Practical Experiments with Symbolic Loop Bound Computation for WCET Analysis[1]

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr[2]

Vienna University of Technology

Abstract. We present an automatic method for computing tight upper bounds on the iteration number of special classes of program loops. These upper bounds are further used in the WCET analysis of programs. Program flow is refined using SMT reasoning, rewriting multi-path loops into single-path ones. Single-path loops are further translated into a set of recurrence relations over program variables. To solve recurrences, we deploy pattern-based technique, instantiating precomputed closed forms for a restricted class of recurrence equations. In practice, these recurrences describe the behavior of a large set of program loops relevant in the WCET community. Our technique is implemented in the r-TuBound tool and was successfully tried out on a number of WCET benchmarks.

1 Introduction

The *worst case execution time (WCET)* analysis of programs aims at deriving an accurate time limit, called the WCET of the program, ensuring that all possible program executions terminate within the computed time limit. One of the main difficulties in WCET analysis comes with the presence of loops, as the WCET crucially depends on the number of loop iterations. To overcome this limitation, state-of-the-art WCET analysis tools, see e.g. [4, 12, 6], rely on user-given program assertions describing loop bounds. Manual annotations are however a source for imprecision and errors. The effectiveness of WCET analysis thus crucially depends on whether bounds on loop iterations can be deduced automatically.

In this paper we address the problem of *automatically inferring iteration bounds* of imperative program loops and *evaluate the proposed technique on a number of relevant benchmarks*. We identified special classes of loops with assignments and conditionals, where updates over program variables are linear expressions (Sections 3.1-3.3). For such loops, we deploy recurrence solving and theorem proving techniques and automatically derive tight iteration bounds, as follows.
(i) A loop with multiple paths arising from conditionals is first transformed into a loop with only one path (Section 3.3). We call a loop with multiple paths, respectively a single path, a multi-path loop, respectively a single-path loop. To this end, the control flow of the multi-path loop is analyzed and refined using *satisfiability modulo theory (SMT) reasoning over arithmetical expressions* [9, 1]. The resulting single-path loop soundly over-approximates the multi-path loop: the iteration bound is a safe bound of the multi-path loop. (ii) A simple loop is next rewritten into a set of *recurrence equations* over those scalar variables that are changed at each loop iteration (Section 3.1). To

[1] This research was partly supported by Dassault Aviation.
[2] The second author is supported by an FWF Hertha Firnberg Research grant (T425-N23).

this end, a new variable denoting the loop counter is introduced and used as the summation variable. Thus, the recurrence equation of the loop iteration variable captures the dependency between various iterations of the loop. (iii) Recurrence equations of loop variables are solved and the values of loop variables at arbitrary loop iterations are computed as functions of the loop counter and the initial values of loop variables (Section 3.1). (iv) Closed forms of loop variables together with the loop condition are used to express the value of the loop counter as a function of loop variables (Section 3.1). The upper bound on the number of loop iterations is derived by computing the *smallest value of the loop counter* such that the loop is terminated. The inferred iteration bound is further used to calculate an accurate WCET of the program loop.
Solving arbitrary recurrence equations is undecidable, however, in our approach we only consider loops with linear updates, yielding linear recurrences with constant coefficients (C-finite [2]). As C-finite recurrences can always be solved, our method succeeds in computing the closed forms of loop variables. We deploy a *pattern-based recurrence solving algorithm*, instantiating unknowns in the closed form pattern of C-finite recurrences by the symbolic constant coefficients of the recurrence to be solved. Unlike algorithmic combinatorics techniques that can solve complex recurrences, our framework only solves a particular class of recurrence equations. However, in WCET analysis the recurrences describing the iteration behavior of program loops are not arbitrarily complex and many can be solved by our approach (Section 4).

This paper summarizes material that was presented at the 8th Ershov Informatics Conference 2011 [8], the Workshop on Worst Case Execution Time [7] and findings from the Worst Case Execution Time Tool Challenge 2011 [13]. The paper is structured as follows. Section 2 presents a motivating example, followed by a detailed explanation of the techniques in Section 3. Section 4 presents in detail the results of the evaluation of the method on a number of benchmark programs.

