Source-Level Support for Timing Analysis [1]

Gergö Barany and Adrian Prantl [2]

Institute of Computer Languages
Vienna University of Technology
email: {gergo, adrian}@complang.tuwien.ac.at

Abstract Timing analysis is an important prerequisite for the design of embedded real-time systems. 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. While actual timings can only be derived from the machine code, many of the supporting analyses (deriving timing-relevant data such as points-to and loop bound information) operate much more effectively on the source code level. At this level, they can use high-level information that would otherwise be lost during the compilation to machine code.

During the optimization stage, compilers often apply transformations, such as loop unrolling, that modify the program's control flow. Such transformations can invalidate information derived from the source code. In our approach, we therefore apply such optimizations already at the source-code level and transform the analyzed information accordingly. This way, we can marry the goals of precise timing analysis and optimizing compilation.

In this article we present our implementation of this concept within the SATIrE source-to-source analysis and transformation framework. SATIrE forms the basis for the TuBound timing analyzer. In the ALL-TIMES EU FP7 project we extended SATIrE to exchange timing-relevant analysis data with other European timing analysis tools. In this context, we explain how timing-relevant information from the source code level can be communicated to a wide variety of tools that apply various forms of static and dynamic analysis on different levels.

[1] This work was supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contracts P18925-N13, *Compiler Support for Timing Analysis (CoSTA)*, http://costa.tuwien.ac.at/ and P21842, *Optimal Code Generation for Explicitly Parallel Processors*, http://www.complang.tuwien.ac.at/epicopt/, and the Commission of the European Union within the 7th EU R&D Framework Programme under contract 215068, *Integrating European Timing Analysis Technology (ALL-TIMES)*, http://www.mrtc.mdh.se/projects/all-times/.

[2] The author is now at Lawrence Livermore National Laboratory, P. O. Box 808, 94551 Livermore, CA.

1 Introduction

Many aspects of our lives are controlled by embedded computer systems with real-time constraints. Embedded real-time systems used in aerospace and automotive applications are especially delicate: Since errors in such computer systems may endanger lives, these systems are deemed safety-critical. Adherence to the specification, both functionally and in non-functional aspects such as timing, is therefore of the utmost importance. Timing analysis aims to predict worst-case timing behavior in order to give feedback to developers and provide information to validation processes.

There are many forms of timing analysis, each having a number of specific advantages and disadvantages. Static analysis predicts timing behavior based on the program's code. This form of analysis is able to cover all possible eventualities; however, static analyses must typically make some simplifying assumptions that coarsen the analysis information, suggesting possible program behaviors that cannot be realized in actual executions. This may lead to overestimations of the worst-case execution time (WCET). Various kinds of static analysis techniques can reduce overestimations, but often at considerable costs in analysis time and memory consumption and, thus, scalability. Wilhelm et al. [WEE+08, pp. 39–41] summarize studies which found that practical static analysis methods overestimate cache miss penalties by 15–30 %, and up to 50 % on more modern and complex architectures.

In contrast to static approaches, dynamic analysis observes actual executions of the program and examines dynamically collected information. Such information can include things such as traces of paths taken through the program, relationships between program paths (such as mutual exclusion), numbers of iterations of loops in the program, and execution times of basic blocks or larger pieces of code. These data points describe actual executions and are therefore very precise. However, the major difficulty of dynamic analysis is that one must ensure that measured data are also safe: The application must be run under conditions and with input data that are capable of eliciting worst-case behavior. Finding such input data can be a very difficult task.

In practice, several forms and levels of analysis can be combined: For example, static analysis may be used in the search for worst-case inputs for subsequent dynamic analysis. Dynamically measured (safe) timings of parts of a program may be combined with statically derived information on worst-case execution frequencies.

This paper provides a high-level overview of past work at Vienna University of Technology's Compilers and Languages group. We focus on the combination of analysis *levels*: The derivation of symbolic auxiliary information at the source-code level and its relevance to timing analysis (Section 3); and communication of the results to tools that perform lower-level timing analysis, usually involving the application binary (Section 4). As far as information about the program's control flow is concerned, source-level information may be invalidated by optimizing compilers during translation. For this reason, we also discuss how to perform

some optimizations at the source-code level and update the flow information accordingly (Section 5).

