Tighter Loop Bound Analysis

Pavel Čadek[1], Jan Strejček[2], and Marek Trtík[3]

[1] Faculty of Informatics, Vienna University of Technology, Austria
[2] Faculty of Informatics, Masaryk University, Brno, Czech Republic
[3] LaBRI, University of Bordeaux, France

Abstract. We present a new algorithm for computing upper bounds on the number of executions of each program instruction during any single program run. The upper bounds are expressed as functions of program input values. The algorithm is primarily designed to produce bounds that are relatively tight, i.e. not unnecessarily blown up. The upper bounds for instructions allow us to infer loop bounds, i.e. upper bounds on the number of loop iterations. Experimental results show that the algorithm implemented in a prototype tool LOOPERMAN often produces tighter bounds than current tools for loop bound analysis.

1 Introduction

The goal of *loop bound analysis* is to derive for each loop in a given program an upper bound on the number of its iterations during any execution of the program. These bounds can be parametrized by the program input. The loop bound analysis is an active research area with two prominent applications: *program complexity* analysis and *worst case execution time* (WCET) analysis.

The aim of program complexity analysis is to derive an asymptotic complexity of a given program. The complexity is commonly considered by programmers in their everyday work and it is also used in specifications of programming languages, e.g. every implementation of the standard template library of C++ has to have the prescribed complexities. Loop bound analysis clearly plays a crucial role in program complexity analysis. In this context, emphasis is put on large coverage of the loop bound analysis (i.e. it should find some bounds for as many program loops as possible), while there are only limited requirements on tightness of the bounds as asymptotic complexity is studied.

A typical application scenario for WCET analysis is to check whether a given part of some critical system finishes its execution within an allocated time budget. One step of the decision process is to compute loop bounds. Tightness of the bounds is very important here as an untight bound can lead to a spuriously negative answer of the analysis (i.e. 'the allocated time budged can be exceeded'), which may imply unnecessary additional costs, e.g. for system redesign or for hardware components with higher performance. The WCET analysis can also be used by schedulers to estimate the run-time of individual tasks.

The problem to infer loop bounds has recently been refined into the *reachability-bound problem* [8], where the goal is to find an upper bound on the number of

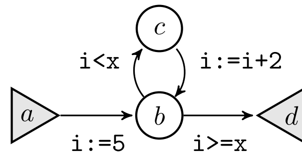
executions of a given program instruction during any single run of a given program. One typically asks for a reachability bound on some resource demanding instruction like memory allocation. Reachability bound analysis is more challenging than loop bound analysis as, in order to get a reasonably precise bound, branching inside loops must be taken into account.

This paper presents a new algorithm that infers reachability bounds. More precisely, for each instruction of a given program, the algorithm tries to find an upper bound on the number of executions of the instruction in any single run of the program. The bounds are parametrized by the program input. The reachability bounds can be directly used to infer loop bounds and asymptotic program complexity. Our algorithm builds on *symbolic execution* [10] and *loop summarisation* adopted from [14]. In comparison with other techniques for reachability bound or loop bound analysis, our algorithm brings the following features:

- It utilizes a loop summarisation technique that computes precise values of program variables as functions of loop iteration counts.
- It distinguishes different branches inside loops and computes bounds for each of them separately.
- If more different bounds arise, it handles all of them while other techniques usually choose nondeterministically one of them.
- It can detect logarithmic bounds.
- Upper bounds for nested loops are computed more precisely: while other techniques typically multiply a bound for the outer loop by a maximal bound on iterations of the inner loop during one iteration of the outer loop, we sum the bounds for the inner loop over all iterations of the outer loop.

All these features have a positive effect on tightness of produced bounds.

We can explain the basic idea of our algorithm on the flowgraph on the right. The node a is the entry location, d is the exit location, and locations b, c form a loop. An initial value of `x` represents program input. We symbolically execute each path in the loop and assign an iteration counter to it. Then



we try to express the effect of arbitrarily many iterations of the loop using the iteration counters as parameters. The loop in our example has just one path bcb that increments `i` by 2. Hence, the value of `i` after κ iterations is $\underline{i}' + 2\kappa$, where $\underline{i}'$ denotes the value of `i` before the loop execution starts. We combine this loop summary with the program state just before entering the loop, which is obtained by symbolic execution of the corresponding part of the program. In our example, we get that the value of `i` after κ iterations of the loop is $5+2\kappa$. To enter another iteration, the condition `i<x` must hold. If we replace the variables `i` and `x` by their current values, we get the condition $5 + 2\kappa < \underline{x}$, where $\underline{x}$ refers to the initial value of `x`. This condition is satisfied only if $\kappa < \frac{\underline{x}-5}{2}$. As κ is an iteration counter, it has to be a non-negative integer. Hence, we get the bound on the number of loop iterations $\mathbf{max}\{0, \lceil\frac{\underline{x}-5}{2}\rceil\}$, which is assigned to all edges in the loop. Edges outside the loop are visited at most once. The situation is

more complicated if we have loops with more loop paths, nested loops, or loops where a run can cycle forever. The algorithm is described in Section 3.