2 Motivating Example

Consider the C program in Figure 1. Between lines 5-21, the method `func` iterates over a two-dimensional array `a` row-by-row[1], and updates the elements of `a`, as follows: In each visited row `k`, the array elements in columns 1, 4, 13, and 53 are set to 1 according to the C-finite update of the simple loop from lines 6-9. Note that the number of visited rows in `a` is conditionalized by the non-deterministic assignment from line 2. Depending on the updates made between lines 6-9, the multi-path loop from lines 10-14 conditionally updates the elements of `a` by -1. Finally, the abrupt termination of the multi-path loop from lines 15-18 depends on the updates made throughout lines 6-14.

Computing an accurate WCET of the `func` method requires tight iteration bounds of the four loops between lines 5-21. The difficulty in computing the number of loop iterations comes with the presence of the non-deterministic initialization and shift updates of the loop from lines 5-21; the use of C-finite updates in the simple loop from lines 6-9; the conditional updates of the multi-path loop from lines 10-14; and the presence of abrupt termination in the multi-path loop from lines 15-18.

We overcome these difficulties as follows.

[1] We denote by `a[k][j]` the array element from the `k`th row and `j`th column of `a`.

```
1  void func() {
2    int i = nondet();
3    int j, k = 0;
4    int a[32][100];
5    for (; i > 0; i = i >> 1) {
6      for (j = 1; j < 100; j = j * 3 + 1) {
7        a[k][j] = 1;
8        wcet_loopbound(4)
9      }
10     for (j = 0; j < 100; j++) {
11       if (a[k][j] == 1) j++;
12       else a[k][j] = -1;
13       wcet_loopbound(100)
14     }
15     for (j = 0; j < 100; j++) {
16       if (a[k][j] != -1 && a[k][j] != 1)
17         break;
18       wcet_loopbound(100)
19     }
20     k++;
21     wcet_loopbound(32)
22 }}
```

Fig. 1. C program annotated with the result of loop bound computation.

- We apply a pattern-based recurrence solving algorithm (Section 3.1) to infer the loop bound from lines 6-9 to be precisely 4.
- We deploy SMT reasoning to translate multi-path loops into simple ones (Section 3.3) via flow refinement. The iteration bounds of the loops from lines 10-14, respectively lines 15-18, are both over-approximated to 100.
- We over-approximate non-deterministic initializations (Section 3.2). As a result, the upper bound of the loop from lines 5-21 is derived to be 31.

3 Symbolic WCET Analysis

In our approach to derive tight loop bounds, we identify special classes of loops and infer upper bounds on loop iterations, as follows. We apply a pattern-based recurrence solving algorithm to get bounds for simple loops with linear arithmetic updates (Section 3.1) and over-approximate non-deterministic initializations (Section 3.2). We translate multi-path loops with abrupt termination and monotonic conditional updates into simple loops (Section 3.3). The computed loop bounds are further used to obtain accurate WCET of programs.

3.1 Recurrence Solving in Simple Loops

We identified a special class of loops with linear updates and conditions, as below.

$$\begin{array}{l}\texttt{for}\,(\texttt{i} = \texttt{a};\ \texttt{i} < \texttt{b};\ \texttt{i} = \texttt{c} * \texttt{i} + \texttt{d})\,;\\ \text{where } a, b, c, d \in \mathbb{K} \text{ do not depend on } i, \text{ and } c \neq 0.\end{array} \quad (1)$$

Hence, in loops matching (1), loop iteration variables (i.e. i) are bounded by symbolic constants and updated by linear expressions over iteration variables. Throughout this paper, $\mathbb{N}$ and $\mathbb{R}$ denote the set of natural and real numbers, respectively, $\mathbb{K}$ is a field of characteristic zero (e.g. $\mathbb{R}$).

