TUBOUND –A CONCEPTUALLY NEW TOOL FOR WORST-CASE EXECUTION TIME ANALYSIS [1]

Adrian Prantl,[2] Markus Schordan[2] and Jens Knoop[2]

Abstract

TUBOUND *is a conceptually new tool for the worst-case execution time (WCET) analysis of programs. A distinctive feature of* TUBOUND *is the seamless integration of a WCET analysis component and of a compiler in a uniform tool.* TUBOUND *enables the programmer to provide hints improving the precision of the WCET computation on the high-level program source code, while preserving the advantages of using an optimizing compiler and the accuracy of a WCET analysis performed on the low-level machine code. This way,* TUBOUND *ideally serves the needs of both the programmer and the WCET analysis by providing them the interface on the very abstraction level that is most appropriate and convenient to them.*

In this paper we present the system architecture of TUBOUND*, discuss the internal work-flow of the tool, and report on first measurements using benchmarks from Mälardalen University.* TUBOUND *took also part in the WCET Tool Challenge 2008.*

1. Motivation

Static WCET analysis is typically implemented by the implicit path enumeration technique (IPET) [15, 19] 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, because of the undecidability of certain analysis problems or imprecision of the analyses. Current WCET analysis tools, as they are used by the industry, therefore allow the user to annotate the machine code with flow constraints.

The goal of the TuBound approach is to lift the level of user annotations from machine code to source code, while still performing WCET analysis on the machine code level. In addition to keeping the precision of low-level WCET analysis, this has the following benefits:

- *Convenience and Ease*: For the user, annotating the source code is generally easier and less demanding as 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.

[1]This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, *Compiler Support for Timing Analysis*, `http://costa.tuwien.ac.at/`, the ARTIST2 Network of Excellence, `http://www.artist-embedded.org/`, and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

[2]Institut für Computersprachen, Vienna University of Technology, Austria
email: {adrian,markus,knoop}@complang.tuwien.ac.at

- *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.

The major obstacle, which has to be overcome for realizing such an approach, is imposed by the fact that compiler optimizations can modify the control-flow of a program and thus invalidate source code annotations. In TUBOUND, this is taken care of by transforming flow constraints according to the performed optimizations. Technically, this is achieved by a special component, called FLOWTRANS, which is a core component of TUBOUND and described in Section 3.2. FLOWTRANS performs source-to-source transformations. Therefore, our overall approach is retargetable to other WCET tools; currently we are using CALCWCET$_{167}$.

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. Since the output of a source-based analysis is again annotated source code, it is also possible to create a feedback loop where the user can run the static analysis and fill in the annotations where the analysis failed to produce satisfying results. Afterwards, the analysis could be rerun with the enriched annotations to produce even tighter estimates.

TUBOUND is based on earlier work by Kirner [14] who formulates the correct flow constraint updates for common compiler transformations. TUBOUND goes beyond this approach by extending it to source-to-source transformations and by adding interprocedural analysis. Optimization traces for flow constraint transformations are also used by Engblom et al. [8]. With FLOWTRANS, we are taking this concept to a higher level, by performing control-flow altering transformations already at the source code level. Another approach towards implementing flow constraint transformation was recently described by Schulte [22]. In contrast to TUBOUND, this approach is based on the low-level intermediate representation of the compiler. The integration of static flow analysis and low-level WCET analysis is also implemented in the context of SWEET, which uses a technique called abstract execution to analyse loop bounds [9, 10]. Again, our approach uses a higher level of abstraction by performing static analyses directly at the source code level. The interaction of compiler optimizations and the WCET of a program has been covered by Zhao et al. [24], where feedback from a WCET analysis was used to optimize the worst-case paths of a program.

2. The architecture of TuBound

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

The architecture and work flow of TUBOUND is summarized in Figure 1. The connecting glue between the components is the *Static Analysis Tool Integration Engine* (SATIrE) [20, 6]. SATIrE enables using data flow analyzers specified with the *Program Analyzer Generator* (PAG) [16, 3] together with the C++ infrastructure of the ROSE compiler [21]. SATIrE internally transforms programs into its own intermediate representation, which is based on an abstract syntax tree (AST). 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 even line and column information of the respective expressions. The terms are also annotated with the results of any preceding static analy-

