TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

DISSERTATION

High-level compiler support for timing analysis

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

Univ. Prof. Dipl.-Inf. Dr. rer. nat. Jens Knoop
E185/1
Institut für Computersprachen

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Adrian Prantl
Matr.-Nr.: 0025274
adrian@complang.tuwien.ac.at
Neustiftgasse 45/12
1070 Wien

Wien, am 16. Mai 2010

Kurzfassung

Um robuste eingebettete Systeme zu entwerfen, ist die Analyse der Ausführungszeit von großer Bedeutung. Ziel der vorliegenden Dissertation ist es einen Weg aufzuzeigen, wie das Zeitverhalten von Computerprogrammen mit geringstmöglicher Unterstützung von menschlicher Seite untersucht werden kann. Um möglichst straffe, aber sichere Schranken für die Ausführungszeit eines Programms zu erhalten, sind exakte Informationen über dessen Kontroll- und Datenfluss notwendig. Bisweilen wird diese Information von Hand gesammelt und an das übersetzte Binärprogramm annotiert. Im letzten Jahrzehnt konnte ein deutlicher Trend festgestellt werden, diesen Vorgang durch den Einsatz statischer Programmanalysen zu automatisieren. Da das Problem im Allgemeinen jedoch unlösbar ist, werden manuelle Annotationen wohl immer vonnöten sein. Akzeptiert man diese Tatsache, so ist es unumgänglich, diesen mühsamen und fehlerbehafteten Vorgang so einfach und sicher wie möglich zu machen.

Wie die Sicherheit ist auch Leistung ein zentraler Punkt, wenn eingebettete Systeme in großen Stückzahlen produziert werden sollen. Während sich die durchschnittliche Leistung vor allem auf den Stromverbrauch auswirkt, so ist die Leistung im schlechtestmöglichen Fall (Worst-case-Performance) der bestimmende Faktor eines Echtzeitsystems: Eine höhere Worst-case-Performance bewirkt, dass das System kosteneffizienter dimensioniert werden kann, wodurch die Produktionskosten gesenkt werden, ohne die Sicherheit zu gefährden. Daher ist in diesem Bereich auch die Verbindung von Zeitanalyse und Programmoptimierungen eine der Kernaufgaben der Forschung. Ziel soll jedoch nicht sein, den hochoptimierten Binärcode, wie er von einem Compiler erzeugt wird, von Programmieren annotieren zu lassen. Ziel ist vielmehr, Kontrollflussannotationen bereits auf Quelltextebene durchzuführen und diese Informationen dann gemeinsam mit dem Programm durch die Optimierungsschritte zu führen und entsprechend zu transformieren.

In dieser Arbeit zeigen wir (1) einen Weg, das Annotationsniveau von der Maschinensprache auf die Quelltextebene anzuheben, um Annotationen auf der Abstraktionsebene der Programmiersprache zu präsentieren. Weiters (2) stellen wir auf dem Quelltext arbeitende statische Programmanalysen vor, mit denen die Notwendigkeit, Annotationen manuell vornehemen zu müssen, auf ein Mindestmaß reduziert werden kann. Damit (3) präsentieren wir eine portable Lösung für die oben genannten Probleme, die mit minimalem Aufwand auf andere Zielarchitekturen übertragen werden kann.

Die Teilnahme an der WCET Tool Challenge 2008 hat bestätigt, dass unser Ansatz mit anderen Implementierungen konkurrenzfähig ist, insbesondere in Bezug auf die automatische Kontrollflussanalyse. Damit, dass unsere Implementierung eine Teilmenge von C++ unterstützt, greifen wir auch einen aufkommenden Trend für eingebettete Systeme auf, in denen C nach wie vor die vorherrschende Programmiersprache ist, jedoch nach und nach durch C++ abgelöst wird.

Abstract

Timing analysis is an important prerequisite for the design of robust embedded systems. The purpose of this thesis is to show how to analyze the timing of computer programs with as little human assistance as possible. In order to get tight and safe bounds for the timing of a program, precise information about its control flow and data flow is needed. Traditionally, this information is collected by hand and annotated to the binary program. The last decade showed a trend to automate much of this work by employing static analyses. However, due to the theoretical intractability of the general problem, manual input is still and will aways be necessary. Once we acknowledge the need for manual annotations, it is important to make this cumbersome and error-prone process as easy and safe as possible.

Performance is also a critical issue for systems that are produced in large quantities. While the average performance has an influence on power consumption, the worst-case performance is what is critical for a real-time system: A better worst-case performance means that the hardware can be dimensioned more cost-effective, thus lowering production costs, without sacrificing safety. The main research question is therefore how to adequately combine code optimizations and timing analysis. We do not intend to force the programmer to manually annotate control flow information to highly optimized code produced by the compiler. Instead, flow annotations should be made at the source code level and be transformed alongside the program during the optimization phase.

In this thesis we (1) show how to lift the annotation level from the machine code to the source code, which is the adequate and more natural representation for the programmer, and (2) reduce the need for manual annotations by using static analysis at the source code level, thus (3) providing a portable solution to the problems mentioned above that is largely independent from the target architecture.

By entering the implementation into the WCET Tool Challenge 2008 we demonstrated that this approach is on par with competing approaches, especially with respect to automatic control flow and data flow analysis. With our implementation we also embrace a continuing trend in the embedded systems market, by supporting a subset of C++ instead of just C, which is still the prevailing programming language in this field.

Acknowledgements

I am very grateful to my advisor Jens Knoop, who has been highly encouraging and supportive throughout the last four years. It was his enthusiasm that sparked my interest in program analysis in the first place. I would like to thank Björn Lisper for reviewing this thesis and for insightful discussions on previous WCET Workshops. I would like to thank Markus Schordan for many hours we spent discussing and hacking SATIrE and also for encouraging me to work on what later would become Termite. I am also thankful to Raimund Kirner, who was the principal investigator of the CoSTA project; many ideas presented in this thesis were formed during discussions with him. I am also grateful to Albrecht Kadlec, who was very supportive and shared many insights from his industry experience. I am very grateful to Gergő Bárány who implemented the connection of PAG and ROSE that later became the basis of SATIrE, the points-to analysis and the `melmac` compiler. He also helped me many times with the debugging of analyzers and his intimate knowledge of ROSE internals. Markus Triska implemented the `clpfd` constraint solver library for SWI-Prolog and repeatedly helped me understand more about constraint logic programming. Finally, I would like to thank my colleagues Florian Brandner and Dietmar Ebner for the great time we had working at the laboratory.

All this would not have been possible without the love and support of Margarita and my daughter Luise.

Funding. This work was partially funded by the Austrian Science Fund (FWF) within the research project “Compiler-Support for Timing Analysis” (CoSTA) under contract № P18925-N13, and the research project “Integrating European Timing Analysis Technology” (ALL-TIMES) under contract № 215068 funded by the 7th framework programme of the European Commission.

Contents

List of Figures

List of Tables

Chapter 1

Overview

In our every-day lives there occur many situations in which we trust ourselves to an embedded computer system. Although these computers are generally designed to become invisible to the unsuspecting passerby, it is important to keep in mind that every time we ride cars, board trains or aircraft, or are exposed to medical equipment it is very likely that these are largely controlled by computer software. To put this into perspective, of course, embedded systems are in no way restricted to being used only in such expensive high-end products: In terms of volume, embedded microprocessors and microcontrollers make up $> 98\%$ of the total number of shipped units [as of 2001, see FFY04]. What should be obvious, however, is that all of the above examples have in common that malfunctions can have severe consequences, and potentially even endanger human lives. It is therefore imperative to ensure the correct functioning of such systems.

The key to designing robust embedded systems is to guarantee that the behaviour of the system always meets the specification. Typically, embedded systems perform a set of tasks in a periodical fashion and do so in close interaction with a (physical) environment. It is the software engineer's responsibility to ensure the correctness of the computer programs that perform these tasks. It is important that correctness here not only concerns the results of calculations performed by a program, but includes also non-functional properties, most importantly correct timing. It is the embedded system's interaction with the environment which often calls for real-time constraints. In a typical configuration, multiple tasks share resources like the processor and are periodically called according to a schedule. One way to contain the complexity of the overall design is to make the schedule static and calculate it off-line beforehand. The necessary schedulability analysis can only be performed once the timing characteristics of each task are known. The most important timing characteristic in this context is the worst-case execution time (WCET). Analogously there exist also the average-case (ACET) and best-case execution times (BCET). The average time is mostly of interest to reason about the expected real-world performance of a program and is thus used in pertinent benchmarks. Applications for the best-case execution time are not as numerous, but when interacting with peripheral devices that exhibit initialization delays it can be necessary to guarantee that a program does not finish too early before performing particular I/O operations. It is worth noting that best-case timing analysis can be seen as the dual problem to worst-case timing analysis and can be performed by cleverly inverting the objective function of a worst-case timing analysis.

There are two ways to approach the timing analysis problem. Static analyses use the semantics of the programming language to safely approximate the behaviour of a program without actually executing it. Measurement-based techniques use specially prepared test data to observe the worst-case execution time in a series of measured executions of the program [WEE$^+$08; WKRP08]. In this thesis a static timing analysis work-flow is considered. Intuitively, static WCET analysis is performed by searching for the most expensive path from the start to the end of the control flow graph (CFG) of the analyzed program. This search can be broken down into two independent steps:

High-level: Identification of feasible paths in the CFG via control flow analysis and user feedback.

Low-level: Calculation of the costs (= the execution time) of each of these paths.

High-level timing analysis gathers all information that can be used to identify the set of feasible paths as precisely as possible. It is important to keep in mind that the CFG is generally a cyclic graph and thus contains infinite paths. Cycles in the CFG can arise from loop constructs, goto statements and recursive function calls. It is the task of high-level analysis to prove that every execution of these program constructs will terminate and to supply bounds for the number of iterations of each cycle in the CFG. For this reason the first part of this work focuses on the static analysis of loops (cf. Chapter 3).

In some cases, the static analysis may not be able to yield a satisfying result. It is well known that it is not generally possible to prove termination of a program, not only due to the halting problem but also because the termination conditions are not necessarily part of the program; loops can depend on input data or on interaction with other parts of the run-time environment. To fill this gap, it is thus necessary to aid the static analysis by manually annotating the program. The second part of this work therefore explores mechanisms to improve the handling of manual annotations, by providing user-friendly annotation at the source-code level (cf. Chapter 5).

To derive a proper cost model for the instructions in the CFG a precise model of the target hardware (including processor pipeline and memory hierarchy) is needed [SP10]. It is important to note that the costs for instructions are generally not constant, but are influenced by the execution history (imagine the influence of the contents of the data memory cache) which can lead to a counter-intuitive behaviour, commonly called timing anomalies [KKP10a]. Hardware modelling is a broad field of research and is beyond the scope of this work. Since it is so closely tied to a specific platform, this step will subsequently be called low-level WCET analysis.

1.1 State of the art analyzers and languages

This section gives an overview of both pioneering and state-of-the-art WCET analysis systems and their annotation languages. Focusing on the annotation language is important since a major goal of this work is to increase the usability for the programmers who are performing WCET analysis of their programs. More information about the respective systems and languages is also available in two dedicated survey articles [WEE+08; KKP+10b].

1.1.1 Tree-based WCET analysis

In the early days of WCET analysis, researchers used a syntax-directed approach to compute an upper bound for the execution time of a program [PS97]. In this approach the execution time of instructions on the target hardware is typically modelled with static costs. The program can then be decomposed into blocks of instructions, branches of alternatives and loops or recursions. In the next step, an upper bound n for each loop trip count is needed. Then, each loop is unrolled n times, forming a sequence of repeated loop bodies. After this preprocess, the modelled cost is substituted for each instruction to form a summation expression. Each branch is replaced with an instance of the *max* function. The resulting expression is then evaluated, yielding an upper bound for the program's execution time.

While a dynamic cost model could be incorporated, the drawback of tree-based WCET bound calculation is that infeasible paths cannot easily be excluded from the bound computation, thus leading to a potentially large overestimation. For this reason most newer WCET analyses left this approach for an IPET-based one (cf. Section 1.1.3).

Real-Time Euclid (University of Toronto)

Real-time Euclid is one of the pioneering programming languages with explicit real-time support [KS86]. The language was extended to allow the specification of upper bounds for loops, either in terms of iterations or in absolute *real-time units* which are the smallest granularity of time used by the WCET analysis. In order to ensure that programs are analyzable the language features only a subset of the full Euclid language. Real-time Euclid is no longer maintained.

Modula/R (Vienna University of Technology)

Similar in spirit to Real-time Euclid, the Modula/R language is a real-time oriented version of Modula-2 with extensions tailored for timing and memory consumption analysis [Vrc92; Vrc94]. The extended syntax allows the user to annotate control flow using loop bounds and generic scoped markers. The Modula/R compiler has an option to check the validity of the annotations at runtime.

SPARK Ada (University of York)

SPARK Ada is a subset of Ada 83 originally targeted to program verification which was subsequently adopted for WCET analysis [CBW94; CBW96]. The SPARK language uses a special comment syntax to embed annotations into the program. The annotation language allows the user to specify multiple behaviours for a function that is called in different contexts and/or with different input values. The WCET bound calculation is done via a tree-based approach that works with graph-rewriting rules to find the most expensive path through the program.

TAS (University of Texas at Austin)

The Timing Analysis System (TAS) is a set of tools based on the C programming language [Che87; MACT89]. The general workflow is a four stage process. First, the “annotate” tool analyzes the C language argument program and produces C code that is annotated with default assumptions about the dynamic program behaviour. Next, a modified C compiler translates the annotated C program into annotated assembler code. Using the source code annotations, the compiler produces a script skeleton in the Timing Analysis Language (TAL). User assistance is required to manually refine the TAL-script skeleton of the previous step. A graphical user interface is provided to aid the user. Finally, the “timetool”, interprets the refined TAL-script to perform the actual WCET bound calculation on the assembler code of the argument. In this process, the TAL script thereby effectively acts as a program that calculates the WCET bound. This approach is often called *timing schema* in the literature.

Timing Tool (University of Washington)

The Timing Tool is unique in that it makes the annotating of a program an interactive process [Sha89; PS91]. The Timing Tool supports a subset of the C language and addresses MC68010-based SUN workstations as target platform. The tool allows the user to interactively specify “software timing property”-annotations at the source code level. In a typical Timing Tool session, the user is shown the source code and requested by the tool to specify both upper and lower bounds for loops. The WCET bound calculation is then done with a tree-based timing schema approach.

1.1.2 Path-based WCET analysis

Having experienced the limitations of tree-based WCET analysis, it became clear to the research community that the specification of infeasible paths was one of the weak points. Path-based annotation languages and WCET analyzers seek to overcome this limitation by making the path a first-class object in the control-flow analysis and specification.

PL and IDL (University of Washington)

The Path Language (PL) and the Information Description Language (IDL) are WCET annotation languages designed to specify (in)feasible paths in a program [Par92; Par93]. PL uses a bounded regular expression syntax to describe patterns representing sets of paths. In contrast to the minimal syntax of PL, which consists merely of regular expressions, its sister-language IDL is a more practically oriented language that introduces special language constructs for often needed kinds of information like loop bounds or, for instance, the fact that two locations always occur on a path together.

Although PL and IDL are the only annotation languages that allow to explicitly describe the execution order of code blocks—a feature that could prove very relevant with respect to cache modelling WCET analysis—their success was hampered; probably by the difficulties to concisely express relative execution frequencies [KKP^{+}10b].

Path-based research prototype (Florida State University)

The research prototype originating at Florida State University is unique in that it is the only running implementation of the path-based approach to WCET bound computation [HSR+00; HW02]. The system includes cache and pipeline analysis. Loop analysis is performed on the intermediate code generated by the compiler [vpo, see ADR98] and works by extracting equation systems from the loop conditions that are subsequently solved by a symbolic algebra package. Infeasible paths can be singled out by a branch constraint analysis which detects dependencies between blocks that allow to statically determine the outcome of branch tests for specific paths. Current research is underway to adopt the algorithms towards providing quality-of-service (QOS) information for soft real-time systems [WEE+08].

1.1.3 IPET-based WCET analysis

While path-based annotation languages and analyzers allow to precisely model feasible paths and explicitly specify the execution order of basic blocks, they are not as convenient when it comes to expressing control flow with a relative point of reference, like an encompassing loop. Systems that use the implicit path enumeration technique (IPET) [LM95; PS97] excel at this: The control flow is modelled by a system of linear constraints that express how often certain edges in the control flow graph are executed relative to arbitrary other edges. This constraint system is then presented to an integer linear programming (ILP) solver together with an objective function which then returns the maximal execution time. Cache and pipeline analysis is typically performed on a basic-block level thus yielding static costs for each basic block. The majority of WCET analyzers and research prototypes built around the year 2000 and later uses an IPET-based approach.

aiT (AbsInt GmbH)

The aiT WCET analyzer is a commercial product developed by AbsInt Angewandte Informatik GmbH, Saarbrücken, and is available for different hardware architectures including ARM7, Motorola Star12/HCS12, and PowerPC 555 [FHT03; Fer04; HF05; Abs10a]. Analysis-wise, aiT features a value analysis to predict jump targets and a cache and pipeline analysis. Apart from that, the user can provide annotations in the AIS language. The tool operates on the binary level, and therefore supports object code annotations, like, e. g., jump targets, to reconstruct the CFG. By using debug information for the mapping, annotations can also be specified at the source code level. However, there is no guarantee that a direct mapping will always be possible. The AIS language also offers annotation variables (register variables) for annotating context-sensitive information. Any annotation can make assignments to these variables, which can later be checked for specific values. The values of annotation variables are taken into account by the abstract interpretation when analyzing the program.

Bound-T (Tidorum Ltd)

Bound-T is a commercial WCET analysis tool originally developed at Space Systems Finland Ltd and now marketed by Tidorum Ltd [HLS00; HLS05].

Bound-T operates on the object code level and relies on debug information and user-provided annotations. The analysis process comprises four stages, including call-graph construction, loop-bound computation, user annotation of missing loop-bounds and WCET bound calculation. The first stage performs a control-flow analysis of the argument program and constructs its call graph. In the second stage, each loop body is analyzed by rewriting individual statements into Presburger arithmetic, a decidable subset of integer arithmetic. The implementation uses the Omega framework provided by the University of Maryland. By expressing each loop body as composition of decidable formulæ, it is possible to compute the increment values of loop counters and based thereon bounds for all counter-based loops. The third stage involves user-assistance for loops which could not automatically be bounded in the second stage. Bound-T emits a warning for each missing bound, together with the context of the loop in question. For such loops, the user is prompted to provide an assertion which specifies the missing bound. The fourth stage performs the WCET bound computation, which is done bottom-up on the call graph. The call graph has to be acyclic. The WCET calculation is then performed by transforming the flow information and the program structure into an ILP problem which is subsequently passed to the lp_solve tool [BDEN10].

Chronos (National University of Singapore)

The Chronos WCET analysis tool is based on the processor models of SimpleScalar, a cycle-accurate architectural simulator [LLMR07; ALE02]. Its strengths lie in the modelling of microarchitectural features like out-of-order execution and dynamic global branch prediction. Chronos does not have an annotation language of its own, but allows the user to directly specify constraints for basic blocks in the format of the external CPLEX ILP solver [IBM10]. Chronos also features an automatic infeasible path analysis.

Heptane (IRISA, Rennes)

The Heptane tool combines two different WCET bound calculation methods, one based on C source code and one based on machine code [CP01]. Loop bounds and absolute time bounds for procedures can be annotated by the user. For this purpose, the C syntax was extended with additional constructs. Optimizations, however, potentially invalidate information annotated at source code level. Thus, Heptane also supports machine code annotations, which are to be provided in a separate XML file. Heptane has models for branch prediction instruction caches and a pipeline.

Hume (University of St. Andrews)

The EU-funded Embounded project applies cost metrics to the functional programming language Hume [HDF$^+$05; HBH$^+$07]. The Hume language consists of purely functional “boxes” which are interconnected via communication wires. The source code is compiled into the Schopenhauer high-level intermediate representation; on this representation a type-based “amortized cost analysis” is performed. The output of this analysis is a set of constraints that describe the behaviour of the low-level byte code. The cost of each byte-code instruction is statically determined by invoking the aiT tool once per byte-code instruction and target hardware. By applying the instruction costs and solving the constraint system a WCET bound is calculated.

Otawa **(IRIT, Toulouse)**

The Open Tool for Adaptive WCET Analysis (Otawa) is a framework for building research WCET analyzers, and is available under a free software license [CS06; dMBCS08; BCN$^+$08]. It provides WCET analysis using IPET and is targeted towards binary programs. Static program analysis is provided by the source-level oRange analyzer which extracts loop bounds and infeasible paths from the input program. In Otawa, annotations can either be specified in the Flow Fact XML format (FFX) which is an XML representation of flow information, or in the Flow Facts File Format (F4) which is a simpler textual format with a more compact syntax.

Sweet **(Mälardalen University)**

The Swedish Execution-Time Tool (Sweet) is a research prototype combining control flow analysis of C programs and WCET bound computation for the ARM9 and NEC V850E processors [EE00; EES02; ESG$^+$07; GEL$^+$09]. Recent versions of Sweet are based on the ALF intermediate representation, which is a target-independent language at an abstraction level close to the C language. In Sweet there are several ways to compute infeasible paths and loop bounds, including a combination of program slicing and value analysis but also a technique called "abstract execution" that works by executing the source program symbolically. The timing analysis backend models the pipeline for traces by interfacing with cycle-accurate processor simulators.

Wcc **(Dortmund University of Technology)**

The Wcc compiler is a WCET-aware C compiler focusing on producing an executable that is optimized for a minimal WCET [FLT06; LCFM09; LM09]. The compiler allows for source-level control flow annotations. A distinctive feature of Wcc is a component called the "flow fact manager" that is in charge of keeping all control flow information up to date during the whole compilation process. The Wcc compiler also includes a polyhedral loop model which is the basis for both loop optimizations and loop analysis. Optimizations are either carried out on a high-level intermediate representation (ICD-C) or on a machine-near representation (LLIR). The compiler outputs annotations and code for the Infineon TriCore TC1796, which can then be analyzed by a separate tool like aiT.

wcetC **(Vienna University of Technology)**

The tool chain surrounding the CalcWCET$_{C167}$ tool uses the wcetC language as input language [Kir01; Kir02; Kir08]. wcetC is a C dialect extended by constructs needed to annotate timing information. The distinctive feature of the tool chain is that the compiler (an extended version of GCC 2.7) correctly transforms all control flow annotations from source-code to machine-code level.

The WCETC tool chain does not include any additional control-flow analyses or data-flow analyses.

1.1.4 Measurement-based WCET analysis

The biggest challenge for the correctness of static WCET analysis is the accuracy of the theoretical model of the target hardware, since it makes or breaks the accuracy of the calculated bound. Any errors in the specification or the modelling, any deviations in the implementation of later generations of the used processor, can render any prediction of the WCET inaccurate. Dynamic (measurement-based) WCET analysis tackles this problem by instrumenting the actual target hardware to get accurate time values. However, the gained accuracy is traded for an entirely new problem: It is now necessary to find input data to the target program that triggers the actual worst-case path, otherwise the safety of the measured timings cannot be guaranteed.

Measurement-based research prototype (Chalmers University of Technology)

The research prototype developed at Chalmers University of Technology uses a WCET bound calculation backend that is based on a cycle-accurate simulation of the target processor [LS99a; LS99b; LS99c]. In order to contain the state space the simulation is performed symbolically where possible. To prohibit the repeated simulation of identical path segments, path-merging is performed. The tool contains a data-cache analysis for a certain class of "predictable" data structures. Further, the tool can perform program transformations to eliminate timing anomalies [WKPR05] in the machine code.

RapiTime (Rapita Systems Ltd)

RapiTime is a commercial product developed by Rapita Systems Ltd., York, that combines measurements of the execution-time on the actual target hardware with path analysis to calculate a WCET bound estimate for programs written in C and Ada [BCP03; RAP06]. RapiTime is an extension of the probabilistic WCET analysis developed at the University of York. The measurement passes are controlled by test data that has to be supplied by the user. Annotations are used to identify sensible points for the instrumentation necessary for execution time measurements.

SymTA/P (TU Braunschweig)

SymTA/P is a measurement-based WCET analysis tool tailored to microcontrollers [WSE02; SE06]. It features static analyses for the behaviour of instruction and data caches. The framework was also adapted to estimate the power consumed by a program running on the target hardware. For the execution time measurements traces called "single feasible paths" are identified by a static analysis.

Measurement-based research prototype (Vienna University of Technology)

The research prototype from Vienna University of Technology combines automatic generation of test data with execution time measurements [WRKP05]. To minimize the number of measurement runs, the program is automatically partitioned according to the results of a static analysis. The tool currently does not make use of an annotation language.

1.1.5 Other annotation concepts

Assertion language (Mälardalen University)

This assertion language is a model for a generic annotation language with a broad field of applications [Lis05; KKP$^+$10b]. In principle it can be used to describe quite different aspects of program behaviour. In particular, it can be used to describe flow constraints for WCET analysis, though it was not specifically designed for this purpose. The assertion language is derived from the existing assertion language used in Floyd/Hoare-style logic [Flo67; Hoa69], which is extended by a notion of time and the inclusion of execution counters. Moreover, in the assertion language assertions are no longer necessarily tied to a single program location.

Symbolic annotations and discrete loops (Vienna University of Technology)

Discrete loops are a generalized variant of for-loops [Bli94]. In addition to the parameters of a standard for-loop, in a discreet loop, the programmer can specify constraints that symbolically describe the possible values the iteration variable can take in the immediately following loop iteration. These constraints can either be validated at compile-time or—if that is not possible—at run-time.

The seamless integration of annotation and programming language is an appealing feature; still, discrete loops, which have been proposed for the Ada language have yet to find their way into other programming languages.

1.1.6 Other directions for control flow analysis

Invariant generation (Microsoft Research)

Symbolic bounds describing the asymptotic complexity of programs can also be used as input to WCET bound analysis [GJK09]. Especially in concert with SMT solvers, invariant generation techniques have become increasingly powerful in the recent past. Control-flow constraints (invariants) can be gathered via an abstract interpretation and contribute to construct a state transition system similar to the control flow automata used by model checking software. Complexity bounds can then be found by proving hypotheses on these transition systems using an off-the-shelf SMT solver.

Recurrence relations (Complutense University of Madrid)

Symbolic complexity bounds can also be found by reformulating loop invariants as recurrence relations [AAGP08]. It is possible to approximate recurrence relations

System	CFA	WCETA	Annotations	Optim.	Reference
Euclid	—	tree	srclang	—	[KS86]
Modula/R	—	tree	srclang	—	[Vrc92]
SPARK Ada	—	tree	srclang	—	[CBW94]
TAS	—	tree	bin,ext	—	[Che87]
Timing Tool	—	tree	interactive	—	[Sha89]
PL/IDL	—	path	src	—	[Par92]
Florida	interm.	path	—	—	[HSR$^+$00]
aiT	bin	ILP	src+bin	—	[FHT03]
Bound-T	bin	ILP	bin,ext	—	[HLS00]
Chronos	—	ILP	src,ext	—	[LLMR07]
Heptane	—	ILP	srclang+bin	—	[CP01]
Hume	interm.	ILP	—	high-level	[HDF$^+$05]
OTAWA	src	ILP	src,ext	—	[CS06]
SWEET	interm.	ILP	src	—	[EE00]
WCC	interm.	ILP	src	backend	[FLT06]
WCETC	—	ILP	srclang	backend	[Kir01]
Chalmers	bin	simul.	—	—	[LS99a]
RapiTime	—	measure	—	—	[BCP03]
SymTA/P	src	measure	—	—	[WSE02]
Vienna	—	measure	—	—	[WRKP05]
TuBound	src	ILP	src	high-level	[PSK08], hic!

Table 1.1: Comparison of different WCET analysis systems

by a non-recursive "closed form", which is generally an upper bound. This upper bound formulation represents a symbolic bound to the original program.

1.1.7 Comparison and summary

A summarizing comparison of the presented WCET analysis systems and annotation languages is given in Table 1.1. The first column "System" of Table 1.1 gives the name of the tool(-chain) or language. In the second column "CFA" the abstraction level of the control flow analysis performed by the tool is noted, where applicable: The options are none [—], binary/object code [bin], the compiler's intermediate representation [interm.] and the source code level [src]. In column three "WCETA" the method of WCET calculation employed by the tool is stated. In column four "Annotations" the supported level of annotations is stated: The options include being extensions of the programming language [srclang], being embedded in the source code as comments or pragma declarations [src], referring to the object code or the final binary [bin]. If the annotations are located in an external file, this is also mentioned [ext]. The second-to-last column "Optim." states whether the tool chain can perform optimizations of the program code and on which abstraction level they are performed ([high-level] or in the compiler's [backend]).

