Symbolic Loop Bound Computation for WCET Analysis

Jens Knoop, Laura Kovács, and Jakob Zwirchmayr*

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. To do so, we refine program flows using SMT reasoning and rewrite multi-path loops into single-path ones. Single-path loops are further translated into a set of recurrence relations over program variables. Recurrence relations are solved and iteration bounds of program loops are derived from the computed closed forms. For solving recurrences we deploy a pattern-based recurrence solving algorithm and compute closed forms only for a restricted class of recurrence equations. However, in practice, these recurrences describe the behavior of a large set of program loops. Our technique is implemented in the r-TuBound tool and was successfully tried out on a number of challenging 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 and/or recursive procedures, as the WCET crucially depends on the number of loop iterations and/or recursion depth.

To overcome this limitation, state-of-the-art WCET analysis tools, see e.g. [8, 20, 12], rely on user-given program assertions describing loop and/or recursion 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 and/or recursion depth can be deduced automatically.

In this paper we address the problem of *automatically inferring iteration bounds* of imperative program loops. For doing so, we identified special classes of loops with assignments and conditionals, where updates over program variables are linear expressions (Sections 5.1-5.3). 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 5.3). We call a loop with multiple paths, respectively with a single path, a multi-path loop, respectively a simple 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* [13, 4]. The resulting simple loop soundly over-approximates the multi-path loop. Iteration bounds of the simple loop are thus safe iteration bounds of the multi-path loop.

* This research is supported by the CeTAT project of TU Vienna. The second author is supported by an FWF Hertha Firnberg Research grant (T425-N23). This research was partly supported by Dassault Aviation and by the FWF National Research Network RiSE (S11410-N23).

(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 5.1). To this end, a new variable denoting the loop counter is introduced and used as the summation variable. The recurrence equation of the loop iteration variable captures thus the dependency between various iterations of the loop.

(iii) Recurrence equations of loop variables are next 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 5.1). In other words, the *closed forms of loop variables* are derived. Our framework overcomes the limitations of missing initial values by a simple *over-approximation of non-deterministic assignments* (Section 5.2).

We note that solving arbitrary recurrence equations is undecidable. However, in our approach we only consider loops with linear updates. Such loops yield linear recurrences with constant coefficients, called C-finite recurrences [5]. As C-finite recurrences can always be solved, our method succeeds in computing the closed forms of loop variables.

For solving C-finite recurrences we deploy a *pattern-based recurrence solving algorithm*. In other words, we instantiate unknowns in the closed form pattern of C-finite recurrences by the symbolic constant coefficients of the recurrence to be solved. Unlike powerful algorithmic combinatorics techniques that can solve complex recurrences, our framework hence only solves a particular class of recurrence equations. However, it turned out that in WCET analysis the recurrences describing the iteration behavior of program loops are not arbitrarily complex and can be solved by our approach (Section 6).

(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 5.1). The upper bound on the number of loop iterations is finally derived by computing the *smallest value of the loop counter* such that the loop is terminated. To this end, we deploy SMT reasoning over arithmetical formulas. The inferred iteration bound is further used to infer an accurate WCET of the program loop.

Experiments. Our approach is implemented as an extension to the TuBound tool [20], denoted r-TuBound, and successfully evaluated on WCET benchmarks (Section 6). Our experimental results give practical evidence for the applicability of our method for the WCET analysis of programs.

Contributions. In this paper, we make the following contributions:

- we introduce a pattern-based recurrence solving approach for WCET analysis of imperative program loops;
- we describe a flow refinement approach for translating multi-path loops into simple loops, by proving arithmetical properties using SMT technology;
- we present the summary of an experimental evaluation of our approach on challenging WCET benchmarks.

Our work advances the state-of-the-art in WCET analysis by a conceptually new and fully automated approach to loop bound computation. Moreover, our approach extends the application of program analysis methods by integrating WCET techniques with recurrence solving and SMT reasoning.

2 Related Work

Our work is closely related to the WCET analysis of software and invariant generation of program loops.