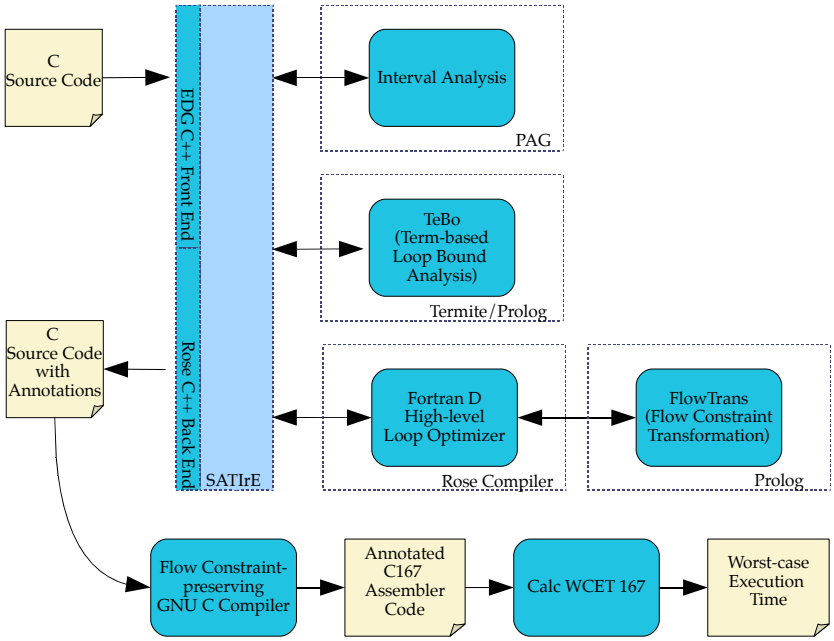

Figure 1. The collaboration of TUBOUND's components

C source code	Term representation

```
7  for (i = 0; i < 100; i++) {
```

```
for_statement(
  for_init_statement( [ expr_statement( assign_op(
    var_ref_exp(
      var_ref_exp_annotation(type_int,"i",0,
                             null,analysis_result(null,null)),
      file_info("triang.c",7,10)),
    int_val(null,value_annotation(0,analysis_result(null,null)),
            file_info("triang.c", 7, 12)),
    ... ], default_annotation(null, analysis_result(null,null)),
                        file_info("triang.c", 7, 3)),
    expr_statement( less_than_op(
      var_ref_exp(var_ref_exp_annotation(type_int,"i",0,null,
        ...
```

Figure 2. The external AST term representation of SATIrE

sis. The key feature, however, is the syntax of the term representation. It was designed to match the syntax of Prolog terms. A Prolog program can thus access and manipulate these terms 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 [2].

The ROSE compiler is a source-to-source transformation framework that includes the EDG C++ front end, a loop optimizer and a C++ unparser [21, 5]. 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. The Program Analyzer Generator (PAG) by AbsInt Angewandte Informatik GmbH allows the specification of data flow analyses in a specialised functional language [16, 3]. Using PAG, we implemented a variable interval analysis for TUBOUND. CALCWCET$_{167}$is a tool that performs WCET analysis for the Infineon C167 micro-controller [4]. CALCWCET$_{167}$ 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.

3. The Work Flow of TuBound

Conceptually, the work flow of analysing a program with TUBOUND comprises three stages:

3.1. 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. ROSE then creates a C++ data structure of the AST and performs consistency checks to verify its integrity. The ROSE loop optimizer performs analysis and transformations based on the AST data structure.

Interval Analysis. The AST is traversed by SATIrE to generate the interprocedural control flow graph (ICFG), an amalgam of call graph and *intra*procedural CFG [23]. This data structure is the interface for the PAG-based interval analysis that calculates the possible variable value ranges at all program locations. The context-sensitive 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 pre-process of the loop bounding algorithm, which is otherwise unable to analyze iteration counts that depend on variable values that stem from different calling contexts. 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 term-based loop bounder (TEBO) which was 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 would be $lb = (buf + 8 - buf)/1$. The right-hand side expression can then be reduced by TEBO's term rewriting rules. The loop bounding algorithm also ensures that the iteration variable is not modified inside the loop body. This is implemented with a control flow-insensitive analysis [17] 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 TEBO, we are using constraint logic programming to yield generalized flow constraints that describe the iteration space more accurately. An example is shown in Figure 3. The nested loop in the example has a triangular iteration space, where the innermost basic block is executed $n * \frac{n-1}{2}$ times. Our analyzer finds the following equation system for this loop nest:

$$
\begin{array}{lll}
m3 & = \sum_{n=0}^{99} m3_n(\{i := n\}) & (1) \\
m3_n(env) & = n = i & (2) \\
m2 & = m1 * 100 & (3)
\end{array}
$$