Following this table some general design trends in WCET analyzers became obvious: The most obvious one is the move towards an ILP-based backend, as already discussed in Section 1.1.3. The majority of newer WCET analyzers also provide some kind of control flow analysis (CFA). There seems to be no consensus regarding the abstraction level to use for performing the CFA. The choice of abstraction level also comes into play if an optimization stage is available. If the CFA is performed before the program optimizations, i. e., on a higher abstraction level, then any alterations of the control flow introduced by the optimizer need to be taken into account. Providing CFA is important to reduce the amount of manual annotations needed for the WCET analysis. Not only does this save time by lifting the burden from the programmer, it is also important to minimize the amount of *trusted* information the result is based on, thus increasing the likelihood of a correct WCET bound. For the remaining annotations, there is a trend towards moving annotations into the source code. While earlier systems advocated extensions of the syntax of the programming language, a less invasive annotation mechanism using a special comment syntax seems to be preferred by most newer tools. A plausible explanation for this is the move towards using off-the-shelf compilers which are not open for adaptions by the WCET community.

What becomes obvious from this analysis is that the landscape of WCET analyzers is quite heterogeneous, especially in respect to control flow analysis, annotation level and support of optimizations. In response to this situation a number of efforts were undertaken to increase the comparability of tools and the cooperation between research groups:

1. The WCET Tool Challenge was initiated in 2006 [Gus06]. A detailed report was recently published in the International Journal on Software Tools for Technology Transfer [Tan09]. The success from the first tool challenge was the incentive to continue the effort in 2008 with the second WCET Tool Challenge [HGB$^+$08].

2. Backed by funding from the 7th Framework Programme of the European Commission, the research project ALL-TIMES was commenced. The goal of this project is to increase the interoperability of academic and commercial WCET analysis tools [GLS$^+$08; UGoT$^+$10].

The following section details how the work presented in this thesis integrates with the current landscape of WCET analyzers and where it goes further to advance the state of the art.

1.2 Objectives and contributions

The goal of this work is to prove that by making information from the source code available to WCET analysis, both precision and usability can be improved significantly. Not only should static analyses be able to harness the information present in the source code, programmers (who are guiding the automatic analysis) should also be much more productive when, instead of annotating the object code, they can make annotations at the same abstraction level they are writing programs, i. e., at the source code. It is important to ensure that the source-level support does not come at the expense of *safety* and *portability*. The feasibility of the approach needs to be demonstrated in a realistic environment.

Provided with the study of state-of-the-art WCET analysis systems summarized in Table 1.1, the following requirements can be identified for a WCET analyzer to be competitive:

1. Control-flow analysis must be available and must be as powerful as possible.
2. Annotations must be available at the source-code level and before the compilation.
3. The resulting code must be optimized for the worst case behaviour; minimizing the WCET.
4. The WCET bound calculation must support constraints on basic blocks and paths.

However, there are more objectives that are special to this thesis that go beyond recreating work already present in existing analysis systems:

5. Due to advancements in compiler technology and increasing processing power, there is an emerging trend to move away from using C towards using C++ to program embedded systems. Our tool encourages this move by providing support for a sensible subset of C++.
6. Annotations are generated automatically whenever possible. If it is necessary to fall back to manual annotations, their correctness is verified, if that is possible. Annotations are always safely transformed to the assembler language level.
7. We turn WCET analysis and manual annotating into an iterative process of progressive refinement, as described below.

In traditional WCET analysis systems, the user annotates the program after compilation, often at the assembler or object code level. The goal of this work is in fact to make annotating and analysis an interwoven and iterative process. The results of automatic analyses are annotated to the source code, where they can be reviewed by the programmer, who can then decide whether to refine them by hand. In this respect, optimizations performed by the compiler have the unfavorable property of altering the control flow of the program. This is in direct conflict with annotations, whose sole purpose is the description of control flow properties. By making annotations subject to a conjoint transformation together with the program optimizations, this problem can be overcome. In order to provide this as a lightweight addition to the tool chain, the key ingredient here are source-to-source transformations: By carrying out the transformation step on the source code level, the system is remains portable to other architectures, as long as the target's compiler is instructed to perform a direct translation to machine code.

1.2.1 Practical aspects

The algorithms described in this thesis have been implemented by using and extending the SATIrE framework [Vie10b] and the Rose compiler [Law10] for the high-level parts and CalcWCET$_{C167}$ [Vie10a] and GCC [Fre10] as low-level backends. This work resulted in a new WCET analysis software called TuBound.

All program transformations and many of the analyses are implemented on a term-based representation of the C++ abstract syntax tree. As a result of this work a Prolog support library called Termite [Vie10c] originated, which is included in newer SATIrE releases.

The success of the approach was demonstrated by entering TuBound into the WCET Tool Challenge 2008, where the capabilities of the analyzer were compared and benchmarked against several other WCET analyzers. The feedback from the WCET analysis community was generally positive; in the course of the EU FP7-project “ALL-TIMES” our project partners encouraged providing interfaces to TuBound’s static analyses. As a direct result, we incorporated all analyses available in TuBound into the SATIrE distribution which is available under a free software license. Towards the end of the ALL-TIMES project these analyses are now available to aiT, RapiTime, and SWEET. The connection with aiT is realized by generating annotations in aiT’s annotation language AIS. Similarly an annotation generator for RapiTime exists. SWEET is connected via the `melmac` compiler [Bár10], an ALF (the intermediate representation used by SWEET [GEL+09]) backend for SATIrE.

1.3 Outline of this thesis

The following list of chapters gives an outline of this thesis and points to relevant publications that originated from this work:

- Chapter 1 provides a gentle introduction to the field of worst-case execution time analysis, and a survey of WCET analysis methods and tools. We first treated this topic within the paper “WCET Analysis: The Annotation Language Challenge” presented at WCET 2007 [KKP+07]. This has led to the article “Beyond Loop Bounds: Comparing Annotation Languages for Worst-Case Execution Time Analysis”, published in a special issue of the Journal on Software and System Modeling (Springer) [KKP+10b].

- In Chapter 2 the general architecture of TuBound is described. This has led to the paper “TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis” presented at WCET 2008 [PSK08].

- The algorithms and the theory behind the static analyses implemented within TuBound are described in Chapter 3. The paper “Constraint solving for high-level WCET analysis”, based on this work, was presented at WLPE 2008 [PKST08].

- In the first part of Chapter 4 the methods used to ensure the correctness of the analysis implementations are detailed. The technique of “Persistent Analysis Results” was presented at the 26th meeting of the German Gesellschaft für Informatik e.V., Fachgruppe „Programmiersprachen und Rechenkonzepte“ [PKS09]. The second part deals with the ramifications of applying this technique to manual annotations—the paper “From trusted annotations to verified knowledge” was presented at WCET 2009 [PKK+09].

- In Chapter 5 we discuss the problems caused by compiler optimizations and present source-to-source transformations as a portable solution. This includes the automatic transformation of flow information. The underlying idea “The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation” was presented at the 14th Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS ’07)“ [Pra07b]. The fully-fledged approach was later described in the article “Transforming Flow Information during Code Optimization for Timing Analysis” published in the Journal on Real-Time Systems (Springer) [KPP10].

- Chapter 6 serves as a collection of technical details regarding the implementation of TuBound and its interdependence with the SATIrE program analysis framework.

- Chapter 7 gathers benchmark results demonstrating the power of our approach. Among others, this chapter also contains more detailed results from the WCET Tool Challenge than published in the paper “WCET Tool Challenge 2008: Report” from WCET 2008 [HGB+08].

- Chapter 8 concludes this thesis by summarizing notable contributions and pointing to collaborative work that originated from it. Finally it also provides an outlook into worthwhile directions for future extensions.

In the following chapter, we will start by introducing terminology used all throughout thesis.

Chapter 2

Introducing TuBound

2.1 Preliminaries: Flow information

All of the concepts presented in this thesis are centred around the handling of flow information. We define flow information to include any kind of control or data flow information that can be used to describe the feasible paths in a program. Since the actual feasible paths are only a subset of the set of paths represented by the program's control flow graph, we will also use the term *flow constraints* synonymously for flow information. Flow information exists in various degrees of abstraction ranging from value restrictions of variables down to explicit execution counts of certain paths. Loop and recursion bounds comprise essential flow information, without which a timing analysis is rendered impossible. A detailed taxonomy of all the possible kinds of flow information is published in a dedicated survey paper [KKP$^+$08].

Flow information can have originating sources: If the information can be found via a static analysis of the program, then it is inherently defined by the program itself and obviously valid. In practise it is, however, often necessary to let the user provide additional flow information in the form of annotations. Such information is per definition unverified and must be trusted. In some cases (if an automatic analysis was just not strong enough) it is, however, possible to validate such information (cf. Chapter 4 [PKK$^+$09]). Annotations can also serve as a convenient way to implement persistent storage medium for the results of a static analysis. It is therefore of advantage that a tool has a solid support for annotations. A detailed survey of a broad range of WCET analysis tools and their respective annotation languages can be found in [KKP$^+$07].

It was already mentioned that there exists a wealth of different types of flow information. One way to categorize these types is to introduce a hierarchy of flow information, based on its originating location in the chain of timing analysis steps. At the lowest level, flow information consists of cumulative frequency information for all execution paths of the analyzed program. To visualize this, the control flow graph can be imagined as a flow system where the flow of the unique start node and end node is 1. If the timing analysis assumes a constant machine model (where the execution time of one instruction is independent of the execution history), path information can be efficiently stored as execution frequencies of edges in the CFG, as shown in Figure 2.1. If the underlying

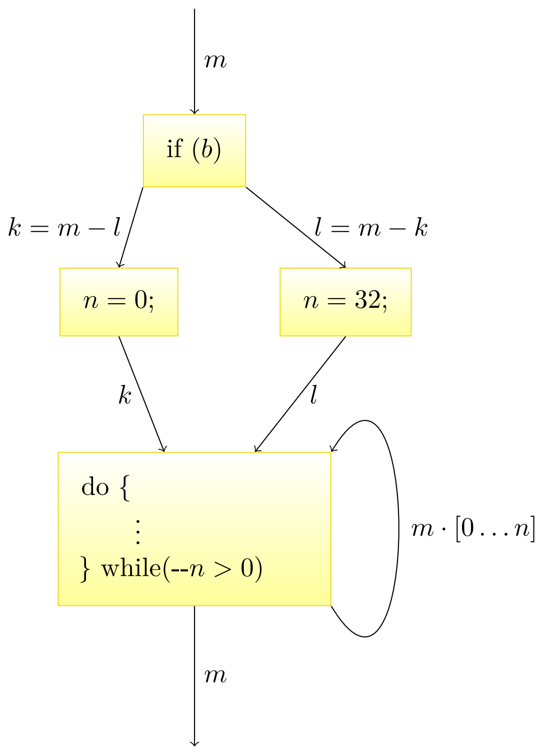


Figure 2.1: Control flow graph with frequency information on the edges

machine model uses dynamic costs (this is brought about by the introduction of pipelines, caches or other shared resources, such as multiple-issue CPUs) it is advantageous to store the frequency information for longer paths and also include other information, such as on mutually exclusive paths. Since the amount of data in this case is much larger it becomes more important to introduce new abstractions such as execution traces [FFY04; WA08].

In order to calculate upper bounds for the particular execution frequencies, a majority of WCET analyzers use integer linear programming (ILP) to maximize a system of flow constraints (IPET, cf. Section 1.1.3). Other approaches, based either on regular expressions or graph rewriting could obviously not prevail in more recent analyzers. For this reason linear flow constraints constitute a de facto standard as a low-level representation of flow information. They also dominate the annotation language of TuBound (cf. Section 2.5).

In our TuBound implementation, we use several levels of flow information, produced and consumed by different analysis steps:

- A points-to analysis generates sets of aliasing pointers.
- An interval analysis uses this information to yield value-ranges for all variables, which are used by
- multiple loop analyses that generate upper bounds for loop iterations as well as more complex flow constraints.

With the exception of the points-to analysis, these steps are decoupled such that the analysis information can be written as annotations to the source code. This flexibility is the basis for the flow information transformation described in Chapter 5.

2.2 Design goals

In this chapter we provide an overview of the architecture and components of the TuBound WCET analyzer software. This description extends on an introductory paper previously published at the WCET'08 workshop [PSK08].

As established in Section 1.1.7, static WCET analysis is typically implemented by the implicit path enumeration technique (IPET) which works by searching for the longest path in the interprocedural control flow graph (ICFG). This search space is described by a set of flow constraints (also called flow facts), which include, e. g., upper bounds for loops and relative frequencies of branches. Flow constraints can generally be determined by statically analyzing the program. However, there are many cases where a tool has to rely on annotations that are provided by the programmer, be it because of the imprecision of the analyses, or due to the fact that information about the execution environment inherently has no representation in the programming language [KKP+08]. An example for this is a loop whose iteration count depends on input data. Current WCET analysis tools, as they are used by industry and in academia, therefore allow the user to annotate the machine code with flow constraints. A central idea behind TuBound is to lift the annotation level from machine code to source code and to smoothly close the gap between source code annotations and machine-specific WCET analysis. We argue that providing high-level annotation support at the source code level has several benefits:

- *Convenience and ease*: For the user, annotating the source code is generally easier and less demanding than annotating the assembler output of the compiler.
- *Reuse and portability*: Source code annotations, which specify hardware-independent behaviour, can directly be reused when the program is ported to another target hardware.
- *Feedback and tuning*: Source code annotations can be used to present the results of static analyses to the programmer for inspection and further manual refinement.

These benefits, however can only be materialized, if the actual longest-path search of the WCET calculation can still be performed on the machine code that will be executed on the target hardware. Compiler optimizations thus represent an obstacle to using source code annotations, as they can change the control flow of the program and hence invalidate annotations.

In TuBound, this is taken care of by transforming flow constraints according to the performed optimizations. Technically, this is achieved by a special flow constraint transformer, which is a core component of TuBound. The technological basis for this component are source-to-source transformations. Therefore, our overall approach is retargetable to other WCET tools; currently we are using CalcWCET$_{C167}$.

From the tool developer's point of view, this source-based approach offers the advantage that analyses can use high-level information that is present in the source code, but would be lost during the lowering to an intermediate representation. A typical example for such information is the differentiation between bounded array accesses and unbounded pointer dereference operations. Moreover, since the output of a source-based analysis is again annotated source code, this approach allows to create a feedback loop where the user can run the static analysis and fill in the annotations where the analysis failed to produce satisfactory results. Afterwards, the analysis can be rerun with the enriched annotations to produce even tighter estimates.

2.3 The Architecture of TuBound

TuBound was built by integrating several components that were developed independently of each other. The majority of the components are designed to operate on source code. This decision was motivated by gains in flexibility for both tool developers and users.

The architecture and work flow of TuBound is summarized in Figure 2.2. The connecting glue between the components is the Static Analysis Tool Integration Engine (SATIrE) [Sch07a; Vie10b]. With SATIrE it is possible to specify data flow analyzers with the Program Analyzer Generator (PAG) [Mar98] and run them on C++ programs, using the infrastructure of the Rose compiler [Law10]. SATIrE transforms programs into its own intermediate representation, which is an interprocedural control-flow graph (ICFG) based on the abstract syntax tree (AST) exported by Rose. An external term representation of the AST can be exported and read by SATIrE. This term representation is generated by a traversal of the AST and contains all information that is necessary to correctly unparse the program. This information is very fine-grained and includes in particular line and column information of the respective expressions. Additionally the terms are annotated with the results of any preceding static analysis. The key feature, however, is the syntax of the term representation. It is designed carefully to match the syntax of Prolog terms. In this form it can be manipulated as and by a Prolog program very easily. A similar approach of using Prolog terms to represent the AST of a program is used in the JTransformer framework for the Java language [Uni10].

The Rose compiler [Law10] is a source-to-source transformation framework that includes the EDG (Edison Design Group) C++ front end, a loop optimizer and a C++ unparser [SQ03]. The loop optimizer was ported from the Fortran D compiler. In TuBound we are using the front end and the high-level loop optimizer that is part of Rose. In the course of this work we also adopted the `clang` front end from the LLVM project [LLV10] to be used as an alternative to the EDG's one (cf. Section 6.2.2). The Program Analyzer Generator (PAG) [Mar98; Abs10b] by AbsInt Angewandte Informatik GmbH allows the specification of data flow analyses in a specialized functional language. Using PAG, we implemented a variable interval analysis for TuBound. To operate on the external term representation of the AST that SATIrE can export and import, we developed the Prolog library Termite [Vie10c]. We implemented our own term-based loop bound

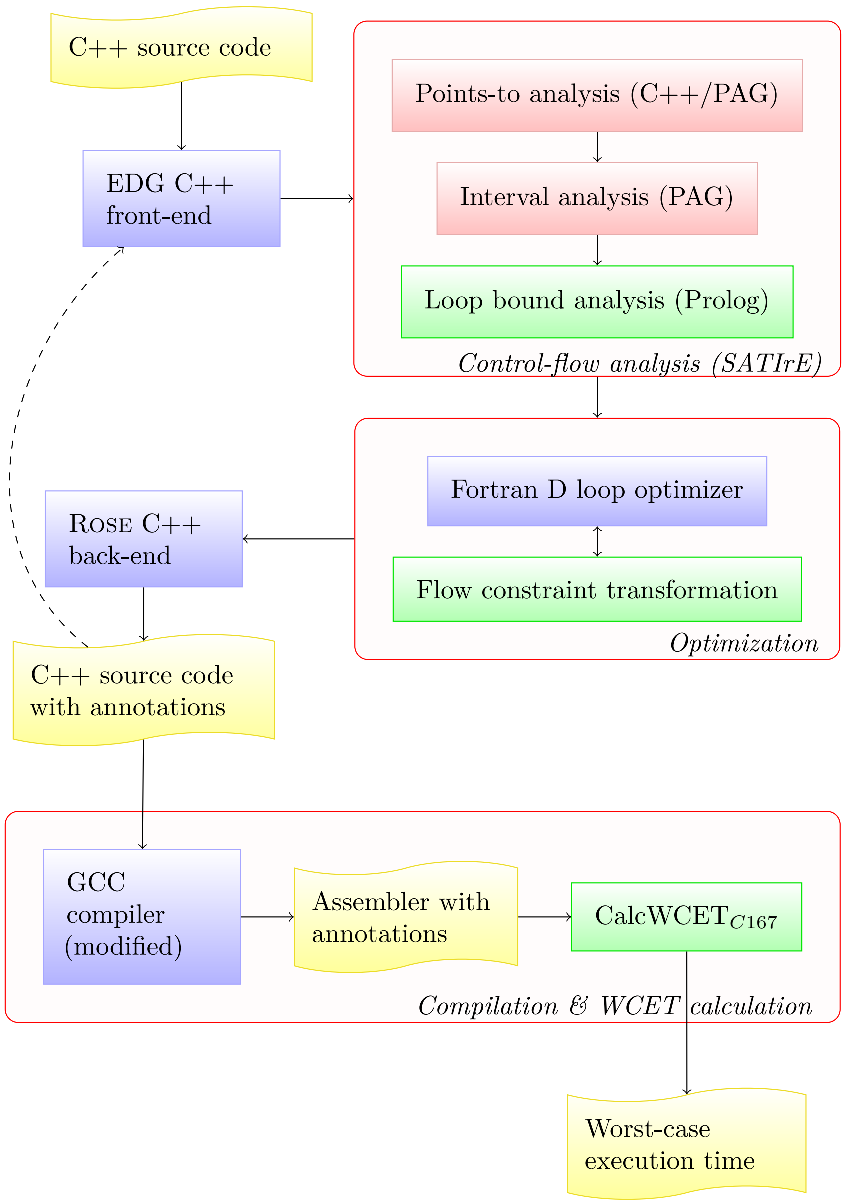

C++ source code
EDG C++ front-end
Points-to analysis (C++/PAG)
Interval analysis (PAG)
Loop bound analysis (Prolog)
Control-flow analysis (SATIrE)
Fortran D loop optimizer
Flow constraint transformation
Optimization
ROSE C++ back-end
C++ source code with annotations
GCC compiler (modified)
Assembler with annotations
CalcWCET$_{C167}$
Compilation & WCET calculation
Worst-case execution time

Figure 2.2: The collaboration of TuBound's components

analyzer with the help of the Termite library (cf. Section 3.2.1). We are using a customized unparser to generate annotated source code. CalcWCET$_{C167}$ [Vie10a] is a tool that performs WCET analysis for the Infineon C167 micro-controller. CalcWCET$_{C167}$ expects annotated C167 assembler code as input. The tool is complemented by a customized version of the GNU C compiler that translates annotated C sources into annotated assembler code for the C167 micro-controller.

2.4 The work flow of TuBound

Conceptually, the work flow of analyzing a program with TuBound comprises three stages:

2.4.1 Start-up and annotation

Parsing. In the first phase, the source code of the program is parsed by the EDG C++ front end that is integrated into the ROSE compiler or, alternatively, by the `clang` front end. ROSE then creates a C++ data structure of the AST and performs consistency checks to verify its integrity. The AST is traversed by SATIrE to generate the interprocedural control flow graph (ICFG), an amalgam of call graph and *intra*-procedural CFG [SP81].

Points-to analysis. Based on the ICFG of the AST, a points-to analysis is performed. This analysis is unification-based [Ste96] with extensions similar to work of Lattner et al. [LLA07]. The points-to graphs resulting from this analysis are available to all subsequent steps via a query interface [Bár09].

Interval analysis. The ICFG data structure is also the interface for the PAG-based interval analysis that calculates the possible variable value ranges at all program locations. The interval analysis operates on a normalized representation of the source code that is generated during the creation of the ICFG. The interval analysis is formulated as an interprocedural data-flow problem and is a preprocess of the loop bounding algorithm, which is otherwise unable to analyze iteration counts that depend on variable values that stem from different calling contexts (context-sensitive analysis). Once the interval analysis converges to a fixed point, the results are mapped back to the AST.

Loop Bound Analysis. The next step is the loop bound analysis. This analysis operates on the external term representation of SATIrE. We exploit this fact with our loop bound analysis which is written entirely in Prolog. Our loop bounding algorithm exploits several features of Prolog: To calculate loop bounds, a symbolic equation is constructed, which is then solved by a set of rules. It is thus possible for identical variables with unknown, but constant values to cancel each other out. For example, in the code

```
for (p = buf; p < buf+8; p++)
```

the symbolic equation is $\texttt{bound} = \frac{\texttt{buf}+8-\texttt{buf}}{1}$. The right-hand side expression can then be reduced by term rewriting rules implemented in the loop bound analysis. The loop bounding algorithm also checks that the iteration variable

Original program

```
int main()
{
    int i,j;
    for (i=0; i<100; i++)
    {
        for (j=0; j<i; j++)
        {
            // body
        }
    }
}
```

Annotations generated by TuBound

```
int main() {
#pragma wcet_marker(m1)
    int i;
    int j;
    for (i=0; i<100; i++) {
#pragma wcet_constraint(
  m2=<m1*100)
#pragma wcet_marker(m2)
#pragma wcet_loopbound(100)
        for (j=0; j<i; j++) {
#pragma wcet_constraint(
  m3=<m_1*4950)
#pragma wcet_marker(m3)
#pragma wcet_loopbound(99)
            // body
        }
    }
    return 0;
}
```

Figure 2.3: Finding flow constraints with TuBound

is not modified inside the loop body. This is implemented with a control flow-insensitive analysis [Muc97] that ensures that the iteration variable does not occur at the left-hand side of an expression inside the loop body and its address is never referenced within its scope.

In the case of nested loops with non-quadratic iteration spaces, loop bounds alone would lead to an unnecessary overestimation of the WCET. In our algorithm, we are using constraint logic programming to yield generalized flow constraints that describe the iteration space more accurately. An example is shown in Figure 2.3. The nested loop in the example has a triangular iteration space, where the innermost basic block is executed $n \cdot \frac{n-1}{2} = 4950$ times. Our analyzer finds the following equation system for this loop nest:

$$m_3 = \sum_{n=0}^{99} m_{3.n}(\{i := n\}) \tag{2.1}$$

$$m_{3.n}(i) = n = i \tag{2.2}$$

$$m_2 = m_1 \cdot 100 \tag{2.3}$$

The equations are constructed with the help of an *environment* that consists of the assignments of variables at the current iteration. The variable m_1 stands for the execution count of the `main()` function, m_2 for the count of the outer loop, and m_3 for the count of the innermost loop relative to the function entry. The variables $m_{3.n}$ describe count of the innermost loop relative to the loop entry for the n^{th} iteration of the outer loop. Equation 2.1 describes the value of i for each iteration of the outer loop, running from $0 \ldots 99$. Equation 2.2 describes the generic behaviour of the inner loop, stating that its iteration count is equal to the value of n in the current environment, where the current environment merely consists of the variable i. The final Equation 2.3 describes the behaviour of the outer loop.

The use of constraint logic programming allows for a lightweight implementation that does not rely on additional tools. In earlier work, Healy et al. [HSR$^+$00] used analysis data to feed an external symbolic algebra system that solves the equation systems for loop bounds. More details on our use of constraint logic programming can be found in Section 3.3.

Eventually, the results of the loop bound analysis are inserted into the term representation as annotations of the source code. We use the `#pragma` directive to attach annotations to basic blocks. The annotations consist of markers, scopes, loop bounds and generic constraints. Markers are used to provide unique names for each basic block, which can then be referred to by constraints. Constraints are inequalities that express relationships between the execution frequencies of basic blocks. Loop bounds are declared within a loop body and denote an upper bound for the execution count of the loop relative to the loop entry. Scopes are a means to limit the area of validity of markers, allowing for example to express relationships that are local to a sub-graph of the ICFG.

2.4.2 Program optimization and WCET annotation transformation

The flow constraint transformation phase is concerned with program sources that are already annotated with WCET constraints, stemming from either an earlier analysis pass or a human. WCET constraints describe the control flow of the program in order to reduce the search space for feasible paths. During the compilation, however, optimizations are performed that modify the control flow. This concerns for example optimizations like loop unrolling, loop fusion and inlining, whereas others like constant folding and strength reduction do not affect the control flow graph. To guarantee the correctness of the annotations of the program sources, we either have to disable these control-flow modifying optimizations and sacrifice performance or transform the annotations accordingly. To achieve the latter, we implemented a transformation component for flow constraints.

A large number of CFG-altering optimizations are loop transformations. For this reason, we based our implementation on the Fortran D loop optimizer that is part of Rose. Keeping optimizations of interest separate from the compiler, our transformation framework is very flexible and also portable to other optimizers. The input to the flow constraint transformation is an optimization trace, consisting of a list of all transformations the optimizer applied to the program, and a set of rules that describe the correct constraint update for each optimization. The concept of using an optimization trace can be applied to any existing compiler. The rules need to be written only once per optimization/compiler pair. The rules, as well as the transformation of the flow constraints are written in Prolog and operate on the term representation of the AST. As a matter of fact, the syntax used to express the flow constraints is identical to that of Prolog terms, too, thus rendering the manipulation of flow constraints very easy. Figure 2.4 gives an example of such a transformation. We currently have rules implementing loop blocking, loop fusion and loop unrolling. With all support predicates, the definitions of the rules range from 2 (loop fusion) to 25 (loop unrolling) lines of Prolog [Pra07b].

2.4.3 Compilation and WCET calculation

As indicated by the dashed arrow in Figure 2.2, the users are free to inspect the annotated source code at this point. If they chose to manually refine the analysis results they are free to do so. It is also possible to restart the automatic analyses at this point, such that they can make use of any newly added information.

Compilation to assembler code. The annotated source code resulting from the previous stage is now converted into the slightly different syntax of the WCETC-language that is expected by the compiler [Kir02]. This is a version of the GNU C Compiler (GCC) that is customized to parse WCETC and to guarantee preservation of all flow constraints and their validity on their way to the C167 machine language level. The output of this customized GCC version is annotated assembler code.

WCET calculation. CalcWCET$_{C167}$ reads the annotated assembler code that is produced by the customized GCC and generates the control flow graph of every function. CalcWCET$_{C167}$ implements the IPET method and contains timing tables for the instruction set and memory of the supported hardware configurations which are used to construct a system of inequalities describing the weighted control flow graph of each function. The weights of the edges correspond to the execution time of each basic block. This system of inequalities is then used as input for an integer linear programming (ILP) solver that searches for the longest path through the weighted CFG. The resulting information can then be mapped back to the assembler code and can optionally also be associated with the original source code [Vie10a].

2.5 The annotation language of TuBound

Annotations in TuBound serve two purposes: Firstly, they provide a programmer-friendly user interface to support the timing analysis with domain-specific knowledge; secondly, they are a textual intermediate representation for results automatically generated by the static program analysis component. Since the result of the static analysis is attached to the program source code, the programmer can inspect it and then decide precisely where to manually refine the annotations, thus keeping the amount of human intervention at a minimum.