WCET Analysis. WCET analysis is usually two-tiered. A low-level analysis estimates the execution times of program instructions on the underlying hardware and computes a concrete time value of the WCET. On the other hand, a high-level analysis is, in general, platform-independent and is concerned, for example, with loop bound computation. In [14] a framework for parametric WCET analysis is introduced, and symbolic expressions as iteration bounds are derived. Instantiating the symbolic expressions with specific inputs yields then the WCET of programs. The approach presented in [15] automatically identifies induction variables and recurrence relations of loops using abstract interpretation [2]. Recurrence relations are further solved using precomputed closed form templates, and iteration bounds are hence derived. In [20] iteration bounds are obtained using data flow analysis and interval based abstract interpretation. However, state-of-the-art WCET analysis tools, including [8, 15, 20], compute loop bounds automatically only for loops with relatively simple flow and arithmetic [10]. For more complex loops iteration bounds are supplied manually in the form of auxiliary program annotations. Unlike the aforementioned approaches, we require no user guidance but automatically infer iteration bounds for special classes of loops with non-trivial arithmetic and flow.

Invariant Generation and Cost Analysis. Loop invariants describe loop properties that are valid at any, and thus also at the last loop iteration. Invariant generation techniques therefore can be used to infer bounds on program resources, such as time and memory – see e.g. [6, 7, 1, 3, 9].

The work described in [6] instruments loops with various counters at different program locations. Then an abstract interpretation based linear invariant generation method is used to derive linear bounds over each counter variable. Bounds on counters are composed using a proof-rule-based algorithm, and non-linear disjunctive bounds of multi-path loops are finally inferred. The approach is further extended in [7] to derive more complex loop bounds. For doing so, disjunctive invariants are inferred using abstract interpretation and flow refinement. Next, proof-rules using max, sum, and product operations on bound patterns are deployed in conjunction with SMT reasoning in the theory of linear arithmetic and arrays. As a result, non-linear symbolic bounds of multi-path loops are obtained. Abstract interpretation based invariant generation is also used in [1] in conjunction with so-called cost relations. Cost relations extend recurrence relations and can express recurrence relations with non-deterministic behavior which arise from multi-path loops. Iteration bounds of loops are inferred by constructing evaluation trees of cost relations and computing bounds on the height of the trees. For doing so, linear invariants and ranking functions for each tree node are inferred. Unlike the aforementioned techniques, we do not use abstract interpretation but deploy a recurrence solving approach to generate bounds on simple loops. Contrarily to [6, 7, 1], our method is limited to multi-path loops that can be translated into simple loops by SMT queries over arithmetic.

Recurrence solving is also used in [3, 9]. The work presented in [9] derives loop bounds by solving arbitrary C-finite recurrences and deploying quantifier elimination

over integers and real closed fields. To this end, [9] uses some algebraic algorithms as black-boxes built upon the computer algebra system Mathematica [23]. Contrarily to [9], we only solve C-finite recurrences of order 1, but, unlike [9], we do not rely on computer algebra systems and handle more complex multi-path loops. Symbolic loop bounds in [3] are inferred over arbitrarily nested loops with polynomial dependencies among loop iteration variables. To this end, C-finite and hypergeometric recurrence solving is used. Unlike [3], we only handle C-finite recurrences of order 1. Contrarily to [3], we however design flow refinement techniques to make our approach scalable to the WCET analysis of programs.

3 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.

```
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       ♯pragma 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      ♯pragma wcet_loopbound(100)
14    }
15    for (j = 0; j < 100; j++) {
16      if (a[k][j] != -1 && a[k][j] != 1) break;
17      ♯pragma wcet_loopbound(100)
18    }
19    k++;
20    ♯pragma wcet_loopbound(32)
21  }
22 }
```

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

Computing an accurate WCET of the `func` method requires thus 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.

- We design a pattern-based recurrence solving algorithm (Section 5.1). Using this algorithm, the iteration bound of the loop from lines 6-9 is inferred to be precisely 4.
- We deploy SMT reasoning to translate multi-path loops into simple ones (Section 5.3). Using our flow refinement approach, 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 5.2). As a result, the upper bound of the loop from lines 5-21 is derived to be 31.

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

The iteration bounds inferred automatically by our approach are listed in Figure 1, using the program annotations `♯pragma wcet_loopbound(...)`.

The rest of the paper discusses in detail how we automatically compute iteration bounds for simple and multi-path loops.

4 Theoretical Considerations

This section contains a brief overview of algebraic techniques and WCET analysis as required for the development of this paper. For more details, we refer to [5, 17].

Algebraic Considerations. Throughout this paper, $\mathbb{N}$ and $\mathbb{R}$ denote the set of natural and real numbers, respectively.