2 Static Timing Analysis Techniques

The standard approach to static timing analysis consists of three conceptual stages (that need not be implemented separately in actual tools): high-level analysis, low-level analysis, and calculation.

High-level analysis derives information about the control- and data flow of the program. In particular, this involves finding static bounds on the number of iterations of each loop in the program; if it is not known after how many iterations some loop terminates, the WCET must be conservatively estimated to be infinite. Other kinds of flow information may include conflicting paths: For instance, whenever some path π_1 is taken at some branch, it may be impossible to take another path π_2 at a subsequent branch.

Low-level analysis is concerned with finding actual timing information for parts of the program, typically basic blocks. As a very rough approximation, this may consist simply of adding up individual worst-case instruction timings. However, this approach gives much too high numbers on modern processor architectures that feature pipelines and caching mechanisms. Sophisticated low-level analyzers therefore track models of the pipeline and cache states, which allows them to compute a safe approximation of the performance gain due to overlapped execution of instructions as well as cache hits.

Finally, the calculation phase integrates flow information with low-level timings. The common approach is the implicit path enumeration technique (IPET): The entire timing analysis problem is formulated as an integer linear program. Each basic block's execution frequency is represented by a variable in the ILP formulation; flow constraints such as loop bounds and mutual exclusions are expressed as linear inequalities. Basic block timings are encoded in the problem's objective function, which is then maximized using an off-the-shelf ILP solver. The maximal solution gives the worst-case execution time, and the values of the ILP variables give some information about the worst-case path.

Both high-level and low-level analysis may profit from various kinds of auxiliary information: For instance, precise information about pointer targets may both eliminate false paths due to indirect function calls (high-level) and improve the prediction of cache effects (low-level). In the following section, we describe our source-level analyses supporting timing analysis.

3 Source-Level Analyses for Timing Analysis

We use the SATIrE (Static Analysis Tool Integration Engine)[1] framework to perform high-level parts of timing analysis. SATIrE is a source-level analysis and transformation framework that has been under development at Vienna University of Technology since 2004.

[1] http://www.complang.tuwien.ac.at/satire/

3.1 The SATIrE Framework

SATIrE allows users to build tools that analyze or transform C (and, to some extent, C++) programs. Its internal program representation is based on the object-oriented abstract syntax tree (AST) provided by the ROSE[2] source-to-source transformation system. The AST may carry arbitrary annotations as extracted from annotations in the program or computed by program analyzers. The AST is either built by ROSE, which is based on the C and C++ frontend by Edison Design Group[3], or by a modified version of the Clang frontend[4].

Given the AST, there are several ways of analyzing and manipulating the program it represents. ROSE includes a number of analyses and transformations, including a loop optimizer that can perform common operations such as loop unrolling. SATIrE includes a component that builds an interprocedural control flow graph (ICFG) suitable for data-flow analysis. Bindings to the Program Analyzer Generator (PAG)[5] allow the generation of such data-flow analyzers from simple functional specifications. Another component[6] can export and import ASTs represented as Prolog terms. Prolog's symbolic processing capabilities provide excellent tools for the manipulation of tree-shaped data, which makes it a good match for our purposes.

ASTs that have been transformed or annotated with analysis results (in the form of comments or `#pragma` statements) can be unparsed to C source code. This code can then be passed to any other source-based tool or regular C compiler for further processing.

3.2 High-Level Analyses Supporting Timing Analysis

SATIrE allows us to implement a number of program analyses supporting the high-level component of timing analysis. On the source-code level, we can derive information regarding pointer relationships, including function pointer targets; information about the possible ranges of the values of integer variables; and, based on these, tight bounds for the number of iterations of many loops and loop nests. This section describes our implementations of these analyses.

Points-To Analysis. SATIrE includes a flow-insensitive unification-based analysis based on Steensgaard's well-known analysis [Ste96]. The basic analysis performs a single pass over the program, assigning an 'abstract memory location' to each program variable, function, or dynamic memory allocation site. Locations representing structures have special edges to their members' locations. All elements of an array are collapsed into a single summary location. Several distinct objects may be assigned the same location, as described below.

