From Trusted Annotations to Verified Knowledge*

Adrian Prantl,Jens Knoop,Raimund Kirner,Albrecht Kadlec, and Markus Schordan

Vienna University of Technology, Institute of Computer Languages, Austria,
{adrian,knoop}@complang.tuwien.ac.at
Vienna University of Technology, Institute of Computer Engineering, Austria,
{raimund,albrecht}@vmars.tuwien.ac.at
University of Applied Sciences Technikum Wien, Austria,
schordan@technikum-wien.at

Abstract. WCET analyzers commonly rely on user-provided annotations such as loop bounds, recursion depths, region- and program constants. This reliance on user-provided annotations has an important drawback. It introduces a Trusted Annotation Basis into WCET analysis without any guarantee that the user-provided annotations are safe, let alone sharp. Hence, safety and accuracy of a WCET analysis cannot be formally established. In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds. Fundamental to our approach is to apply model checking in concert with other more inexpensive program analysis techniques, and the coordinated application of two algorithms for *Binary Tightening* and *Binary Widening*, which control the application of the model checker and hence the computational costs of the approach. Though in this paper we focus on the control of model checking by Binary Tightening and Widening, this is embedded into a more general approach in which we apply an array of analysis methods of increasing power and computational complexity for proving or disproving relevant time bounds of a program. First practical experiences using the sample programs of the Mälardalen benchmark suite demonstrate the usefulness of the overall approach. In fact, for most of these benchmarks we were able to empty the trusted annotation base completely, and to tighten the computed WCET considerably.

* This paper has been published in the Proceedings of the 9th Int'l Workshop on Worst-Case Execution Time Analysis (WCET'09). This work has been supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) within the research project "Compiler-Support for Timing Analysis" (CoSTA) under contract No P18925-N13 and the research project "Integrating European Timing Analysis Technology" (ALL-TIMES) under contract No 215068 funded by the 7th EU R&D Framework Programme.

1 Motivation

The computation of loop bounds, recursion depths, region- and program constants is undecidable. It is thus commonly accepted that WCET analyzers rely to some extent on user-assistance for providing bounds and constants. Obviously, this is tedious, complex, and error-prone. State-of-the-art approaches to WCET analysis thus provide for a fully automatic preprocess for computing required bounds and constants using static program analysis. This unburdens the user since his assistance is reduced to bounds and constants, which cannot automatically be computed by the methods employed by the preprocess. Typically, these are classical data-flow analyses for constant propagation and folding, range analysis and the like, which are particularly cost-efficient but may fail to verify a bound or the constancy of a variable or term. WCET analyzers then rely on user-assistance to provide the missing bounds which are required for completing the WCET analysis. This introduces a *Trusted Annotation Base* (*TAB*) into the process of WCET analysis. The correctness (safety) and optimality (tightness) of the WCET analysis depends then on the safety and tightness of the bounds of the TAB provided by the user.

In this paper we propose a uniform approach, which reduces the trusted annotation base to a minimum, while simultaneously yielding sharper (tighter) time bounds.

Figure 1 illustrates the general principle of our approach. At the beginning the entire annotation base that is added by the user where static analysis fails to establish the required information, is assumed and trusted to be correct, thus we call it *Trusted Annotation Base* (*TAB*). Using model checking we aim to verify as many of these user-provided facts as possible. In this process we shrink the trusted fraction of the annotation base and establish a growing verified annotation base. In Figure 1 the current state in this process is visualized as the horizontal bar. In our approach we are lowering this bar, representing the decreasing fraction of trust to an increasing fraction of verified knowledge, and thus transfer *trusted user-belief* into *verified knowledge*.

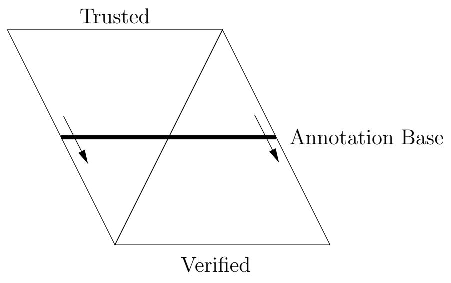

Fig. 1. The annotation base: shrinking the trusted annotation base and establishing verified knowledge about the program

2 Shrinking the Trusted Annotation Base – Sharpening the Time Bounds

2.1 Shrinking the Trusted Annotation Base

The automatic computation of bounds by the preprocesses of up-to-date 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 practice, we often succeed to empty it completely.