Let $\mathbb{K}$ be a field of characteristic zero (e.g. $\mathbb{R}$) and $f : \mathbb{N} \to \mathbb{K}$ a *univariate sequence* in $\mathbb{K}$. Consider a rational function $R : \mathbb{K}^{r+1} \to \mathbb{K}$. A *recurrence equation* for the sequence $f(n)$ is of the form $f(n+r) = R(f(n), f(n+1), ..., f(n+r-1), n)$, where $r \in \mathbb{N}$ is the *order* of the recurrence.

Given a recurrence equation of $f(n)$, one is interested to compute the value of $f(n)$ as a function depending *only* on the summation variable n and some given initial vales $f(0), \dots, f(n-1)$. In other words, a *closed form* solution of $f(n)$ is sought. Although solving arbitrary recurrence equations is an undecidable problem, special classes of recurrence equations can be effectively decided using algorithmic combinatorics techniques.

We are interested in solving one particular class of recurrence equations, called *C-finite recurrences*. An *inhomogeneous* C-finite recurrence equation is of the form $f(n+r) = a_0 f(n) + a_1 f(n+1) + ... + a_{r-1} f(n+r-1) + h(n)$, where $a_0, \dots, a_{r-1} \in \mathbb{K}$, $a_0 \neq 0$, and $h(n)$ is a linear combination over $\mathbb{K}$ of exponential sequences in n with polynomial coefficients in n. The C-finite recurrence is *homogeneous* if $h(n) = 0$. C-finite recurrences fall in the class of decidable recurrences. In other words, closed forms of C-finite recurrences can always be computed [5].

Example 1. Consider the C-finite recurrence $f(n+1) = 3f(n) + 1$ with initial values $f(0) = 1$. By solving $f(n)$, we obtain the closed form $f(n) = \frac{3}{2} * 3^n - \frac{1}{2}$.

WCET Analysis. Efficient and precise WCET analysis relies on program analysis and optimization techniques. We overview below four methods on which our WCET analysis framework crucially depends on. Throughout this paper, we call a loop *simple* if it has a single path (i.e. the loop body is a sequence of assignment statements). A loop is called *multi-path* if it has multiple paths (i.e. the loop body contains conditionals and/or loops). We also assume that integers are represented using 32-bits.

(1) Interval and points-to analysis implements a forward directed data flow interval analysis, yielding variable values and aliasing information for all program locations.

(2) Counter-based loop analysis derives bounds for simple loops with incremented/decremented updates, by constructing symbolic equations that are solved using pattern matching and term rewriting. If no values are available for variables in the loop initialization, condition, and increment expression, then the analysis fails in deriving loop bounds. Note that in certain cases variables with unknown values can be discarded, and thus loop bounds can be derived.

```
void
func (void)
{
  a;
  while (b)
    {
      if (c)
        d;
      e;
    }
  f;
}
```

Fig. 2. Abstracted C program, where `b` and `c` are boolean expressions and `a`, `d`, `e`, and `f` denote program statements.

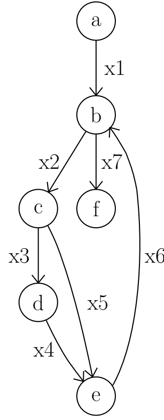

Fig. 3. Graph representation of program flow. The execution frequencies are listed on edges.

$$x1 = 1 \quad (4.1)$$
$$x2 = x1 * L \quad (4.2)$$
$$x3 = x2 - x5 \quad (4.3)$$
$$x4 = x3 \quad (4.4)$$
$$x5 = x2 - x3 \quad (4.5)$$
$$x6 = x5 + x4 \quad (4.6)$$
$$x7 = 1 \quad (4.7)$$

Fig. 4. Execution frequencies of program expressions. Equations (4.1) and (4.7) state that the program is entered and exited once. Equation (4.2) states that the loop body is executed L times, where L denotes the number of loop iterations.

(3) Constraint-based loop analysis models and enumerates the iteration space of nested loops. To this end, a system of constraints is constructed that reflects dependencies between iteration variables and thus yields better results than just multiplying the bounds of nested loops. The solution to the constraint system is computed by a constraint logic programming solver over finite domains.
(4) Implicit path enumeration technique (IPET) first maps the program flow to a set of graph flow constraints describing relationships between execution frequencies of program blocks. Execution times for expressions are evaluated by a low-level WCET-analysis. The execution times are then summed up to get execution times for program blocks. Finally, the longest execution path (exposing the WCET) of the program is found by maximizing the solution to the constraint system. We illustrate the IPET in Example 2.