For a loop like (1), we derive an *exact* upper bound on the number of loop iterations: (i) We first model the loop iteration update as a recurrence equation over a new variable $n \in \mathbb{N}$ denoting the loop counter. To do so, we write $i(n)$ to denote the value of variable

i at the nth loop iteration. The recurrence equation of i corresponding to (1) is given below.

$$i(n+1) = c * i(n) + d \quad \text{with the initial value} \quad i(0) = a. \tag{2}$$

Note that (2) is a C-finite recurrence of order 1 as variable updates of (1) are linear.
(ii) Next, recurrence (2) is solved and the closed form of i as a function over n is derived. Depending on the value of c, the closed form of $i(n)$ is given below.

$$\begin{array}{lcl} i(n) \;=\; \alpha * c^n + \beta, \;\text{ if } c \neq 1 & & i(n) \;=\; \alpha + \beta * n, \;\text{ if } c = 1 \\ \text{where} & \text{and} & \text{where} \\ \begin{cases} \alpha + \beta & = a \\ \alpha * c + \beta & = a * c + d \end{cases} & & \begin{cases} \alpha & = a \\ \alpha + \beta & = a + d \end{cases} \end{array} \tag{3}$$

(iii) The closed form of $i(n)$ is further used to derive a tight upper integer bound on the number of loop iterations of (1). To this end, we are interested in finding the value of n such that the loop condition holds at the nth iteration and is violated at the $n+1$th iteration. We are thus left with computing the (smallest) positive integer value of n such that the below formula is satisfied:

$$n \in \mathbb{N} \;\wedge\; i(n) < b \;\wedge\; i(n+1) \geq b. \tag{4}$$

The smallest n derived yields a tight upper bound on the number of loop iterations. This upper bound is further used in the WCET analysis of programs containing a loop matching (1).

For automatically inferring exact loop bounds for (1) we designed a *pattern-based recurrence solving* algorithm which is not based on a computer algebra system (CAS). Our method relies on the crucial observation that in our approach to WCET analysis we do not handle arbitrary C-finite recurrences. We only consider loops matching the pattern of (1), where updates describe C-finite recurrences of order 1. Closed forms of such recurrences are given in (3). Therefore, to compute upper bounds on the number of loop iterations of (1) we do not deploy the general C-finite recurrence solving algorithm given in [2], but instantiate the closed form pattern (3). In other words, whenever we encounter a loop of the form (1), the closed form of the iteration variable is derived by instantiating the symbolic constants $a, b, c, d, \alpha, \beta$ of (3) with the concrete values of the loop under study. Hence, we do not make use of general purpose C-finite recurrence solving algorithms, but handle loops (1) by pattern-matching C-finite recurrences of order 1. However, our approach can be further extended to handle loops with more complex linear updates than those in (1).

Finally let us make the observation that while reasoning about (1), we consider the iteration variable i and the symbolic constants $a, b, c, d, \alpha, \beta$ to have values in $\mathbb{K}$. That is, when solving recurrences of (1) the integer variables and constants of (1) are safely approximated over $\mathbb{K}$. However, when deriving upper bounds of (1), we restrict the satisfiability problem (4) over integers. If formulas in (4) are linear, we obtain a tight loop bound satisfying (4).

3.2 Non-deterministic Analysis in Shift-Loops

We call a loop a *shift-loop* if updates over the loop iteration variables are made using the bit-shift operators $\ll$ (left shift) or $\gg$ (right shift). Let us recall that the operation $i \ll m$ (respectively, $i \gg m$) shifts the value of i left (respectively, right) by m bits.

Consider a shift-loop with iteration variable i, where i is shifted by m bits. Hence, updates over i describe a C-finite recurrence of order 1 and upper bounds on the number of loop iterations can be derived as described in Section 3.1, whenever the initial value of i is specified as a symbolic constant. However, the initial value of i might not always be given or derived by interval analysis. A possible source of such a limitation can, for example, be that the initialization of i uses non-deterministic assignments, as given in (5)(a) and (b).

$$\texttt{for}\,(\texttt{i} = \texttt{nondet}(); \texttt{i} > \texttt{d};\ \ \texttt{i} \gg\!= \texttt{m})\,; \qquad\qquad \texttt{for}\,(\texttt{i} = \texttt{nondet}(); \texttt{i} < \texttt{d};\ \ \texttt{i} \ll\!= \texttt{m})\,;$$

(a) (b) (5)

where $d \in \mathbb{K}$ does not depend on i and $m \in \mathbb{N}$.