[2] http://www.rosecompiler.org
[3] http://www.edg.com
[4] http://clang.llvm.org
[5] http://www.absint.de/pag/
[6] http://www.complang.tuwien.ac.at/adrian/termite/

The effects of pointer assignments are modeled using points-to edges between locations: There is a points-to edge from location ℓ_1 to location ℓ_2 if at some point during some execution of the program, one of the objects represented by ℓ_1 may hold a pointer value that points to one of the objects represented by ℓ_2.

Each location is constrained to have at most one outgoing points-to edge; if some location might point to two or more different locations, those locations are merged into a new combined location. If merged locations pointed to different locations before, those target locations must also be merged recursively. This merging ensures that the analysis can be implemented in almost-linear time using a fast Union/Find data structure [Tar75]; however, it is also a source of imprecision as it may introduce spurious points-to relations that cannot be realized in any actual run of the program.

In its basic form, the algorithm suffers from imprecision because it is context-insensitive: If a function that receives a pointer argument is called at several different sites, the analysis will merge all the objects that may be pointed to at *any* call site. We made the analysis context-sensitive by analyzing each function several times (once for each context), and linking the analysis data according to the calling structure between contexts. This approach of cloning contexts is similar to Lattner et al.'s context-sensitive points-to analysis [LLA07].

The points-to analysis supports other source-level analyses in SATIrE. However, it is also directly relevant to timing: Precise points-to information can help other tools reduce the number of candidates at indirect call sites; allow better value analysis by reducing the number of candidates for indirect data accesses; and allow better modeling of cache effects.

Value Interval Analysis. SATIrE uses a flow-sensitive interval analysis (or 'value range analysis') to associate each integer variable with a value interval for each location in its scope. If at some point a variable is associated with an interval $[a, b]$, this means that at that point, the variable's value is definitely somewhere between a and b. The interval analysis is implemented as an abstract interpretation [CC77]. The declarative analysis specification is translated to an executable program by PAG.

In certain cases, the analysis can make use of assert statements in the program that were inserted by programmers with domain knowledge, or by some other program analysis/transformation. The information in a statement like `assert(x >= 0 && x <= 10);` can be used by the interval analysis to infer that at the program point following that statement, the value of variable `x` must be in the range $[0, 10]$, regardless of what was known about its value before. SATIrE includes a component that annotates a program with such assertions capturing the results of interval analysis; thus, such assertions are well suited for storing analysis information for later use without full-scale recomputation. These assertions are also useful in testing the analysis itself, and in verifying annotations provided by users or by other tools [PKK+09].

The analysis is integrated with SATIrE's points-to analysis. Integer assignments or reads through pointer expressions can therefore be resolved to sets

of possibly referenced variables. This allows us to avoid some conservative assumptions that would be necessary without points-to information: Indirect reads yield the union of all involved variables' intervals, indirect writes only affect the analysis data associated with the possible pointer targets. Similarly, arrays are modeled as sets of aliased variables.

The interval analysis is inter-procedural, i. e., intervals associated with argument expressions of function calls are propagated into the corresponding functions. For programs that use indirect calls through function pointers, the points-to analysis is consulted to propagate information to and from all possible targets of the given call. Using facilities provided by PAG, the interval analysis can be used in a context-sensitive way with arbitrarily long call strings.

In the context of timing analysis, interval analysis is mostly of interest for the computation of loop bounds (see below). It some cases, it can also identify infeasible paths: Branches on the values of function parameters may be resolved statically (in context-sensitive ways) if the analysis can identify the value ranges of actual arguments.

Loop Bounds Analysis. To implement the TuBound timing analysis tool, SATIrE was extended with a component which computes bounds for loops based on iteration variables [PKST08]. It uses results of the interval analysis and structural information about the program to build equations or set of inequalities, which are solved to yield bounds on the number of loop iterations.

The loop analyzer looks for loops preceded by the initialization of an iteration variable, a loop condition consisting of an inequality involving the variable (or a set of such inequalities connected by 'logical or' operations), and exactly one increment or decrement of the variable inside the loop with a bounded (but not necessarily constant) step size. Assuming the loop variable is i, the initialization expression is *Init*, the test expression is $i < Max$, and the minimum step size *Step* is known to be positive, we can set up an equation like $n = (Max - Init)/Step$ to describe the number of loop iterations.