2.5.1 Annotation syntax and source code integration

TuBound uses the `#pragma` directive to embed annotations into C++ sources. With this mechanism, it is possible to place an annotation at each sequence point of the program. In its current version, annotations comprise the following four types:

1. *Loop bounds* are the most basic type of annotation available in TuBound. For a successful timing analysis it is mandatory that every loop construct

Original annotated program

```
int* f(int* a)
{
    int i;
#pragma wcet_marker(m_func)
    for (i = 0; i < 48; i += 1) {
#pragma wcet_loopbound(48)
#pragma wcet_marker(m_for)
            if (test(a[i])) {
#pragma wcet_marker(m_if)
                // Domain-specific knowledge
#pragma wcet_constraint(m_if =< m_for/4)

            a[i]++;
        }
    }
    return a;
}
```

After loop unrolling by factor 2

```
int *f(int *a)
{
    int i;
    for (i = 0; i <= 47; i += 2) {
#pragma wcet_marker(m_f_1_1)
#pragma wcet_loopbound(24)
        if ((test(a[i]))) {
#pragma wcet_marker(m_f_1_1_1)
#pragma wcet_constraint(m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
            a[i]++;
        }
        if ((test(a[1 + i]))) {
#pragma wcet_marker(m_f_1_1_2)
#pragma wcet_constraint(m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
            a[1 + i]++;
        }
    }
    return a;
}
```

Figure 2.4: Prolog terms everywhere: WCET constraints before and after loop unrolling

```
 1  { // block b1
 2  #   pragma wcet_marker(m1)
 3      do { // block b2
 4          // Variant A: Generic constraint
 5  #       pragma wcet_marker(m2)
 6  #       pragma wcet_constraint( m2 =< m1*42 )
 7
 8          // Variant B: Constraint with scoped marker
 9  #       pragma wcet_marker(m3)
10  #       pragma wcet_scope(m3)
11  #       pragma wcet_constraint( m3 =< 42 )
12
13          // Variant C: Loopbound
14  #       pragma wcet_loopbound(0..42)
15          ...
16      } while (!done);
17  }
```

Figure 2.5: Three annotations (lines 5–6, 9–11 and 14) carrying the same information

contains such an annotation. Loop bound annotations can be located anywhere inside the scope of the loop body. They always refer to the directly enclosing loop construct.

Loop bounds provide an upper and lower bound for the number of times a loop is executed in relation to the number of times the block right before the loop entry is executed. By convention the special loop bound of -1 is used to indicate an infinite (main)loop.

2. *Markers* are a type of annotation closely related to labels to identify addressable units in the program. A marker creates a symbolic name for a block that can be used in constraint specifications. A marker is always associated with a scope, defaulting to the global scope.

3. *Constraints* are the most generic and powerful annotation kind supported by TuBound. They allow to express arbitrary relations between the execution counts of blocks, referred to by markers.

4. (Marker)*scope* declarations are syntactic sugar to reduce the amount of typing when performing annotation manually. Consider the two blocks b_1 and b_2 in Figure 2.5: In a constraint expression, an occurrence of the marker m_3, which has the scope b_2 is equivalent to the expression m_2/m_1, where m_2 and m_1 are global markers.

Figure 2.5 shows an example utilizing three different types of annotations to express the same fact that the loop construct is executed up to 42 times upon entering the parent scope. It is important to note that both scoped markers and loop bound annotations exist only as a convenience notation to increase readability for the programmer. Both types of annotations are syntactic sugar and are translated to constraint inequalities with global markers before the final

⟨Annotation⟩ → ⟨Loopbound⟩ | ⟨Marker⟩ | ⟨Constraint⟩ | ⟨Scope⟩
⟨Loopbound⟩ → **wcet_loopbound(**⟨Expr⟩**..**⟨Expr⟩**)**
⟨Marker⟩ → **wcet_marker(**⟨Marker⟩**)**
⟨Constraint⟩ → **wcet_constraint(**⟨Expr⟩ ⟨RelOp⟩ ⟨Expr⟩**)**
⟨Scope⟩ → **wcet_scope(**⟨Marker⟩**)**
⟨Expr⟩ → ⟨Expr⟩⟨BinOp⟩⟨Expr⟩ | ⟨UnOp⟩⟨Expr⟩ | ⟨NaturalNumber⟩
⟨Rel⟩ → ⟨Rel⟩⟨BinOp⟩⟨Rel⟩ | ⟨UnOp⟩⟨Rel⟩ | ⟨NaturalNumber⟩ | ⟨Marker⟩
⟨BinOp⟩ → + | - | * | / | ** | **mod**
⟨UnOp⟩ → + | -
⟨Marker⟩ → **[a-z]**(**[a-z]**|**[A-Z]**|**[0-9]**|_)∗
⟨NaturalNumber⟩ → ⟨Digit⟩ | ⟨Digit⟩⟨NaturalNumber⟩
⟨Digit⟩ → **[0-9]**
⟨RelOp⟩ → < | > | =< | >=

Figure 2.6: Grammar of annotations

WCET computation. The static loop analyses in TuBound generate both loop bound annotations and constraints with global markers. The formal BNF-style grammar of annotation pragma declarations is given in Figure 2.6.

Chapter 3

Static analysis

In this chapter we describe in detail the static analyses that were developed to support our approach. These include an interval analysis and two kinds of loop bound analyses for unnested and nested loops, respectively. Interval and loop analyses differ in their method of operation; while the first one is a control-flow-sensitive data-flow problem the latter two are control-flow-insensitive analyses. This difference is also reflected in the implementations. For data-flow analysis the functional specification language (FuLa/OptLa) of the Program Analyzer Generator (PAG) was the language of choice, whereas the loop analyses were implemented with the help of the Termite library in Prolog (cf. Section 6.2). This also enabled the use of methods from the world of constraint logic programming. More details regarding the implementation can be found in Chapter 6. Considering the dependencies between the analyses, the interval analysis can be viewed as an important prerequisite for the subsequent loop analyses. This is also the reason why the loop analyses do not need to follow the control flow because the analysis result of the preceding analysis already encodes all the necessary information.

Parts of this chapter were published in an earlier research paper [PKST08]. Starting with version 0.8.6, the described analyses are included in the SATIrE distribution.

3.1 An interprocedural interval analysis

The purpose of an interval analysis is to associate each variable with a value interval for each location within its scope. This interval contains all values the variable can take at the given location, regardless through which path the location was reached in the execution context. A first treatise on interval analysis, then also called value range propagation, was done by Harrison [Har77]. Interval analysis is also used as running example in the seminal article on abstract interpretation by Cousot and Cousot [CC77]. While interval analysis can also be extended to floating point arithmetic [Gou01], loops depending on floating point decision variables are rarely encountered in typical embedded code. For this reason, this text focuses on integer arithmetic.

Interval analysis is most fittingly specified as an interprocedural data-flow problem, using the extensions provided by the theory of abstract interpretation.

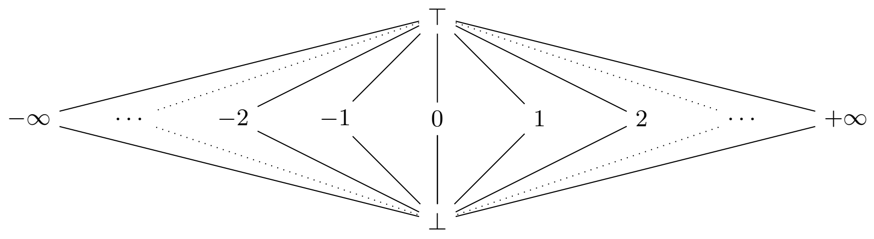


Figure 3.1: Lattice for constant propagation

The variant described here is an extension of the constant propagation analysis specified in the book by Nielson, Nielson and Hankin [NNH99]. The design parameters are sketched in Table 3.1. Just like constant propagation, interval analysis is a forward-directed data-flow problem. The carrier lattice of the analysis is extended to pairs of integers representing the value interval of each integer variable. To further illustrate this, Figure 3.1 shows the "flat" lattice used for constant propagation. The extended lattice used for interval analysis is shown in Figure 3.2. The two elements of the pairs denote the lower and upper bounds of each variable: In constant propagation a join of two different values inevitably results in $\top$. With the deeper structure of the interval analysis lattice, the interval is widened to encompass the whole range of values of all joined intervals. If one bound of the interval is unknown, it is reported as $\pm\infty$; due to the modulo arithmetic implemented in CPUs this is equivalent to the minimum and maximum value that is representable by the underlying data type. The $\bot$ element of the lattice means that a value has not yet been calculated, whereas $\top$ represents an unknown bound, which is equivalent to $(-\infty, +\infty)$ respectively $\pm$`MAXINT`. The transfer functions for each statement capture the ramifications of the statement on the *State* lattice by abstractly interpreting the statement with interval arithmetic. This detail is hidden in the function $\mathcal{A}_{Itvl}$ which is described in detail in Section 3.1.1. At a control-flow join, the combine function is applied pairwise for each variable and merges the interval information coming from different incoming branches. The widening operator, which is used to speed up the fixed-point search, is defined very aggressively, and can be used to fine-tune the trade-off between analysis speed and analysis precision. The transfer functions for conditional branches return different results for the *true* and *false* edges. Since we focus on the C language family, we can also make some assumptions about other data types as well: If the branch condition statically evaluates to either $(1, 1)$ or $(0, 0)$, the state for the other branch is set to $\bot$, such that dead code can not influence the analysis result for live branches.

3.1.1 Arithmetic in the interval lattice

Definitions for standard arithmetic operators, like $+, -, *, /$ on intervals can be found in the literature [AH74]. In this section, we show how to extend these definitions to account for the $\top$ element, infinite bounds and the specifics of C and C++. Furthermore, we also give approximations for typical bit-wise and comparison operators. To this end, we define for each (binary) operator $\circ$ a corresponding operator $\circledcirc$, taking two intervals with possibly infinite bounds

Direction:	*forward*
Lattice:	$State = (Var \to (\mathbb{Z}^{-\infty}, \mathbb{Z}^{\infty}), \sqsubseteq, \sqcup, \sqcap, \bot, \lambda x\ .\ (-\infty, \infty))$
Init:	$\lambda x\ .\ (-\infty, \infty)$
Combine:	$\lambda(a_{\min}, a_{\max})(b_{\min}, b_{\max})\ .\ (\min(a_{\min}, b_{\min}), \max(a_{\max}, b_{\max}))$
Widening:	$\lambda(a_{\min}, a_{\max})(b_{\min}, b_{\max})\ .\ (c_{\min}, c_{\max})$ where $c_{\min} = \begin{cases} a_{\min} & \text{if } a_{\min} = b_{\min} \\ -\infty & \text{otherwise} \end{cases}$ $c_{\max} = \begin{cases} a_{\max} & \text{if } a_{\max} = b_{\max} \\ \infty & \text{otherwise} \end{cases}$
Transfer:	$[x := a]^l\colon f_l^{Itvl}(\sigma) = \begin{cases} \bot & \text{if } \sigma = \bot \\ \sigma[x \mapsto \mathcal{A}_{Itvl}[\![a]\!]\sigma] & \text{otherwise} \end{cases}$
	$[\text{if } (c)]^l_{edge}\colon f_l^{Itvl}(\sigma) = \begin{cases} \bot & \text{if } \sigma = \bot \\ \bot & \text{if } [\mathcal{A}_{Itvl}[\![a]\!]\sigma] = edge \\ f_{l'}^{Itvl}(\sigma), [c]^{l'} & \text{otherwise} \end{cases}$
	where $\mathcal{A}_{Itvl}[\![x]\!]\sigma = \sigma(x)$; $\mathcal{A}_{Itvl}[\![n]\!]\sigma = (n, n)$; $\mathcal{A}_{Itvl}[\![a\ op\ b]\!]\sigma = \mathcal{A}_{Itvl}[\![a]\!]\sigma \ \textcircled{\scriptsize op}\ \mathcal{A}_{Itvl}[\![b]\!]\sigma$

Table 3.1: Specification of the interval analysis

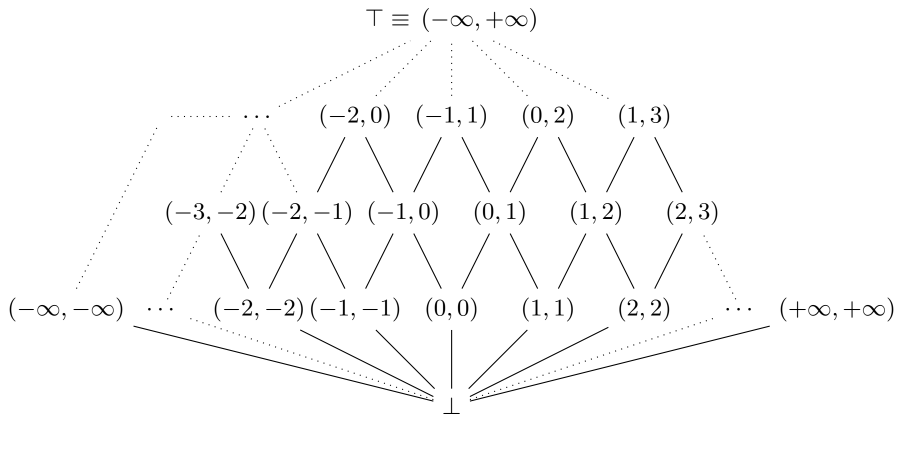


Figure 3.2: Extended lattice for integer interval analysis

as arguments and return an interval which is an upper and lower bound to the operator applied to all pairs i, j out of the two intervals:

$$(a_{\min}, a_{\max}) \circledcirc (b_{\min}, b_{\max}) = (r_{\min}, r_{\max})$$

where

$$\begin{aligned} r_{\min} &\leq \min(R) \\ r_{\max} &\geq \max(R) \\ R &= \{i \circledcirc j \mid i \in [a_{\min} \ldots a_{\max}], j \in [b_{\min} \ldots b_{\max}]\} \end{aligned}$$

For the purpose of the interval analysis, we are satisfied with a safe over- or under-approximation of the operator's effect that is within a certain error margin. As a notational convention, the shorthand a will be used as a placeholder for a tuple $(a_{\min}, a_{\max})$.

Arithmetic operators

In the following definitions for the $\oplus$ and $\ominus$ operators, the approximation is precise, i. e., the error is actually zero:

$$\begin{aligned} a \oplus b &= (a_{\min} + b_{\min}, a_{\max} + b_{\max}) \\ a \ominus b &= (a_{\min} - b_{\max}, a_{\max} - b_{\min}) \end{aligned}$$

For these operators the monotonicity property $x < y \Rightarrow x \pm c < y \pm c$ holds. It is therefore safe to compute the bounds by applying the operator to the extreme values. In the case of $\oplus$ and $\ominus$ it is not necessary to make any extra provisions for infinite values; if the lower or upper bound overflows the corresponding bound the result is interpreted as $\pm\infty$. For the multiplication operator we can additionally exploit the fact that the zero element[1] neutralizes any other operand regardless of its value. This way it is possible to "reanimate" values that were previously reported as $\top$.

$$\begin{aligned} (0,0) \odot b &= (0,0) \\ a \odot (0,0) &= (0,0) \\ a \odot b &= \begin{matrix} \big(\min(a_{\min} \cdot b_{\min}, a_{\min} \cdot b_{\max}, a_{\max} \cdot b_{\min}, a_{\max} \cdot b_{\max}), \\ \max(a_{\min} \cdot b_{\min}, a_{\min} \cdot b_{\max}, a_{\max} \cdot b_{\min}, a_{\max} \cdot b_{\max})\big) \end{matrix} \end{aligned}$$

In contrast to multiplication, integer division is not commutative. It is, however, still monotone. The extreme values are those where the divisor is closest to zero. For C and C++, the result of a division by zero is undefined [WG110; JTC10]:

$$a \oslash b = \begin{cases} \big(\min(a_{\min}/b_{\max}, a_{\max}/b_{\min}, a_{\max}/b_{\max}), & \\ \;\max(a_{\min}/b_{\min}, a_{\min}/b_{\max})\big) & \text{if } b_{\max} < 0 \\ \big(\min(a_{\min}/b_{\max}, a_{\min}/b_{\min}), & \\ \;\max(a_{\min}/b_{\max}, a_{\max}/b_{\min}, a_{\max}/b_{\max})\big) & \text{if } b_{\min} > 0 \\ \top & \text{otherwise} \end{cases}$$

[1]Actually in the modular arithmetic implemented in computer languages, e. g., $\mathbb{Z}_{2^n-1}$ there exists more than one zero element.

The following definition of the modulus operator is no longer precise. We can, however, increase the precision slightly by including a special case for the zero element and the neutral element:

$$\begin{aligned}
a \circledast a &= 1 \text{ if } a_{\min} = a_{\max} \\
a \circledast (1,1) &= a \\
a \circledast b &= \begin{cases} \quad (0, & x-1) \quad \text{if } sgn(a_{\min}) = sgn(a_{\max}), b_{\min} > 0 \\ (1-x, & 0) \quad\quad\; \text{if } sgn(a_{\min}) = sgn(a_{\max}), b_{\min} < 0 \\ (1-x, & x-1) \quad \text{otherwise} \end{cases} \\
&\text{where } x = \max(|b_{\min}|, |b_{\max}|)
\end{aligned}$$

Bit-wise operators

For the bit-wise shift operators, it is impossible to predict the outcome of a shift operation with negative operands. For C and C++ the standard leaves the result of left-shifting a negative number undefined, whereas the result of right-shifting a negative number is defined as implementation-specific. If the right operand is negative, the result is also undefined. The left-shift operators in C and C++ always perform a bit-wise shift filling new bits with zero [WG110; JTC10]:

$$\begin{aligned}
a \circledll b &= \begin{cases} \top & \text{if } a_{\min} < 0 \text{ or } b_{\min} < 0 \\ (a_{\min} \ll b_{\min}, a_{\max} \ll b_{\max}) & \text{otherwise} \end{cases} \\
a \circledgg b &= \begin{cases} \top & \text{if } a_{\min} < 0 \text{ or } b_{\min} < 0 \\ (a_{\min} \gg b_{\max}, a_{\max} \gg b_{\min}) & \text{otherwise} \end{cases}
\end{aligned}$$

For some bit-wise operations we can exploit the fact that in the two's complement representation of -1 all bits are set to one. The helper function *nextpow2(x)* calculates the next-higher power of two for a given integer. The notation $a > 0$ is shorthand for ($a_{\min} > 0$ and $a_{\max} > 0$). Next we present the approximations for the bit-wise *and*, *or* and *xor* operations.
Bit-wise and:

$$\begin{aligned}
a \circledast (0,0) &= (0,0) \\
(0,0) \circledast b &= (0,0) \\
a \circledast (-1,-1) &= a \\
(-1,-1) \circledast b &= b \\
a \circledast b &= \begin{cases} \quad (0, & \max(a_{\max}, b_{\max})) \quad \text{if } a \geq 0 \text{ and } b \geq 0 \\ (-x, & 0) \quad\quad\quad\quad\quad\quad\quad\;\; \text{if } a < 0 \text{ and } b < 0 \\ (-x, & \max(a_{\max}, b_{\max})) \quad \text{otherwise} \end{cases} \\
&\text{where } x = \text{nextpow2}(|\min(a_{\min}, b_{\min})|)
\end{aligned}$$

Bit-wise inclusive or:

$$\begin{aligned}
a \textcircled{|} (0,0) &= a \\
(0,0) \textcircled{|} b &= b \\
a \textcircled{|} (-1,-1) &= (-1,-1) \\
(-1,-1) \textcircled{|} b &= (-1,-1) \\
a \textcircled{|} b &= \begin{cases} (0, x-1) & \text{if } a \geq 0 \text{ and } b \geq 0 \\ (-x, x-1) & \text{otherwise} \end{cases} \\
&\quad \text{where } x = \text{nextpow2}(\max(|a_{\min}|, |a_{\max}|, |b_{\min}|, |b_{\max}|))
\end{aligned}$$

Bit-wise exclusive or:

$$\begin{aligned}
a \textcircled{\wedge} a &= (0,0) \\
a \textcircled{\wedge} b &= \begin{cases} (0, x-1) & \text{if } \text{sgn}(a_{\min}) = \text{sgn}(a_{\max}) = \text{sgn}(b_{\min}) = \text{sgn}(b_{\max}) \\ (-x, x-1) & \text{otherwise} \end{cases} \\
&\quad \text{where } x = \text{nextpow2}(\max(|a_{\min}|, |a_{\max}|) \mid \max(|b_{\min}|, |b_{\max}|))
\end{aligned}$$

Comparison operators

Comparison operators are important to support because they are typically used in branch decisions. The following definition approximates the behaviour of the equality operator:

$$a \textcircled{==} b = \begin{cases} a_{\min} = b_{\min} & \text{if } a_{\min} = a_{\max} \text{ and } b_{\min} = b_{\max} \\ \textit{false} & \text{if } a_{\max} < b_{\min} \text{ or } a_{\min} > b_{\max} \\ \top & \text{otherwise} \end{cases}$$

Dually, we obtain the definition for the inequality operator:

$$a \textcircled{!=} b = \begin{cases} a_{\min} \neq b_{\min} & \text{if } a_{\min} = a_{\max} \text{ and } b_{\min} = b_{\max} \\ \textit{true} & \text{if } a_{\max} < b_{\min} \text{ or } a_{\min} > b_{\max} \\ \top & \text{otherwise} \end{cases}$$

The following definitions for relative comparison are arguably more straightforward to define and also more precise than most of the bit-wise operations:

$$\begin{aligned}
a \textcircled{<} b &= \begin{cases} \textit{true} & \text{if } a_{\max} < b_{\min} \\ \textit{false} & \text{if } a_{\min} \geq b_{\max} \\ \top & \text{otherwise} \end{cases} \\
a \textcircled{\leq} b &= \begin{cases} \textit{true} & \text{if } a_{\max} \leq b_{\min} \\ \textit{false} & \text{if } a_{\min} > b_{\max} \\ \top & \text{otherwise} \end{cases} \\
a \textcircled{>} b &= \begin{cases} \textit{true} & \text{if } a_{\min} > b_{\max} \\ \textit{false} & \text{if } a_{\max} \leq b_{\min} \\ \top & \text{otherwise} \end{cases} \\
a \textcircled{\geq} b &= \begin{cases} \textit{true} & \text{if } a_{\min} \geq b_{\max} \\ \textit{false} & \text{if } a_{\max} < b_{\min} \\ \top & \text{otherwise} \end{cases}
\end{aligned}$$

This concludes the definitions of operations in the interval arithmetic. In the C language family there exist a couple more operators, including Boolean operations, like && and ||. These can be defined analogously to their bit-wise counterparts. The largest other group are assignment operators, like +=, which also share the definitions with the regular operators. Pre/post increment and decrement operators can be treated like a +=1 expression. The result of the `sizeof()`-operator can be evaluated statically, by the compiler front end.

What has been omitted from this discussion so far is the handling of pointers. In the actual implementation, the results from a unification-based points-to analysis are available [Ste96; Bár09]. To take this additional information into account, the pointer dereferencing operator is implemented via a lookup into the points-to database. Similar considerations apply to the member selection operators "." and "→".

In the future, the accuracy of the interval analysis can further be improved by increasing the memory and run-time budget: It can be modified to report a set of possible intervals instead of one merged interval for each variable. In order to reduce the number of necessary sets, several compression strategies can be used. A simple strategy is to always merge the two sets that are closest to each other. A more intricate strategy is to also store strides for each set, such as in the set of even numbers between 1 and 10 (stride 2).

3.2 Analyzing loop bounds

3.2.1 Single loops

The loop bound analysis is a control flow insensitive analysis that builds upon the results of the above interval analysis. The analysis takes as input an iteration-variable based loop L, variable intervals, and context information, such as the scope of the iteration variable i. The preceding interval analysis is technically not a strict requirement, however, it serves as a means to significantly increase the precision of the resulting bounds.

Definition. An iteration-variable based loops contains

- an (optional) initialization statement $[i := \mathit{start}]^{l_1}$,
- one or more exit conjunctive conditions $[(i\ \mathit{rel}\ \mathit{end}_1) \wedge \ldots]^{l_2}$,
- a monotone iteration step statement $[i := i \pm \mathit{step}]^{l_3}$,

where i is an integer variable (that is not a field member) with a $\mathit{scope}(i) \supseteq L$. Further, as a safety condition, there must be no statement $s \in L \setminus \{l_1, l_2, l_3\}$ where i is modified (such as an assignment or increment statement). Also, no pointer may point to the address of i.

The single loop bound analysis works on all iteration-variable based loops, where the step size is either strictly positive or strictly negative:

$$\mathrm{sgn}(\mathit{step}_{\min}) = \mathrm{sgn}(\mathit{step}_{\max})$$

rel	$expr_{min}$	$expr_{max}$	$expr_{step}$
$i < end$:	$start$	end	$step$
$i \leq end$:	$start$	$end + 1$	$step$
$i > end$:	end	$start$	$step$
$i \geq end$:	end	$start - 1$	$step$

Table 3.2: Deriving the loop parameters

The result of the analysis is an upper bound n for the number of times the loop entry is executed in relation to its direct predecessor statements p outside of the loop, where *excnt(s)* denotes the execution count of statement s:

$$\sum_{p \in pred(entry) \setminus L} excnt(p) \;\leq\; n \cdot excnt(entry)$$

Since the discrete function described by the iteration step statement is monotone and its gradient is constant, we can set up the following equation for the loop bound:

$$n = \frac{expr_{max} - expr_{min}}{|expr_{step}|}$$

where $expr_{min}$, $expr_{max}$ are lower and upper bounds for i, whereas $expr_{step}$ is the minimum step size of i on a path through the loop L. We call these values *loop parameters*. To derive the loop parameters, it is necessary to examine the relational operator of the exit condition, which must be one of $<, >, \leq, \geq$.

As shown in Table 3.2, the assignment of $expr_{min}$ and $expr_{max}$ depends on the direction (= sign of the step size) of the loop. In our implementation, concrete values of the loop parameters are calculated in two phases:

1. *Simplify.* In this phase, algebraic identities are exploited to simplify the expression $(expr_{max} - expr_{min})/expr_{step}$. This is implemented by a set of rewrite rules that are applied to the expression until a fixed point is reached. This simplification operates on purely symbolic expressions and disregards the analyzed intervals of variables. It can, however, use the information that an expression is *loop invariant* or *constant*, i. e., no variable occurring in it appears on the left-hand side of any statement in L.

2. *Evaluate.* Using the results of the interval analysis as state, we can evaluate the simplified expression using interval arithmetic ($\mathcal{A}_{Itvl}$, cf. Table 3.1). The return value is an interval (m, n) where n is the upper bound for the iteration count of the loop L.

Lower loop bounds are constructed analogously, by swapping $expr_{min}$ and $expr_{max}$. During the evaluation phase it is important to use the right interval bounds: When we want to derive upper loop bounds, we need to take the lower interval bound for $expr_{min}$, and the upper interval bound for $expr_{max}$. For lower loop bounds, this situation is reversed.

The complexity of this algorithm is bounded by the number of exit conditions in the loop, the depth of the $expr_{min}$, $expr_{max}$, and $expr_{step}$ expressions and the number of rules in the simplification term rewriting system. Implementing the

Direction	Init	Test	Step
up	`I #>= InitExpr`	`I #=< TestExpr`	`(I-InitExpr) mod`
down	`I #=< InitExpr`	`I #>= TestExpr`	➥ `StepExpr #= 0`

Table 3.3: Deriving the constraints

```
1  expr_constr(add_op(E1, E2, _, _, _), Map, Expr) :-
2    expr_constr(E1, Map, Expr1),
3    expr_constr(E2, Map, Expr2),
4    Expr #= Expr1 + Expr2.
```

Figure 3.3: Constraint translation for the + operator

analysis for the source code level does not only increase the precision (because we can use type information), but serves to improve the performance as well: Approaches operating on the machine language level [CM07; HLS05] need to reconstruct the loop's iteration variable from the data flow beforehand.

3.3 Analysis of whole loop nests

TuBound contains an implementation of a loop-bound algorithm that works for nested loops. If the iteration space described by the iteration variables is rectangular or cuboid-shaped, the resulting bounds will be optimal, i. e., precise. Often, however, the iteration variables of nested loops depend on each other, forming, e. g., a triangular iteration space. Loop bounds would then be an overestimation of the iteration space, describing the enclosing rectangle. It is thus desirable to formulate more general *flow constraints* in addition to loop bounds. The flow constraints we are generating describe the execution counts of the loop bodies in relation to the scope containing the outermost loop. Our constraint analysis works by transforming the whole loop nest into finite domain logic constraints.