A key observation is that a user-provided bound – which the preprocessing analyses were unable to compute – can not be checked by them either. Hence, verifying the correctness of the corresponding user annotation in order to move it *a posteriori* from the trusted annotation base to the verified knowledge base requires another, more powerful and usually computationally more costly approach. For example, there are many algorithms for the detection of copy constants, linear constants, simple constants, conditional constants, up to finite constants detecting different classes of constants at different costs [7]. This provides evidence for the variety of available choices for analyses using the example of constant propagation and folding. 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* [3, 6]. Like 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 are now available off-the-shelf. The Blast [1] and the CBMC [2] model checkers are two prominent examples. In our experiments reported in Section 4 we used the CBMC model checker.

The following example demonstrates the ease and elegance of using a model checker to verify a loop bound, which we assume could not be automatically bounded by the program analyses used. The program fragment on the left-hand side of Figure 2 shows a loop together with a user-provided annotation of the loop. The program on the right-hand side shows the transformed program which is presented to CBMC to verify or refute the user-provided annotation shown in the program on the right-hand side:

In this example the CBMC model checker comes up with the answer "yes," i.e., the loop bound provided by the user is safe; allowing thus for its movement from the trusted to the verified annotation base. If, however, the user were to provide $\_\_bound \leq 3$ as annotation, model checking would fail and produce a

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */


  while(low <= up){
#pragma wcet_trusted_loopbound(7)
    mid = low + up >> 1;
    if (data[mid].key == x) { /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;


  }


  return fvalue;
}
```

$\longrightarrow$

```
int binary_search(int x) {
  int fvalue, mid, low = 0, up = 14;
  fvalue = (-1); /* all data are positive */

  unsigned int __bound = 0;
  while(low <= up){

    mid = low + up >> 1;
    if (data[mid].key == x) { /* found  */
      up = low - 1;
      fvalue = data[mid].value;
    }
    else if (data[mid].key > x)
    up = mid - 1;
    else low = mid + 1;

    __bound += 1;
  }
  assert(__bound <= 7);

  return fvalue;
}
```

Fig. 2. Providing loop bound annotations for the model checker

counter example as output. Though negative, this result would still be most valuable. It allows for preventing usage of an unsafe trusted annotation base in a subsequent WCET analysis. Note that the counter example itself, which in many applications is the indispensable and desired output of a failed run of a model checker, is not essential for our application. It might be useful, however, to present it to the user when asking for another candidate of a bound, which can then be subject to *a posteriori* verification in the same fashion until a safe bound is eventually found.

Next we introduce a more effective approach to come up with a safe and even tight bound, if so 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.

2.2 Sharpening the Time Bounds

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. In particular, this will hold for user-provided bounds. In order to exclude channeling an unsafe bound into the trusted annotation base, the user will generously err on the side of caution when providing a bound. This suggests the following iterative approach to tighten the bound, which is an application of the classical pattern of a 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: call procedure *binaryTightening* with the interval $[0..b_0]$ as argument, where $binaryTightening([low..high])$ is defined as follows:

```
1. Let $m = \lceil \frac{low+high}{2} \rceil$.
2. ModelCheck($m$ is a safe bound):
3.    yes:   $low = m$: **return** $m$
               $low = m - 1$: ModelCheck($low$ is a safe bound)
                                   yes: **return** $low$     no: **return** $m$
               otherwise: $binaryTightening([low..m])$
4.    no:    $high = m$: **return** $false$
               $high = m + 1$: ModelCheck($high$ is a safe bound)
                                   yes: **return** $high$     no: **return** $false$
               otherwise: $binaryTightening([m..high])$
```

Obviously, *binaryTightening* terminates. If it returns *false*, a safe bound tighter than that of the initial bound b_0 could not be established. Otherwise, i.e., if it returns value b, this value is the least safe bound. This means b is tight. If it is smaller than b_0, we succeeded to sharpen the bound.

Binary widening described next allows for proceeding in the case where 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 Binary widening is dual to binary tightening. Its functioning is inspired by the risk-aware gambler playing roulette, 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, whatever is lower, prevent this strategy to work out in reality. Nonetheless, the idea of an externally given limit yields the inspiration for the *Binary Widening* algorithm to avoid looping if no safe bound exists. A simple approach is to limit the number of recursive calls of binary widening to a predefined maximum number. The version of binary widening we present below uses a different approach. It comes up with a safe bound, if one exists, and terminates, if the size of the bound is too big to be reasonable, or does not exist at all. This directly corresponds to the limit set by a casino to a maximum bet.

Let b_0 be an arbitrary number, $b_0 \geq 1$, and let max be the maximum value for a safe bound considered reasonable. Then: Call procedure $binaryWidening$ with b_0 and max as arguments, where $binaryWidening(b, limit)$ is defined as follows:

```
1. if $b > limit$: **return** false
2. ModelCheck($b$ is a safe bound):
3.    yes: **return** $b$
4.    no:  $binaryWidening(2 * b, limit)$
```

Obviously, $binaryWidening$ terminates.[1] If it returns $false$, at most a bound of a size considered unreasonably big exists, if at all. Otherwise, i.e., if it returns value b, this value is a safe bound. The ratio 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 more sophisticatedly, by using application dependent information, e.g., such as a coarse estimate of the execution time of a single execution of the loop and a limit on the overall execution time this loop shall be allowed for.

Coordinating Binary Tightening and Widening Once a safe bound has been determined using binary widening, binary tightening can be used to compute the uniquely determined safe tight bound. Because of the exponential resp. logarithmic behaviour in the selection of 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 in our WCET analyzer TuBound, as described in Section 3. The results of practical experiments we conducted with the prototype implementation are promising. They are reported in Section 4.

3 Implementation within TuBound

3.1 TuBound

TuBound [9] is a research WCET analyzer tool working on a subset of the C++ language. It is unique for uniformly combining static program analysis, optimizing compilation and WCET calculation. Static analysis and program optimization are performed on the abstract syntax tree of the input program. TuBound is built upon the SATIrE program analysis framework [12] and the TERMITE program transformation environment.[2] TuBound features an array of algorithms for loop analysis of different accuracy and computation cost including sophisticated analysis methods for nested loops. A detailed account of these methods can be found in [8].

3.2 Implementation

The Binary Widening/Tightening algorithms are implemented by means of a dedicated TERMITE source-to-source transformer T. For simplicity and uniformity we assume that all loops are structured. In our implementation unstructured goto-loops are thus transformed by another source-to-source transformer

[1] In practice, 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.

[2] `http://www.complang.tuwien.ac.at/adrian/termite`

T' into while-loops beforehand, where possible. On while-loops the transformer T works by locating the first occurrence of a `wcet_trusted_loopbound(N)` annotation in the program source and then proceeding to rewrite the encompassing loop as illustrated in the example of Figure 3.[3] Surrounding the loop statement,

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

  counter_decl('__bound', ..., CounterDecl),
  counter_inc('__bound', ..., Count),
  counter_assert('__bound', N, ..., CounterAssert),

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

Fig. 3. Excerpt from the source-to-source transformer T

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 (cf. Figure 2).

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 a sharper, now verified bound. Figure 4 shows a snapshot of processing the *janne_complex* benchmark. In this figure, the status and value changes are highlighted by different colors.

4 Experimental Results

We implemented our approach as an extension of the TuBound WCET analyzer 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, which took part in the 2008 WCET Tool Challenge [5], later on called the basic version of TuBound. In the spirit of the WCET Tool Challenge [4, 5] we do encourage authors of other WCET analyzers to carry out similar experiments.

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

```
...
int complex(int a, int b)
{
 while(a < 30) {
#pragma wcet_trusted_loopbound(30)
   while(b < a) {
#pragma wcet_trusted_loopbound(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;
}
...
```

Containing two trusted loop annotations

→