The equations are constructed with the help of an *environment* that consists of the assignments of variables at the current iteration. The variable $m1$ stands for the execution count of the `main()` function, $m2$ for the count of the outer loop and $m3$ for the count of the innermost loop. Equation 1 describes the fact that the values of i as well as the iteration counts for the individual runs of the inner loop are $0..99$, respectively. Equation 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. The last equation describes the behaviour of the outer loop. The use of constraint logic programming allows for a lightweight

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 3. Finding flow constraints with constraint logic programming

implementation that does not rely on additional tools. In earlier work, Healy et al. [11] are using analysis data to feed an external symbolic algebra system that solves the equation systems for loop bounds.

Eventually, the results of the loop bound analysis are inserted into the term representation as annotations of the source code. We are using 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 mechanism to limit the area of validity of markers which allows us to express relationships that are local to a sub-graph of the ICFG.

3.2. Program Optimization and WCET Annotation Transformation

The FLOWTRANS phase deals with program sources which are already annotated by flow constraints. These can stem from either an earlier analysis pass or from a human. Flow constraints describe the control flow of the program in order to reduce the search space for feasible paths. These constraints, however, can be invalidated in the course of the compilation process by the application of optimizations that modify the control flow. This applies to optimizations such as loop unrolling, loop fusion and inlining, whereas optimizations such as constant folding and strength reduction do not affect the control flow. In order to ensure validity of the flow constraints throughout the compilation, a naive approach would be to disable control-flow modifying optimizations. This, however, would sacrifice the performance of the compiled code. As a part of TuBound, we thus implemented the FLOWTRANS component, a transformation framework for flow constraints which transforms the annotations according to the optimizations applied.

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 of FLOWTRANS 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