Each loop in the loop nest must be iteration-variable based. In contrast to the traditional loop bound analysis, a few additional restrictions are imposed on the loop: The step size must be loop invariant. When the loop has a stride greater than 1 ($|step| > 1$), *start* should be constant (remember the loop parameters as described in Table 3.2). Otherwise the results produced by the analysis will be an overestimation which is bounded by the term $start_{\max} - start_{\min}$. Furthermore, the exit test expression must either test for $\leq$ or $\geq$. The expression $expr_{min} < expr_{max}$ can be transformed into the equivalent $expr_{min} \leq expr_{max}-1$, just as $expr_{max} > expr_{min}$ can be transformed into $expr_{max} \geq expr_{min} + 1$.

The algorithm works recursively, beginning with the outermost loop. First, a new logic variable `I` is created that is associated with the iteration variable. Then, the initialization, exit condition and iteration step statements are translated into constraints, as sketched in Table 3.3. The remaining arithmetic expressions, `InitExpr`, `TestExpr`, and `StepExpr` are then recursively translated into corresponding constraints. Figure 3.3 shows how the + operator (the expression `add_op` in the AST) is translated into a constraint: In line 2 and 3, the constraints describing the operands `E1` and `E2` are collected, which are then used to define the main constraint that describes the actual addition (line 4). The `Map` variable holds the interval analysis results to generate constraints for variable AST nodes.

After the constraints are determined, the constraint solver finds the number of possible combinations of all iteration variables that were encountered so far. Since explicit enumeration of all solutions can be prohibitively expensive, we added a new labeling option `upto_in` to our constraint solver (library(clpfd), included with SWI-Prolog [Wie03; TNW08]), which can be used to count the number of possible instantiations if all remaining constraints are trivial.[2] With this method, the running time and memory consumption of the solver no longer depends on the size of the iteration space.

The resulting n is then an upper bound for the number of times the current (= innermost regarded) loop is executed relative to the scope containing the outermost loop. If the constraint analysis is applied to a single loop only, the resulting constraint degenerates into a loop bound. By using this approach, we can leverage a great deal of features from our constraint solver for the loop analysis:

- The order in which the constraints are posted does not influence the behaviour of the solver.
- The termination of the constraint solver is guaranteed.
- The strategy of the solver can be customized through labeling options to improve its efficiency (cf. Section 3.3.1).
- Through the implicit enumeration of the iteration space the results are generally more precise than those of the traditional loop bound analysis.

Since much of the complexity is offloaded into the constraint solver, the implementation is very concise and easy to maintain.

3.3.1 Example

We illustrate the general principle using the following loop nest, for which we want to determine the number of times the inner loop is executed:

```
for (i = 0; i < 10; ++i)
  for (j = i; j > 0; j -= 2)
```

By translating the loop nest accordingly, we get the following constraint program:

```
I #>= 0, I #< 10, I mod 1 #= 0,
J #=< I, J #> 0, (J-I) mod 2 #= 0,
findall((I,J),labeling([], [I,J]), IS),
length(IS, IterationCount).
```

By solving the constraint system, we explicitly enumerate the iteration space `IS` described by (i, j):

[2]This excludes operations like `mod`, where the solver currently falls back to computing the whole iteration space.

```
  [ (1, 1),
    (2, 2),
    (3, 1), (3, 3),
    (4, 2), (4, 4),
    (5, 1), (5, 3), (5, 5),
    (6, 2), (6, 4), (6, 6),
    (7, 1), (7, 3), (7, 5), (7, 7),
    (8, 2), (8, 4), (8, 6), (8, 8),
    (9, 1), (9, 3), (9, 5), (9, 7), (9, 9) ]
```

The number of pairs in the iteration space is then an upper bound for the innermost loop body. In our case, exactly 25 times. For larger bounds, explicit enumeration of all solutions is infeasible. With the help of the `upto_in` labeling option we can directly count the number of possible instantiations. Thus we can reduce or avoid explicit enumeration in many cases. The following example demonstrates that this even works with very large numbers:

```
I #>= 0, I #=< 10000,
J #>= 0, J #=< 500,
labeling([upto_in(IterationCount)], [I,J]).
```

This directly yields IterationCount = 5010501.

Chapter 4

Obtaining flawless annotations and analyses

4.1 Automatic testing of a data-flow analysis

It is well known that the correctness of a program is not only a matter of proving the soundness of the underlying algorithm but in practise mostly one of thoroughly testing of the implementation [Fet88]. This is especially important for compiler construction, since a bug in the compiler can introduce unwanted behaviour in every program that is translated with it. Production compilers like the GCC therefore ship with a huge test suite that every version has to pass before it can be released. These test suites, however, typically consist of small programs that are created to target one specific behaviour of the compiler at a time. In fact most of these test programs are extracted examples from bug reports contributed by frustrated users. When it comes to the testing of a static analysis it would make more sense to directly test correctness of the analysis results rather than the correct behaviour of programs that were transformed because of an analysis result.

In this chapter we show how to achieve this by transforming the program: Given a static analysis that generates a pre- and a post-condition for the statements in a program, that program can be transformed to assert the respective conditions before and after every statement. The program can then be executed to see whether any of the assertions fire (dynamic). It is even possible to pass the program to a model checker to formally verify the correctness of the asserted conditions (static). It is thus possible to reuse existing test cases and transform them into tests for a whole new purpose. This enables a smart reuse of an already existing test suite.

4.1.1 Preparing test cases

The method described above is already a major improvement for the author of a static analysis. In practise some test cases need to be preprocessed to ensure a better coverage. Typical test-suite programs are fragments that were extracted from more complex programs and thus often lack an entry-function `main(argc, argv)`. We thus need to generate an artificial main function that calls the other functions in the test module. To generate all the function calls it is important to adhere the calling conventions of the other functions in the module. For parameterless functions, this is easily achieved, otherwise it is necessary to fake input values depending on the function signature. To gain maximum function coverage, a naïve main-function would call every other function in the module. Since we want to test multiple combinations of input values, it pays off to take the call graph into account to reduce the overall number of function calls. By calculating the transitive closure of the call graph, a minimal set of connected components can be identified. One call to the first node of each cluster of functions guarantees that every function is reachable from the main function.

Once the test cases are preprocessed to constitute full executable programs on their own, the only missing ingredient is the assertion generating source-to-source transformation. In the next section we discuss several routes to introduce automatic assertion generation into the compilation process.

4.2 Assertion-carrying code work flow

In the following, we will refer to code that is outfitted with automatically inserted assertions as *assertion-carrying code* (ACC). This name was selected because the idea of ACC is similar in spirit with proof-carrying code [Nec97]. Figure 4.1 highlights the work flow of ACC. It shows how ACC fits into the compilation process. Steps I and II initiate the process by transforming the information gained from an analyzer into dynamic assertions that are embedded into the program code of some program Π:

I. Perform a series of (potentially complex and expensive) analyses on Π.

II. Transform analysis results into assertions and attach those to the program itself.

Besides informing the user about certain aspects of the run-time behaviour of the program, ACC can be subject to other analyzers extending the behaviour information further. Moreover, at any step, the ACC can be passed on to the compiler to proceed with compiling it and to generate executable code. Transformations applied by the various stages of the compiler, in particular its optimizer stage, can directly refer to this information without need to (re-)compute it; thus speeding up the compilation process.

How ACC can be used to create a feedback loop by making previously found information available to a source-to-source analyzer is also shown in Figure 4.1. This makes up the third and fourth step of the complete workflow:

III. Feed the ACC of a source program to the analyzer, thereby obtaining a new ACC, with possibly extended behaviour information.

IV. Stop analyzing the ACC and pass it on to the compiler.

We already saw how ACC is useful to inform the user on certain aspects of the run-time behaviour of a program and can easily be reused and exploited by other program analyses typically applied throughout the compilation process. Next we demonstrate how to prepare ACC, which is the precondition for the final step:

V. Automatically check analysis results with static or dynamic verifiers.

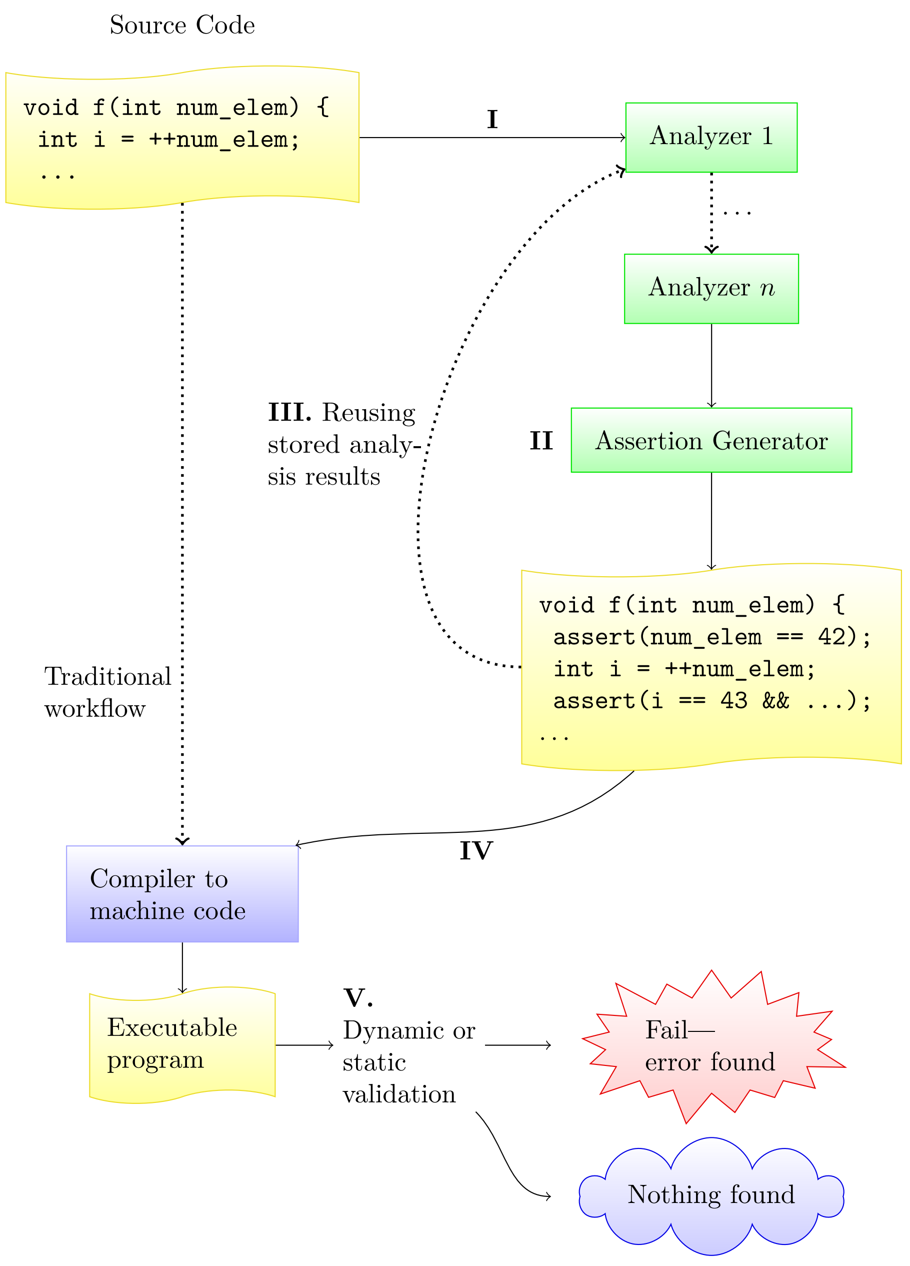


Figure 4.1: The work flow of assertion-carrying code

4.3 Checkable assertions

Intuitively, the results of a program analysis provide information about the possible states of the underlying program at the various program points. Considering the two points immediately before and after the execution of a statement, this information can be thought of in terms of a pre- and a post-condition describing a property of the program states which may be valid immediately before and after executing this statement. In analogy to (valid) Hoare-assertions [Hoa69], we write

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

where P is the precondition of a program fragment Q and R a post-condition capturing the effect of Q with respect to P. The expressions P and R are assertions which ensure a certain property of program Q. In the same fashion the results of a program analysis attached to a statement can also be considered assertions about the program state at this location. Some programming languages offer an integrated concept of assertions: ANSI C in terms of the `assert`-macro and Eiffel in terms of *contracts*. These language-provided assertions can be automatically checked at run-time and are useful for security checks. In the following we present our approach of how to represent analysis results such that they bear the flavour of assertions, which can then be subject of static (off-line) verification and dynamic (on-line) testing.

To this end we define the ACC program transformation that modifies the analysis information attached to a program such that the attached analysis information can be used like an assertion. It is worth noting that this does not affect the semantics of the program (provided the analysis information is correct), while offering the advantage of having the analysis information immediately available. An implementation might even be organized to directly output assertions and skipping the intermediate step of outputting plain analysis information. In our approach, we distinguish assertions addressing *universally* and *existentially* valid program properties. Using the programming language C as the source language for our demonstration, we will show next how to express the two kinds of assertions for this setting.

4.3.1 Notation

We use the following notational conventions: The letter σ denotes the perfect, precise analysis summary. A summary is associated with a certain location l in the program and consists of multiple similar properties p_i. The implemented analyzer, however, actually returns an approximation $\tilde{\sigma}$. Depending on the policy, we can distinguish two ways of approximating analysis information:

- *Optimistic* analyses over-approximate the summary by computing a superset of the precise properties.
- *Pessimistic* analyses under-approximate the summary by computing a subset of the precise properties.

This is illustrated by Figure 4.2, which depicts a typical lattice of analysis results: At the top, the most optimistic (and utterly useless) approximation $\top$, containing all properties is shown. The (useful) optimistic approximation $\tilde{\sigma}_{opt}$ contains all correct properties p_i and a few false positives q_i. In the centre, the precise

$$\begin{tikzcd}
\top \arrow[d, dash] & \text{(all properties)} & \top \arrow[dd, dashed] \\
\tilde{\sigma}_{opt} = \{p_1 \ldots p_n, q_1, q_2\} \arrow[d, dash] & \text{(optimistic approximation)} & \\
\sigma = \{p_1, p_2, p_3, \ldots p_n\} \arrow[d, dash] & \text{(precise solution)} & \\
\tilde{\sigma}_{pess} = \{p_1, p_3, p_{n-1}\} \arrow[d, dash] & \text{(pessimistic approximation)} & \\
\bot & \text{(no properties)} & \bot \arrow[uu, dashed]
\end{tikzcd}$$

Figure 4.2: Optimistic versus pessimistic approximations

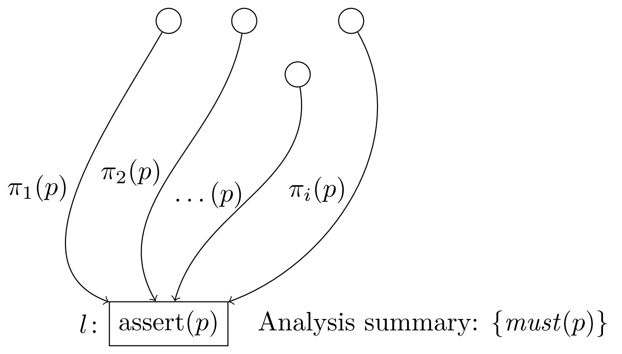


Figure 4.3: Universal properties: p holds on all paths π_i reaching l. The assertion tests whether the analysis was conservative (or safe).

analysis summary σ is shown. Below, we have a pessimistic approximation $\tilde{\sigma}_{pess}$ that is a strict subset of σ. The most pessimistic result is the empty set, or the bottom element of the lattice, $\bot$.

In order to automatically generate assertions, we use a mapping function $f\colon (l, \tilde{\sigma}) \mapsto \mathit{Expr}$ to convert an analysis summary and location pair into an assertion expression.

4.3.2 Assertions for universally valid properties

Intuitively, a property p is *universally valid* at a location l if it is valid, regardless along which path the location l was reached from. Pessimistic program analyses computing universally valid information are commonly called *must*-analyses [NNH99; Hec77]. Figure 4.3 serves as an illustrative example.

Universally valid properties can be specified in terms of an assertion by testing for the respective analysis information. This way a failed assertion indicates a faulty analysis result. This is illustrated by the following example:

Example 1. Constant Propagation is a typical *must*-analysis. For each location l, it yields universally valid information in terms of expression-constant pairs $(expr, c)$ meaning that at the location l, the expression $expr$ is always

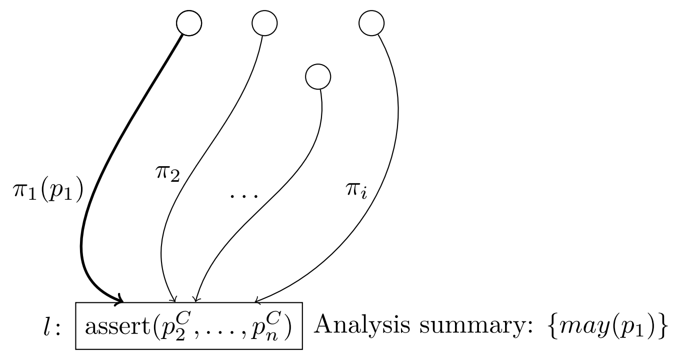

$\pi_1(p_1)$
π_2
$\dots$
π_i
l: assert($p_2^C, \dots, p_n^C$)
Analysis summary: $\{may(p_1)\}$

Figure 4.4: Existential properties: p_1 holds for at least one of the paths p_i. The assertion tests whether the analysis was conservative (or complete).

equal to the constant value c. In the C programming language, the assertion `assert(expr == c)` placed at l expresses this information.

According to [ALSU07], we define:

Definition 1. A universally valid analysis summary $\tilde{\sigma}$ is *conservative* (or safe) if it does not contain any false positives, such that $\tilde{\sigma} \subseteq \sigma$.

Using the notion of safety, Example 1 can be generalized to any universal information as summarized in Observation 1, which follows immediately from the definition.

Observation 1. If an analysis summary $\tilde{\sigma}$ reports a property p to be universally valid at a location l, a statement `assert(`$f(p)$`)` inserted at l will succeed if $\tilde{\sigma}$ is a conservative (safe) approximation of σ (with f being a mapping function from the analysis domain to the source language, cf. Section 4.3.5). If the assertion fails, we know that $p \notin \sigma$ and therefore $\tilde{\sigma}$ is not a safe approximation of σ.

4.3.3 Assertions for existentially valid information

A property is said to be *existentially valid* at a location l if it is valid on at least one path reaching l. Optimistic program analyses computing existentially valid information are commonly called *may*-analyses. Figure 4.4 serves as an illustrative example.

Since existentially valid information need not hold along all paths, an assertion addressing such a property cannot directly test for it the way it would for universal information; the assertion would be executed every time the location is passed, however, the analyzed properties would hold only for some of these executions, if any at all.[1] This is because assertions naturally test for universal properties.

[1]There are two reasons why this might happen: (1) The relevant path is incidentally never reached during the program execution. (2) The approximation $\tilde{\sigma}$ introduced false positives.

Definition 2. An existentially valid analysis summary $\tilde{\sigma}$, consisting of properties $\tilde{p}_i$ calculated by a may-analysis, is *conservative* (or complete) iff it contains all valid properties $p_i \in \sigma$, such that $\tilde{\sigma} \supseteq \sigma$.

A conservative summary returned by an (optimistic) may-analysis implies that any properties p_j which are not part of the analysis summary can never hold for any program execution. This, again, is universal information that can be tested for with assertions. Instead of considering the existential properties $p_i \in \tilde{\sigma}$ the complementary information $p_j^C \notin \tilde{\sigma}$ is a universal property that can be checked as other universal properties. Observation 2 now follows immediately from Definition 2:

Observation 2. If an (optimistic) may-analysis summary $\tilde{\sigma}$ reports a property p to be existentially valid at location l, a sequence of statements `assert(`$f(p_j^C)$`)` for all other $p_j \notin \tilde{\sigma}$ of the same type as p will succeed, if $\tilde{\sigma}$ is a conservative (complete) approximation of σ. If any of the assertions fail, we know that $\tilde{\sigma}$ does not contain all valid results and therefore $\tilde{\sigma}$ is not a conservative approximation of σ.

The following example shows how to apply this idea to an alias analysis:

Example 2. A *may*-alias analysis yields multiple disjoint sets $\{A_1, \ldots, A_n\}$ where each A_j is a set $\{a_1, \ldots, a_n \mid addr(a_i) = addr(a_{i+1}), i < n\}$ of potentially aliasing pointer variables. In the C language, each set in such a summary can be transformed into the assertion `assert(a`$_1$` != b`$_1$` && a`$_1$` != b`$_2$` && ...&& a`$_n$` != b`$_n$`)`, where $b_j \neq a_i$ are all remaining pointer variables. For a program with m pointer variables and a summary describing n variables as potentially aliased via pointers, the assertion consists of $m \cdot n$ comparisons. Under the assumption that the number of aliasing pointer variables is a very limited in typical programs, this approach is feasible in practise.

Pessimistic existential and optimistic universal analyses

In the case of pessimistic existential analyses, it is not generally possible to transform the analysis summary into assertions, because there are no assumptions about the completeness of the summary. It is, however, still possible to provide positive feedback (a witness [ZP08]) during testing by stating that what the analysis found actually occurred during program execution. The same applies for optimistic universal analyses.

4.3.4 May, must, and conservative approximations

The relationship between safety and completeness, existential and universal information is shown in the Venn diagram in Figure 4.5. It depicts the analysis result set for live variables (existential) and the dual problem of dead variables (universal). The rectangular area is the space defined by the carrier set. The small area σ is the set of variables that actually are live in one path to the exit node. The larger area $\tilde{\sigma}$ is the conservative approximation returned by our analysis. The whole area surrounding σ marks the dual information: All variables that are dead on all paths to the exit node. Here we can see that a conservative

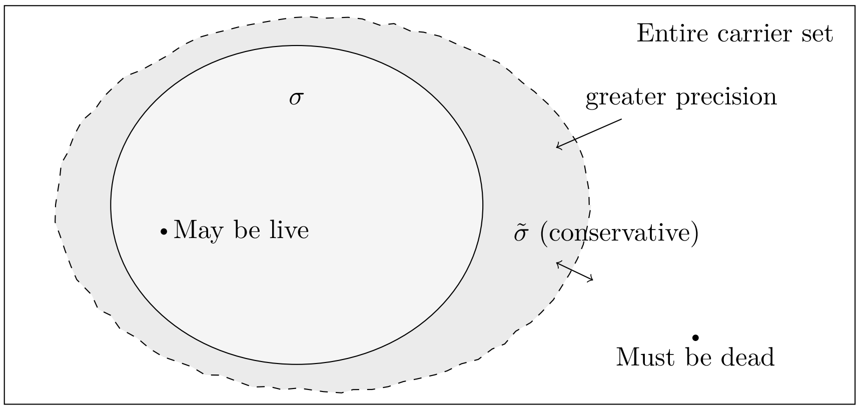


Figure 4.5: Conservative approximation for a live variable analysis (may)

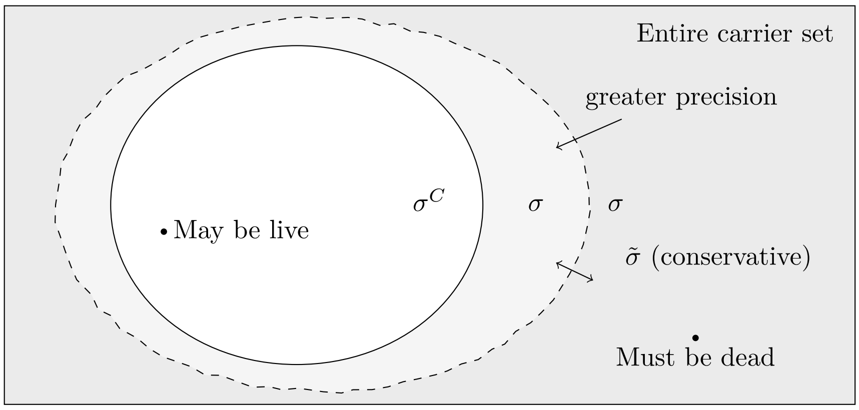


Figure 4.6: Conservative approximation for a dead variable analysis (must)

Analysis	Direction	Type	Summary (σ)
Available Expressions	forward	must	$\big\{(v,e)\big\}, v \in \mathit{VarSym}, e \in \mathit{Expr}$
Constant Propagation	forward	must	$\big\{(v,c)\big\}, v \in \mathit{VarSym}, c \in \mathit{Int}$
Must-alias Analysis	forward	must	$\big\{\{p_i\}\big\}, p_i \in \mathit{PtrSym}$
May-alias Analysis	forward	may	$\big\{\{p_i\}\big\}, p_i \in \mathit{PtrSym}$
Reaching Definitions	forward	may	$\big\{(v,l)\big\}, v \in \mathit{VarSym}, l \in \mathit{Loc}$
Very Busy Expressions	backward	must	$\{e\}, e \in \mathit{Expr}$
Live Variables	backward	may	$\{v\}, v \in \mathit{VarSym}$

Table 4.1: Signatures of typical textbook analyses

may-analysis enlarges the area $\tilde{\sigma}$ by reporting more values than necessary. A conservative (dual) must-analysis would shrink the remaining area, moving the dashed border *in the same direction*, by reporting fewer values than theoretically possible. To stress this point, Figure 4.6 shows a conservative approximation in the dual (must-be-)dead variable analysis. For a precise analysis the areas σ and $\tilde{\sigma}$ would be identical.

4.3.5 From analysis results to assertions

The function f is used to map an analysis summary into an assertion. This function has to be defined for every analysis. In this section we show how to design such a mapping function. Often this translation is straightforward, as we saw in the examples used in the previous section, where we were able to conveniently translate the analysis result into a testable expression of the source language. These analyses are the natural candidates for ACC, we will subsequently call them first-class candidates.

Table 4.1 shows the signatures of typical textbook data-flow analyses [NNH99]: For each analysis the characteristics such as data flow direction and quantification of the analysis result are given. The carrier set of each analysis is given in column 3. The data types used in these sets include variable symbols (*VarSym*), source language expressions symbols (*Expr*), pointer variable symbols (*PtrSym*) and labels to code locations (*Loc*).

We will now show how to define a mapping function for each of these analyses. To this end we provide a function $f\colon (l,\tilde{\sigma}) \mapsto \mathit{Expr}$ which maps a pair of code location and analysis summary to generated source code (written out in typewriter font) to be inserted at that location l. The helper functions *defs* and *uses* return the list of variables that are defined and used by node l, respectively. The mapping functions for Very Busy Expressions and Live Variables differentiate between $\tilde{\sigma}_{pre}$ and $\tilde{\sigma}_{post}$ denoting the analysis summary of just *before* and *after* node l. For better legibility this detail is omitted in the other mapping function definitions.