This expression can be evaluated using interval arithmetic to provide an upper bound. Before numeric evaluation, we also perform a symbolic simplification step that attempts to eliminate common subexpressions between *Max* and *Init*. This allows us to handle some loops that involve unknown quantities, such as the common idiom of iterating over an array using a pointer: `for (p = a; p < a + 10; p++)`. Here, our interval analysis cannot determine an interval for `a`; however, none is needed because after simplification, no occurrences of `a` are left in the loop bound expression.

The above analysis works well for single loops, but it can overestimate nested loops with a triangular or irregular iteration space. We analyze nested loops using more general flow constraints. This analysis works for counting loops as described above, but now we require a constant step size. For each (upwards-counting) loop, we set up a system of inequalities $\{i \geq Init, i \leq Max, (i - Init) \bmod Step = 0\}$.

This translation can be performed recursively for nested loops. The set of distinct integral solutions to the resulting system of inequalities describes the

entire iteration space, i. e., the set of all tuples of values that the iteration variables of the loops can take. The size of this set gives the number of iterations of the innermost loop in the nest. The *clpfd* solver distributed with SWI-Prolog[7] allows efficient computation of the number of solutions without producing them.

Precise data on loop bounds is directly relevant to timing analysis as programs typically spend most of their time in loops, and an overestimation of loop trip counts directly translates into an overestimation of the WCET. Automatic analysis, especially of complex irregular loop nests, is both less time-consuming and less error-prone than manual annotation.

4 Integration of Timing Analysis Tools

This section explains how we integrate the high-level analyses described in the previous section with other timing analysis tools. Such integration is needed because actual timing information cannot be derived at the source code level: An intervening compiler is needed to produce actual machine code. (Compilation to an abstract machine may suffice if a precise timing model of the abstract machine on a given physical machine is available [HBH+07].) Such compilers may be, but need not be, aware of the real-time nature of the software they are compiling.

Here we describe SATIrE's integration with four other tools with different approaches to the WCET analysis problem: Compiler integration; dynamic analysis; static analysis on the binary; flow analysis on a lower-level representation. All of these connections make heavy use of annotation capabilities provided by the respective tools.

The integration with CalcWCET$_{C167}$ was implemented as part of the Austrian CoSTA project, while the other three integrations were part of the ALL-TIMES EU FP7 project. We have working research prototypes for each tool integration.

4.1 Integrated Compilation and WCET Calculation

One compiler designed for integrated compilation and WCET calculation is CalcWCET$_{C167}$ [Kir01] targeting Infineon's C167 family of microcontrollers. This is a modified version of GCC which understands `wcetC`, an extension of C that provides a custom syntax to specify flow constraints and loop bounds in addition to the input program. During code generation, it computes execution times for each basic block it generates; the flow information and basic block timings are used to set up an IPET problem, which is solved using standard techniques. One drawback of this approach is that the compiler is prohibited from performing optimizations that alter the control flow of the program. Section 5 discusses how we can sidestep this problem by performing optimizations on the source-code level.

CalcWCET$_{C167}$'s approach to timing analysis relies on good source-level flow annotations. Without tool support, such annotations must be placed in

[7] http://www.swi-prolog.org

the program by the programmer, which is a tedious and error-prone task. The TuBound tool implemented using SATIrE is able to leverage its loop bounds analysis to compute the necessary information for many loops. Its program transformation capabilities can then be used to insert the annotations in the source code.

4.2 Annotations for Measurement-Based Analysis

RapiTime by Rapita Systems Ltd[8] is a dynamic analysis toolkit. It instruments target applications with measurement code and uses the measurement data to profile performance, provide code coverage information, and perform WCET analysis. As RapiTime uses dynamic analysis to gather information at run-time, one cannot always be sure that all possible executions of certain parts of the code have been covered by its analysis. RapiTime therefore provides the possibility for users to annotate the program's source code with high-level knowledge about issues such as points-to relations or flow constraints. SATIrE can compute some of the relevant information, as detailed below.

