \in \widetilde{\mathfrak{D}}'\}$ is analogous.

Proof 4.3.20 from page 71 (extracted from [NNH99],p.238)

Using the claim for α we define γ by $\gamma(\widetilde{d}) = \sqcup\{d' \mid \alpha(d') \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\}$.

Then we get $\alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d} \;\Rightarrow\; d \in \{d' \mid \alpha(d') \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\} \;\Rightarrow\; d \sqsubseteq \gamma(\widetilde{d})$ where the last implication follows from the definition of γ. For the other direction we first observe that $d \sqsubseteq \gamma(\widetilde{d}) \;\Rightarrow\; \alpha(d) \mathrel{\tilde{\sqsubseteq}} \alpha(\gamma(\widetilde{d}))$ because α is completely additive and hence monotone. Furthermore, we have

$$\begin{aligned} \alpha(\gamma(\widetilde{d})) \quad &= \quad \alpha(\sqcup\{d' \mid \alpha(d') \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\}) \\ &= \quad \sqcup\{\alpha(d') \mid \alpha(d') \mathrel{\tilde{\sqsubseteq}} \widetilde{d}\} \\ &\mathrel{\tilde{\sqsubseteq}} \quad \widetilde{d} \end{aligned}$$

and therefore $d \sqsubseteq \gamma(\widetilde{d}) \;\Rightarrow\; \alpha(d) \mathrel{\tilde{\sqsubseteq}} \widetilde{d}$. If follows that $(\mathfrak{D} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \widetilde{\mathfrak{D}})$ is an adjunction and hence a Galois connection (Theorem 4.3.18).
The proof of the claim for γ is analogous.

Proof 4.3.21 from page 71 (extracted from [NNH99],p.238)

It is $\lambda x.x \sqsubseteq \gamma \circ \alpha$ and since α is monotone we get $\alpha \mathrel{\tilde{\sqsubseteq}} \alpha \circ (\gamma \circ \alpha)$. Similarly, it follows from $\alpha \circ \gamma \sqsubseteq \lambda y.y$ that $(\alpha \circ \gamma) \circ \alpha \mathrel{\tilde{\sqsubseteq}} \alpha$. Thus we get $\alpha \circ \gamma \circ \alpha = \alpha$. The proof for $\gamma \circ \alpha \circ \gamma = \gamma$ is analogous.

Proof 4.3.27 from page 76 (extracted from [Gus00],p.80)

From Definition 4.3.26 we get

$$\forall \widetilde{d} \in \widetilde{\mathfrak{D}} : \; F(\gamma(\widetilde{d})) \;\sqsubseteq\; \gamma(\widetilde{F}(\widetilde{d})) \tag{C.1}$$
$$\forall \bar{d} \in \bar{D} : \; \widetilde{F}(\widetilde{\gamma}(\bar{d})) \;\mathrel{\tilde{\sqsubseteq}}\; \widetilde{\gamma}(\bar{F}(\bar{d})) \tag{C.2}$$

By substituting $\widetilde{d}$ with $\widetilde{\gamma}(\bar{d})$ in Property C.1 it becomes

$$\forall \bar{d} \in \bar{D} : \; F(\gamma(\widetilde{\gamma}(\bar{d}))) \mathrel{\tilde{\sqsubseteq}} \gamma(\widetilde{F}(\widetilde{\gamma}(\bar{d})))$$

Using the monotonicity of γ together with Property C.2 we get the result:

$$\forall \bar{d} \in \bar{D} : \; F(\gamma(\widetilde{\gamma}(\bar{d}))) \mathrel{\tilde{\sqsubseteq}} \gamma(\widetilde{\gamma}(\bar{F}(\bar{d})))$$

Proof 4.3.29 from page 77 (extracted from [Gus00],p.82)

α, γ, F are per definition monotone functions. Since $\circ$ preserves monotonicity, also $\widetilde{F}$ is monotone.

Proof 4.3.30 from page 77

Starting with the expression $\gamma(\widetilde{F}(\alpha(d)))$ and Definition 4.3.28 we get $\gamma(\widetilde{F}(\alpha(d))) = \gamma(\alpha(F(\gamma(\alpha(d)))))$. From the properties of the Galois connection and the monotonicity of $\widetilde{F}$ it follows $F(d) \sqsubseteq \gamma(\alpha(F(\gamma(\alpha(d)))))$. Therefore, $\widetilde{F}$ is safe according to Definition 4.3.26 (safe γ-approximation).