Example 2. Consider the program given in Figure 2. Its program flow is given in Figure 3. The dependency relations between execution frequencies of program blocks are extracted from Figure 3 and are listed in Figure 4. For simplicity, we assume that all expressions take one time unit to execute. Therefore, execution of one loop iteration takes at most 4 time units: 1 unit to check each boolean condition `b` and `c`, and 1 unit to execute each statement `d` and `e`. The longest path through the loop is the node sequence $b \to c \to d \to e$. We consider[2] the loop bound `L` to be 10. The WCET for `func` is further derived by maximizing the sum of block costs in the process of satisfying the dependency relations of Figure 4. As a result, the WCET of `func` is computed to be 42 time units.

The accuracy of a WCET limit crucially depends on the precision of the available loop bounds. The difficulty of inferring precise upper bounds on loop iterations comes from the presence of complex loop arithmetic and control flow. In what follows, we

[2] L can be either inferred by a loop analysis step or given as a manual annotation.

describe an approach for inferring precise upper bounds on the number of loop iterations by using recurrence solving and SMT reasoning in a unified framework. For simplicity, we sometimes say *loop upper bounds* or *loop bound* to mean upper bounds on the number of loop iterations.

5 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 5.1) and over-approximate non-deterministic initializations (Section 5.2). We translate multi-path loops with abrupt termination and monotonic conditional updates into simple loops (Section 5.3). The computed loop bounds are further used to obtain accurate WCET of programs.

5.1 Recurrence Solving in Simple Loops

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

$$\begin{array}{l}\texttt{for}\,(\mathtt{i} = \mathtt{a};\ \mathtt{i} < \mathtt{b};\ \mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d})\,; \\ \text{where } a, b, c, d \in \mathbb{K} \text{ do not depend on } i, \text{ and } c \neq 0.\end{array} \tag{5.1}$$

Hence, in loops matching (5.1), loop iteration variables (i.e. i) are bounded by symbolic constants and updated by linear expressions over iteration variables.

For a loop like (5.1), we derive an *exact* upper bound on the number of loop iterations. In more detail, we proceed as follows.
(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 (5.1) is given below.

$$i(n+1) = c * i(n) + d \quad \text{with the initial value} \quad i(0) = a. \tag{5.2}$$

Note that (5.2) is a C-finite recurrence of order 1 as variable updates of (5.1) are linear.
(ii) Next, recurrence (5.2) is solved (cf. Section 4) and the closed form of i as a function over n is derived. More precisely, depending on the value of c, the closed form of $i(n)$ is given below.

$$\begin{array}{l} i(n) \ = \ \alpha * c^n + \beta, \ \text{ if } c \neq 1 \\ \text{where} \\ \begin{cases} \alpha + \beta & = a \\ \alpha * c + \beta = a * c + d \end{cases} \end{array} \quad \text{and} \quad \begin{array}{l} i(n) \ = \ \alpha + \beta * n, \ \text{ if } c = 1 \\ \text{where} \\ \begin{cases} \alpha & = a \\ \alpha + \beta = a + d \end{cases} \end{array} \tag{5.3}$$

(iii) The closed form of $i(n)$ is further used to derive a tight upper integer bound on the number of loop iterations of (5.1). 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:

$$n \in \mathbb{N} \ \wedge \ i(n) < b \ \wedge \ i(n+1) \geq b. \tag{5.4}$$

```
for (j=1; j<100; j = j*3+1);
```

Fig. 5. Loop with C-finite update.

```
for (i=nondet(); i>0; i≫=1);
```

Fig. 6. Loop with non-deterministic initialization.

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 (5.1).

Example 3. Consider Figure 5. Updates over j describe a C-finite recurrence, whereas the loop condition is expressed as a linear inequality over j. Let $n \in \mathbb{N}$ denote the loop counter. Based on (5.3), the value of j at arbitrary loop iteration n is $j(n) = \frac{3}{2} * 3^n - \frac{1}{2}$. Using (5.4), the upper bound on loop iterations is further derived to be 4 – we consider 0 to be the starting iteration of a loop.