In order to compute a worst-case timing for a function call, RapiTime must know all the possible functions that may be called at that site (in a certain context). Since embedded system programs often contain indirect calls through function pointers, this information is typically not immediately available. During the execution of the system, the code instrumented by RapiTime can record all *observed* functions called from a certain site, but as noted above, it may not always be sure that these were all the *possible* call targets for that call site. Without this information, it must make a conservative approximation or reject the program.

SATIrE's points-to analysis statically computes conservative approximations of the sets of targets of each indirect function call. This automatic analysis is much faster and more reliable than manual annotations; this is particularly true for context-sensitive annotations. Thus SATIrE's information can tell RapiTime whether it has observed all possible call targets during its tests, or which other possible targets it must take into account. Similarly, RapiTime may observe certain numbers of iterations for loops in the application. SATIrE's static analysis of loop bounds may confirm that the observed iterations are indeed the worst case, or provide information for appropriate computation of a guaranteed time bound.

Our source-level static analysis thus helps in ensuring the safety of the dynamic analysis, or in proving that a given dynamic analysis result is indeed safe.

4.3 Annotations for Binary-Level Static Analysis

The aiT family of WCET analysis tools from AbsInt Angewandte Informatik GmbH[9] performs static analysis directly on the application binary. Using abstract

[8] http://www.rapitasystems.com
[9] http://www.absint.com

interpretation, aiT derives possible value ranges of registers and memory locations; it also computes upper bounds on the WCET of basic blocks, taking cache and pipelining effects into account. The overall WCET is computed using the IPET approach.

While the analyses of aiT and SATIrE compute some similar information, they have different ways of computing that information. aiT's value analysis subsumes its pointer analysis, treating pointer values simply as integers. A value interval determined for a pointer thus implicitly denotes the set of all objects that could be addressed by that pointer. In contrast, SATIrE's points-to analysis is symbolic, identifying objects by abstract symbols, not by memory addresses. Where a pointer points to non-adjacent functions or global symbols in the program, aiT's analysis will determine that it may also point to any intervening function or object. In such cases, SATIrE's symbolic analysis can derive the possibly much more precise result that the pointer may only reference one of a discrete set of functions or objects, but not any arbitrary memory address in between. This more precise pointer analysis may also add precision to SATIrE's interval analysis in some cases. We express results using aiT's existing annotation mechanism.

aiT's annotation mechanism includes a notion of 'user-defined registers', which are virtual machine registers whose values can be read or written by annotations. In particular, annotations can be made conditional on a user-defined register's value. This allows us to formulate context-sensitive annotations by encoding call string information in virtual registers. Such annotations may, in turn, tighten aiT's timing results computed for certain contexts.

4.4 Integration with Other High-Level Tools

SWEET is a research tool from Mälardalen University's WCET group[10]. Its flow analysis is based on abstract execution [GESL06] and can derive complex flow constraints relating execution frequencies for different points in the program. The integration of SWEET and SATIrE involves various issues. First, as SWEET works on programs represented in the ALF format [GEL+09], it needs translators to ALF in order to analyze source code. The connection between SATIrE and SWEET in the ALL-TIMES project therefore included a C-to-ALF compiler.[11] One advantage of having control over this translator is that it can output useful meta-information besides the ALF code.

The translator therefore outputs information mapping ALF code positions (identified by jump labels) to SATIrE's internal position identifiers as well as to source code locations. It also exports information on the call strings used by SATIrE. Using this information, SATIrE's context-sensitive analysis results regarding points-to information and value intervals can also be communicated to SWEET. In contrast to the other connections described above, SWEET and SATIrE have similar notions of program objects (ALF allows named, scoped variables like C does). Thus SATIrE's analysis information referring to program

[10] http://www.mrtc.mdh.se/projects/wcet/
[11] http://www.complang.tuwien.ac.at/gergo/melmac/

variables and pointer relations is directly useful to SWEET. The tight correspondences between program positions as well as variables allow SATIrE to exchange even flow-sensitive information with SWEET, which is not the case for the other connections. This tight integration can ease the implementation burden on SWEET's developers, who at the time of writing do not have a context-sensitive points-to analysis.