Available Expressions analyzes which expressions e have been computed at a program point l and can be made available in a new variable v. The mapping function generates an assertion that the contents of that variable v is equal to the expression e:

$$f(l,\tilde{\sigma}) \mapsto \forall (v,e) \in \tilde{\sigma}\colon \texttt{assert(}v\texttt{==}e\texttt{)}$$

Constant Propagation tracks whether the value stored in a variable v is a constant c. The mapping function generates an assertion that v is equal to c:

$$f(l, \tilde{\sigma}) \mapsto \forall (v, c) \in \tilde{\sigma} \colon \texttt{assert(}v\texttt{==}c\texttt{)}$$

Must-alias Analysis yields sets of pointer variables that point to the same memory location. The mapping function generates an assertion that the values of all pointers in each set is equal:

```
$f(l, \tilde{\sigma}) \mapsto$
  $\forall P \in \tilde{\sigma} \colon$
    $\forall p_i, p_j \in P \colon \texttt{assert(}p_i\texttt{==}p_j\texttt{)}$
```

May-alias Analysis yields sets of all pointer variables that potentially point to the same memory location. The mapping function covers two cases: The first one asserts that no pointer from the may-alias sets aliases with the remaining pointers or with pointers from other may-alias sets. The second case asserts that the values of all remaining pointers variables are distinct:

```
$f(l, \tilde{\sigma}) \mapsto$
  $\forall P_i \in \tilde{\sigma} \colon$
    $\forall p_i \in P_i \colon$
      $\forall q_j \in \mathit{PtrSym} \setminus P_i \colon \texttt{assert(}p_i \neq q_j\texttt{)}$
  $\forall q_i, q_j \in \mathit{PtrSym} \setminus \bigcup \tilde{\sigma} \colon \texttt{assert(}q_j \neq p_i\texttt{)}$
```

Reaching Definitions analyzes which assignments may have been made at a location. The dual information is that a definition was definitely overwritten when the current location is reached. To encode this information in an assertion, the mapping function introduces a flag carrying the information that a certain definition is valid. The flag is reset when a variable is redefined. The assertions compare all the flags against the analysis summary:

```
$f(\mathit{atinit}, \_) \mapsto \texttt{defined} = \overrightarrow{0}$
$f(l, \tilde{\sigma}) \mapsto$
  $\forall v' \in \mathit{defs}(l) \colon$
    $\texttt{defined}[(v', l)] = \texttt{1}$
    $\forall l' \in \mathit{Loc} \setminus l \colon \texttt{defined}[(v', l')] = \texttt{0}$
  $\forall (v, l') \in (\mathit{VarSym} \times \mathit{Loc}) \setminus \tilde{\sigma} \colon \texttt{assert(!defined[}(v, l')\texttt{])}$
```

Very Busy Expressions analyzes which expressions will definitely be used in the future. An expression is very busy if it is always going to be used before any of its variables are redefined. The mapping function introduces a busy-flag for each expression which is initialized to zero at the program entry point. The flag is toggled whenever an expression changes its busy-status at the current location l. For each location an assertion is generated, stating that no variable that is redefined ($\mathit{defs}(l)$) there could have been part of a very busy expression unless it was used by the current statement itself ($\mathit{uses}(l)$). The function $\mathit{VarSym}(e)$ returns all variable symbols occurring in the expression e. Finally, at the exit node it is asserted that no very busy expressions remain:

```
$f(atinit, \_) \mapsto$ busy $= \overrightarrow{0}$
$f(atexit, \_) \mapsto$ assert(busy==$\overrightarrow{0}$)
$f(l, \tilde{\sigma}) \mapsto$
  $\forall e \in \tilde{\sigma}_{post} \setminus \tilde{\sigma}_{pre}$: busy[e] = 1
  $\forall e \in \tilde{\sigma}_{pre} \setminus \tilde{\sigma}_{post}$: busy[e] = 0
  $\forall e \in Expr$:
    if $(defs(l) \in VarSym(e) \wedge VarSym(e) \cap uses(l) = \emptyset)$
        assert(!busy[e])
```

Live Variables yields all variables that may be used in the future. The mapping function for Live Variables was designed with the method introduced in Section 4.3.3. It operates on the dual universal problem of definitely dead variables.

```
$f(atinit, \_) \mapsto$ dead $= \overrightarrow{0}$
$f(l, \tilde{\sigma}) \mapsto$
  $\forall v \in \tilde{\sigma}_{post} \setminus \tilde{\sigma}_{pre}$: dead[v] = 1
  $\forall v \in \tilde{\sigma}_{pre} \setminus \tilde{\sigma}_{post}$: dead[v] = 0
  $\forall v \in uses(l)$: assert(dead[$v$]==$\overrightarrow{0}$)
```

The last three examples represent *second-class* candidates for ACC, which share the common characteristic that the analyzed information is not directly accessible from the programming language. A remedy for this situation is to introduce flags into the program that carry the analysis information. This way the transformed program dynamically recomputes the analysis information and again has the analyzed information accessible from the programming language as is the case with first-class candidates.

To perform such a "dynamic data-flow analysis" it generally suffices to toggle corresponding flags once a certain condition is fulfilled at runtime. There is naturally no need to define merge operators, as there is only a single execution path (at least for single-threaded programs). We will now show how to apply this to the two kinds of analyses:

- For universal problems, the assertions generated by the mapping function have to check that the summary computed at runtime is a superset of the static analysis summary $\tilde{\sigma}$.
- For optimistic existential problems, the assertions have to check for the dual (and universal) property that the summary computed live is a subset of the complementary static analysis summary $\tilde{\sigma}^C$.

Figure 4.7 illustrates this by encoding both *may* and *must*-information into dynamically computed flags. At the location l, the assertions test whether the computed information is a sub- or super-set of the analyzed information.

While this ACC representation for second-class candidates is sufficient to test the safety of the analyzer, it is not particularly efficient as data store: A second analysis run (cf. Figure 4.1) will not be able to recover the information. Judging from Table 4.1 the decision whether a particular analysis is a first-class candidate

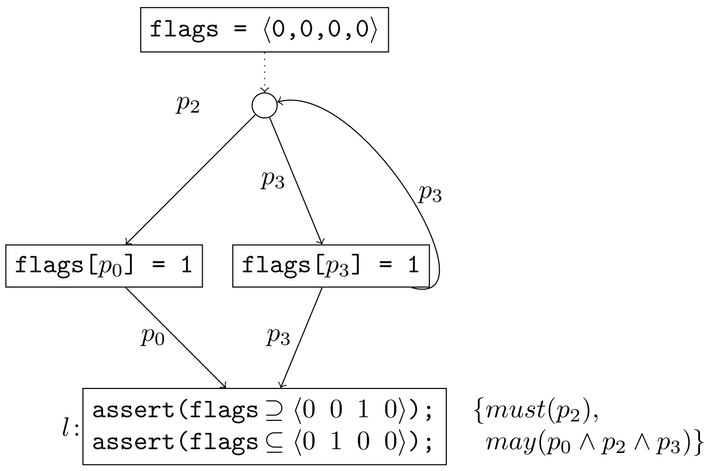


Figure 4.7: Dynamically recomputing some bit-vector problems

for ACC is obviously correlated with the direction of the analysis and type of analyzed information. This is easily explained: When a program is executed, it is always executed in the forward direction, thus restricting the types of properties that can be computed dynamically. Additionally, it also depends on the degree of reflection of the source language. Analyses that compute information that is readily available from within the code are more likely to be first-class candidates for ACC. The following table summarizes the fitness of first-class and second-class candidates for the different usage scenarios of ACC:

Analysis	Testing	Storage	Retrieval
First-class candidate	✓	✓	✓
Second-class candidate	✓	✓	

Assertions for backwards-directed analyses

We already mentioned that backwards-directed analyses inherently are second-class candidates. Backwards-directed analyses compute information that is valid for some (existential) or all (universal) paths leading from the current point to the end of the program. These analyses operate on a reversed control flow graph and yield summaries that contain “predictions” about the (potential) *future* of the program execution. Examples for backward analyses are Very Busy Expressions (universal) and Live Variables (existential). Naturally these properties cannot be asserted at just one single location. Here we can apply the same trick we already used for the results of may-analyses: By introducing flags representing the state of the carrier set, the predictions of the analysis are embedded into the program. The flags are toggled whenever the corresponding analysis result changes. For example, a flag f_i, which corresponds to property p_i in the analysis result, is set to zero whenever $p_i \in \tilde{\sigma}_{post}$ but $p_i \notin \tilde{\sigma}_{pre}$. To verify the correctness of the predictions, assertions are inserted along each path leading from the current location to the exit node. This is visualized in Figure 4.8 where the property p_1 is found to come true somewhere after location l. Therefore

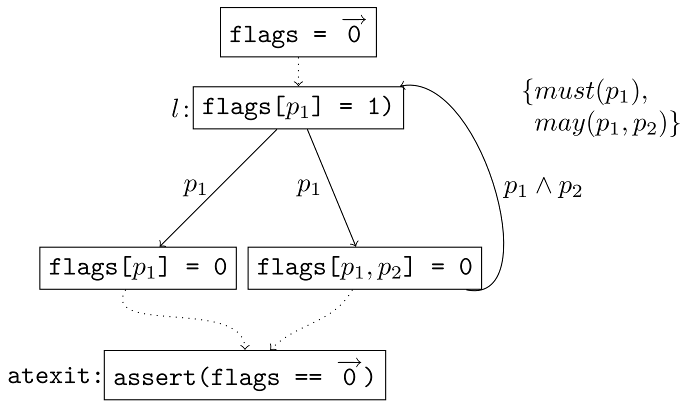


Figure 4.8: Backward-directed data flow problems

the flag for p_1 is set at l. To assert that this prediction was actually true, an assertion at the exit node verifies that the flag for p_1 was really cleared again. Further examples for mapping functions are given in Section 4.3.5, page 47.

4.4 Interprocedural ACC

ACC is particularly powerful when combined with interprocedural analyses. In this section we highlight how to extend ACC inter-procedurally, by taking context information tethered to call strings into account. Intuitively, call strings allow to keep call contexts up to a predefined depth n (= length of the call string) separate [SP81; Muc97]. To account for context-sensitivity in assertion-carrying code, several options are available:

1. Introducing an implicit *call-string* argument to every function call that keeps track of the current context. On machine code level this can be implemented efficiently by using the return address on the stack, similar to the way call-graph-based profiling information is generated [GKM82]. For source-to-source analyzers this has a slight impact on performance and the dynamic memory footprint caused by additional bookkeeping. As an illustrating example, the ACC for an interprocedural interval analysis with call string length 1 and its merged (context-insensitive) counterpart is given in Figure 4.9: Here the current execution context is stored in the global variable `__context` which is updated upon function entry as suggested with the call to `__update_context()`. In the example `CS_F` and `CS_H` are used for representing the call-strings emanating from the calls to function f and h respectively. This particular implementation is not yet thread-safe but serves to illustrate our purpose.

2. Generating explicit copies of the deepest n functions in the expanded call graph. While this transformation is likely to increase execution speed, it has the drawback of increasing the code size significantly.

Both methods involve a slight refactoring of the original program. While a less invasive method would seem preferable, it has to be noted that, because of the more precise analysis information, the modified program also carries an increased potential for subsequent optimizations—this is comparable to inlining.

4.5 The trusted annotation base

In the previous sections we showed how to automatically generate assertion-carrying code (ACC), a technique that can be used to quickly get to a flawless analysis implementation. In this section, we will show how ACC can also be applied to improve the quality of manual annotations.

Manual annotations introduce an obvious risk into the WCET analysis workflow. Not only is it very expensive to use software engineers' hours to perform control flow analysis by hand—it is also inherently error-prone. In fact all annotations that stem from an unknown source have to be trusted as they have the potential to introduce errors into the WCET analysis result. An important goal for us is therefore to shrink this *trusted annotation base* to keep the influence of trusted sources at a controlled minimum. We pursue this goal by transforming the trusted annotation base into a verified annotation base. Formal verification techniques can be used to transform trusted information into verified knowledge. We can often do this because it is often cheaper to verify the correctness of a result than computing it from scratch. Later on, we also discuss how these techniques can be applied to improve the precision of the verified annotations. This section extends on a paper previously published at the WCET'09 workshop [PKK+09].

4.5.1 Lifting environmental information to the program layer

Step zero in shrinking the trusted annotation base is actually to make as much information as possible automatically analyzable. Not only does this save time—it can also serve to increase the reliability of the code by making implicit assumptions explicit. Table 4.2 shows an example of this technique [KKP+08]. If we assume that only the environment dictates that the return value of `read_from_sensor()` is in the interval [0,47], the upper loop bound of 48 would not be found by any analysis. On the left side, the information was thus annotated. Once the information is made part of the program (as shown on the right side of Table 4.2) the loop bound information can be easily inferred by the WCET analyzer. This example shows how to write programs in an analyzable fashion.

4.5.2 Shrinking and verifying the trusted annotation base

The automatic computation of bounds by current approaches to WCET analysis is a step towards keeping the trusted annotation base small. In our approach we go a step further to shrinking the trusted annotation base. In practise, we often succeed to empty it completely.

A key observation is that a user-provided bound—which the automatic analyses were unable to compute—cannot be checked by them either. But this would be necessary in order to move it a posteriori from the trusted annotation base to the verified knowledge base. Hence, verifying the correctness of the

```
int g(int x) {
    int r;

    /*[ACC]*/ __update_context(CS_G); // update call string
    /*[ACC]*/ switch(__context) { // pre-info
    /*[ACC]*/   case CS_F: assert(x >= 0 && x <= 0); break;
    /*[ACC]*/   case CS_H: assert(x >= 42 && x <= 42); break;
    /*[ACC]*/   default:  assert(false);
    /*[ACC]*/ }
    /*[ACC]*/ assert(x >= 0 && x <= 42); // merged pre-info

    r = x + x;

    /*[ACC]*/ switch(__context) { // post-info
    /*[ACC]*/   case CS_F: assert(x >= 0 && x <= 0
    /*[ACC]*/                 && r >= 0 && r <= 0); break;
    /*[ACC]*/   case CS_H: assert(x >= 42 && x <= 42
    /*[ACC]*/                 && r >= 84 && x <= 84); break;
    /*[ACC]*/   default:  assert(false);
    /*[ACC]*/ } // merged post-info
    /*[ACC]*/ assert(x >= 0 && x <= 42 && r >= 0 && r <= 84);

    return r;
}

int f(int x) {
    /*[ACC]*/ __update_context(CS_F); // update call string
    return g(0);
}

int h(int x) {
    /*[ACC]*/ __update_context(CS_H); // update call string
    return g(42);
}
```

Figure 4.9: Calling contexts and merged interval information

before	after
	`int c = read_from_sensor();`
`int c = read_from_sensor();`	`    if (c < 48) {`
`while (c >= 0) {`	`        while (c >= 0) {`
`#pragma` *`wcet_loopbound`*`(0..48)`	`            c--;`
`     c--;`	`            ...`
`     ...`	`    } else error_handler();`
	`}`

Table 4.2: Lifting environmental information

corresponding user annotation requires another, more powerful and usually computationally more costly approach. For example, there are many algorithms for constant propagation, detecting different classes of constants at different costs, such as copy constants, linear constants, simple constants, conditional constants, up to finite constants [Muc97]. This provides evidence for the variety of available choices for analyses using the example of constant propagation. While some of these algorithms might in fact well be able to verify a user annotation, none of these algorithms is especially prepared and suited for solely verifying a data-flow fact at a particularly chosen program location, a so-called data-flow query. This is because these algorithms are exhaustive in nature. They are designed to analyze whole programs. They are not focused towards deciding a data-flow query, which is the domain of *demand-driven program analyses* [DGS97; HRS95]. As for the more expressive variants of constant propagation and folding, however, demand-driven variants of program analyses are often not available.

In our approach, we thus propose to use *model checking* for the *a posteriori* verification of user-provided annotations. Model checking is tailored for the verification of data-flow queries. Moreover, the development of software model checkers made tremendous progress in the past few years and they are now available off-the-shelf, such as Blast [BHJM07] and CBMC [CKL04]. In our experiments reported in Section 7.2 we used CBMC, which was already successfully used to provide loop bounds in the context of measurement-based WCET analysis [RPW08].

4.5.3 Sharpening the time bounds

We now introduce an effective approach to come up with a safe and even tight bound, if existing, which does not even rely on any user interaction. Fundamental for this are the two algorithms *binary tightening* and *binary widening* and their coordinated interaction. The point of this coordination is to make sure that model checking is applied with care as it is computationally expensive.

Binary tightening

Suppose a loop bound has been proven safe, e. g., by verifying a user-provided bound by model checking or by a program analysis. Typically, this bound will not be tight, especially if it was user-provided. Users tend to generously err on the side of caution when manually providing a bound, in order to avoid sacrificing safety. This suggests the following iterative approach to tighten the bound, which is an application of the classical pattern of the binary search algorithm, thus called *binary tightening* in our scenario.

Let b_0 denote the value of the initial bound, which is assumed to be safe. Per definition b_0 is a positive integer. Then we call procedure *binaryTightening* with the interval $[0 \dots b_0]$ as argument, where binaryTightening($[low \dots high]$) is defined as shown in Algorithm 1. Obviously, *binaryTightening* terminates. If it returns *false*, a safe bound tighter than that of the initial bound b_0 could not be established. Otherwise it returns the value b, which is the least safe bound. This means b is tight. If it is smaller than b_0, we succeeded to sharpen the bound.

Note that it is not essential to invoke *binaryTightening* with an interval whose high value has been proven to be safe. Obviously, this does not affect termination of *binaryTightening*. If *binaryTightening* returns a value of m

Algorithm 1 Binary tightening

```
$m \leftarrow \left\lceil \frac{low+high}{2} \right\rceil$
if ModelCheck(Is $m$ a safe bound?) then
  if $low = m$ then
    return $m$
  if $low = m - 1$ then
    if ModelCheck(Is $low$ a safe bound?) then
      return $low$
    else
      return $m$
  else
    return binaryTightening($[low \ldots m]$)
else
  if $high = m$ then
    return $false$
  if $high = m + 1$ then
    if ModelCheck(Is $high$ a safe bound?) then
      return $high$
    else
      return $false$
  else
    return binaryTightening($[m \ldots high]$)
```

smaller than the initial value of $high$, then this value is the tight safe bound of this loop relative to the precision of the model checker that is used. If it returns $false$, then the initial value of $high$ finally has to be model checked for safety. In the positive case, it is the tight safe bound. In the negative case, it is not a safe bound at all and must be removed from the annotation base.

Binary widening

We will now describe how to proceed if a safe bound is not known a priori. If a safe bound (of reasonable size) exists, *binary widening* will find one, without any further user interaction.

Binary widening is dual to binary tightening. It is operating like a risk-aware roulette player, who exclusively bets on 50% chances like red and black. Following this strategy, in principle, any loss can be flattened by doubling the bet the next game. In reality, the maximum bet allowed by the casino or the limited monetary resources of the gambler, whichever is lower, prevent this strategy from working in reality. Nonetheless, such an externally given limit is also what prevents the binary widening algorithm from avoid looping if no safe bound exists: The implementation shown in Algorithm 2 comes up with a safe bound, if one exists, and terminates, if the size of the bound is too big to induce a useful WCET bound, or does not exist at all. This directly corresponds to the limit set by a casino for a maximum bet.

Let b_0 be an arbitrary number, $b_0 \geq 1$, and let *limit* be the maximum value for a safe bound considered reasonable. Then we call procedure binaryWidening with

Algorithm 2 Binary widening

```
if $b > limit$ then
  return $false$
else
  if ModelCheck(is $b$ a safe bound?) then
    return $b$
  else
    return binaryWidening($2 \cdot b$, $limit$)
```

b_0 and *limit* as arguments, where binaryWidening(b, *limit*) is defined as shown in Algorithm 2. Obviously, $binaryWidening$ terminates.[2] If it returns *false*, at most an unreasonably large bound exists, if at all. Otherwise, it returns value b, which is a safe bound. The rationale behind this approach is the following: If a safe bound exists, but exceeds a predefined threshold, it can be considered practically useless. In fact, this scenario might indicate a programming error and should thus be reported to the programmer for inspection. A more refined approach might set this threshold in a more sophisticated way, by using application dependent information, such as a coarse estimate of the execution time of a single execution of the loop and a limit on the overall execution time budgeted for this loop.

Coordinating binary widening and tightening.

Once a safe bound has been determined using binary widening, binary tightening can be used to compute the uniquely determined safe and tight bound. Because of the exponential or rather logarithmic behaviour in the growth of the arguments for binary widening and tightening, model checking is called moderately often. This is the key for the practicality of our approach, which we implemented within TuBound (cf. Chapter 2). The results of practical experiments we conducted with the prototype implementation are promising. They are reported in Section 7.2.

4.6 Implementation details

The interfacing with the model checker necessary for the binary widening/tightening algorithms is implemented by means of a dedicated Termite source-to-source transformer (cf. Section 6.2). This requires the assertion-carrying code transformation described in Section 4.3 to translate user-provided annotations into `assert()` statements. On while-loops the transformer works by locating the first occurrence of a `wcet_trusted_loopbound(Lower..Upper)` annotation in the program source and then proceeds to rewrite the encompassing loop as illustrated in the example of Figures 4.11 and 4.12. An excerpt of the source-to-source transformer is given in Figure 4.10.[3] For simplicity and uniformity we assume that all loops are structured. In our implementation unstructured goto loops are thus transformed into while loops where possible. This is done in a pre-pass by

[2]In practise, the model checker might run out of memory before verifying a bound, if it is too large, or may take too much time for completing the check.

[3]For better readability, the extra arguments containing file location and other bookkeeping information are replaced by "…".

```
assertions(..., Statement, AssertedStatement) :-
    Statement = while_stmt(Test, basic_block(Stmts, ...), ...),
    get_annot(Stmts, wcet_trusted_loopbound(Lower..Upper), _),

    var_decl(unsigned_int, '_bound', 0, ..., CounterDecl),
    var_ref_exp('_bound', unsigned_int, Counter),
    plusplus_expr_stmt(Counter, ..., Count),
    build_expr_stmt(
      assert((Counter =< Upper) && (Counter >= Lower)),
                   ..., CounterAssert),

    AssertedStatement =
      basic_block([CounterDecl,
                   while_stmt(Test, basic_block([Count|Stmts], ...),
                        ...),
                   CounterAssert], ...).
```

Figure 4.10: Excerpt from the ACC generator

another Termite transformer. If such a transformation is not possible, the loop cannot be analyzed.

Surrounding the loop statement, a new compound statement is generated, which accommodates the declaration of a new unsigned counter variable which is initialized to zero upon entering the loop. Inside the loop, an increment statement of the counter is inserted at the very first location. After the loop, an assertion is generated which states that the count is at most of value N, where this value is taken from the annotation.

The application of the transformer is controlled by a driver, which calls the transformer for every trusted annotation contained in the source code. Depending on the result of the model checker and the coordinated application of the algorithms for binary widening and tightening, the value and the status of each annotation is updated. In the positive case, this means the status is changed from *trusted annotation* to *verified knowledge*, and the value of the originally trusted bound is replaced by the now verified, possibly sharper, bound. Figure 4.12 shows a snapshot of processing the *janne_complex* benchmark. In this figure, the outer loop was already tightened to a smaller interval (line 4), whereas the inner loop is prepared for another run of the model checker, by introducing the new counter variable `_bound` (lines 9, 20–21).

```
...
int complex(int a, int b)
{
   while(a < 30) {
#pragma wcet_trusted_loopbound(0..30)
      while(b < a) {
#pragma wcet_trusted_loopbound(0..30)
         if (b > 5)
            b = b * 3;
         else
            b = b + 2;
         if (b >= 10 && b <= 12)
            a = a + 10;
         else
            a = a + 1;
      }
      a = a + 2;
      b = b - 10;
   }
   return 1;
}
...
```

Figure 4.11: Code containing two trusted loop annotations

```
...
int complex(int a, int b)
{
   while(a < 30) {
#pragma wcet_loopbound(0..16)
       {
           unsigned int _bound = 0;
           while(b < a){
#pragma wcet_trusted_loopbound(0..30)
              ++_bound;

             if (b > 5)
                 b = b * 3;
             else
                 b = b + 2;
             if (b >= 10 && b <= 12)
                 a = a + 10;
             else
                 a = a + 1;
         }
         assert(_bound >= 0
             && _bound <= 30U);
       }
       a = a + 2;
       b = b - 10;
   }
   return 1;
}
...
```

Figure 4.12: Outer loop annotation verified and tightened, inner loop transformed for checking

Chapter 5

Transforming flow constraints

As indicated in Chapter 1, compiler optimizations are root to several problematic issues when it comes to timing-critical embedded systems:

First there is the issue of the optimization target. An off-the-shelf compiler generally offers only one of two options: Optimize for the (average) execution speed or optimize for the minimal size of the executable. For a real-time system, the average-case performance is not the main point of interest—although it contributes to making the system more energy efficient. The targeted optimization goal should obviously be the worst-case performance. Sadly, there are only few compilers that offer this [LM09].

The second issue is with annotations. We already established that not only annotation support is generally necessary, but also that the source code is a more suitable target to perform automatic control flow analysis on. The gathered information (be it manually or automatically) now needs to be transformed alongside the code and should be available at the machine code level without any loss of correctness and, when possible, without any degradation in quality.

In this chapter, we first investigate compiler optimizations and their effect on flow information. Then we continue to design a language to concisely describe how to transform annotations. Finally, we will show how to apply this language to describe flow information updates for several types of loop optimizations. In Chapter 7, results from the Mälardalen and DSPstone benchmark suites are shown. Research results related to this chapter are published in the Journal on Real-Time Systems [KPP10].

5.1 Classification of optimizations

Depending on the types of optimizations that are performed by the compiler, different requirements are imposed on the flow constraint transformer. To illustrate this, we categorize the transformations by their effects on the control flow graph (CFG) and the annotation semantics. In general, we can distinguish three kinds of transformations, classifying whether they always/never/sometimes modify the shape of the control flow graph and affect other aspects of the annotated information.

	Affects		
Optimization	CFG	Labels	Constraints
Loop Unrolling	maybe	no	yes
Loop Blocking	yes	yes	yes
Loop Fusion	yes	yes	yes
Loop Interchange	no*	yes	yes

(*Remember that in this case the shape of the graph remains the same—only the contents of the nodes changes)

Table 5.1: Classification of common program transformations

These other aspects can be broken down in two categories: The first one concerns the location of the annotation. If a program transformation alters the CFG, e. g., by fusing two loops of identical iteration space into a single new loop, it is necessary to update the location of the annotations. In a source-based annotation language this could be achieved by physically moving them to the newly created loop.

The other aspect concerns the contents of the annotations: If a loop is unrolled by a factor that is a integer divisor of the iteration count, the locations of the basic blocks in the CFG will not necessarily change, but the *information* described by the contained annotations has to be adjusted accordingly.

5.2 Defining transformation rules

Recalling Section 2.5, there are four types of annotations available in TuBound: Loop bounds, markers, constraints and scopes. Of these, however, only markers and constraints are canonical. Loop bounds exist mostly for convenience purposes and are syntactic sugar for a constraint-marker combination.[1] The same considerations apply for scopes as well, which again are syntactic sugar for an (albeit longer) marker-constraint combination (cf. Section 2.5). For this reason we will mostly deal with constraints and markers in this section. We also restrict our considerations to *linear* flow constraints.

In the annotation language, markers are associated with flow. In the control flow graph (CFG), flow can either be associated with nodes or edges. In compiler literature nodes are usually holding the instructions, whereas program analyzers often use control flow automata where nodes denote the state of the machine between executing instructions. In the following we will use a representation where nodes are instructions and flow is associated with edges. The constraint system consists of flow variables f_e—one for each edge e—that are described by a set of inequalities. To slim down this representation it is sufficient to generate only one flow variable per basic block, since sequences of instructions (basic blocks) always share the same flow. Markers are translated into flow variables via their name (cf. Section 5.4.2). In its current form the TuBound annotation language uses the location of a marker in the source code to reference basic

[1]A loop bound (l, u) can be translated into a marker *entry* (placed before the loop entry), a marker *body* (placed at the top of the loop body) and a constraint $\langle l{\cdot}entry \leq body \leq u{\cdot}entry\rangle$.

blocks. In the future, edge-markers like, for instance, loop back edges could be supported by introducing reserved names.

In this chapter we define rules that describe how to transform annotations consisting of loop bounds, markers and constraint terms according to an optimization trace. We start by introducing the vocabulary needed in these rules: Loop bounds can be transformed simply by updating the lower and upper bound. For markers every necessary operation can be expressed by either renaming, deleting, or introducing new markers. For constraints we additionally need more fine-grained operations, as discussed below.

Control-flow-altering compiler optimizations delete and insert edges (and nodes) in the transformed CFG (henceforth called CFG$'$). Since these optimizations need to preserve the semantics of the original program, the total flow of the program (on a per-basic-block level) typically remains the same. However, it will be differently distributed. For our flow constraint transformation rules we therefore need to deal with two scenarios [KPP10]:

1. The flow f_e at an edge e can be split into flow at multiple edges e'_i in the optimized program. In this case each occurrence $\langle n \cdot f_e \rangle$ of a linearly scaled marker (=flow variable) in a constraint expression is transformed into the sum of multiple scaled variables f'_{e_i}.

$$\langle n \cdot f_e \rangle \quad \longrightarrow \quad \langle n_1 \cdot f_{e'_1} \rangle + \langle n_2 \cdot f_{e'_2} \rangle + \dots$$

2. Conversely, the control flow of several edges e_i can be merged into one edge e', which implies that we need one transformation for each edge e_i,

$$\langle n \cdot f_{e_i} \rangle \quad \longrightarrow \quad \langle [n_{\text{lhs}} \dots n_{\text{rhs}}] \cdot f_{e'} \rangle$$

   where n_{lhs} is used on the left-hand-side of a $\leq$ or $<$ constraint and n_{rhs} is used on the right-hand-side and vice versa with $\geq, >$.

With the help of these operations (typically combinations thereof) we can model all types of linear flow constraint transformations. A transformation is called *safe* if each flow variable f_e is transformed into a new set of flow variables $f_{e'_i}$,

$$\langle n \cdot f_e \rangle \quad \longrightarrow \quad \sum_{e'_i \in \text{CFG}'} [n_{\text{lhs}} \dots n_{\text{rhs}}] \cdot f_{e'_i}$$

such that

$$\sum_{e'_i \in \text{CFG}'} n_{\text{lhs}} \cdot f_{e'_i} \;\leq\; n \cdot f_e \;\leq\; \sum_{e'_i \in \text{CFG}'} n_{\text{rhs}} \cdot f_{e'_i}$$

□

The term

$$\sum_{e'_i \in \text{CFG}'} |n_{\text{lhs}} - n_{rhs}| \cdot f_{e'_i}$$

describes the precision of a transformation rule.