Solving the recurrence equation (5.2) can be done using powerful symbolic computation packages, such as [16, 11]. These packages are implemented on top of the computer algebra system (CAS) Mathematica [23]. Integrating a CAS with program analysis tools is however problematic due to the complexity and closed-source nature of CAS. Moreover, the full computational power of CAS algorithms is hardly needed in applications of program analysis and verification. Therefore, for automatically inferring exact loop bounds for (5.1) we designed a *pattern-based recurrence solving* algorithm which is not based on 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 (5.1), where updates describe C-finite recurrences of order 1. Closed forms of such recurrences are given in (5.3). Therefore, to compute upper bounds on the number of loop iterations of (5.1) we do not deploy the general C-finite recurrence solving algorithm given in [5], but instantiate the closed form pattern (5.3). In other words, whenever we encounter a loop of the form (5.1), the closed form of the iteration variable is derived by instantiating the symbolic constants $a, b, c, d, \alpha, \beta$ of (5.3) 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 (5.1) 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 (5.1).

Finally let us make the observation that while reasoning about (5.1), 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 (5.1) the integer variables and constants of (5.1) are safely approximated over $\mathbb{K}$. However, when deriving upper bounds of (5.1), we restrict the satisfiability problem (5.4) over integers. If formulas in (5.4) are linear, we obtain a tight loop bound satisfying (5.4) by using SMT reasoning over linear arithmetic. Otherwise, a loop bound satisfying (5.4) is computed using SMT reasoning over non-linear arithmetic (with bounded integers).

5.2 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. Upper bounds on the number of loop iterations can thus be derived as described in Section 5.1, 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 (5.5)(a) and (b).

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

(a) (b) (5.5)

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

For shift-loops matching pattern (5.5), the method presented in Section 5.1 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. (5.5)(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. (5.5)(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.

Example 4. Consider the shift-loop of Figure 6 with right shift updates. The initial value of i is set to `INT_MAX`. The upper bound on the number of loop iteration is then derived to be 31.

Let us note that the treatment of non-deterministic initial assignments as described above is not only restricted to shift-loops, but can be also extended to (5.1). For doing so, one would however need to investigate the monotonic behavior of C-finite recurrences in order to soundly approximate the initial value of loop iteration variables. We leave this study for future research.

5.3 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.

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

Fig. 7. Loop with abrupt termination.

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

Fig. 8. Loop with monotonic conditional update.

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 5.1 and 5.2, 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.

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 (5.6) (a)–(c).

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

```
int i = nondet();
for (; i > e; i >>= m)
  if (nondet()) break;
```
(b)

```
int i = nondet();
for (; i < e; i <<= m)
  if (nondet()) break;
```
(c)

(5.6)

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 (5.6). Therefore, we safely over-approximate the number of loop iterations of (5.6) by assuming that the abruptly terminating loop path is not taken. In other words, the non-deterministic conditional statement causing the abrupt termination of (5.6) is ignored, and we are left with a simple loop as in (5.1) or (5.5). Upper bounds on the resulting loops are then computed as described in Sections 5.1 and 5.2, from which an over-approximation of the WCET of (5.6) is derived.

Example 5. Consider Figure 7. Assuming that the abruptly terminating loop path is not taken, we obtain a simple loop as in (5.1). We thus get the loop bound 100.

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 (5.7).

```
for (i = a; i < b; i = c * i + d)
  if (B) i = f1(i);
  else  i = f2(i);
```
(5.7)

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 (5.7). 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 (5.7), we aim at computing the *worst case* iteration time of (5.7). To do so, we ignore B and transform (5.7) into a single-path loop by *safely over-approximating* the multi-path behavior of (5.7), 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 (5.7).
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 (5.7).
(ii) Loop (5.7) is then over-approximated by the simple loop (5.8) capturing the worst case iteration time of (5.7).

$$\begin{cases} \textbf{for}\ (\mathtt{i} = \mathtt{a}; \mathtt{i} < \mathtt{b}; \{\mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d}; \mathtt{i} = \mathtt{f_1(i)}\}), & \text{if } c > 0 \text{ and } m = \Delta + \Delta_1 \\ \textbf{for}\ (\mathtt{i} = \mathtt{a}; \mathtt{i} < \mathtt{b}; \{\mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d}; \mathtt{i} = \mathtt{f_2(i)}\}), & \text{if } c > 0 \text{ and } m = \Delta + \Delta_2 \\ \textbf{for}\ (\mathtt{i} = \mathtt{a}; \mathtt{i} < \mathtt{b}; \{\mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d}; \mathtt{i} = \mathtt{f_1(i)}\}), & \text{if } c < 0 \text{ and } m = \Delta + \Delta_1 \\ \textbf{for}\ (\mathtt{i} = \mathtt{a}; \mathtt{i} < \mathtt{b}; \{\mathtt{i} = \mathtt{c} * \mathtt{i} + \mathtt{d}; \mathtt{i} = \mathtt{f_2(i)}\}), & \text{if } c < 0 \text{ and } m = \Delta + \Delta_2 \end{cases} \tag{5.8}$$

Hence, the control flow refinement of (5.7) 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 (5.8). To this end, we need to make additional constraints on the monotonic functions f_1 and f_2 of (5.7) so that the approach of Section 5.1 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 (5.8) correspond to C-finite recurrences of order 1. Upper bounds on loop iterations of (5.8) can thus be derived as presented in Section 5.1.