```
...
int complex(int a, int b)
{
  while(a < 30) {
#pragma wcet_loopbound(16)
   {
    unsigned int __bound = 0U;
    while(b < a){
#pragma wcet_trusted_loopbound(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 <= 30U);
  }
  a = a + 2;
  b = b - 10;
 }
 return 1;
}
...
```

Outer loop annotation verified and tightened, inner being checked

Fig. 4. Illustrating trusted bound verification and tightening

Our experiments conducted were guided by two questions: "Can the number of automatically bounded loops be significantly increased?" 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 $2^{13} + 1$. The "compress" and "whet" benchmarks contained unstructured goto-loops; as indicated in Section 3.2 these were automatically converted into do-while loops beforehand by a source-to-source transformation.

Our findings are summarized in Table 1. Column three of this table shows the percentage of loops that can be bounded by the basic version of TuBound; column four shows the total percentage of loops the extended version of TuBound was able to bound. The last column shows the accumulated runtime of the model checker for the remaining loops.

Comparing columns three and four reveals the superiority of the extended version of TuBound over its basic variant. The extended version succeeds to bound 67% of the loops the basic version could not bound.

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

Benchmark	Loops	TuBound basic	with Model Checking	Runtime
bs	1	0.0%	100.0%	0.03s
duff	2	50.0%	50.0%	0s
fft1	11	54.5%	81.8%	0.43s
janne_complex	2	0.0%	100.0%	0.18s
minver	17	94.1%	100.0%	0.06s
nsichneu	1	0.0%	100.0%	5.59s
qsort-exam	6	0.0%	66.6%	0.02s
statemate	1	0.0%	100.0%	0.06s
whet	11	90.9%	90.9%	0s
adpcm	18	83.3%	83.3%	timeout
compress	8	25.0%	25.0%	timeout
fir	2	50.0%	50.0%	timeout
insertsort	2	0.0%	0.0%	timeout
lms	10	60.0%	60.0%	timeout
select	4	0.0%	0.0%	timeout
bsort100	3	100.0%	100.0%	–
cnt	4	100.0%	100.0%	–
cover	3	100.0%	100.0%	–
crc	3	100.0%	100.0%	–
edn	12	100.0%	100.0%	–
expint	3	100.0%	100.0%	–
fdct	2	100.0%	100.0%	–
fibcall	1	100.0%	100.0%	–
jfdctint	3	100.0%	100.0%	–
lcdnum	1	100.0%	100.0%	–
ludcmp	11	100.0%	100.0%	–
matmult	5	100.0%	100.0%	–
ndes	12	100.0%	100.0%	–
ns	4	100.0%	100.0%	–
qurt	1	100.0%	100.0%	–
sqrt	1	100.0%	100.0%	–
st	5	100.0%	100.0%	–
recursion	0	–	–	–
Total Percentage		77.0%	84.7%	

Table 1. Results for the Mälardalen benchmarks

invalidated by introducing a faux dependency on e.g. `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.

The fully-fledged variant of our approach, which we highlight in the next section is tailored towards this goal.

5 Extensions: The Fully Fledged Approach

The shrinking of the trusted annotation base and sharpening of time bounds, as described in Section 2, is based on model checking. Based on our experience, we believe that the model checking approach can be especially valuable in the real world when (i) it is combined with advanced program slicing techniques to reduce the state space and (ii) the results of static analyses (like TuBound's variable-interval analysis) are used to narrow the value ranges of variables, thus regaining a feasible problem size. This leads to the following extension of our approach to improve efficiency:

1. By using a pool of analysis techniques with different computational complexity: As shown in Figure 5, model checking is considered as one of the most complex analysis methods. On the other side, techniques like *constant propagation* or *interval analysis* are relatively fast. Thus we are interested in exploiting the fast techniques wherever beneficial and using the relatively complex techniques rarely.
2. By using a smart activation mechanism for the different analysis techniques: As shown on the right of Figure 5 we are interested in the interaction of the different analysis techniques. We do not aim to use the pool of analysis techniques in waves of different complexity, i.e., first applying the fast techniques and then gradually shifting towards the more complex techniques. Instead we aim for a smart interaction of the different analysis techniques. For example, whenever a technique with relatively high computational complexity has been applied, we can again apply techniques of relatively simple complexity to compute the closure of flow information based on previously obtained results; thus *squeezing* the annotation base.

We also think that profiling techniques are useful to guide the heuristics to be used within our static analysis techniques. For example, execution samples obtained by profiling can be used to elicit propositions to be verified by model checking.