Original program	Loop-interchanged program′

`for (i = 0; i < 8; ++i) {`
`  // l1`
`  for (j = 0; j < n; ++j) {`
`    // l2`
`    if (even(i))`
`      // then`
`      ...`
`    else ...`
	`for (j = 0; j < n; ++j) {`
	`  // l1`
	`  for (i = 0; i < 8; ++i) {`
	`    // l2`
	`    if (even(i))`
	`      // then`
	`      ...`
	`    else ...`

Loop bounds	Loop bounds′
$\langle l_1, 8 \dots 8\rangle$	$\langle l_1, 1 \dots 4\rangle$
$\langle l_2, 1 \dots 4\rangle$	$\langle l_2, 8 \dots 8\rangle$

Constraints	Constraints′
$f_{l_1} \leq 8$	$f_{l_2}/4 \leq 8$
$f_{\text{then}} \leq f_{l_1} \cdot 2$	$f_{\text{then}} \leq f_{l_2}/1 \cdot 2$
$f_{l_1} \leq f_{\text{then}} \cdot 2$	$f_{l_2}/4 \leq f_{\text{then}} \cdot 2$

Figure 5.1: Example: Loop interchange

Example. Loop interchange is a loop optimization that alters the iteration pattern of two nested loops. Figure 5.1 shows the effect of interchanging the loops l_1 and l_2 in an example program. The flow transformation rules for loop interchange are (cf. Section 5.4.1, and Figure 5.3) defined as follows: The loop bounds $(lo_1 \dots up_1)$ for $l1$ and $(lo_2 \dots up_2)$ for $l2$ are interchanged:

$$\langle l_1, lo_1 \dots up_1\rangle \quad \longrightarrow \quad \langle l_2, lo_1 \dots up_1\rangle$$

$$\langle l_2, lo_2 \dots up_2\rangle \quad \longrightarrow \quad \langle l_1, lo_2 \dots up_2\rangle$$

The flow variable referring to the outer loop body needs to be transformed as well:

$$\langle n \cdot f_{l1}\rangle \quad \longrightarrow \quad \langle [1/up_2 \dots 1/lo_2] \cdot f_{l2}\rangle$$

In the above rule, the flow variable referencing the outer loop body is replaced by one referencing the inner loop body. Since the inner loop body is executed more often than the original outer loop body, the flow variable is scaled by the lower or upper loop bound of the new outer loop, respectively. This is highlighted with colour in Figure 5.1. Since the flow of the innermost loop body remains the same after the interchange, it is not necessary to transform it.

The effect of these rules on actual flow constraints is shown in the lower part of Figure 5.1. Loop bounds are given as pairs of loop name and lower... upper bound. The first constraint describes that the outer loop body is executed up to 8 times. The second and third constraints describe that the *then*-block is executed exactly half as many times as l_2 is entered. Since the flow transformation rules are not precise, this information is weakened by the transformation. Nevertheless, the information is safe.

5.3 Designing a new work flow

Earlier proof-of-concept implementations of flow constraint transformation frameworks were performed in the following compiler systems:

1. The earliest implementation we are aware of is the *co-transformer*, which uses optimization traces generated by the compiler to transform the flow information separately from the program [EEA98]. This is facilitated by an optimization description language (ODL) that describes the code optimizations performed by the compiler.

2. A second implementation was done inside the GNU C Compiler (GCC) version 2.7.2 [Kir03]. This implementation could only show some basic capabilities of a flow constraint transformation framework, since at the time, GCC 2.7.2 implemented only a small number of code transformations that would change the control flow of the compiled program.

3. In more recent work, flow constraint transformation was also integrated into the Wcc compiler, which is also able to perform WCET-aware code optimizations [Sch07b]. The Wcc compiler implements transformations for annotations supported by the WCET analysis tool aiT.

The last two of the above implementations tightly integrate the handling and transformation of flow information with the compiler. Such an approach involves a deep understanding of the compiler internals and is only feasible when it is integrated with the main development branch of the compiler. Our experiences with the GCC implementation showed that maintaining the handling of flow constraints as an add-on patch can be very time-consuming. These complications inspired us to the development of a more portable solution, thereby taking the concept of the co-transformer to a higher level: Considering that not every optimizing transformation alters the control flow graph (think of local optimizations like expression rewriting), the optimizations can be divided into two groups, control-flow-invariant and control-flow-modifying. A majority of the control-flow-modifying transformations are loop optimizations [Kir03], which can be implemented effectively as source-to-source transformations. This approach is traditionally taken by Fortran compilers [AK02], but also by recent versions of GCC, which are using a near-source internal representation to perform high-level loop optimizations [PCB+06]. By carrying out control-flow-modifying optimizations as source-to-source transformations, the subsequent target compiler needs only to perform a direct translation to machine code, leaving the control flow intact. It is still safe to apply control-flow-invariant transformations in the target compiler.[2]

The prototypical work-flow of the high-level source-to-source compiler is shown in Figure 5.2 [Pra07b]: In the first step [1. Unweave], annotations and source code are separated. Each annotation is associated with a unique label that identifies its location in the source code. This way the optimizer need not be aware of the placement or contents of annotations. In the second step [2. Optimize], the source-to-source optimization is performed. The optimizer needs

[2]In current compiler systems like GCC or LLVM this can be achieved by starting with the `-O0` optimization level and manually enabling safe transformations using the respective command line flags.

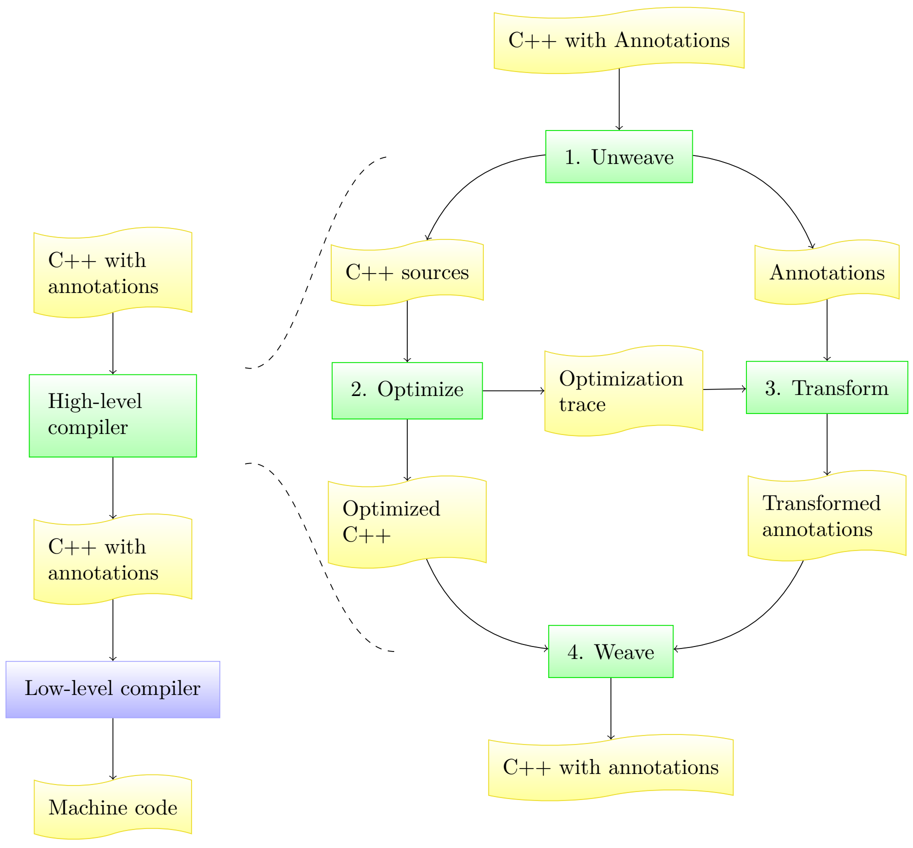


Figure 5.2: Workflow of the source-to-source high-level compiler

to generate a trace of the performed program transformations. The optimization trace together with the original annotations is the input for the flow information transformation engine [3. Transform]. This engine contains a rule base describing the flow constraint update for each type of optimization. These rules need to be specified only once for each program transformation that is implemented in the optimizer. In the final step, the transformed annotations are merged with the transformed source code [4. Weave].

5.4 Implementation in TuBound

The TuBound WCET analysis tool contains an implementation of this source-to-source work flow, based on a C++ port of the Fortran D loop optimizer included with the Rose [QSMK04] compiler. Even though many of the performed high-level optimizations target the average-case execution time instead of the worst case, our measurements indicate a positive effect also on the analyzed worst-case performance of the optimized programs [Pra07a; PSK08; KPP10]. A detailed account on these benchmarks is given in Section 7.3.

5.4.1 Transformation rules examples

While the source-to-source infrastructure is targeting a subset of C++, the low-level compiler and rest of the tool chain is currently restricted to C as input language. The analysis results (loop bounds, flow constraints) found by TuBound are annotated into the source code as `#pragma` directives. The concrete syntax of these annotations is designed such that each annotation is also a legal Prolog term. This makes it possible to use Prolog as specification language for the flow information update rules.

Figure 5.3 shows the rules for *loop interchange*: Processing the optimizations in the trace, these rules are applied to each annotation. The first two arguments contain information from the optimization trace (the labels of the interchanged loops `Loop1` and `Loop2`). The third argument is the list of all annotations before the transformation. This is followed by the annotation to be transformed: An annotation is associated with a location (a marker) and a body: Valid bodies are loop bounds, constraints and marker names. Markers are labels of basic blocks or specific edges in the CFG and follow a unique hierarchical naming scheme that encodes the location in the abstract syntax tree of the program. The last argument is unified with a list of annotations generated by the transformation rule. The first two clauses in Figure 5.3 swap the loop bounds of the interchanged loops. The third clause updates flow constraints The helper predicate `replace` is used to replace all occurrences of `Loop1` on the left-hand-side with `Loop2` divided by its lower loop bound and vice versa. After the rules are applied, the resulting constraints are normalized to remove the division operator and allow further processing by other tools.

Figure 5.4 shows an excerpt of the rules we implemented for loop unrolling. The rules in this example are parametrized with the label of the unrolled loop body and the unroll factor. They are written in Prolog and are applied to each annotation of the original program. The first rule updates the loop bounds according to the unroll factor k, whereas the second rule clones constraints for each unrolled basic block in the loop body. The program on the right of

```
% loop interchange
% -----------------
% interchanged(+Loop1, +Loop2, +Annotations, +OldAnnotation,
%              -NewAnnotations)
%
interchanged(Loop1, Loop2, _,
             annotation(Loop1, wcet_loopbound(Lo..Up)),
            [annotation(Loop2, wcet_loopbound(Lo..Up))]).

interchanged(Loop1, Loop2, _,
             annotation(Loop2, wcet_loopbound(Lo..Up)),
            [annotation(Loop1, wcet_loopbound(Lo..Up))]).

interchanged(Loop1, Loop2, Annotations,
             annotation(M, wcet_constraint(Lhs=<Rhs)),
            [annotation(M, wcet_constraint(Lhs1=<Rhs1))]) :-
    member(annotation(Loop2, wcet_loopbound(Lo..Up)),
           Annotations),
    replace(Lhs, Loop1, Loop2/Up, Lhs1),
    replace(Rhs, Loop1, Loop2/Lo, Rhs1).
```

Figure 5.3: Flow transformation specification for loop interchange

Table 5.2 shows the effect of applying these rules to the program on the left: In the example, the for loop is unrolled with a factor 2, which means that the control flow is now equally distributed over the two copies of the original loop body. This is reflected in the updated constraints (note that the original marker names have been replaced by generic ones), where the marker `m_if` is replaced by the sum of the two new if-blocks `f_1_1_1` and `f_1_1_2`, whereas the right-hand side `m_for`/4 was replaced by (`f__1__1` $\cdot$ 2)/4, which was then simplified to `f_1_1`/2. This last replacement was necessary because the unrolled `for` loop is executed half as often as the original loop. Finally, the loop bound of the unrolled loop was updated to $48/2 = 24$.

Figure 5.5 shows the only rule necessary to implement loop fusion. The idea is that annotations referencing the second loop, `M_fused`, are merged into the first and only loop remaining after the optimization, `M_orig`.

5.4.2 A hierarchical naming scheme for AST scopes

When annotations are sent through the transformation rules, they are detached from the source code. The mapping from annotations to source code locations is achieved by a hierarchical naming system. This way, we can even refer to nodes that are not present in the original program. The name of a node is derived from its location in the AST in the following way: From the function root-node, which by convention is called like the function name, we concatenate the number of each child scope starting with 1 and separated by an underscore. As an example, Figure 5.6 shows a function and the derived names for each scope.

```
% loop unrolling
% --------------
unrolled(M, K, _, annotation(M, wcet_loopbound(Lo..Up)),
               [annotation(M, wcet_loopbound(Lo1..Up1))]) :-
   Lo1 is floor(Lo/K),
   Up1 is ceiling(Up/K).

unrolled(M_Loop, K, _, annotation(M_Annot, wcet_constraint(Term)),

                    NewAnnots) :-
   replace(Term, M_Loop, M_Loop*K, Term1),
   (  nested_in(M_Annot, M_Loop)
   -> list_from_to(1, K, Ns),
      M_AnnotBase = M_Annot,
      maplist(unroll_clone(M_AnnotBase, M_Annot, Term1, Ns), Ns,
           NewAnnots) ;
     NewAnnots = [annotation(M_Annot, wcet_constraint(Term1))]).

unroll_clone(M_AnnotBase, M_Annot, ConstrTerm, Ns, N, NewAnnot) :-

   atomic_list_concat([M_AnnotBase, '_', N], M_clone), % Pos. of
       Constr
   sumterm(M_AnnotBase, Ns, M_sum), % New ConstrTerm
   replace(ConstrTerm, M_Annot, M_sum, ConstrTerm1),
   NewAnnot = annotation(M_clone, wcet_constraint(ConstrTerm1)).
```

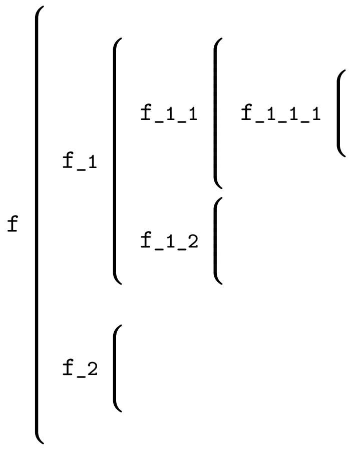
Figure 5.4: Example from the transformation rules: loop unrolling

```
% loop fusion
% --------------
fused(M_orig, M_fused, _, annotation(M_orig, Annot),
                        [annotation(M_fused, Annot)]).
```

Figure 5.5: Example from the transformation rules: loop fusion

Original user-annotated program	After 2× loop unrolling
	`int *f(int *a)`
	`{`
	`  int i;`
`int* f(int* a)`	`  for (i = 0; i <= 47; i += 2) {`
`{`	`#pragma wcet_marker(f_1_1)`
`  int i;`	`#pragma wcet_loopbound(24..24)`
`#pragma wcet_marker(m_func)`	`    if ((test(a[i]))) {`
`  for (i = 0; i < 48; i += 1) {`	`#pragma wcet_marker(f_1_1_1)`
`#pragma wcet_loopbound(48..48)`	`#pragma wcet_constraint(`
`#pragma wcet_marker(m_for)`	`  f_1_1_1+f_1_1_2=<f_1_1/2)`
`    if (test(a[i])) {`	`      a[i]++;`
`#pragma wcet_marker(m_if)`	`    }`
`#pragma wcet_constraint(`	`    if ((test(a[1 + i]))) {`
`  m_if=<m_for/4)`	`#pragma wcet_marker(f_1_1_2)`
`      a[i]++;`	`#pragma wcet_constraint(`
`    }`	`  f_1_1_1+f_1_1_2=<f_1_1/2)`
`  }`	`      a[1 + i]++;`
`  return a;`	`    }`
`}`	`  }`
	`  return a;`
	`}`

Table 5.2: Flow annotations before and after loop unrolling



```
int f() {
    if (A) {
        while (B) {
            // f_1_1_1
        }
    }
    else {
        // f_1_2
    }
    // f
    if (C) {
        // f_2
    }
}
```

Figure 5.6: The node naming scheme in action

5.4.3 Extensions

The current implementation of the flow constraint transformation rules allows only markers and integer values as part of flow constraints. In the future this could be extended to allow other kinds of symbolic expressions, such as function parameters or variable values, to be used within flow constraints. This way, it would become possible to derive parametric WCET bounds [AAG$^+$07].

Currently we use a notation where flow variables represent the global flow of a program location. To get the most benefit from inter-procedural flow information, the syntax of the flow information has to be extended to support call contexts [KKP$^+$08]. Typical inter-procedural code optimizations that require the update of inter-procedural flow information are *procedure cloning* [LFMT08] or *procedure inlining*.

Chapter 6

The implementation environment

6.1 SATIrE

All of the algorithms presented in this thesis are implemented in the SATIrE source-to-source program analysis framework. The SATIrE framework started as a connection of the Rose compiler and the Program Analyzer Generator (PAG) and is developed at Vienna University of Technology. Rose is a C++ to C++ compiler developed at Lawrence Livermore National Laboratory that provides a C++ object representation of the abstract syntax tree and a line-and-column precise unparser [Law10]. PAG is developed at AbsInt GmbH and takes a specification for a program analyzer and generates C code that performs the fixed-point search [Abs10b]. The specification uses a special-purpose functional language called OPTLA[1]. The main contribution of SATIrE is the construction of an interprocedural control flow graph (ICFG) which is achieved by connecting the nodes in the abstract syntax tree (AST) provided by Rose. Since early in 2009, SATIrE was extended by other components such as Termite (described in Section 6.2). Now, many of the analyses written with SATIrE (such as the interval and loop bound analyses) are also distributed together with it in the same package.

To get a better grasp of what is available under the SATIrE hood, we will now dissect TuBound into its components and describe which parts of SATIrE they are using. Recalling the structure of TuBound shown in Figure 2.2 on page 19, every component that operates on the high-level language representation is connected to SATIrE.

SATIrE's points-to analysis is written in C++ and operates directly on the ICFG constructed by SATIrE. The interval analysis is written in FuLa; SATIrE is responsible for connecting the ICFG with the analyzer generated by PAG. The information from the points-to analysis is available via the foreign function interface in FuLa. All results from the interval analysis are stored in the ICFG, using Rose's attribute mechanism. In a typical automated TuBound session, the AST is then exported in the Termite format. This format, which retains all

[1]OPTLA again is based on the FuLA functional language.

PAG analysis information and also condensed points-to information, is described next.

6.2 Termite

The TERM Iteration and Transformation Environment (Termite) is a Prolog library that allows easy manipulation and analysis of C++ programs [Vie10c]. It is particularly well suited for specifying source-to-source program transformations, static program analyses and program visualizations. Termite builds upon the intermediate representation of SATIrE.

6.2.1 The Termite term representation

To fully support the work flow, we extended SATIrE to export an external term representation of the abstract syntax tree (AST) of a C++ program. This term representation contains all information that is necessary to correctly unparse the program, including line and column information of every expression. The terms are also annotated with the results of any preceding PAG analysis. The syntax of the term representation is designed to match the syntax of Prolog terms. This allows to very naturally manipulate it by Prolog programs.

Depending on the desired architecture there are several ways to integrate Termite into the work flow of a larger tool chain. For a flexible recombination of several analyses and/or transformations it is best to treat Termite programs as interpreted scripts that read/write AST terms from the standard input and output. If performance and stability are sought for, it is also possible to call Termite programs transparently from a SATIrE analyzer. A complete overview of all combinations possible is shown in Figure 6.1.

6.2.2 Using Termite for a standalone process

The most flexible and convenient way to work with the Termite library is by using it to define filter operations on streams of source code. This way one can follow the UNIX tradition of having a collection of small self-contained programs that can be combined to create larger work flows. Depending on the expected input and generated output, several types of Termite programs can be distinguished. Typical examples are:

A source-to-source transformer is a program that reads in an AST, then performs some transformation and outputs the transformed AST. (Example: loop unrolling)

An analyzer is a program that reads in an AST, performs some analysis and outputs the analysis result as attributes of the AST. (Example: loop bound analysis)

A visualization is a program that reads in an AST and outputs a visualization, e. g., in a GUI window or a PostScript file. (Example: Call-graph → Graphviz (DOT))

A source generator is a program that reads in a specification and outputs an AST in termite format.

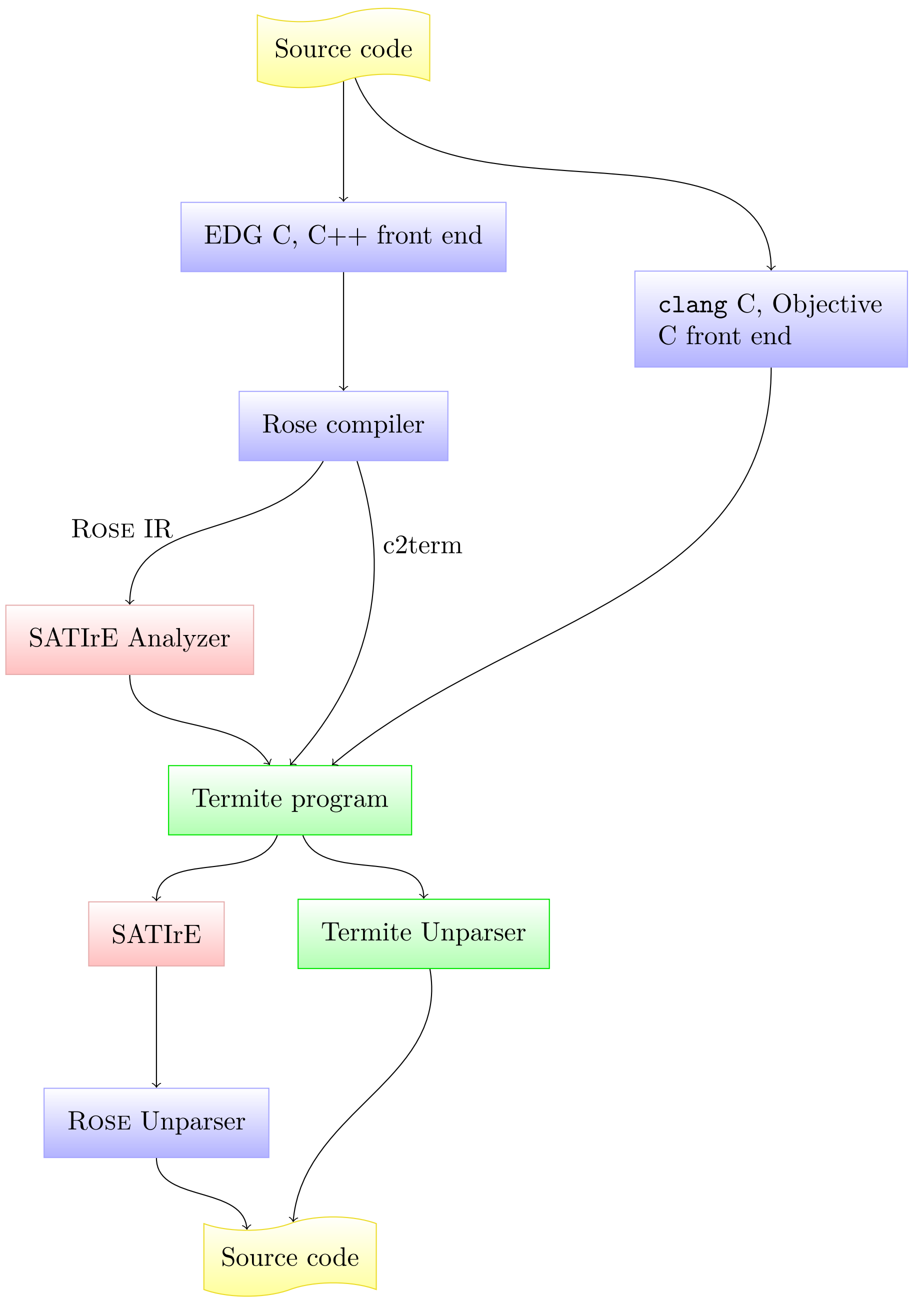


Figure 6.1: Termite usage scenarios

A compiler is a program that translates an AST into a different language, e. g., `melmac` [Bár10] or wcetcc (cf. Section 6.3).

In order to generate a Termite term from one or more source files a compiler front end must be invoked. Two possibilities are supported and available in the SATIrE distribution: The commercial EDG front end, and the open-source `clang` front end. Next, we discuss the differences between the two.

EDG C/C++ front end from the Rose compiler

If SATIrE is configured with the Rose connection enabled[2], conversion tools are available to translate source code to term files and vice versa. To translate source code into a Termite term we developed the `c2term` program:

> c2term

```
Usage:
  c2term [FRONTEND OPTIONS] [--dot] [--pdf] src1.c src2.cpp ...
         [-o termfile.pl]
  Parse one or more source files and convert them into a
  TERMITE file.  Header files will be included in the term
  representation.

Options:
  [FRONTEND OPTIONS] will be passed to the C/C++ frontend.

  -o, --output <termfile.pl>
    Write the output to <termifile.pl> instead of stdout.

  --dot
    Create a dotty graph of the syntax tree.

  --pdf
    Create a PDF printout of the syntax tree.

This program was built against SATIrE 0.8.6-rc2,
please report bugs to adrian@complang.tuwien.ac.at.
```

The `c2term` program invokes the commercial EDG C++ front end embedded into the Rose compiler to parse one or more source files. The abstract syntax tree is then translated into the Rose intermediate representation which in turn is converted into the textual term serialization. The program passes additional options to the EDG front end.

The reverse direction is managed by the `term2c` conversion utility. It works by reading in a term file and then rebuilding the Rose intermediate representation. Finally, this data structure is passed to the Rose unparser. The EDG front end is not involved in this step any more.

[2]Rose must be installed separately beforehand from `http://www.rosecompiler.org`

> term2c

```
Usage: term2c [OPTION]... [FILE.term]
Unparse a term file to its original source representation.

Options:
  -o, --output sourcefile.c
    If specified, the contents of all files will be concatenated
    into the sourcefile.

  -s, --suffix '.suffix'  Default: '.unparsed'
    Use the original file names with the additional suffix.

  -d, --dir DIRECTORY
    Create the unparsed files in DIRECTORY.

  --dot
    Create a dotty graph of the syntax tree.

  --pdf
    Create a PDF printout of the syntax tree.

This program was built against SATIrE 0.8.6-rc2,
please report bugs to adrian@complang.tuwien.ac.at.
```

Since both converters use standard input and output per default it is possible to concatenate multiple Termite programs with the help of UNIX pipes. This way it is possible to build new chains of program transformations or analyzers on the fly without having to recompile the whole project.

Example:

```
c2term a.c b.c | ./transform1.pl | term2c -s '.transformed'
```

In this example pipeline, two C source file are joined into one project which is dumped to a stream in the Termite format. The stream is then transformed by a Prolog program. Finally the two source files are unparsed by the `term2c` converter with the new suffix ".transformed" attached to the file names.

Using the `clang` C/Objective C front end

While the commercial EDG front end offers a high-quality C++ parser, license restrictions encumber its free distribution together with other tools. Most notably, the ROSE compiler redistributes only a 32-bit precompiled binary version of the EDG front end. It is, however, possible to buy other licenses from the Edison Design Group.

If C++ support is not needed, there is a free alternative available from the LLVM compiler project. Designed especially for use with LLVM a front end for C-like languages called `clang` is published under a BSD-style license. The `clang` front end can be downloaded at `http://clang.llvm.org/`. The front end is written in C++ and creates an intermediate representation very similar to that of Rose and therefore makes it a good candidate to replace the EDG front end in SATIrE. The C99 and Objective C languages are supported very well by `clang`. Support for C++ is in progress; at the time of writing `clang`, which is written in a subset of C++, was already able to compile itself.

In order to connect SATIrE with the `clang` front end, we decided to take the route via the Termite representation. This way, the front end is cleanly decoupled from the rest of the system and uses the Termite terms as a stable interface. The Termite term generator is implemented as a pass over the `clang` intermediate representation and is available via the `-emit-term` command line option. The term generator is not integrated with upstream `clang`, but distributed as a patch against a current SVN version together with SATIrE.

To build the `clang` front end for use with SATIrE, a special `make clang` target is available at the top level which fetches the needed version of `clang` from the subversion repository, applies the patch, and compiles and installs the patched front end to `$prefix/bin`.

Unparsing Termite terms without SATIrE

The `term2c` always writes its output to one or several files, using the file names embedded in the `file_info` nodes in the Termite representation. This is sometimes too heavyweight, for instance, when only a few expressions should be unparsed for debugging purposes. For these occasions an independent term → C converter is implemented in pure Prolog and available both in the Termite library and as a stand-alone script. The predicate is called `unparse/1` and expects a Termite term as argument.