We have implemented our algorithm in an experimental tool LOOPERMAN. Comparison with several leading loop bound analysis tools shows that our approach often provides tighter loop bounds. For example, our tool is currently the only one that detects that the inner cycle of the BubbleSort algorithm makes $\frac{n\cdot(n-1)}{2}$ iterations in total (i.e. during all iterations of the outer loop) when sorting an array of n elements, while other tools provide only the bound n^2 or $\mathcal{O}(n^2)$. Section 4 presents the comparison with the best performing tool LOOPUS [12]. Experimental comparison with more tools, a detailed description of the algorithm, and discussion of the BubbleSort example can be found in [15].

2 Preliminaries

For simplicity, this paper focuses on programs without function calls, manipulating only integer scalar variables $\mathtt{a}, \mathtt{b}, \ldots$ and read-only multidimensional integer array variables $\mathtt{A}, \mathtt{B}, \ldots$. As usual in the context of loop bound analysis, integers are interpreted in the mathematical (i.e. unbounded) sense.

Flowgraph, backbone, loop, induced flowgraph An analysed program is represented as a *flowgraph* $P = (V, E, l_{beg}, l_{end}, \iota)$, where (V, E) is a finite oriented graph, $l_{beg}, l_{end} \in V$ are different *begin* and *end nodes* respectively, and $\iota : E \to \mathcal{I}$ labels each edge e by an *instruction* $\iota(e)$. The out-degree of l_{end} is 0 and out-degrees of all other nodes are positive. We use two kinds of instructions: an assignment $\mathtt{a}{:=}\mathit{expr}$ for some scalar variable $\mathtt{a}$ and some program expression *expr* over program variables, and an assumption $\mathtt{assume}(\gamma)$ for some quantifier-free formula γ over program variables. For example, a statement $\mathtt{if}\ \gamma\ \mathtt{then}\ldots$ corresponds to a node with two outgoing edges labelled with $\mathtt{assume}(\gamma)$ and $\mathtt{assume}(\neg\gamma)$. We often omit the keyword `assume` in flowgraphs.

A *path* in a flowgraph is a (finite or infinite) sequence $\pi = v_1 v_2 \cdots$ of nodes such that $(v_i, v_{i+1}) \in E$ for all v_i, v_{i+1} in the sequence. Paths are denoted by Greek letters. A *backbone* in a flowgraph is an acyclic path leading from the begin node to the end node.

Let π be a backbone with a prefix αv. There is a *loop* C with a *loop entry* v along π, if there exists a path $v\beta v$ such that no node of β appears in α. The loop C is then the smallest set containing all nodes of all such paths $v\beta v$.

Each *run* of the program corresponds to a path in the flowgraph starting at l_{beg} and such that it is either infinite, or it is finite and ends in l_{end}.[4] Every run follows some backbone: it can escape from the backbone in order to perform one or more iterations in a loop along the backbone, but once the last iteration in the loop is finished (which need not happen if the run is infinite), the execution continues along the backbone again. We thus talk about a *run along* a backbone.

[4] We assume that crashes or other undefined behaviour of program expressions are prevented by safety guards, e.g. an expression $\mathtt{a/b}$ is guarded by $\mathtt{assume}(\mathtt{b} \neq 0)$.

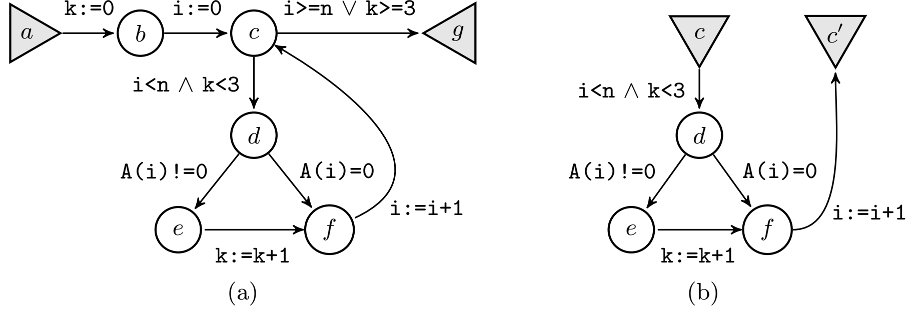


Fig. 1. (a) A flowgraph representing a program that gets an array `A` of size `n` and counts up to three non-zero elements in the array. (b) Its induced flowgraph $P(\{c,d,e,f\},c)$.