The fully fledged approach envisioned in this section provides the promising potential as a research platform for complementing program analysis techniques.

6 Conclusions

Model checking has been used before in the context of WCET analyzers. Examples of our own related work are the ForTAS [13], MoDECS [14], and ATDGEN

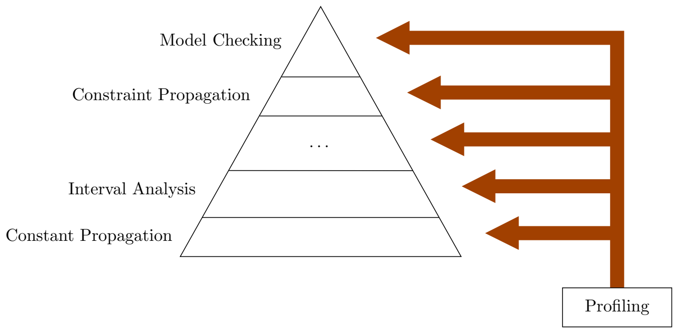

Fig. 5. Pool of complementary analysis techniques with different complexity

projects [10, 11], which are concerned with measurement-based WCET analysis. In these three projects, model checking is used to generate test data for the execution of specific program paths. Intuitively, in these applications the model checker is presented with formulae stating that a specific program path is infeasible. If these formulae can be refuted by the model checker, the counter examples generated provide the test data ensuring the execution of the particular paths. Otherwise, the paths are known to be infeasible. Hence, the search for test data is in vain. In these applications the counter examples generated in the course of failing model checker runs are the truly desired output, whereas successful runs are of less interest stopping just the search for test data for the path under consideration. This is in contrast and opposite to our application of shrinking the trusted annotation base. In our application, the counter example of a failed model checker run is of little interest. We are interested in successful runs of the model checker as they allow us to change a *trusted annotation* into *verified knowledge*. This opens a new application domain for model checking in the field of WCET analysis. Our preliminary practical results demonstrate the practicality and power of this approach.

References

1. BEYER, D., HENZINGER, T., JHALA, R., AND MAJUMDAR, R. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT) 9*, 5-6 (October 2007), 505–525.
2. CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 168–176.
3. DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems 19*, 6 (1997), 992 – 1030.
4. GUSTAFSSON, J. The WCET tool challenge 2006. In *Preliminary Proc. 2nd International IEEE Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Paphos, Cyprus, November 2006), pp. 248 – 249.

5. HOLSTI, N., GUSTAFSSON, J., (EDS.), G. B., BALLABRIGA, C., BONENFANT, A., BOURGADE, R., CASSÉ, H., CORDES, D., KADLEC, A., KIRNER, R., KNOOP, J., LOKUCIEJEWSKI, P., MERRIAM, N., DE MICHIEL, M., PRANTL, A., RIEDER, B., ROCHANGE, C., SAINRAT, P., AND SCHORDAN, M. WCET Tool Challenge 2008: Report. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)* (Prague, Czech Republic, July 2008), Österreichische Computer Gesellschaft, pp. 149–171. ISBN: 978-3-85403-237-3.
6. HORWITZ, S., REPS, T., AND SAGIV, M. Demand interprocedural dataflow analysis. In *Proc. 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering* (*FSE-3*) (1995), pp. 104 – 115.
7. MUCHNICK, S. S. *Advanced Compiler Design & Implementation.* Morgan Kaufmann Publishers, Inc., 1997. ISBN 1-55860-320-4.
8. PRANTL, A., KNOOP, J., SCHORDAN, M., AND TRISKA, M. Constraint solving for high-level wcet analysis. In *Proc. 18th International Workshop on Logic-based methods in Programming Environments (WLPE 2008)* (Udine, Italy, December 2008), pp. 77–89.
9. PRANTL, A., SCHORDAN, M., AND KNOOP, J. TuBound – a conceptually new tool for worst-case execution time analysis. In *Proc. 8th International Workshop on Worst-Case Execution Time Analysis (WCET 2008)* (Prague, Czech Republic, July 2008), Österreichische Computer Gesellschaft, pp. 141–148. ISBN: 978-3-85403-237-3.
10. RIEDER, B. *Measurement-Based Timing Analysis of Applications written in ANSI-C.* PhD thesis, Technische Universität Wien, Vienna, Austria, 2009 (forthcoming).
11. RIEDER, B., WENZEL, I., AND PUSCHNER, P. Using model checking to derive loop bounds of general loops within ANSI-C applications for measurement-based WCET analysis. In *Proc. 6th International Workshop on Intelligent Solutions in Embedded Systems (WISES 2008)* (Regensburg, Germany, July 2008).
12. SCHORDAN, M. Source-To-Source Analysis with SATIrE – an Example Revisited. In *Proceedings of Dagstuhl Seminar 08161: Scalable Program Analysis* (April 2008), Germany, Dagstuhl.
13. VIENNA UNIVERSITY OF TECHNOLOGY AND TU DARMSTADT. The ForTAS project. Web page (`http://www.fortastic.net`). Accessed in June 2009.
14. WENZEL, I., KIRNER, R., RIEDER, B., AND PUSCHNER, P. Measurement-based timing analysis. In *Proc. 3rd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation* (Porto Sani, Greece, October 2008).