5 Source-Level Optimization and Timing Analysis

As mentioned in Section 4.1, some types of compiler optimizations alter the program's control flow and invalidate flow information annotated at the source-code level. One prominent example for this is loop unrolling: An unrolled loop will perform fewer iterations than expected from the source code, but will accordingly do more work in each iteration. For a loop unrolled by a factor of k, preserving the original annotation will result in an overestimation of the WCET by a similar factor of about k. For this reason, such optimizations must be disabled in the back-end compiler if source-level annotations are to be used. The CalcWCET$_{C167}$ compiler disables all loop optimizations that change the control flow.

However, such optimizations are desired because they can often result in considerable speedups. We have therefore combined source-level optimization with corresponding transformation of source-code annotations. This way, programs can take advantage of loop optimizations at the source level, while such optimizations are still disabled in the back-end. At the same time, annotations remain correct and tight. The result is usually a reduction in the calculated WCET.

5.1 Transformation of Flow Information

We consider flow information represented by a variable f_e for each edge e in the program's control-flow graph. Flow constraints are given as inequalities of linear expressions involving possibly scaled flow variables, which we write as $\langle n \cdot f_e \rangle$. We transform constraints by replacing expressions referring to transformed edges with other expressions [KPP10]. In general, there are two cases to consider:

- The flow f_e at edge e is split into multiple edges e'_i. We replace the corresponding scaled flow variable by appropriately scaled variables for the new edges:
$$\langle n \cdot f_e \rangle \longrightarrow \langle n_1 \cdot f_{e'_1} \rangle + \langle n_2 \cdot f_{e'_2} \rangle + \cdots$$
- The flow of several edges e_i is merged into one edge e'. For each e_i, we perform the following transformations for $\leq$ or $<$ constraints:
$$\langle n \cdot f_{e_i} \rangle \longrightarrow \begin{cases} \langle n_{\text{lhs}} \cdot f_{e'} \rangle & \text{on the left-hand-side of the constraint} \\ \langle n_{\text{rhs}} \cdot f_{e'} \rangle & \text{on the right-hand-side of the constraint} \end{cases}$$
and vice versa for $\geq$ or $>$ constraints.

In either case, the values of the newly introduced scalar factors depend on the details of the program transformation.

Transformation of flow annotations is guided by an optimization trace. The trace, a log of each modification of the control flow, is produced by the loop optimizer. We use it to access data about the original and transformed loops as well as the flow variables involved. Our implementation of optimization traces and the transformation rules can handle the correct transformation of annotations for loop unrolling, loop blocking, loop fusion, and loop interchange.

Original program	Loop-interchanged program′

`for (i = 0; i < 8; ++i) {`
`  // $l_1$`
`  for (j = 0; j < n; ++j) {`
`  // $l_2$`
`  if (even(i))`
`  // then`
`  ...`
`  else ...`
	`for (j = 0; j < n; ++j) {`
	`  // $l_1$`
	`  for (i = 0; i < 8; ++i) {`
	`  // $l_2$`
	`  if (even(i))`
	`  // then`
	`  ...`
	`  else ...`

Loop bounds	**Loop bounds′**
$\langle l_1, 8 \dots 8 \rangle$ $\langle l_2, 1 \dots 4 \rangle$	$\langle l_1, 1 \dots 4 \rangle$ $\langle l_2, 8 \dots 8 \rangle$
Constraints	**Constraints′**
$f_{l_1} \leq 8$ $f_{\text{then}} \leq f_{l_1} \cdot 2$ $f_{l_1} \leq f_{\text{then}} \cdot 2$	$f_{l_2}/4 \leq 8$ $f_{\text{then}} \leq f_{l_2}/1 \cdot 2$ $f_{l_2}/4 \leq f_{\text{then}} \cdot 2$

Figure 1. Example: Transformation of annotations after loop interchange [Pra10]

Example. Figure 1 illustrates the transformation of flow constraints for interchanged loops. Note that each occurrence of f_{l_1} is replaced by the flow variable f_{l_2}, scaled appropriately according to the analyzed lower or upper bounds of iterations of the new outer loop.