6.2.3 Invoking Termite as part of a SATIrE analyzer

If execution speed is an issue, the steps of writing the Termite representation to disk (or a pipe) and parsing the terms (which, when output as a text, are significantly larger than the original source files) can be optimized away. If SATIrE is configured with SWI-Prolog support enabled, the term representation will be built in memory using the external interface of an embedded SWI-Prolog interpreter. Using this in-memory term, a Termite program can be executed without leaving the current process. The resulting term can again be translated to the Rose intermediate representation directly from memory using the SWI-Prolog interface.

Using this work flow, the whole analyzer (or transformer, ...) can be distributed as a single self-contained executable.

6.2.4 A logical data structure for AST traversals

The Termite interface is designed to facilitate the analysis and transformation of abstract syntax trees. The most important and fundamental operation for these applications is to traverse the AST in a meaningful fashion. The requirements for a meaningful traversal are that it must allow to

I. recognize specific patterns,

II. freely inspect the context of the current node, and

III. modify the AST during the traversal.

Requirement (I) was the main motivation to choose Prolog as implementation language. Prolog has language-level support for pattern matching and thus already saves much implementation effort compared to functionally similar traversal implementations written in C++ (most notably the Rose compiler whose intermediate representation formed the basis for the Termite terms).

The `transformed_with/6`-predicate provides efficient AST traversals that recursively visit every node in any order and call a visitor goal. The goal

```
transformed_with(Node, Transformation, Order, Info, InfoT, NodeT)
```

traverses an AST `Node` according to `Order` and calls the goal `Transformation` at each sub-node. `Transformation` is expected to implement the interface

```
transformed(Info, InfoT, Node, NodeT)
```

`Info` can be used by `Transformation` to pass any data between nodes. `Order` must be either `preorder` or `postorder`. `InfoT` and `NodeT` are unified with the result of the transformation.

This interface satisfies requirements (I) and (III) but has the drawback that the transformation predicate only sees the current node and its children at a time. It is completely ignorant of the parent or siblings of the current node. In a C++ implementation this problem would be solved by passing a pointer to the parent node. In a language lacking mutable data structures this is not possible without violating requirement (III).

A zipper [Hue97] is a functional data structure that allows to bidirectionally navigate through lists. By combining this idea with that of difference lists [SS94] it is possible to design a logical data structure that also adheres to requirement (II). Similar to the functional zipper, a *context* is attached to the data structure, storing the current location in the tree. However, similar to logical difference lists, the current node is replaced by a variable that is also part of the context data structure. This way the current node can be replaced by unifying this variable with a new node. Figure 6.2 illustrates how this data structure can be applied to a binary tree: In this example we want to visit node *c*, which is found somewhere in the middle of a larger tree. First, *c* and its children are cut off from its parents. This extraction process leaves a gap in the parent tree where the visited node and its children were originally. A copy of the parent tree with the gap replaced by a logic variable is stored as context. The variable is also explicitly stored in the context such that it is easily accessible. Further, the path taken downwards in the tree is stored in the context. In a balanced binary tree, the space requirement for this is logarithmic in the number of elements. In a typical AST, however, most nodes are stored in sequential lists (such as compound statements), which can be expressed as n-ary nodes, thus reducing the space needed to store the path to a node even further. The interfaces exposes predicates like `down/3`, `up/3`, `top/2`, `left/3` and `right/3`, but also more specific ones like `goto_function/3`. There is also `next_tdlr/2`, which visits the next node in a top-down-left-to-right fashion. This can be used to implement preorder and postorder traversals. Using this interface the parents of every node are just one call to `up/2` away.

```
down(zipper(X,Ctx), N, zipper(Child,[down(X1,Gap,N)|Ctx])) :-
```

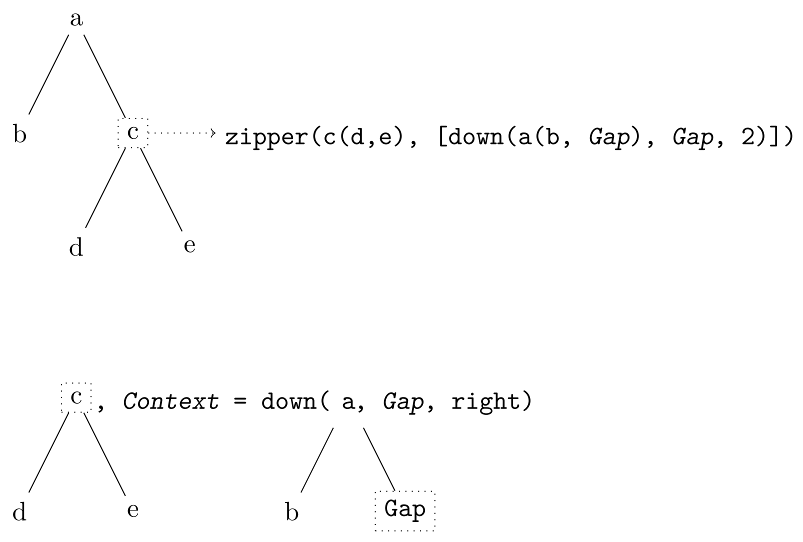

a
b
c
zipper(c(d,e), [down(a(b, Gap), Gap, 2)])
d
e
c
, Context = down(a, Gap, right)
d
e
b
Gap

Figure 6.2: Packing a zipper that points to node **c**

```
    X =.. List,
    replace_nth(List, N, Gap, Child, List1),
    X1 =.. List1.

up(zipper(X,[down(Parent,Gap,_)|Ctx]), zipper(Parent,Ctx)) :-
    X = Gap.
```

6.3 WCETC code generation

The WCETC language is an extension of the ANSI C programming language [Kir02]. It is used as the output language for TuBound and the input language of CalcWCET$_{C167}$. It is not possible to use the ROSE unparser to generate WCETC syntax, therefore a specialized backend called `wcetcc` exists for this purpose. This backend reuses the Termite unparser to do most of its work, but replaces the code generation for those AST nodes that contain syntax extensions in WCETC. In practise this affects all loop and compound statements. For each compound statement (such as a loop body) the list of child nodes is processed in the following order:

1. Scope declarations are transformed into `WCET_SCOPE` blocks.
2. Variable declarations are emitted.
3. Marker declarations are emitted.
4. The implicit marker of the current compound statement is declared (cf. Section 5.4.2).
5. The remaining nodes are recursively unparsed, while skipping any remaining annotation nodes in the current compound statement.

6. After the closing curly brace, all flow constraints are translated into `WCET_CONSTRAINT`s. This translation is also done recursively and consists mostly of the renaming of some operators.

Due to the different handling of scopes, the output of `wcetcc` is not column-exact any more. The `wcetcc` backend was successfully tested with the Mälardalen and DSPstone as well as the DEBIE benchmarks.

6.4 Analysis-guided loop unrolling

Most of the information that is needed for WCET analysis is about the iteration behaviour of loops. Since loop information is also crucial for many optimizations it is only natural to make all this information also available to the optimizer. Prototypical for other optimizations, we implemented a version of loop unrolling that is guided by loop bound annotations in the source code. When the worst-case behaviour is used as optimization target, it is sufficient to implement only a basic version of loop unrolling: First of all, in the WCET sense, loop unrolling is only beneficial when the unroll factor is statically known to be a divisor of the loop trip count. This is because in most cases, compensation code introduced for the remaining loop iterations potentially increases the WCET.

The loop unrolling process is controlled by a maximum unroll factor k_{max}. Each loop with a loop bound of $b_{min} = b_{max}$ is then unrolled by the largest possible n, where $n \in [1 \ldots k]$, $n \bmod b \equiv 0$. This way, it is never necessary to generate any compensation code. Loop unrolling is only applied to innermost loops.

Chapter 7

Evaluation

In this chapter we set out to prove the practicability of our approach by using standardized benchmarks to assess the algorithms presented in this thesis and their respective implementations. This chapter is made up of three sections. Each of them focuses on one specific part of the TuBound implementation. Section 7.1 investigates and discusses the flow analysis results from the *WCET Tool Challenge 2008* which were produced by the '08 version of the static analyses presented in Chapter 3. In Section 7.2 we analyze the effects of using model checkers to reduce the trusted annotation base as described in Chapter 4. Finally, Section 7.3 details the effects of program optimizations on the analyzed worst-case execution time, as discussed in Chapter 5.

7.1 Flow Problems from the WCET Tool Challenge 2008

The WCET Tool Challenge 2008 was a continuation of the first Tool Challenge held in 2006 and organized by the ARTIST2 Network of Excellence on Embedded Systems Design funded by the European Commission within the 7th Framework Programme [Tan09; HGB+08]. In total there were six contenders, both industrial and academic (cf. Table 1.1 on page 10): Bound-T (Tidorum Ltd), MTime (Measurement-based research prototype of Vienna University of Technology), Otawa (IRIT, Toulouse), RapiTime (Rapita Systems Ltd), *TuBound (Vienna University of Technology)* and Wcc (Dortmund University of Technology). Learning from problems at the previous instance of the Tool Challenge, two new additions to the rules were made to increase the comparability of the results, namely

1. the introduction of a common target hardware platform (ARM-based),
2. the introduction of hardware-independent flow analysis problems.

For practical reasons (an ARM backend of CalcWCET$_{C167}$ was not available) we chose to report WCET analysis results only for the C167 platform. Since these results were not comparable to the timings reported by the other contenders, we will focus on the hardware-independent flow analysis problems in this section. Neither MTime (measurement-based, [WRKP05]) nor RapiTime (also measurement-based, [RAP06]) had the infrastructure to answer flow analysis questions at the time of the WCET Tool Challenge 2008. For this reason, they are omitted from the tables. The detailed results from the Tool Challenge can be found at the official wiki.[1]

The main benchmark used in the Tool Challenge was the DEBIE-1 control software which was provided by Space Systems Finland Ltd (SSF) for this specific purpose. Tables 7.1–7.5 show the flow analysis results for the `debie1`-benchmark. The DEBIE-1 is a satellite instrument for measuring impacts of micro-meteoroids and small space debris [KDM+01]. The DEBIE-1 instrument was successfully launched into orbit aboard the PROBA I satellite in 2001. Its mostly-identical successor DEBIE-2 is installed on the ESA Columbus module which is now a part of the International Space Station. The code of the `debie1`-benchmark consists of about 10 000 lines of C code, driven by an infinite loop that is entered from the `main()` function. The actual I/O with the A/D converters reporting the sensor data was replaced by a test harness that systematically triggers interesting constellations in the driver code. Typical flow problems posed questions such as the maximum number of times a certain function or C macro is executed for each time another encompassing function is entered.

To answer these questions we used only the source-code-based parts of the TuBound tool chain. In the first steps all the static analyses are performed. Beginning from the encompassing function given in the problem description, a constraint logic program is constructed in the same way ILP constraints are formulated for classical WCET analysis (cf. Section1.1.3, [KKP+10b]) that, when solved, yields the maximum number of times the label in question is executed [Pra09].

Of the total six problem groups there were four with flow problems associated: The Telemetry (TM) Interrupt [2] is triggered when a transmission over the I/O channels is completed. The Hit Trigger Interrupt [3] is responsible for handling an impact event. The TeleCommand (TC) interrupt [4] is triggered whenever a complete word is received over the I/O channel. The periodic monitor task [6] performs housekeeping activities such as the measurement of voltage levels and the computation of checksums to detect radiation-related memory errors. It invokes an error handler if necessary.

Tables 7.1–7.5 contain the flow analysis results for all participants of the Challenge. The name of the tool is given in the first column. In the second column the compiler used by the tool is noted. Technically—at least speaking for TuBound—invoking the compiler is not strictly necessary to answer the Challenge questions. In the following columns, the answers for the questions found by the respective tools are given as upper and lower bounds in the form *low . . . up*. Empty cells [—] mean that a tool could not compute an answer. In problem 2a-F1, the tools reporting 0 . . . 0 were correct, as the problem asked for the execution count of an infeasible path. The cases where TuBound reported the most precise results are highlighted in green colour. Since the 2008 version of TuBound could not compute lower bounds (this feature was not present until late in 2009) it always reported 0 for the lower bound.

From this comparison we can see that the overall flow analysis performance of TuBound is quite competitive with that of the other tools. The most interesting

[1] `http://www.mrtc.mdh.se/projects/WCC08/doku.php`

Tool	Compiler	2a-F1	2b-F1	2c-F1
Bound-T	gcc-if07	0 . . . 0	1 . . . 1	1 . . . 1
OTAWA		0 . . . 1	1 . . . 1	1 . . . 1
TuBound	gcc-c16x	0 . . . 1	0 . . . 1	0 . . . 1
WCC	WCC	0 . . . 0	1 . . . 1	1 . . . 1

Table 7.1: Problem group 2: TM Interrupt Handler

Tool	Compiler	3a-F1	3a-F2	3b-F1	3c-F1
Bound-T	gcc-if07	0 . . . 12	—	—	0 . . . 150
OTAWA		0 . . . 12	1 . . . 1	0 . . . 12	0 . . . 156
TuBound	gcc-c16x	0 . . . 156	0 . . . 5	0 . . . 156	0 . . . 156
WCC	WCC	0 . . . 12	0 . . . 0	0 . . . 123	0 . . . 150

Table 7.2: Problem group 3: Hit Trigger Interrupt Handler

Tool	Compiler	4a-F1
Bound-T	gcc-if07	0 . . . 32
OTAWA		—
TuBound	gcc-c16x	0 . . . 32
WCC	WCC	0 . . . 38

Table 7.3: Problem group 4: TC Execution Task

Tool	Compiler	6a-F1	6a-F2	6b-F1	6c-F1
Bound-T	gcc-if07	0 . . . 18	0 . . . 0	0 . . . 20	0 . . . 58
OTAWA		—	—	—	—
TuBound	gcc-c16x	—	0 . . . 8	—	—
WCC	WCC	0 . . . 18	0 . . . 0	0 . . . 36	0 . . . 360

Table 7.4: Problem group 6, a–c: Monitoring Task, no errors

Tool	Compiler	6d-F1	6e-F1	6e-F2
Bound-T	gcc-if07	0 . . . 8	0 . . . 2295	0 . . . 8
OTAWA		—	—	—
TuBound	gcc-c16x	0 . . . 8	—	0 . . . 8
WCC	WCC	0 . . . 12	0 . . . 2295	0 . . . 12

Table 7.5: Problem group 6, d–e: Monitoring Task, some errors

result from the benchmarks is that there is no tool that is consistently better than any other. For example, when compared to WCC, the results reported by TuBound for the benchmarks 3a-F1 and 3c-F1 are largely overestimated. On the other hand, the TuBound results for 4a-F1, 6d-F1 and 6e-F2 are consistently better than those of WCC. OTAWA produced much better results than TuBound for benchmarks 3a–3c, but failed to produce results for the benchmark groups 4–6. While the commercial product Bound-T produced very good bounds for most benchmarks, it could not yield results for benchmarks 3a-F2 and 3a-F1—problems that OTAWA, TuBound and WCC could solve.

The enhancements of TuBound that were added throughout the year 2009, including improved flow analysis by incorporating model checking into the work flow are discussed in the following section.

7.2 Loop analysis with model checking

In this section we show how the concert of model checking and binary widening/-narrowing can improve the results from a classical loop analysis. We implemented our approach as an extension of TuBound and applied the extended version to the well-known Mälardalen WCET benchmark suite. As a baseline for comparison we used the 2008 version of TuBound, as it was used for the WCET Tool Challenge 2008, later on referred to as the basic version of TuBound. Our experiments were guided by two questions: "Can the number of automatically bounded loops be increased significantly?" and "How expensive is the process?". The benchmarks were performed on a 3 GHz Intel Xeon processor running 64-bit Linux. The model checker used was CBMC 2.9, which we applied to testing loop bounds up to the size of $2^{13} = 8192$ using a timeout of 120 seconds and a maximum unroll factor of $2^{13} + 1$[2] The `compress` and `whet` benchmarks contained unstructured goto loops; as indicated in Section 4.6 these were automatically converted into do-while loops beforehand by a separate Termite transformation.

Our findings are summarized in Table 7.6. The third column of this table shows the number of loops that can be bounded by the basic version of TuBound; column four shows the total number of loops the extended version of TuBound was able to bound. It is important to note that the model checker was only invoked for benchmarks where at least one loop could not be bounded by the basic version of TuBound. The last column shows the accumulated runtime of the model checker.

Comparing columns three and four reveals the superiority of the extended version of TuBound over its basic variant. The extended version raises the total number of bounded loops from 77% to 85%. Considering column five, it can be seen that the model checker terminates quickly on small problems but that the runtime and space requirements can increase to practically infeasible amounts on problems suffering from the state explosion problem. Such a behaviour can be triggered if the initialization values which are part of the majority of the Mälardalen benchmarks are manually invalidated by introducing a faux dependency on, for instance, `argc`. This demonstrates that model checking is to be used with care or the model checker be fed with additional information guiding and simplifying the verification task.

[2]This setting controls the depth of loop analysis in the model checker.

		TuBound		
		basic	+ model checking	
Benchmark	Loops	bounded/out of	bounded/out of	Runtime
bs	1	0/1	1/1	$0.03s$
janne_complex	2	0/2	2/2	$0.18s$
nsichneu	1	0/1	1/1	$5.59s$
statemate	1	0/1	1/1	$0.06s$
qsort-exam	6	0/6	4/6	$0.02s$
fft1	11	6/11	9/11	$0.43s$
minver	17	16/17	17/17	$0.06s$
duff	2	1/2	1/2	$\sim 0s$
whet	11	10/11	10/11	$\sim 0s$
adpcm	18	15/18	15/18	timeout
compress	8	2/8	2/8	timeout
fir	2	1/2	1/2	timeout
insertsort	2	0/2	0/2	timeout
lms	10	6/10	6/10	timeout
select	4	0/4	0/4	timeout
bsort100	3	3/3	3/3	–
cnt	4	4/4	4/4	–
cover	3	3/3	3/3	–
crc	3	3/3	3/3	–
edn	12	12/12	12/12	–
expint	3	3/3	3/3	–
fdct	2	2/2	2/2	–
fibcall	1	1/1	1/1	–
jfdctint	3	3/3	3/3	–
lcdnum	1	1/1	1/1	–
ludcmp	11	11/11	11/11	–
matmult	5	5/5	5/5	–
ndes	12	12/12	12/12	–
ns	4	4/4	4/4	–
qurt	1	1/1	1/1	–
sqrt	1	1/1	1/1	–
st	5	5/5	5/5	–
recursion	0	–/–	–/–	–
Total	170	131/170	144/170	–

Table 7.6: Number of bounded loops in the Mälardalen benchmarks

7.3 Flow transformation benchmarks

The results reported in Figure 7.1 and 7.2 indicate the potential of high-level loop optimizations. Figure 7.1 shows results for the standardized set of WCET analysis benchmarks collected by Mälardalen University [Mä10a]. In Figure 7.2 results for the fixed-point version of the DSPstone [ŽVSM94] benchmarks are shown.

Using the benchmarks that could be fully analyzed (and automatically annotated at the source code level) by an unassisted TuBound, the diagram shows the WCET bound for several combinations of optimizations. The WCET bound calculation was done using the CalcWCET$_{C167}$ back end of TuBound. From left to right each diagram shows four columns for each benchmark:

1. WCET bound of the unoptimized program.
2. WCET of the high-level loop optimized program.
3. WCET with low-level control-flow insensitive optimizations.
4. WCET when combining both types of optimizations.

Each value is normalized by the WCET bound of the unoptimized program (column 1). Loop optimizations are performed by the source-to-source optimizer which uses the upper and lower loop bound information found by TuBound. Therefore the applied loop optimizations improve the analyzed WCET in most cases. The low-level optimizations are performed by the target compiler and do not alter the control flow any more. In the last group of bars the average performance impact is shown: For each of the four categories, the bars represent the geometric mean of the scaled execution speed of each of the benchmarks.

The benchmarks indicate that the potential for optimizations is significant. It is important to keep in mind that high-level loop optimizations (loop unrolling, fusion, interchange, splitting), usually target the average-case performance. For this reason it is important to keep an eye on the optimization parameters. It is, for instance, counterproductive to perform loop unrolling with an unroll factor that divides the actual trip count of a loop with a remainder since the compensation code necessary for the remaining iterations would in fact decrease the worst-case performance. For this reason, we specialized the loop unrolling algorithm to become WCET-aware, as described in Section 6.4.

Outliers like `whet` furthermore show that careful selection of the different optimization phases is very important. This process, however, can be supported by an automatic WCET analysis, which can be used to guide the optimizer by judging the improvement of a program transformation [LM09].

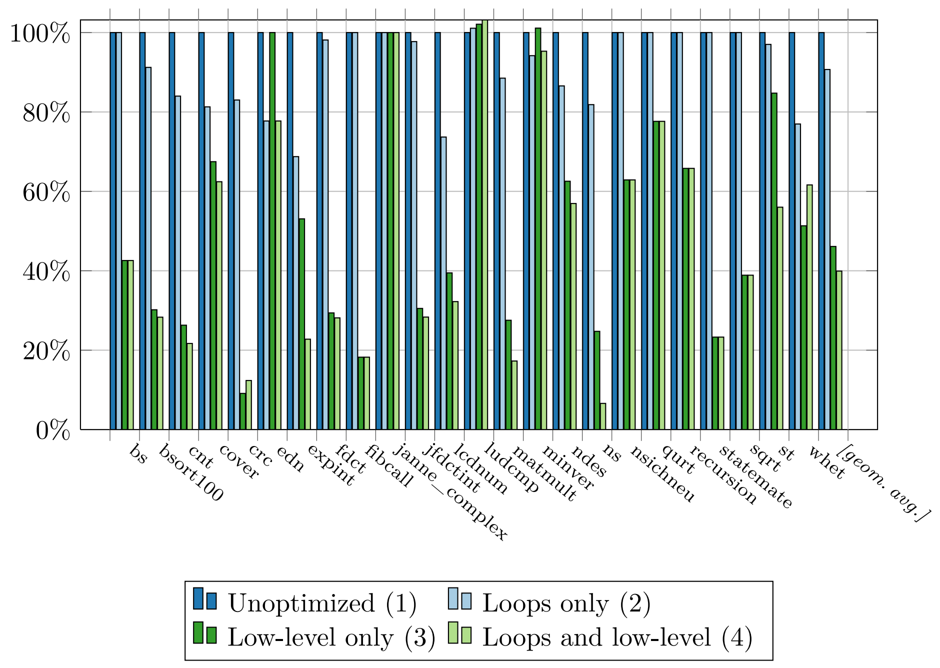


Figure 7.1: Benchmark results for the Mälardalen benchmarks (Vertical bars show analyzed WCET relative to the unoptimized program)

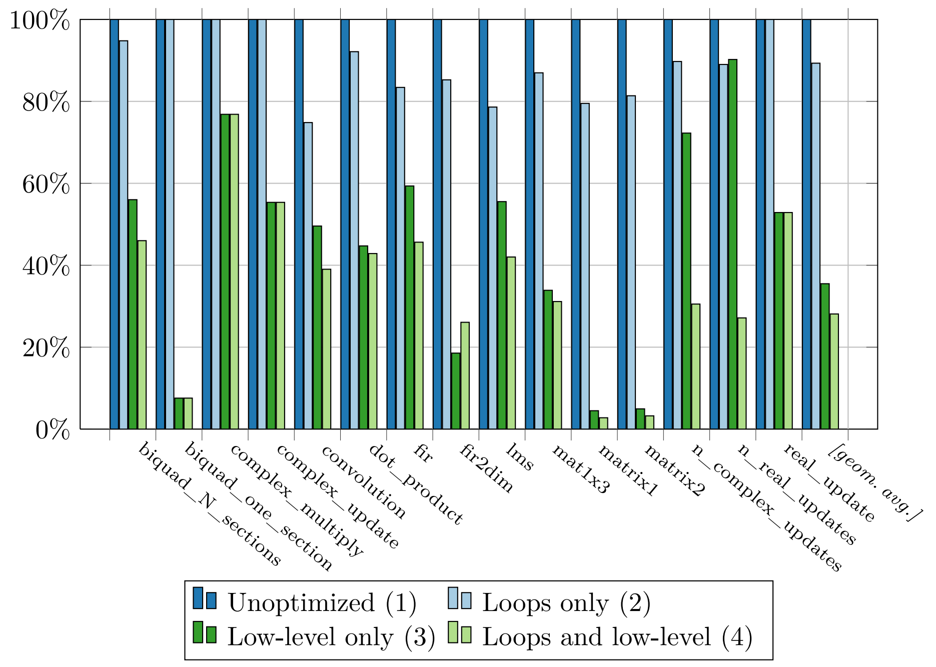


Figure 7.2: Benchmark results for the fixed-point DSPstone benchmarks (Vertical bars show analyzed WCET relative to the unoptimized program)

Chapter 8

Conclusions and perspectives

In this thesis we set out to show that it is possible to lift both control-flow analysis and program annotations to the source code level without sacrificing the performance of executables or the accuracy of WCET analysis. To demonstrate this new concept we implemented the WCET analyzer TuBound, which followed a radical approach of performing control-flow analysis on the source code level and compiler optimizations as source-to-source transformations. We further included support for fully embedded annotations in the source code without sacrificing accuracy: Source code annotations are transformed alongside the compiler optimizations using a generic and portable flow constraint transformer. In many cases the correctness of annotations can even be checked by employing external tools for formal program verification.

To answer the question whether we could reach our goal of demonstrating the usefulness of such an approach, we found the following indicators pointing towards success:

- With the initial presentation of the TuBound source-to-source concept at the WCET Workshop 2008 we demonstrated the general feasibility of our platform-independent approach.
- With the participation in the WCET Tool Challenge 2008 we were able to make the step from a proof-of-concept implementation towards becoming a viable contributor to the WCET community. The interest expressed by several of our partners in the ALL-TIMES[1] project in interfacing with our static analyses can be seen as a confirmation of this transition.
- With funding from the ALL-TIMES project we were able to provide interfaces for three of our partners' tools, all implemented by the ALL-TIMES team at Vienna University of Technology: For AbsInt's aiT [Abs10a], we provide a backend that is capable of generating annotations in the AIS language. Rapita Systems' RapiTime [Rap10] is connected via an abstract source location interface. The connection with Mälardalen University's

[1]ALL-TIMES is a multilateral research project funded by the 7th framework programme of the European Commission, `http://www.all-times.org`.

SWEET [Mä10b] builds upon the Termite-based `melmac` compiler, which generates annotated code in the ALF intermediate representation.

- Apart from the ALL-TIMES project we also found the opportunity to cooperate with the authors of the OTAWA tool and the oRange control flow analyzer from the Institut de Recherche en Informatique de Toulouse (IRIT): In a joint bilateral French-Austrian Amadeus/Amadée project funded by both the Égide scientific exchange service of the MAEE[2] and MESR[3] in France and the scientific cooperation programme (WTZ) of the OEAD[4] in Austria, "Trends in Timing Analysis: Uniform WCET Annotation Language" we cooperate on developing a flow annotation language which can serve as a community standard.

Regarding future work, we outline streams of research which we consider particularly important and rewarding based on experiences with our approach: Firstly, as already indicated in the previous chapters, there is still room to further improve the quality of the static analyses presented in this thesis. Future extensions to the presented algorithms may either incorporate more expensive but powerful techniques, or go towards supporting more language features, such as templates.

Secondly, on a conceptual level, an important direction for future research is the extension towards using symbolic expressions in annotations and control flow analysis. Symbolic expressions are already used internally by the loop bound analysis. By making them visible to the outside, users would gain a valuable tool to write generic and more reusable annotations.

Additionally, the control flow analysis could be expanded to become less loop-centric. Currently, no WCET analysis tool exists that could answer questions like the relative execution frequencies of certain branches in a program. However, the answers to such questions often have a critical impact on the analyzed WCET.

Finally, our results for the flow information transformation benchmarks in Chapter 7.3 indicated a lot of potential for WCET-aware optimizations. In order to further increase the performance of embedded code, it would be most rewarding to investigate new types of optimizations specifically targeting the worst-case performance. Since this involves reconsidering all cost metrics typically applied by a compiler, this opens a broad field of research on its own.

[2]French Ministry of Foreign and European Affairs (Ministère des Affaires Étrangères et Européennes)

[3]French Ministry for Higher Education and Research (Ministère de l'Enseignement Supérieur et de la Recherche)

[4]Austrian Exchange Service (Österreichischer Austauschdienst) of the Federal Ministry of Science and Research (Bundesministerium für Wissenschaft und Forschung)

Bibliography

[AAG$^+$07] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. Cost Analysis of Java Bytecode. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2007. Cited on page 69.

[AAGP08] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In María Alpuente and Germán Vidal, editors, *SAS*, volume 5079 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2008. Cited on page 9.

[Abs10a] AbsInt Angewandte Informatik GmbH. aiT. Web page (`http://www.absint.com/ait/`), 2010. Cited on pages 5 and 86.