For shift-loops matching pattern (5), the method presented in Section 3.1 would thus fail in deriving loop upper bounds. To overcome this limitation, we proceed as below. (i) We soundly approximate the non-deterministic initial assignment to i by setting the initial value of i to be the integer value that allows the maximum number of shift operations within the loop (i.e. the maximum number of loop iterations). To this end, we distinguish between left and right shifts as follows. If i is updated using a right-shift operation (i.e. (5) (a)), the initial value of i is set to be the maximal integer value, yielding a maximum number of shifts within the loop. The initial value of i is hence assumed to have the value 2147483647, that is $010\ldots0$ in a 32-bit binary representation (the most significant, non-sign, bit of i is set). If i is updated using a left-shift operation (i.e. (5)(b)), we assume the initial value of i to be the integer value resulting in the maximum number of shift operations possible: i is set to have the value 1, that is $0\ldots01$ in a 32-bit binary representation (the least significant bit of i is set). (ii) The upper bound on the number of loop iterations is then obtained by first computing the difference between the positions of the highest bits set in the initial value of i and d, and then dividing this difference by m. If no value information is available for m, we assume m to be 1.

3.3 Flow Analysis on Multi-path Loops

Paths of multi-path loops can interleave in a non-trivial manner. Deriving tight loop upper bounds, and thus accurate WCET, for programs containing multi-path loops is therefore a challenging task.

In our approach to WCET analysis, we identified special classes of *multi-path loops with only conditionals* which can be translated into simple loops by refining the control flow of the multi-path loops. Loop bounds for the obtained simple loops are further derived as described in Sections 3.1 and 3.2, yielding thus loop bounds of the original multi-path loops. In what follows, we describe in more detail what type of multi-path loops our approach can automatically handle and overview the flow analysis techniques deployed in our work. For simplicity reasons, in the rest of the paper we consider multi-path loops (arising from conditionals) with only 2 paths.

```
for (j = 0; j < 100; j++)
 if (nondet())  break ;
```

Fig. 2. Loop with abrupt termination.

```
for (j = 0; j < 100; j++)
 if (nondet()) j++;
```

Fig. 3. Loop with monotonic conditional update.

Loops with abrupt termination. One class of multi-path loops that can automatically be analyzed by our framework is the class of linearly iterating loops with abrupt termination arising from non-deterministic conditionals, as given in (6) (a)–(c).

```
int i = a;                      int i = nondet();          int i = nondet();
for (; i < b; i = c * i + d)    for (; i > e; i >>= m)     for (; i < e; i <<= m)
  if (nondet()) break;            if (nondet()) break;       if (nondet()) break;     (6)
           (a)                            (b)                        (c)
```

where $a, b, c, d, e \in \mathbb{K}$ do not depend on i, $c \neq 0$, and $m \in \mathbb{N}$.

We are interested in computing the *worst case* execution time of a loop from (6). Therefore, we safely over-approximate the number of loop iterations of (6) by assuming that the abruptly terminating loop path is not taken. In other words, the non-deterministic conditional statement causing the abrupt termination of (6) is ignored, and we are left with a simple loop as in (1) or (5). Upper bounds on the resulting loops are then computed as described in Sections 3.1 and 3.2, from which an over-approximation of the WCET of (6) is derived.

Loops with monotonic conditional updates. By analyzing the effect of conditional statements on the number of loop iterations, we identified the class of multi-path loops as listed in (7).

```
for (i = a; i < b; i = c * i + d)
  if (B) i = f1(i);                                                    (7)
  else  i = f2(i);
```

where $a, b, c, d \in \mathbb{K}$ do not depend on i, $c \neq 0$, B is a boolean condition over loop variables,

and: $\begin{cases} f_1, f_2 : \mathbb{K} \to \mathbb{K} \text{ are monotonically increasing functions, if } c > 0 \\ f_1, f_2 : \mathbb{K} \to \mathbb{K} \text{ are monotonically decreasing functions, if } c < 0 \end{cases}$.

We refer to the assignment $i = f_1(i)$ as a *conditional monotonic assignment (or update)*, as its execution depends on the truth value of B.