5.2 Experimental Evaluation

We evaluated the impact of source-level loop optimization and annotation transformation on the analyzed WCETs of the programs from the WCET benchmark suite from Mälardalen University and the DSPstone benchmark suite. On average, our source-based optimizations succeed in reducing the analyzed WCET by about 13 % on the Mälardalen benchmarks and by about 21 % on DSPstone.

Figure 2 shows results for the standard WCET analysis benchmarks collected by Mälardalen University[12]. In Figure 3 results for the fixed-point version of the DSPstone benchmarks [ŽVSM94] are shown: Using the benchmarks that could be fully analyzed (and automatically annotated at the source code level) by an unassisted TuBound, the diagrams show how the WCET bound is affected by the source-level loop optimizations. Each value is normalized by the WCET bound of the program with just the low-level optimizations applied, marked as the 'original' analyzed WCET in the graphs. The WCET bound calculation was done using the CalcWCET$_{C167}$ back end of TuBound. The source-level loop optimizations use the upper and lower loop bound information found by TuBound. All low-level optimizations performed by the target compiler do not alter the control flow any more. The rightmost bar represents the geometric mean of the scaled execution speed of each of the benchmarks.

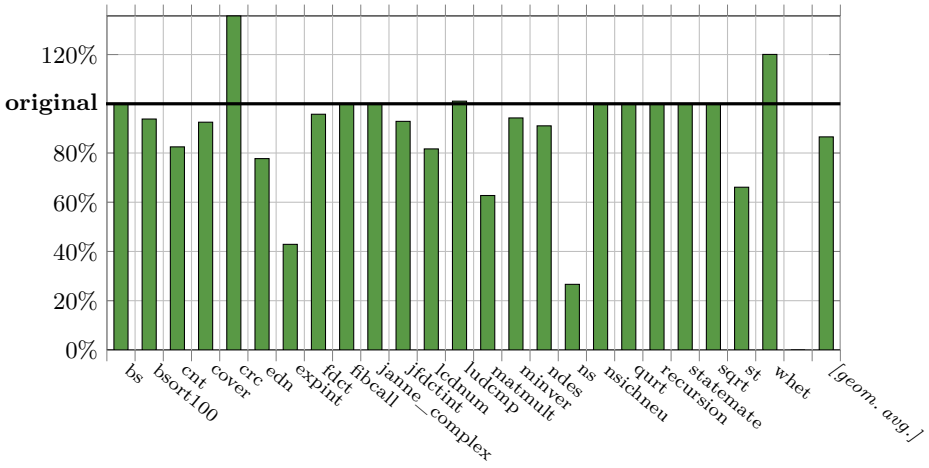

Figure 2. Tighter analyzed WCETs due to high-level loop optimizations: Mälardalen benchmarks

The benchmarks show that the potential for optimizations is significant. Unsurprisingly, the implemented optimizations (loop unrolling, fusion, interchange, splitting), perform well on DSP kernels (e. g., `matrix2`) and show little to no impact on branch-intensive code (e. g., `nsichneu`) that lacks tight loops with a high trip-count.

Outliers like `crc` 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].

[12] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

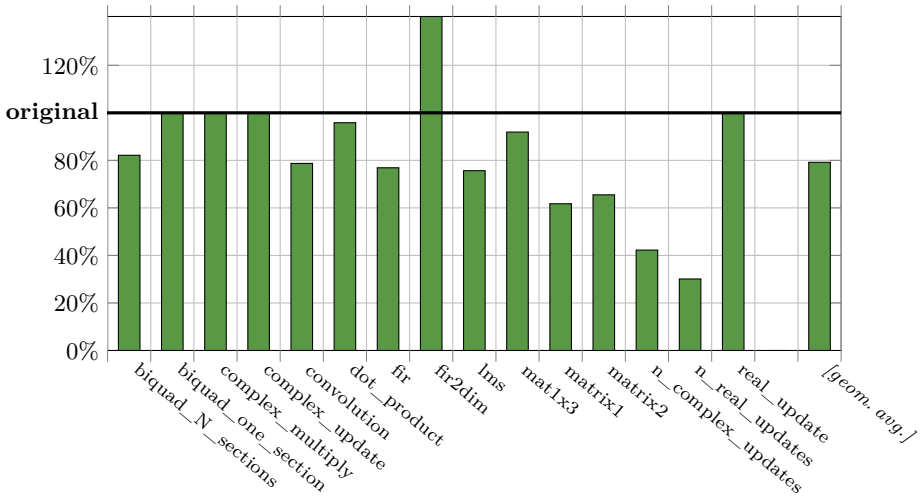

Figure 3. Tighter analyzed WCETs due to high-level loop optimizations: DSPstone benchmarks