Note that the additional constraints imposed over f_1 and f_2 restrict our approach to the multi-path loops (5.9) with linear conditional updates.

```
for (i = a; i < b; i = c * i + d)
{
  if (B) i = u1 * i + v1;
  else  i = u2 * i + v2;
}
```
(5.9)

where $a, b, c, d, u_1, u_2, v_1, v_2 \in \mathbb{K}$ do not depend on i, and $c, u_1, u_2 \neq 0$,

B is a boolean condition over loop variables, and $\begin{cases} u_1 > 0 \text{ and } u_2 > 0, \text{ if } c > 0 \\ u_1 < 0 \text{ and } u_2 < 0, \text{ if } c < 0 \end{cases}$

Loops (5.9) form a special case of (5.7). Let us however note, that extending the approach of Section 5.1 to handle general (or not) C-finite recurrences with monotonic behavior would enable our framework to compute upper bounds for arbitrary (5.7). We leave the study of such extensions for future work.

```
for (i = x;
       i < 65536;
       i *= 2) ;
```

(a) Loop with C-finite update. Iteration bound is 12.

```
int s = x;
while (s)
   s >>= 1;
```

(b) Shift-loop. Iteration bound is 31.

```
while (i > 0)
   if (i >= c)
      i = -c;
   else
      i -= 1;
```

(c) Loop with conditional updates. With no initial value information on i, the bound is INT_MAX.

```
M = 12;
for (i = 0; i < M; i++)
   A[i] = malloc(N)
   if (!A[i])
      break;
```

(d) Abruptly terminating loop. Iteration bound is 12.

Fig. 9. Examples from[10].

Example 6. Consider Figure 8. The conditional update over j is linear, and hence the multi-path loop is transformed into the simple loop **for** (j = 0; j < 100; j++). The loop bound of Figure 8 is therefore derived to be 100.

6 Experimental Evaluation

We implemented our approach to loop bound computation in the TuBound tool [19]. TuBound is a static analysis tool with focus on program optimization and WCET analysis of the optimized program. TuBound is based on the Static Analysis Tool Integration Engine (SATIrE) framework and developed at VUT. Given a C program, the work flow of TuBound consists of a forward-directed flow-sensitive interval analysis and a unification based flow-insensitive points-to analysis, followed by a loop bound analysis. The analysis results are weaved back into the program source as compiler-pragmas (i.e. annotations). A WCET-aware GNU C compiler translates then the source to annotated assembler code. Finally, a WCET bound of the input C program is calculated using the IPET approach. For more details about TuBound we refer to [19, 17].

We extended TuBound's loop bound inference engine by our approach to infer iteration bounds of loops with non-trivial linear updates and refinable control flow. However, to make our approach applicable to real world examples, we also extended TuBound's loop preprocessing step. The preprocessing step transforms arbitrary loop constructs into `for`-loops, and applies simple loop rewriting techniques.

The new version of TuBound implementing our approach to loop bound computation is called *r-TuBound*.

WCET Benchmarks. We investigated two benchmark suites that originate from the WCET community: the Mälardalen Real Time and the Debie-1d benchmark suites [10]. The Mälardalen suite consists of 152 loops. r-TuBound was able to infer loop bounds for 121 loops from the Mälardalen examples. By analyzing our results, we observed that 120 out of these 121 loops involved only incrementing/decrementing updates over iteration variables, and 1 loop required more complex C-finite recurrence solving as described in Section 5.1. When compared to TuBound [19], we noted that TuBound also computed bounds for the 120 Mälardalen loops with increment/decrement updates. However, unlike r-TuBound, TuBound failed on analyzing the loop with more complex C-finite behavior.