Let $g : \mathbb{K} \to \mathbb{K}$ denote the function $i \mapsto c * i + d$ describing the linear updates over i made at *every* iteration of a loop matching (7). Note that the monotonic behavior of g depends on c and coincides with the monotonic properties of f_1 and f_2. To infer loop upper bounds for (7), we aim at computing the *worst case* iteration time of (7). To do so, we ignore B and transform (7) into a single-path loop by *safely over-approximating* the multi-path behavior of (7), as given below. In what follows, let $\Delta = |g(i+1) - g(i)|$, $\Delta_1 = |f_1(i+1) - f_1(i)|$, and $\Delta_2 = |f_2(i+1) - f_2(i)|$, where $|x|$ denotes the absolute value of $x \in \mathbb{K}$. (i) If c is positive, let $m = \mathtt{min}\{\Delta + \Delta_1,\ \Delta + \Delta_2\}$. That is, m captures the minimal value by which i can be increased during an arbitrary iteration of (7). Alternatively, if c is negative, we take $m = \mathtt{max}\{\Delta + \Delta_1,\ \Delta + \Delta_2\}$. That is, m captures the maximal value by which i can be decreased during an arbitrary iteration of

(7). (ii) Loop (7) is then over-approximated by the simple loop (8) capturing the worst case iteration time of (7).

$$\begin{cases} \texttt{for}\,(\mathtt{i}=\mathtt{a};\mathtt{i}<\mathtt{b};\{\mathtt{i}=\mathtt{c}*\mathtt{i}+\mathtt{d};\mathtt{i}=\mathtt{f_1(i)}\}), & \text{if } c>0 \text{ and } m=\Delta+\Delta_1 \\ \texttt{for}\,(\mathtt{i}=\mathtt{a};\mathtt{i}<\mathtt{b};\{\mathtt{i}=\mathtt{c}*\mathtt{i}+\mathtt{d};\mathtt{i}=\mathtt{f_2(i)}\}), & \text{if } c>0 \text{ and } m=\Delta+\Delta_2 \\ \texttt{for}\,(\mathtt{i}=\mathtt{a};\mathtt{i}<\mathtt{b};\{\mathtt{i}=\mathtt{c}*\mathtt{i}+\mathtt{d};\mathtt{i}=\mathtt{f_1(i)}\}), & \text{if } c<0 \text{ and } m=\Delta+\Delta_1 \\ \texttt{for}\,(\mathtt{i}=\mathtt{a};\mathtt{i}<\mathtt{b};\{\mathtt{i}=\mathtt{c}*\mathtt{i}+\mathtt{d};\mathtt{i}=\mathtt{f_2(i)}\}), & \text{if } c<0 \text{ and } m=\Delta+\Delta_2 \end{cases} \tag{8}$$

Hence, the control flow refinement of (7) requires checking the arithmetical monotonicity constraints of f_1 and f_2. We automatically decide this requirement using arithmetical SMT queries. (iii) We are finally left with computing loop upper bounds of (8). To this end, we need to make additional constraints on the monotonic functions f_1 and f_2 of (7) so that the approach of Section 3.1 can be applied. Namely, we restrict f_1 (respectively, f_2) to be a linear monotonic function $i \mapsto u * i + v$, where $u, v \in \mathbb{K}$ do not depend on i and $u \neq 0$. As linear monotonic functions are closed under composition, updates over the iteration variable i in (8) correspond to C-finite recurrences of order 1. Upper bounds on loop iterations of (8) can thus be derived as presented in Section 3.1.

Note that the additional constraints imposed over f_1 and f_2 restrict our approach to multi-path loops with linear conditional updates.

4 Experimental Evaluation

The approach of [8] is implemented in the r-TuBound software tool. r-TuBound extends the TuBound framework [11, 6] by an SMT-based control flow refinement approach and a pattern-based recurrence solving algorithm. In this section we report on experimental results obtained by applying r-TuBound on a number of challenging examples. These examples include (i) WCET benchmarks, such as the Mälardalen and Debie benchmark suite [5], (ii) scientific benchmarks, such as the SciMark2 benchmark suite, and (iii) industrial benchmarks using a number of examples sent by Dassault Aviation.