Proof 4.3.31 from page 78 (based on [JN94],p.27)

Assume having a function $\widetilde{f} : \; \widetilde{\mathfrak{D}} \to \widetilde{\mathfrak{D}}$ fullfilling $\forall d \in \mathfrak{D} : \; \alpha(F(d)) \sqsubseteq \widetilde{f}(\alpha(d))$. Then it holds for all $\widetilde{d} \in \widetilde{\mathfrak{D}}$:

$$\begin{array}{rcl}
\widetilde{F}(\widetilde{d}) & = & \text{(def. of induced function)} \\
\alpha(F(\gamma(\widetilde{d}))) & \mathrel{\tilde{\sqsubseteq}} & \text{(from safe def. of } \widetilde{f}) \\
\widetilde{f}(\alpha(\gamma(\widetilde{d}))) & \mathrel{\tilde{\sqsubseteq}} & \text{(from } \langle \alpha, \gamma \rangle \text{) -abstraction and } \widetilde{f} \text{ monotone)} \\
\widetilde{f}(\widetilde{d}) & &
\end{array}$$

Therefore, the induced function $\widetilde{F}$ gives at least the same precision than any other safe function $\widetilde{f}$.

List of Publications

[1] Jan Gustafsson, Björn Lisper, Raimund Kirner, and Peter Puschner. Input-Dependency Analysis for Hard Real-Time Software. In *Proc. 9th IEEE International Workshop on Object-Oriented Real-time Dependable Systems (WORDS 2003F)*, Capri Island, Italy, October 2003.

[2] Raimund Kirner and Peter Puschner. Discussion of Misconceptions about WCET Analysis. In *Proc. 3rd Euromicro International Workshop on WCET Analysis*, Porto, Portugal, July 2003.

[3] Raimund Kirner and Peter Puschner. A Simple and Effective Fully Automatic Worst-Case Execution Time Analysis for Model-Based Application Development. In *Proc. 1st Workshop on Intelligent Solutions in Embedded Systems*, Vienna, Austria, June 2003.

[4] Peter Puschner and Raimund Kirner. Avoiding Timing Problems in Real-Time Software. In *Proc. IEEE Computer Society's Workshop on Software Technologies for Future Embedded Systems*, May 2003.

[5] Raimund Kirner and Peter Puschner. Timing Analysis of Optimised Code. In *Proc. 8th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Guadalajara, Mexico, January 2003.

[6] Raimund Kirner. Enforcing Composability for Ubiquitous Computing Systems. In *Proc. 7th Cabernet Radicals Workshop*, Bertinoro, Italy, October 2002.

[7] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter Puschner. Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In *Proc. 14th Euromicro Conference on Real-Time Systems*, pages 31–40, Vienna, Austria, Jun. 2002. Vienna University of Technology, IEEE.

[8] Wilfried Elmenreich, Lukas Schneider, and Raimund Kirner. A Robust Certainty Grid Algorithm for Robotic Vision. In *Proc. 6th International Conference on Intelligent Engineering Systems (INES'02)*, Opatija, Croatia, May 2002. IEEE.

[9] C. Scheidler, S. Boutin, U. Virnich, J. Rennhack, G. Grnsteidl, M. Pisecky, R. Lang, R. Kirner, and Y. Papadopoulos. Systems Engineering von zeitgesteuerten Systemen - die SETTA Methodik. In *VDI/VDE GMA Fachtagung, Steuerung und*

Regelung von Fahrzeugen und Motoren - AutoReg 2002, pages 663–676, Mannheim, Germany, Apr. 2002.

[10] Raimund Kirner, Roland Lang, and Peter Puschner. WCET Analysis for Systems Modelled in Matlab/Simulink. In *Proc. 22nd IEEE Real-Time Systems Symposium, Work in Progress Session*, pages 33–36, London, UK, Dec. 2001. University of York, Department of Computer Science, Report YCS 337 (2001).

[11] Pavel Atanassov, Raimund Kirner, and Peter Puschner. Using Real Hardware to Create an Accurate Timing Model for Execution-Time Analysis. In *International Workshop on Real-Time Embedded Systems RTES (in conjunction with 22nd IEEE RTSS 2001)*, London, UK, Dec. 2001.

[12] Raimund Kirner and Peter Puschner. Transformation of Path Information for WCET Analysis during Compilation. In *Proc. 13th IEEE Euromicro Conference on Real-Time Systems*, pages 29–36, Delft, The Netherlands, Jun. 2001. Technical University of Delft.

[13] Raimund Kirner, Roland Lang, Peter Puschner, and Christopher Temple. Integrating WCET Analysis into a Matlab/Simulink Simulation Model. In *Proc. 16th IFAC Workshop on Distributed Computer Control Systems*, Sydney, Australia, Nov. 2000. School of Computer Science and Engineering, UNSW.