The Debie benchmark suite contains 75 loops. r-TuBound successfully analyzed 59 loops. These 59 loops can be classified as follows: 57 simple loops with increment/decrement updates, 1 shift-loop with non-deterministic initialization, and 1 multi-

Benchmark Suite	# Loops	TuBound	r-TuBound	Types
Mälardalen	152	120	121	CF
Debie-1d	75	58	59	SH, CU
Scimark	34	24	26	AT, CF
Dassault	77	39	46	AT, CF-CU-AT (3), CU(4)
Total	338	241	252	AT (2), SH, CU (5), CF (2), CF-CU-AT (3)

Table 1. Experimental results and comparisons with r-TuBound and TuBound.

path loop with conditional update. When compared to TuBound, we observed that TuBound could analyze the 58 simple loops and failed on the multi-path loop. Moreover, r-TuBound derived a tighter loop bound for the shift-loop than TuBound.

Scientific Benchmarks. We ran r-TuBound on the SciMark2 benchmark suite containing 34 loops. Among other arithmetic operations, SciMark2 makes use of fast Fourier transformations and matrix operations. r-TuBound derived loop bounds for 26 loops, whereas TuBound could analyze 24 loops. The 2 loops which could only be handled by r-TuBound required reasoning about abrupt-termination and C-finite updates.

Industrial Benchmarks. We also evaluated r-TuBound on 77 loops sent by Dassault Aviation. r-TuBound inferred loop bounds for 46 loops, whereas TuBound analyzed only 39 loops. When compared to TuBound, the success of r-TuBound lies in its power to handle abrupt termination, conditional updates, and C-finite behavior.

Experimental Results. Altogether, we ran r-TuBound on 4 different benchmark suites, on a total of 338 loops and derived loop bounds for 253 loops. Out of these 253 loops, 243 loops were simple and involved only C-finite reasoning, and 10 loops were multi-path and required the treatment of abrupt termination and conditional updates. Unlike r-TuBound, TuBound could only handle 241 simple loops. Figure 9 shows example of loops that could be analyzed by r-TuBound, but not also by TuBound.

We summarize our results in Table 1. Column 1 of Table 1 lists the name of the benchmark suite, whereas column 2 gives the number of loops contained in the benchmarks. Columns 3 and 4 list respectively how many loops were successfully analyzed by TuBound and r-TuBound. Column 5 describes the reasons why some loops, when compared to TuBound, could only be analyzed by r-TuBound. 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). Column 5 of Table 1 also lists, in parenthesis, how many of such loops were encountered and could only be analyzed by r-TuBound. For example, among the loops sent by Dassault Aviation 4 multi-path loops with monotonic conditional update, denoted as CU(4), could only be analyzed by r-TuBound. Some loops on which only r-TuBound succeeds are, for example, multi-path loops with C-finite conditional updates and abrupt termination; such loops are listed in Table 1 as CF-CU-AT.

Analysis of Results. Table 1 shows that 74.55% of the 338 loops were successfully analyzed by r-TuBound, whereas TuBound succeeded on 71.30% of the 338 loops. That is, when compared to TuBound, the overall quality of loop bound analysis within r-TuBound has increased by 3.25%. This relatively low performance increase of r-TuBound might thus not be considered significant, when compared to TuBound.

```
while (abs(diff) >= 1.0e-05)
 {
   diff = diff * -rad * rad/
        (2.0 * inc) * (2.0 * inc + 1.0);
   inc++;
 }
```
(a)

```
while (k < j)
 {
   j -= k;
   k /= 2;
 }
```
(b)

```
while (((int)p[i]) != 0)
  i++;
```
(c)

Fig. 10. Limitations of r-TuBound.

Let us however note that the performance of r-TuBound, compared to TUBound, on the WCET and scientific benchmarks was predictable in some sense. These benchmarks are used to test and evaluate WCET tools already since 2006. In other words, it is expected that state-of-the-art WCET tools are fine tuned with various heuristics so that they yield good performance results on loops occurring in ”classical” WCET benchmarks, including Debie-1D, Mälarden, or even Scimark.

The benefit of r-TuBound wrt TuBound can be however evidenced when considering new benchmarks, where loops have more complicated arithmetic and/or control flow. Namely, on the 77 examples coming from Dassault Aviation, r-TuBound derives loop bounds for 46 programs. That is, 60% of new benchmarks can be successfully analysed by r-TuBound. When compared to TuBound, we note that r-TuBound outperforms TuBound on these new examples by a performance increase of 9%. The programs which can only be handled by r-TuBound require reasoning about multi-path loops where updates to scalars yield linear recurrences of program variables (in many cases, with $c \neq 1$ in (5.1)) . These recurrences cannot be solved by the simple variable increment/decrement handling of TuBound. Moreover, TuBound fails in handling multi-path loops. Based on the results obtained on these new benchmark suite, we believe that our pattern-based recurrence solving approach in conjunction with SMT reasoning for flow refinement provides good results for computing bounds for complex loops with r-TuBound.