6 Related Work

Wilhelm et al. [WEE+08] present a thorough discussion of the worst-case execution time problem and various tools and methods to approach it. Gustafsson et al. [GLS+08] give a more detailed overview of the ALL-TIMES project and the tools involved. Schordan [Sch08] presents the SATIrE system in more detail and considers some challenges of annotating source code with analysis information.

There does not appear to be much previous work that deals specifically with integration of source code analysis and WCET calculation. This paper summarizes and unifies several threads of work in this area performed at Vienna University of Technology in between 2006 and 2010 within the ALL-TIMES and CoSTA projects. Prantl et al. discuss the TuBound tool in much more detail [PSK08,Pra10], while Barany [Bar09] gives more details on the integration of SATIrE in the ALL-TIMES project.

Schulte [Sch07] as well as Engblom et al. [EEA98] discuss the transformation of flow annotations along with control-flow altering program optimizations.

Herrmann et al. [HBH+07] use a (conceptual) virtual machine as an intermediate step for combining high-level and low-level analysis of programs written in the functional language Hume: For each target machine, timings of abstract machine instructions are derived using aiT. Source-level analysis and knowledge of compiler internals allows their framework to determine the set of abstract machine instructions that would be generated for each input program. Straightforward composition of this information with the low-level timings yields a WCET bound.

7 Conclusions

We have presented our approach to supporting timing analysis on the source code level using the SATIrE analysis framework. Using this framework, we built the TuBound tool for WCET analysis, combining source-level analysis, source-level optimization, and a back-end compiler performing WCET analysis. We have also implemented connections to several other timing analysis tools of various kinds, demonstrating that source-level analysis information can be useful for a wide range of timing-related analyzers that cover different analysis approaches.

Acknowledgements. The authors would like to thank Viktor Pavlu and Alexander Jordan for many helpful comments on earlier versions of this article. We are grateful to the anonymous reviewers for their comments that helped us improve the paper's focus and presentation.

References

[Bar09] Gergö Barany. SATIrE within ALL-TIMES: Improving timing technology with source code analysis. In *15. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS '09)*, page 230, Maria Taferl, Austria, Oct. 2009.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[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.

[GEL+09] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*, June 2009.

[GESL06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *The 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, December 2006.

[GLS+08] Jan Gustafsson, Björn Lisper, Markus Schordan, Christian Ferdinand, Marek Jersak, and Guillem Bernat. ALL-TIMES - a European project on integrating timing technology. In *Proc. Third International Symposium on Leveraging Applications of Formal Methods (ISOLA'08)*, pages 445–459. Springer, October 2008.

[HBH+07] Christoph A. Herrmann, Armelle Bonenfant, Kevin Hammond, Steffen Jost, Hans-Wolfgang Loidl, and Robert Pointon. Automatic amortised worst-case execution time analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2007.

[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/.

[KPP10] Raimund Kirner, Peter Puschner, and Adrian Prantl. Transforming flow information during code optimization for timing analysis. *Real-Time Systems*, 45(1–2):72–105, June 2010.

[LLA07] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 278–289, New York, NY, USA, 2007. ACM.

[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.

[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.

[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.

[Pra10] Adrian Prantl. *High-level compiler support for timing analysis*. PhD thesis, Vienna University of Technology, 2010.

[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.

[Sch07] Daniel Schulte. Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler. Master's thesis, Universität Dortmund, 2007.

[Sch08] Markus Schordan. Source-to-source analysis with SATIrE - an example revisited. In *Scalable Program Analysis*, number 08161 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[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.

[Tar75] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[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.