In our experiments, we aimed at the automated derivation of loop bounds. In the following tables, column 1 ("BM") denotes the name of the benchmark program, column 2 gives the number of loops in the corresponding file ("#L"). Column 3 lists how many of those loops were successfully analyzed by TuBound ("TB"), whereas column 4 lists how many loops were analyzed by the symbolic loop bound computation technique of r-TuBound("r-TB"). Column 5 ("G") shows which loop pattern could only be handled by r-TuBound[2]. To this end, we distinguish between simple loops with C-finite updates (CF), shift-loops with non-deterministic initializations (SH), multi-path loops with abrupt termination (AT), and multi-path loops with monotonic conditional updates (CU). Some loops require combinations of the proposed techniques, for example, multi-path loops with C-finite conditional updates and abrupt termination; such loops are listed as CF-CU-AT.

Altogether, we ran r-TuBound on 4 different benchmark suites, on a total of 393 loops and derived loop bounds for 296 loops. Out of these 296 loops, 286 loops were simple and involved only C-finite reasoning, and 10 loops were multi-path loops which

[2] Subtracting column 5 from column 4 yields the number of simple loops with constant increments/decrements (i.e. $c = 1$ in (1))

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
class	2	2	2	SH	array	6	5	6	AT
debie	1	0	0	-	fft	8	4	5	CF
harness	45	34	34	-	kernel	9	4	4	-
health	11	9	10	CU	montecarlo	1	1	1	-
hw_if	3	2	2	-	random	4	4	4	-
measure	6	4	4	-	scimark	0	0	0	-
tc_hand	3	3	3	-	sor	3	3	3	-
telem	4	4	4	-	sparsecomprow	3	3	3	-
sum	75	58	59	1	stopwatch	0	0	0	-
					sum	34	24	26	2

Table 1. Evaluation of r-TuBound on the Debie benchmarks (left) and SciMark2 (right).

required the treatment of abrupt termination and conditional updates. TuBound could handle 284 simple loops only. We note that, 75% of the 393 loops were successfully analyzed by r-TuBound, whereas TuBound succeeded on 72% of the 393 loops. When compared to TuBound, the overall quality of loop bound analysis within r-TuBound has increased by 3% (72% to 75%). However, TuBound already performed well on Mälardalen and Debie, and therefore the increase given by r-TuBound is only of 1% (78% to 79% in Mälardalen and 77% to 78% in Debie). For the SciMark2 and the examples from Dassault, the increase in performance given by r-TuBound is of 6% (70% to 76%) and 9% (50% to 59%), respectively. The practical importance of r-TuBound can thus be better evidenced on examples with more complex arithmetic and/or control-flow.

5 Conclusion

We describe an automatic approach for deriving iteration bounds for loops with linear updates and refinable control flow. Our method implements a pattern-based recurrence solving algorithm, applies SMT reasoning to refine program flow, and safely over-approximates non-deterministic initializations. The inferred loop bounds are used in the WCET analysis of programs. Our approach succeeded in generating tight iteration bounds for large number of loops from relevant WCET benchmarks.

In the line of [10], we plan to extend the recurrence solving algorithm in r-TuBound to analyze loops with more complex update-arithmetic. We also intend to integrate our approach with flow refinement techniques based on invariant generation, such as in [3].

References

1. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. of TACAS*, pages 174–177, 2009.
2. G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2003.
3. S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Proc. of PLDI*, pages 292–304, 2010.
4. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. In *Proc. of RTSS*, pages 57–66, 2006.

BM	#L	TB	r-TB	G	BM	#L	TB	r-TB	G
adpcm	18	17	17	-	all_zeros	3	1	1	-
bs	1	0	0	-	array_ptr	3	3	3	-
bsort100	3	3	3	-	asm_memset2	2	1	1	-
cnt	4	4	4	-	behavior	1	1	1	-
cover	3	3	3	-	b_s_o	1	0	0	-
crc	3	3	3	-	break	3	1	1	-
duff	2	1	1	-	bresenham	1	1	1	-
edn	12	12	12	-	bsearch	1	0	0	-
expint	3	3	3	-	bts0041-bis	3	3	3	-
fibcall	1	1	1	-	bts0041	3	3	3	-
janne_c	2	0	0	-	continue	3	0	3	CU,CU,CU
nsichneu	1	0	0	-	copy	1	0	0	-
insertsort	2	0	0	-	count_bits	1	0	0	-
jfdctint	3	3	3	-	dillon4	1	1	1	-
matmult	5	5	5	-	division	1	0	0	-
ns	4	4	4	-	dowhile	1	1	1	-
qurt	1	1	1	-	fs253	1	0	0	-
select	4	0	0	-	fs256	1	1	1	-
statemate	1	0	0	-	fs350	1	1	1	-
sqrt	1	1	1	-	ghost_label	1	1	1	-
fft1	11	6	6	-	heap	2	0	2	AT, CF-AT-CU
lms	10	6	6	-	heapsort	3	1	2	CF-AT-CU
whet	11	11	11	-	inv_perm_minimal	2	1	1	-
ludcmp	11	11	11	-	loop_eq	1	0	0	-
compress	7	2	3	CF	loop_inv	1	0	1	CU
fir	2	1	1	-	malloc	1	1	1	-
minver	17	16	16	-	minimum_sort	2	2	2	-
qsort_exam ...	6	0	1	AT	min_sort ...	2	2	2	-