Let us also note that in WCET analysis, loops, even simple ones, are in general manually annotated with additional information on loop bounds. Automated generation of loop bounds would significantly increase the applicability of WCET tools in embedded software. The techniques presented in this paper can automatically deal with loops which often occur real-time systems, and therefore can be successfully applied for automated WCET analysis of programs.

Limitations. We investigated examples on which r-TuBound failed to derive loop bounds. We list some of the failing loops in Figure 10. Note that the arithmetic used in the simple loop Figure 10(a) requires extending our framework with more complex recurrence solving techniques, such as [22, 11], and deploy SMT solving over various numeric functions, such as the absolute value computations over floats or integers. On the other hand, Figure 10(b) suggests a simple extension of our method to solving blocks of C-finite recurrences. Finally we note that Figure 10(c) illustrates the need of combining our approach with reasoning about array contents, which can be done either using SMT solvers [13, 4] or first-order theorem provers [21]. We leave this research challenges for future work.

7 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, uses SMT reasoning to refine program flow, and over-approximates non-deterministic initializations. The inferred loop bounds are further used in the WCET analysis of programs. When applied to challenging benchmarks, our approach succeeded in generating tight iteration bounds for large number of loops.

In the line of [18], we plan to extend the recurrence solving algorithm in r-TuBound to analyze loops with more complex arithmetic. We also intend to integrate our approach with flow refinement techniques based on invariant generation, such as in [7]. Moreover, we plan to compete in the WCET-Challenge 2011.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *J. Automated Reasoning*, 46(2):161–203, 2011.
2. Z. Ammarguellat and W. L. Harrison, III. Automatic Recognition of Induction Variables and Recurrence Relations by Abstract Interpretation. In *Proc. of PLDI*, pages 283–295, 1990.
3. R. Blanc, T. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic Bound Computation for Loops. In *Proc. of LPAR-16*, pages 103–118, 2010.
4. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. of TACAS*, pages 174–177, 2009.
5. G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2003.
6. S. Gulwani, K. K. Mehra, and T. M. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *Proc. of POPL*, pages 127–139, 2009.
7. S. Gulwani and F. Zuleger. The Reachability-Bound Problem. In *Proc. of PLDI*, pages 292–304, 2010.
8. 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.
9. T. Henzinger, T. Hottelier, and L. Kovács. Valigator: A Verification Tool with Bound and Invariant Generation. In *Proc. of LPAR-15*, pages 333–342, 2008.
10. 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.
11. M. Kauers. SumCracker: A Package for Manipulating Symbolic Sums and Related Objects. *J. of Symbolic Computation*, 41(9):1039–1057, 2006.
12. 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.
13. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. of TACAS*, pages 337–340, 2008.
14. B. Lisper. Fully Automatic, Parametric Worst-Case Execution Time Analysis. In *Proc. of WCET*, pages 99–102, 2003.

15. M. D. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat. Static Loop Bound Analysis of C Programs Based on Flow Analysis and Abstract Interpretation. In *Proc. of RTCSA*, pages 161–166, 2008.
16. P. Paule and M. Schorn. A Mathematica Version of Zeilberger's Algorithm for Proving Binomial Coefficient Identities. *J. of Symbolic Computation*, 20(5-6):673–698, 1995.
17. A. Prantl. *High-level Compiler Support for Timing-Analysis*. PhD thesis, Vienna University of Technology, 2010.
18. 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.
19. A. Prantl, J. Knoop, M. Schordan, and M. Triska. Constraint Solving for High-Level WCET Analysis. In *Proc. of WLPE*, pages 77–89, 2008.
20. A. Prantl, M. Schordan, and J. Knoop. TuBound - A Conceptually New Tool for WCET Analysis. In *Proc. of WCET*, pages 141–148, 2008.
21. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
22. C. Schneider. Symbolic Summation with Single-Nested Sum Extensions. In *Proc. of ISSAC*, pages 282–289, 2004.
23. S. Wolfram. *The Mathematica Book. Version 5.0*. Wolfram Media, 2003.