[Abs10b] AbsInt Angewandte Informatik GmbH. The program analyzer generator PAG. Web page (`http://www.absint.com/pag/`), 2010. Cited on pages 18 and 70.

[ADR98] Andrew Appel, Jack Davidson, and Norman Ramsey. The Zephyr Compiler Infrastructure. Technical report, Princeton University and University of Virginia, November 1998. Cited on page 5.

[AH74] Götz Alefeld and Jürgen Herzberger. *Einführung in die Intervallrechnung*, volume 12 of *Reihe Informatik.* B. I.-Wissenschaftsverlag, Mannheim – Wien – Zürich, 1974. Cited on page 28.

[AK02] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, Inc., 2002. ISBN 1-55860-286-0. Cited on page 63.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002. `http://www.simplescalar.com/`. Cited on page 6.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques and tools.* Pearson Education, second edition, 2007. Cited on page 43.

[Bár09] Gergő Bárány. SATIrE within ALL-TIMES: Improving Timing Technology with Source Code Analysis. In Jens Knoop and Adrian Prantl, editors, *15. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS '09)“*, number 2009-X-1 in Schriftenreihe des Instituts für Computersprachen, pages 27–37. Technische Universität Wien, October 2009. Cited on pages 20 and 33.

[Bár10] Gergő Bárány. melmac. Web page (`http://www.complang.tuwien.ac.at/gergo/melmac/`), 2010. Cited on pages 13 and 73.

[BCN$^+$08] Clément Ballabriga, Hugues Cassé, Fadia Nemer, Christine Rochange, and Pascal Sainrat. *OTAWA Online Program Documentation*. University of Toulouse, France, http://www.otawa.fr/, 2008. Cited on page 7.

[BCP03] Guillem Bernat, Antoine Colin, and Stefan M. Petters. pWCET, a Tool for Probabilistic WCET Analysis of Real-Time Systems. In *3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, pages 21–38, 2003. Cited on pages 8 and 10.

[BDEN10] Michel Berkelaar, Jeroen Dirks, Kjell Eikland, and Peter Notebaert. lp_solve, a Mixed Integer Linear Programming (MILP) solver. Web page (`http://lpsolve.sourceforge.net/`), 2010. Cited on page 6.

[BHJM07] Dirk Beyer, Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, October 2007. Cited on page 53.

[Bli94] Johann Blieberger. Discrete loops and worst case performance. *Computer Languages*, 20(3):193–212, 1994. Cited on page 9.

[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, June 1994. Cited on pages 3 and 10.

[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. Cited on page 3.

[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 (POPL)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. Cited on page 27.

[Che87] Moyer Chen. *A Timing Analysis Language – (TAL)*. Dept. of Computer Science, University of Texas, Austin, TX, USA, 1987. Programmer's Manual. Cited on pages 4 and 10.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. Cited on page 53.

[CM07] Christoph Cullmann and Florian Martin. Data-Flow Based Detection of Loop Bounds. In Rochange [Roc07]. Cited on page 35.

[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, June 2001. Technical University of Delft. Cited on pages 6 and 10.

[CS06] Hugues Cassé and Pascal Sainrat. OTAWA, a framework for experimenting WCET computations. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 25/01/06-27/01/06*, http://www.see.asso.fr, January 2006. Société de l'Electricité, de l'Electronique et des Technologies de l'Information et de la Communication (SEE). 8 pages. Cited on pages 7 and 10.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):992–1030, 1997. Cited on page 53.

[dMBCS08] Marianne de Michiel, Armelle Bonenfant, Hugues Cassé, and Pascal Sainrat. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Kaohsiung, Taiwan, 25/08/2008-27/08/2008*, pages 161–168, http://www.computer.org, August 2008. IEEE Computer Society. Cited on page 7.

[EE00] Jakob Engblom and Andreas Ermedahl. Modeling complex flows for worst-case execution time analysis. In *Proceedings 21st IEEE Real-Time Systems Symposium (RTSS)*, Orlando, Florida, USA, Dec. 2000. Cited on pages 7 and 10.

[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, June 1998. Cited on page 63.

[EES02] Andreas Ermedahl, Jakob Engblom, and Friedhelm Stappert. A unified flow information language for WCET analysis. In *Proc. 2nd International Workshop on Worst Case Execution Time Analysis.* Technical University of Vienna, Austria, June 2002. Cited on page 7.

[ESG$^+$07] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In Rochange [Roc07]. Cited on page 7.

[Fer04] Christian Ferdinand. Worst case execution time prediction by static program analysis. *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, 03:125a, 2004. Cited on page 5.

[Fet88] James H. Fetzer. Program verification: the very idea. *Communications of the ACM*, 31(9):1048–1063, 1988. Cited on page 38.

[FFY04] Joseph A. Fisher, Paolo Faraboschi, and Cliff Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, December 2004. Cited on pages 1 and 16.

[FHT03] Christian Ferdinand, Reinhold Heckmann, and Henrik Theiling. Convenient user annotations for a WCET tool. In *Proc. 3rd International Workshop on Worst-Case Execution Time Analysis*, pages 17–20, Porto, Portugal, July 2003. Cited on pages 5 and 10.

[Flo67] Robert Floyd. Assigning Meaning to Programs. In *Proc. of AMS Symposia in Applied Mathematics*, pages 19–32, 1967. Cited on page 9.

[FLT06] Heiko Falk, Paul Lokuciejewski, and Henrik Theiling. Design of a WCET-aware C compiler. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, July 4, 2006, Dresden, Germany*, volume 06902 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006. Cited on pages 7 and 10.

[Fre10] Free Software Foundation (FSF). GCC, the GNU Compiler Collection. Web page (`http://gcc.gnu.org/`), 2010. Cited on page 12.

[GEL+09] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – A Language for WCET Flow Analysis. In Niklas Holsti, editor, *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*, volume 09004 of *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 2009. Cited on pages 7 and 13.

[GJK09] Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In Michael Hind and Amer Diwan, editors, *PLDI*, pages 375–385. ACM, 2009. Cited on page 9.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler (with retrospective). In Kathryn S. McKinley, editor, *Best of PLDI*, pages 49–57. ACM, 1982. Cited on page 50.

[GLS+08] Jan Gustafsson, Björn Lisper, Markus Schordan, Christian Ferdinand, Peter Gliwa, Marek Jersak, and Guillem Bernat. ALL-TIMES – A European Project on Integrating Timing Technology. In Margaria and Steffen [MS08], pages 445–459. Cited on page 11.

[Gou01] Eric Goubault. Static analyses of the precision of floating-point operations. In Patrick Cousot, editor, *SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer, 2001. Cited on page 27.

[Gus06] Jan Gustafsson. The WCET tool challenge 2006. In *Preliminary Proceedings 2nd Int. IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 248–249, Paphos, Cyprus, November 2006. Cited on page 11.

[Har77] William H. Harrison. Compiler Analysis of the Value Ranges for Variables. *IEEE Transactions on Software Engineering*, 3(3):243–250, 1977. Cited on page 27.

[HBH$^+$07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert F. Pointon. Automatic amortised worst-case execution time analysis. In Rochange [Roc07]. Cited on page 6.

[HDF$^+$05] Kevin Hammond, Roy Dyckhoff, Christian Ferdinand, Reinhold Heckmann, Martin Hofmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert F. Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. The Embounded project (project start paper). In Marko C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 195–210. Intellect, 2005. Cited on pages 6 and 10.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, North-Holland, 1977. Cited on page 42.

[HF05] Reinhold Heckmann and Christian Ferdinand. Combining automatic analysis and user annotations for successful worst-case execution time prediction. In *Embedded World 2005 Conference*, Nürnberg, Germany, Feb. 2005. Cited on page 5.

[HGB$^+$08] Niklas Holsti, Jan Gustafsson, Guillem Bernat (eds.), Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET Tool Challenge 2008: Report. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 149–171, Prague, Czech Republic, July 2-4 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3. Cited on pages 11, 14, and 79.

[HLS00] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. Worst-case execution time analysis for digital signal processors. In *European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000. Cited on pages 5 and 10.

[HLS05] Niklas Holsti, Thomas Långbacka, and Sami Saarinen. *Bound-T timing analysis tool User Manual.* Tidorum Ltd, 2005. Cited on pages 5 and 35.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. Cited on pages 9 and 41.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-3)*, pages 104–115, 1995. Cited on page 53.

[HSR$^+$00] Christopher A. Healy, Mikael Sjodin, Viresh Rustagi, David B. Whalley, and Robert van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000. Cited on pages 5, 10, and 22.

[Hue97] Gérard P. Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, 1997. Cited on page 76.

[HW02] Christopher A. Healy and David B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28:763–781, 2002. Cited on page 5.

[IBM10] IBM. IBM ILOG CPLEX—High-performance mathematical programming engine. Web page (`http://www.ibm.com/software/integration/optimization/cplex/`), 2010. Cited on page 6.

[JTC10] JTC1/SC22/WG21. Working Draft, Standard for Programming Language C++, 2010-02-16. Web page (`http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2010/n3035.pdf`), 2010. Cited on pages 30 and 31.

[KDM$^+$01] J. Kuitunen, G. Drolshagen, J. A. M. McDonnell, H. Svedhem, M. Leese, H. Mannermaa, M. Kaipiainen, and V. Sipinen. DEBIE – first standard in-situ debris monitoring instrument. In Huguette Sawaya-Lacoste, editor, *Proceedings of the Third European Conference on Space Debris, 19–21 March 2001, Darmstadt, Germany*, volume ESA SP-473, Vol. 1, pages 185–190, Noordwijk, Netherlands, 10 2001. ESA Publications Division. ISBN 92-9092-733-X. Cited on page 80.

[Kir01] Raimund Kirner. *User's Manual – WCET-Analysis Framework based on* WCETC. Vienna University of Technology, Vienna, Austria, 0.0.3 edition, July 2001. available at `http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`. Cited on pages 7 and 10.

[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. Cited on pages 7, 23, and 77.

[Kir03] Raimund Kirner. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis.* PhD thesis, Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, May 2003. Cited on page 63.

[Kir08] Raimund Kirner. *Compiler Support for Timing Analysis of Optimized Code: Precise Timing Analysis of Machine Code with Convenient Annotation of Source Code.* VDM Verlag, Germany, July 2008. ISBN: 978-3-8364-6883-1. Cited on page 7.

[KKP$^+$07] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET Analysis: The Annotation Language Challenge. In Christine Rochange, editor, *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 83–99, Pisa, Italy, 2007. Institut de Recherche en Informatique de Toulouse. Cited on pages 13 and 15.

[KKP$^+$08] Raimund Kirner, Albrecht Kadlec, Peter Puschner, Adrian Prantl, Markus Schordan, and Jens Knoop. Towards a common wcet annotation language: Essential ingredients. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 53–65, Prague, Czech Republic, July 2-4 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3. Cited on pages 15, 17, 51, and 69.

[KKP10a] Albrecht Kadlec, Raimund Kirner, and Peter Puschner. Avoiding timing anomalies using code transformations. In *Proceedings of the 13th IEEE International Symposium on Object/component/service-oriented Real-time distributed computing (ISORC 2010)*, Carmona, Spain, May 2010. Cited on page 2.

[KKP$^+$10b] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: Comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 2010. (online edition). Cited on pages 3, 4, 9, 13, and 80.

[KPP10] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 2010. (online edition). Cited on pages 14, 59, 61, and 65.

[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. Cited on pages 3 and 10.

[Law10] Lawrence Livermore National Laboratory. The Rose Compiler. Web page (`http://www.rosecompiler.org/`), 2010. Cited on pages 12, 18, and 70.

[LCFM09] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *CGO*, pages 136–146. IEEE Computer Society, 2009. Cited on page 7.

[LFMT08] Paul Lokuciejewski, Heiko Falk, Peter Marwedel, and Henrik Theiling. WCET-driven, code-size critical procedure cloning. In *Proc. 11th International Workshop on Software and Compilers for Embedded Systems*, pages 21–30, Munich, Germany, Mar. 2008. Cited on page 69.

[Lis05] Björn Lisper. Ideas for annotation language(s). Technical report, Department of Computer Science and Engineering, Mälardalen University, October 25, 2005. Cited on page 9.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007. Cited on page 20.

[LLMR07] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007. `http://www.comp.nus.edu.sg/~rpembed/chronos`. Cited on pages 6 and 10.

[LLV10] The LLVM Project. clang: a C language family frontend for LLVM. Web page (`http://clang.llvm.org/`), 2010. Cited on page 18.

[LM95] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings 32nd ACM/IEEE Design Automation Conference*, pages 456–461, June 1995. Cited on page 5.

[LM09] Paul Lokuciejewski and Peter Marwedel. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In *The 21st Euromicro Conference on Real-Time Systems (ECRTS)*, pages 35–44, Dublin / Ireland, July 2009. IEEE Computer Society. Cited on pages 7, 59, and 84.

[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, 1999. Cited on pages 8 and 10.

[LS99b] Thomas Lundqvist and Per Stenström. A method to improve the estimated worst-case performance of data caching. In *RTCSA*, pages 255–262. IEEE Computer Society, 1999. Cited on page 8.

[LS99c] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. *Real-Time Systems Symposium, IEEE International*, 0:12, 1999. Cited on page 8.

[MACT89] Aloysius K. Mok, Prasanna Amerasinghe, Moyer Chen, and Kamtorn Tantisirivat. Evaluating tight execution time bounds of programs by annotations. In *Proc. 6th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–80, Pittsburgh, PA, USA, May 1989. Cited on page 4.

[Mar98] Florian Martin. PAG – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998. Cited on page 18.

[MS08] Tiziana Margaria and Bernhard Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October*

13-15, 2008. Proceedings, volume 17 of *Communications in Computer and Information Science*. Springer, 2008. Cited on pages 91 and 100.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4. Cited on pages 21, 50, and 53.

[Mä10a] Mälardalen University. Benchmarks for WCET Analysis. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2010. Cited on page 84.

[Mä10b] Mälardalen University. SWEET (SWEdish Execution Time tool). Web page (`http://www.mrtc.mdh.se/projects/wcet/sweet.html`), 2010. Cited on page 87.

[Nec97] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM. Cited on page 39.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag Berlin Heidelberg New York, 1999. Cited on pages 28, 42, and 46.

[Par92] Chang Yun Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, Seattle, USA, 1992. TR 92-08-02. Cited on pages 4 and 10.

[Par93] Chang Yun Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993. Cited on page 4.

[PCB$^+$06] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. GRAPHITE: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developer's Summit*, pages 179–198, June 2006. Cited on page 63.

[PKK$^+$09] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009)*, pages 39–49, Dublin, Ireland, June 2009. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-252-6. Cited on pages 13, 15, and 51.

[PKS09] Adrian Prantl, Jens Knoop, and Markus Schordan. Persistent analysis results. In Michael Hanus and Bernd Braßel, editors, *26. Workshop der „GI-Fachgruppe Programmiersprachen und Rechenkonzepte"*, number 0915 in Technische Berichte des Instituts für Informatik, pages 87–98, Olshausenstr. 40, D – 24098 Kiel, 2009. Institut für Informatik der Christian-Albrechts-Universität zu Kiel. Cited on page 13.

[PKST08] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. In *The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008. Cited on pages 13 and 27.

[Pra07a] Adrian Prantl. Source-to-Source Transformations for WCET Analysis: The CoSTA Approach. In Michael Hanus and Bernd Braßel, editors, *24. Workshop der „GI-Fachgruppe Programmiersprachen und Rechenkonzepte“*, number 0707 in Technische Berichte des Instituts für Informatik, pages 51–60, Olshausenstr. 40, D – 24098 Kiel, 2007. Institut für Informatik der Christian-Albrechts-Universität zu Kiel. Cited on page 65.

[Pra07b] Adrian Prantl. The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation. In Walter Dosch, Clemens Grelck, and Anette Stümpel, editors, *14. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS '07)“*, number A-07-07 in Schriftenreihe A, pages 172–177. Institute für Informatik und Mathematik der Universität zu Lübeck, 2007. Cited on pages 14, 23, and 63.

[Pra09] Adrian Prantl. Towards a static profiler. In Jens Knoop and Adrian Prantl, editors, *15. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS '09)“*, number 2009-X-1 in Schriftenreihe des Instituts für Computersprachen, page 230, Maria Taferl, Austria, October 2009. Technische Universität Wien. Cited on page 80.

[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. Cited on page 4.

[PS97] Peter Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13:67–91, 1997. Cited on pages 3 and 5.

[PSK08] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3. Cited on pages 10, 13, 17, and 65.

[QSMK04] Daniel J. Quinlan, Markus Schordan, Brian Miller, and Markus Kowarschik. Parallel object-oriented framework optimization. *Concurrency and Computation: Practice and Experience*, 16(2-3):293–302, 2004. Cited on page 65.

[RAP06] RAPITA Systems Ltd. Worst-case execution time analysis. White Paper (Automotive), Rev. 1.32, 21st Sep. 2006. Cited on pages 8 and 80.

[Rap10] Rapita Systems Ltd. RapiTime On Target Timing Analysis. Web page (`http://www.rapitasystems.com/rapitime/`), 2010. Cited on page 86.

[Roc07] Christine Rochange, editor. *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*, volume 07002 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007. Cited on pages 90 and 92.

[RPW08] Bernhard Rieder, Peter P. Puschner, and Ingomar Wenzel. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement based WCET analysis. In *WISES*, pages 1–7. IEEE, 2008. Cited on page 53.

[Sch07a] Markus Schordan. Combining tools and languages for static analysis and optimization of high-level abstractions. Proceedings 24th Workshop of „GI-Fachgruppe Programmiersprachen und Rechenkonzepte“. Technical Report, Christian-Albrechts-Universität zu Kiel, 2007. Cited on page 18.

[Sch07b] Daniel Schulte. *Flow Facts für WCET-optimierende Compiler: Modellierung und Transformation.* VDM Verlag, Germany, July 2007. ISBN: 978-3836448130. Cited on page 63.

[SE06] Jan Staschulat and Rolf Ernst. Worst case timing analysis of input dependent data cache behavior. In *ECRTS*, pages 227–236. IEEE Computer Society, 2006. Cited on page 8.

[Sha89] Alan C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989. Cited on pages 4 and 10.

[SP81] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981. Cited on pages 20 and 50.

[SP10] Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for wcet analysis. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 67–76, New York, NY, USA, 2010. ACM. Cited on page 2.

[SQ03] Markus Schordan and Daniel J. Quinlan. A source-to-source architecture for user-defined optimizations. In László Böszörményi and Peter Schojer, editors, *JMLC*, volume 2789 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2003. Cited on page 18.

[SS94] Leon Sterling and Ehud Y. Shapiro. *The Art of Prolog – Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994. Cited on page 76.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM. Cited on pages 20 and 33.

[Tan09] Lili Tan. The worst-case execution time tool challenge 2006. *International Journal on Software Tools for Technology Transfer (STTT)*, 11(2):133–152, 2009. Cited on pages 11 and 79.

[TNW08] Markus Triska, Ulrich Neumerkel, and Jan Wielemaker. A generalised finite domain constraint solver for SWI-Prolog. In Sibylle Schwarz, editor, *In Proceedings of the 22nd Workshop on (Constraint) Logic Programming (WLP 2008)*, pages 89–91, 2008. Cited on page 36.

[UGoT$^+$10] Mälardalen University, AbsInt Angewandte Informatik GmbH, Vienna University of Technology, Gliwa GmbH, Symtavision GmbH, and Rapita Systems Ltd. The All-TIMES project. Web page (`http://www.all-times.org`), 2010. 2007 through 2010, funded by the 7th EU R&D Framework Programme as research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068. Cited on page 11.

[Uni10] University of Bonn. The JTransformer framework. `http://roots.iai.uni-bonn.de/research/jtransformer/`, 2010. Cited on page 18.

[Vie10a] Vienna University of Technology. The CalcWCET$_{167}$ tool. Web page (`http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/`), 2010. Cited on pages 12, 20, and 23.

[Vie10b] Vienna University of Technology. The static analysis tool integration engine SATIrE. Web page (`http://www.complang.tuwien.ac.at/markus/satire/`), 2010. Cited on pages 12 and 18.

[Vie10c] Vienna University of Technology. The term manipulation library Termite. Web page (`http://www.complang.tuwien.ac.at/adrian/termite/`), 2010. Cited on pages 13, 18, and 71.

[Vrc92] Alexander Vrchoticky. Modula/R – Language Definition. Technical Report 02/1992, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Mar. 1992. Cited on pages 3 and 10.

[Vrc94] 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, June 1994. Cited on page 3.

[WA08] Jack Whitham and Neil Audsley. Forming Virtual Traces for WCET Analysis and Reduction. In *Proc. RTCSA*, pages 377–386, 2008. Cited on page 16.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckman, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3), Apr. 2008. Cited on pages 2, 3, and 5.

[WG110] WG14/N1124. Committee Draft – May 6, 2005 – ISO/IEC 9899:TC2. Web page (`http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`), 2010. Cited on pages 30 and 31.

[Wie03] Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th Int. Workshop on Logic Programming Environments (WLPE 2003)*, pages 10–16, December 2003. Cited on page 36.

[WKPR05] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings 5th International Conference on Quality Software*, Sep. 2005. Cited on page 8.

[WKRP08] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In Margaria and Steffen [MS08], pages 430–444. Cited on page 2.

[WRKP05] Ingomar Wenzel, Bernhard Rieder, Raimund Kirner, and Peter P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *DATE*, pages 606–611. IEEE Computer Society, 2005. Cited on pages 9, 10, and 79.

[WSE02] Fabian Wolf, Jan Staschulat, and Rolf Ernst. Associative caches in formal software timing analysis. In *DAC '02: Proceedings of the 39th annual Design Automation Conference*, pages 622–627, New York, NY, USA, 2002. ACM. Cited on pages 8 and 10.

[ZP08] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In Jorge Cuéllar, T. S. E. Maibaum, and Kaisa Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008. Cited on page 44.

[ŽVSM94] Vojin Živojnović, Juan Martínez Velarde, Christian Schläger, and Heinrich Meyr. DSPstone: A DSP-Oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT)*, Dallas, October 1994. Cited on page 84.

Curriculum Vitae

Personal Data

Name	Adrian Prantl
Date of Birth	Aug. 10, 1982
Citizenship	Austria
Home Address	Neustiftgasse 45/12 A-1070 Wien Austria

Affiliation

Institute of Computer Languages
Vienna University of Technology
Argentinierstraße 8 / E185.1
1040 Vienna, Austria

Phone: +43-1-58801-58521
Fax: +43-1-58801-18598
E-Mail: adrian@complang.tuwien.ac.at
WWW: `http://www.complang.tuwien.ac.at/adrian`

Employment

2010 – ongoing	Institute of Computer Languages / Vienna University of Technology Research Assistant in the EU-FP7-funded project *"ALL-TIMES: Integrating European Timing Analysis Technology"*.
2006 – 2010	Institute of Computer Languages / Vienna University of Technology Research Assistant in the FWF-funded project *"Compiler Support for Timing Analysis (CoSTA)"*.
2005 – 2006	Christian Doppler Laboratory *"Compilation Techniques for Embedded Processors"* / OnDemand Microelectronics (paid master's thesis) Design and implementation of a compiler infrastructure for the 24-bit *Ilvy* VLIW Processor, including ports of GCC, Binutils and GDB.
2006	OnDemand Microelectronics (freelance) An implementation of the compiler for the 32-Bit *Chili* VLIW Processor, based on my previous design.
2004 – 2006	Student Assistant (Tutor) at Vienna University of Technology for the courses *Functional Programming* and *Compiler Construction*.

Education

2006 – ongoing	PhD student at Vienna University of Technology Expected graduation in 2010 Thesis topic: *High-level compiler support for timing analysis* Supervisor: Prof. Jens Knoop
2000 – 2006	Computer Science at Vienna University of Technology Graduated with honors as "Dipl.-Ing." (M. Sc. equivalent) Master's Thesis: *Creating a GCC back end for a VLIW-architecture* Supervisor: Prof. Andreas Krall
1992 – 2000	Bundesgymnasium Leoben I (Secondary school) Graduated with honors ("Matura")

Summer Schools

International Summer School on Advances in Programming Languages
25th–28th August, 2009
Heriot-Watt University, Edinburgh, Scotland

Research interests

My research interests include compiler construction, program optimizations, source-to-source transformers, static program analysis, code generators, logic-oriented and functional programming languages, real-time systems and worst-case execution time analysis.

Activities	*Referee* for the following scientific conferences and journals: CC'08, ICS'09, LCTES'09, PACT'09, PLDI'09, JSA. *Co-Editor* of the KPS'09 Proceedings

Research Projects and cooperation

2010 – ongoing	ALL-TIMES: Integrating European Timing Analysis Technology Project funded by the European Commission's 7th Framework Programme on Research, Technological Development and Demonstration under contract № 215068.
2009 – ongoing	Trends in Timing Analysis: Uniform WCET Annotation Language Cooperation with Institut de Recherche en Informatique de Toulouse (IRIT) Bilateral French-Austrian (Amadée) Project
2006 – 2010	CoSTA: Compiler Support for Timing Analysis Project funded by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract № P18925-N13.

Software

TuBound	(main author): A tool for worst-case execution time (WCET) analysis that features source-to-source program transformations and static program analysis. Built upon Termite and SATIrE.
Termite	(main author): A Prolog library to transform and analyze abstract syntax trees of C/C++ programs. `http://www.complang.tuwien.ac.at/adrian/termite`
SATIrE	(co-author): Static analysis tool integration engine. Combines the abstract syntax tree of the ROSE compiler (from LLNL) and the Program Analyzer Generator (from AbsInt) to facilitate source-based static analysis of C/C++ programs. `http://www.complang.tuwien.ac.at/satire`

Publications

[1] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. Beyond loop bounds: Comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 2010. doi: http://dx.doi.org/10.1007/s10270-010-0161-0. (online edition).

[2] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 2010. doi: http://dx.doi.org/10.1007/s11241-010-9091-8. (online edition).

[3] Adrian Prantl. Towards a static profiler. In Jens Knoop and Adrian Prantl, editors, *15. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS '09)"*, number 2009-X-1 in Schriftenreihe des Instituts für Computersprachen, page 230, Maria Taferl, Austria, October 2009. Technische Universität Wien.

[4] Adrian Prantl, Jens Knoop, Raimund Kirner, Albrecht Kadlec, and Markus Schordan. From trusted annotations to verified knowledge. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET 2009)*, pages 39–49, Dublin, Ireland, June 2009. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-252-6.

[5] Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska. Constraint solving for high-level WCET analysis. In *The 18th Workshop on Logic-based methods in Programming Environments (WLPE 2008)*, pages 77–89, Udine, Italy, December 2008.

[6] Adrian Prantl, Markus Schordan, and Jens Knoop. TuBound – A Conceptually New Tool for Worst-Case Execution Time Analysis. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 141–148, Prague, Czech Republic, 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.

[7] Niklas Holsti, Jan Gustafsson, Guillem Bernat (eds.), Clément Ballabriga, Armelle Bonenfant, Roman Bourgade, Hugues Cassé, Daniel Cordes, Albrecht Kadlec, Raimund Kirner, Jens Knoop, Paul Lokuciejewski, Nicholas Merriam, Marianne de Michiel, Adrian Prantl, Bernhard Rieder, Christine Rochange, Pascal Sainrat, and Markus Schordan. WCET Tool Challenge 2008:

Report. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 149–171, Prague, Czech Republic, July 2-4 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.

[8] Raimund Kirner, Albrecht Kadlec, Peter Puschner, Adrian Prantl, Markus Schordan, and Jens Knoop. Towards a common wcet annotation language: Essential ingredients. In *8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, pages 53–65, Prague, Czech Republic, July 2-4 2008. Österreichische Computer Gesellschaft. ISBN: 978-3-85403-237-3.

[9] Adrian Prantl. The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation. In Walter Dosch, Clemens Grelck, and Anette Stümpel, editors, *14. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS '07)"*, number A-07-07 in Schriftenreihe A, pages 172–177. Institute für Informatik und Mathematik der Universität zu Lübeck, 2007.

[10] Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Ingomar Wenzel. WCET Analysis: The Annotation Language Challenge. In Christine Rochange, editor, *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 83–99, Pisa, Italy, 2007. Institut de Recherche en Informatique de Toulouse.

[11] Adrian Prantl. Source-to-Source Transformations for WCET Analysis: The CoSTA Approach. In Michael Hanus and Bernd Braßel, editors, *24. Workshop der „GI-Fachgruppe Programmiersprachen und Rechenkonzepte"*, number 0707 in Technische Berichte des Instituts für Informatik, pages 51–60, Olshausenstr. 40, D – 24098 Kiel, 2007. Institut für Informatik der Christian-Albrechts-Universität zu Kiel.

[12] Adrian Prantl. Creating a GCC back end for a VLIW-architecture. Master's thesis, Vienna University of Technology, May 2006.