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_restriction(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_restriction(
  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_restriction(
  m_f_1_1_1+m_f_1_1_2=<m_f_1_1/2)
      a[1 + i]++;
    }
  }
  return a;
}
```

Figure 4. Prolog terms everywhere: WCET constraints before and after loop unrolling

any existing compiler. The rules need to be written only once per optimization. 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 4 gives an example of such a transformation. We currently implemented rules for 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 [18].

3.3. Compilation and WCET calculation

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 [13]. This compiler is a customized version of GCC 2.7.2 which can parse WCETC and guarantees the preservation of all flow constraints at the C167 machine language level. The output of the GCC is annotated assembler code.

WCET Calculation. CALCWCET$_{167}$ reads the annotated assembler code that is produced by the GCC and generates the control flow graph of every function. CALCWCET$_{167}$ 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 also be associated with the original source code.

4. Measurements

To demonstrate the practicality of our approach, we use a selection of benchmarks that were collected by the Real-Time Research Center at Mälardalen University [1]. For our experiments we selected those benchmarks that can be analysed by TUBOUND without annotating the sources manually. Figure 5 shows the time spent in the different phases of TUBOUND and the estimated WCET for a subset

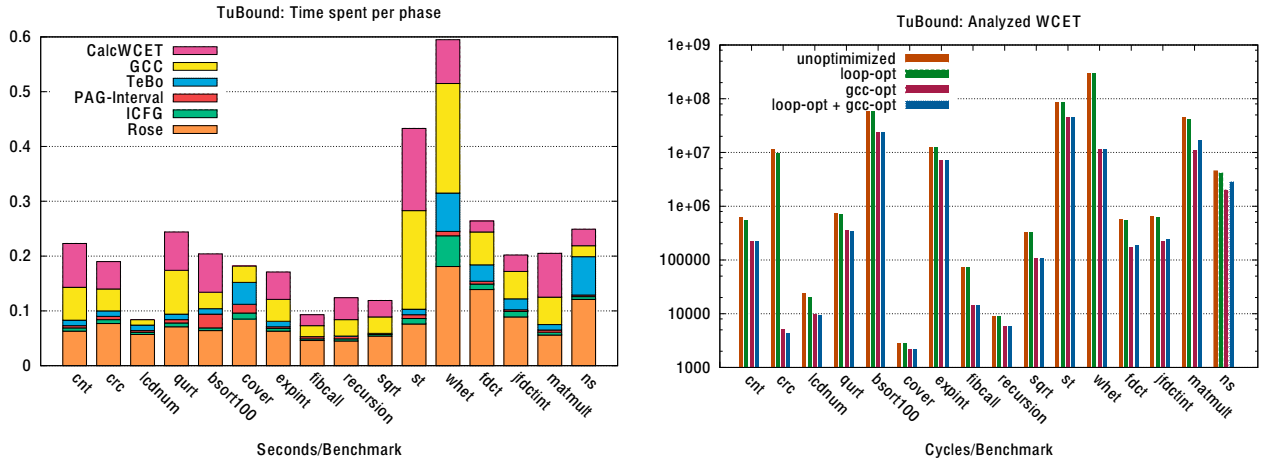

Figure 5. Analysis runtime (left) and analyzed WCET (right) of the selected benchmarks

of benchmarks. It must be noted that a large part (about 45% for the `ns` benchmark) of the time spent in TEBO is currently used to read and parse the term representation from one file and write it to another. This bottleneck can be eliminated by directly generating the data structure via the foreign function interface of the Prolog interpreter process and thus eliminating the expensive parsing and disk I/O. On the right-hand side of Figure 5 the influence of compiler optimizations on the WCET of the benchmarks can be seen, where the different bars per benchmark denote the analyzed WCET of the unoptimized program vs. the program with high-level and/or low-level optimizations turned on. Note that the y-axis uses a logarithmic scale. From the results, three different groups can be observed:

Group 1: cnt, crc, lcdnum, qurt
Group 2: bsort100, cover, expint, fibcall, recursion, sqrt, st, whet
Group 3: fdct, jfdctint, matmult, ns

In the first group, the calculated WCET is always lower for the loop-optimized code. In the second group, the WCET is the same, regardless of loop optimizations. In the third group, the WCET of the loop-optimized program is better than that of the unoptimized program, however, if both kinds of optimizations are enabled, they interfere and less well performing code is generated, which is reflected by the higher WCET. One reason for this is extra spill code that is generated due to higher register pressure.

5. Conclusion

TUBOUND is a WCET analysis tool which is unique for combining the advantage of low level WCET analysis with high level source code annotations and optimizing compilation. The flow constraint transformation framework FLOWTRANS ensures that annotations are transformed according to the optimization trace as provided by the high-level optimizer. This approach allows us to close the gap between source code annotations and machine-specific WCET analysis. TUBOUND took also part in the WCET Tool Challenge 2008 [7], the results of which are published in [12].

Acknowledgements. We would like to thank Raimund Kirner for his support in integrating his tool CALCWCET$_{167}$ and Albrecht Kadlec for many related discussions.

References

[1] Benchmarks for WCET Analysis collected by Mälardalen University. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[2] The JTransformer framework. http://roots.iai.uni-bonn.de/research/jtransformer/.

[3] The program analyzer generator PAG. http://www.absint.com/pag/.

[4] The CALCWCET$_{167}$ tool. http://www.vmars.tuwien.ac.at/~raimund/calc_wcet/.

[5] The ROSE Compiler. http://www.rosecompiler.org/.

[6] The static analysis tool integration engine SATIrE. http://www.complang.tuwien.ac.at/markus/satire/.

[7] The WCET tool challenge 2008. http://www.mrtc.mdh.se/projects/WCC08/.

[8] J. Engblom, P. Altenbernd, and A. Ermedahl. Facilitating worst-case execution time analysis for optimized code. In *Proceedings of the 10th EuroMicro Workshop on Real-Time Systems, Berlin, Germany*, June 1998.

[9] Jan Gustafsson, Andreas Ermedahl, and Björn Lisper. Towards a flow analysis for embedded system C programs. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2005)*, pages 287–297, Feb 2005.

[10] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *27th IEEE Real-Time Systems Symposium (RTSS'06)*, Feb 2006.

[11] 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.

[12] Niklas Holsti, Jan Gustafsson, and Guillem Bernat (eds). WCET tool challenge 2008: Report. In *Proceedings 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)*, Prague, 2008.

[13] Raimund Kirner. The programming language WCETC. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[14] 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.

[15] 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.

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

[17] Steven S. Muchnik. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[18] Adrian Prantl. The CoSTA Transformer: Integrating Optimizing Compilation and WCET Flow Facts Transformation. In Proceedings 14. Kolloquium „Programmiersprachen und Grundlagen der Programmierung (KPS'07)". Technical Report, Universität zu Lübeck, 2007.

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

[20] 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.

[21] 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.

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

[23] Micha Sharir and Amir Pnueli. Two approaches to inter-procedural data-flow analysis. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[24] Wankang Zhao, William C. Kreahling, David B. Whalley, Christopher A. Healy, and Frank Mueller. Improving WCET by Optimizing Worst-Case Paths. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 138–147, 2005.