5. N. Holsti, J. Gustafsson, G. Bernat, C. Ballabriga, A. Bonenfant, R. Bourgade, H. Cassé, D. Cordes, A. Kadlec, R. Kirner, J. Knoop, P. Lokuciejewski, N. Merriam, M. de Michiel, A. Prantl, B. Rieder, C. Rochange, P. Sainrat, and M. Schordan. WCET 2008 - Report from the Tool Challenge 2008 - 8th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis. In *Proc. of WCET*, 2008.
6. R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec. Beyond Loop Bounds: Comparing Annotation Languages for Worst-Case Execution Time Analysis. *J. of Software and System Modeling*, 2010. Online edition.
7. J. Knoop, L. Kovacs, and J. Zwirchmayr. An Evaluation of WCET Analysis using Symbolic Loop Bounds. In *Pre-Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*, Porto, Portugal, 2011.
8. J. Knoop, L. Kovacs, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In *Pre-Proceedings of the 8th International Andrei Ershov Memorial Conference – Perspectives of System Informatics (PSI 2011)*. Springer, 2011.

...					...				
fdct	2	2	2	-	muller	2	2	2	-
lcdnum	1	1	1	-	nb_occ	1	1	1	-
ndes	12	12	12	-	negate	1	1	1	-
sum	207	163	165	2	ovt_vs_poly	1	0	0	-
					permut_search2	1	0	0	-
					permut_search	1	0	0	-
					r_strcpy	1	0	0	-
					string_constant	1	0	0	-
					struct_hack	3	0	0	-
					sum1	1	1	1	-
					sum2	2	2	2	-
					test5_floats	1	0	0	-
					trace.c	2	2	2	-
					vamos	2	0	0	-
					vieira1	2	2	2	-
					vieira2	1	0	0	-
					weber1	1	1	1	-
					weber3	1	0	0	-
					weber4	1	0	0	-
					weber5	1	0	0	-
					weber6	1	0	0	-
					weber8	1	0	0	-
					weber9	1	1	1	-
					sum	77	39	46	7

Table 2. Evaluation of r-TuBound on the Mälardalen benchmarks (left) and examples sent by Dassault Aviation (right), continued.

9. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, pages 337–340, 2008.
10. A. Prantl, J. Knoop, R. Kirner, A. Kadlec, and M. Schordan. From Trusted Annotations to Verified Knowledge. In *Proc. of WCET*, pages 39–49, 2009.
11. A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint Solving for High-Level WCET Analysis. In *Proc. of WLPE*, pages 77–89, 2008.
12. A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for WCET Analysis. In *Proc. of WCET*, pages 141–148, 2008.
13. R. von Hanxleden, N. Holsti, B. Lisper, E. Ploedereder, R. Wilhelm, A. Bonenfant, H. Casse, S. Bünte, W. Fellger, S. Gepperth, J. Gustafsson, B. Huber, N. M. Islam, D. Kästner, R. Kirner, L. Kovacs, F. Krause, M. de Michiel, M. C. Olesen, A. Prantl, W. Puffitsch, C. Rochange, M. Schoeberl, S. Wegener, M. Zolda, and J. Zwirchmayr. WCET tool challenge 2011: Report. In *Proceedings of the 11th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Porto, Portugal, July 2011.