[14] Raimund Kirner and Peter Puschner. Consideration of Optimizing Compilers in the Context of WCET Analysis. In *Informatiktage 2000, Fachwissenschaftlicher Informatik-Kongreß*, pages 123–126, Bad Schussenried, Germany, Oct. 2000. GI Gesellschaft für Informatik e.V.

[15] Raimund Kirner and Peter Puschner. Supporting Control-Flow-Dependent Execution Times on WCET Calculation. In *Deutschsprachige WCET-Tagung*, Paderborn, Germany, Oct. 2000. C-Lab.

Curriculum Vitae

Raimund Kirner

September 29^{th} 1972	Born in Neunkirchen, Lower Austria (Austria)
September 1979 – June 1983	Elementary School in Edlitz
September 1983 – June 1987	Secondary School in Edlitz
September 1987 – June 1992	Engineering School for communications engineering in Mödling
July 1992 – February 1993	Military Service in Wiener Neustadt
April 1993 – September 1996	HF measurement engineer at the Austrian Research Center Seibersdorf
October 1996 – May 2000	Studies of Computer Science at the Vienna University of Technology
March 1997 – December 1998	HF measurement engineer at the Austrian Research Center Seibersdorf
May 2000	Master's Degree in Computer Science (with distinction)
October 1998 – June 2000	Measurement and software technician at the C&P DI Suschnig GmbH, Klagenfurth
May 2000 – June 2000	Research assistant at the Vienna University of Technology
since July 2000	Research/Teaching assistant at the Vienna University of Technology
since October 2000	PhD Studies of Computer Science at the Vienna University of Technology

Errata

The following list shows errata that has been corrected since the release of the original thesis in Mai 2003:

Page	**Line**	**Reads**	**Correction**
59	21	semantics $\mathcal{S}_S^\tau = \tau^{\vec{\infty}}$:	semantics $\mathcal{S}_P^\tau = \tau^{\vec{\infty}}$:
92	2	control of adjacent	control of adjacent loops
106	10	$\widetilde{F}_2 = impl_{\widetilde{F}_2}(\widetilde{F}_1)$	$\widetilde{F}_2 = impl(\widetilde{F}_2/\widetilde{F}_1)$
107	28	in Section 6.3.1	in Section 6.2.2
116	17	done by $\widetilde{F}_t 1$	done by $\widetilde{F}_{t1}$
116	35	for arbitrary nodes	for arbitrary edges
117	3	$\{L_y, i \vert 1 \leq i \leq n-1\}$	$\{L_{y,i} \vert 1 \leq i \leq n-1\}$
122	30	e a unique	t a unique
123	7	$(\langle n, n', t\rangle \in E_P) \wedge (n \notin N_M)$	$\langle n, n', t\rangle \in E_P$
123	9	$(\langle n', n, t\rangle \in E_P) \wedge (n \notin N_M)$	$\langle n', n, t\rangle \in E_N$
125	26	$\langle \left[\frac{X_1}{Y_2} \ldots \frac{X_2}{Y_1}\right] \cdot mAB[s]\rangle$	$\langle \left[n \cdot \frac{X_1}{Y_2} \ldots n \cdot \frac{X_2}{Y_1}\right] \cdot mAB[s]\rangle$
126	4	$\langle k \cdot mAB[s]\rangle$	$\langle n \cdot k \cdot mAB[s]\rangle$
126	30	given in Figure 7.5	given in Figure 7.6
140	22	Y_δ of the original inner	X_δ of the original outer
142	3	$\langle \left[n \frac{1}{\lceil \frac{Y_2}{k_2} \rceil} \ldots n \frac{1}{\lceil \frac{Y_1}{k_2} \rceil}\right] \cdot LME(L_2)\rangle$	$\langle \left[n \frac{X_1 \cdot k_1}{X_1 + k_1 - 1} \ldots n \cdot k_1\right] \cdot LME(L_4)\rangle$
142	14,15	$\cdot LME(L_4)$	$\cdot LME(L_3)$
147	Fig. 8.10	(edge $\langle A, B, seq\rangle$ and $\langle B, C, seq\rangle$ became bold)	
148	Fig. 8.11	$(X_1 - g)$:$(X_2 - g)$	$(X_1 - g + 1)$:$(X_2 - g + 1)$
149	7	$(X_1 - g), (X_2 - g)$	$(X_1 - g + 1), (X_2 - g + 1)$
149	8-11	$+ g \cdot mDA[s]$	$+ (g-1) \cdot mDA[s]$