For a loop C with a loop entry v along a backbone π, a *flowgraph induced by the loop*, denoted as $P(C,v)$, is the subgraph of the original flowgraph induced by C, where v is marked as the begin node, a fresh end node v' is added, and every transition $(u,v) \in E$ leading to v is redirected to v' (we identify the edge (u,v') with (u,v) in the context of the original program). Each single iteration of the loop corresponds to a run of the induced flowgraph. Figure 1(b) shows the flowgraph induced by the loop $\{c,d,e,f\}$ of the program in Figure 1(a).

The program representation by flowgraphs and our definition of loops easily handle many features of programming languages like `break`, `continue`, or `goto`.

Symbolic execution *Symbolic execution* [10] replaces input data of a program by *symbols* representing arbitrary data. Executed instructions then manipulate symbolic expressions over the symbols instead of exact values. *Symbolic expressions* are terms of the theory of integers extended with functions **max** and **min**, rounding functions $\lceil\cdot\rceil$ and $\lfloor\cdot\rfloor$ applied to constant expressions over reals, and

- for each scalar variable `a`, an uninterpreted constant $\underline{a}$, which is a *symbol* representing any initial (input) value of the variable `a`,
- for each array variable `A`, an uninterpreted function $\underline{A}$ of the same arity as `A`, which is a *symbol* representing any initial (input) content of the array `A`,
- a countable set $\{\kappa_1, \kappa_2, \ldots\}$ of artificial variables (not appearing in analysed programs), called *path counters* and ranging over non-negative integers,
- a special symbol $\star$ called *unknown*, and
- for each formula ψ build over symbolic expressions and two symbolic expressions ϕ_1, ϕ_2, a construct $\mathbf{ite}(\psi, \phi_1, \phi_2)$ of meaning "**i**f-**t**hen-**e**lse", that evaluates to ϕ_1 if ψ holds, and to ϕ_2 otherwise.

For symbolic expressions ψ, ϕ and a symbol or a path counter x, let $\psi[x/\phi]$ denote the expression ψ where all occurrences of x are simultaneously replaced by ϕ. Further, $\psi[x_i/\phi_i \mid i \in I]$ denotes multiple simultaneous replacements. We sometimes write $\psi^{\boldsymbol{\kappa}}$ to emphasize that ψ can contain path counters $\boldsymbol{\kappa} = (\kappa_1, \ldots, \kappa_n)$. An expression is $\boldsymbol{\kappa}$*-free* if it contains no path counter.

Symbolic execution stores variable values in *symbolic memory* and all executable program paths are uniquely identified by corresponding *path conditions*. Here we provide brief descriptions of these terms. For more information see [10].

A *symbolic memory* is a function θ assigning to each scalar variable `a` a symbolic expression and to each array variable `A` the symbol $\underline{A}$ (array variables keep their initial values as we consider programs with read-only arrays). We overload the notation $\theta(\cdot)$ to program expressions as follows. Let *expr* be a program expression over program variables $\mathtt{a}_1, \ldots, \mathtt{a}_n$. Then $\theta(\mathit{expr})$ denotes the symbolic expression obtained from *expr* by replacement of all occurrences of the variables $\mathtt{a}_1, \ldots, \mathtt{a}_n$ by symbolic expressions $\theta(\mathtt{a}_1), \ldots, \theta(\mathtt{a}_n)$ respectively.

Symbolic execution of a path in a flowgraph starts with the initial symbolic memory θ, where $\theta(\mathtt{a}) = \underline{a}$ for each variable `a`. The memory is updated on assignments. For example, if the first executed assignment is `a:=2*a+b`, the initial symbolic memory θ is updated to the symbolic memory θ' where $\theta'(\mathtt{a}) = \theta(\mathtt{2*a+b}) = 2\underline{a} + \underline{b}$. If we later update θ' on `a:=1-a`, we get the memory θ'' such that $\theta''(\mathtt{a}) = \theta'(\mathtt{1-a}) = 1 - 2\underline{a} - \underline{b}$.

If ψ is a symbolic expression over symbols $\{\, \underline{a}_i \mid i \in I\}$ corresponding to program variables $\{\, \mathtt{a}_i \mid i \in I\}$ respectively, then $\theta\langle\psi\rangle$ denotes the symbolic expression $\psi[\underline{a}_i/\theta(\mathtt{a}_i) \mid i \in I]$. For example, if $\theta(\mathtt{a}) = \kappa_1$ and $\theta(\mathtt{b}) = \underline{a} - \kappa_2$, then $\theta\langle 2\underline{a} + \underline{b}\rangle = 2\theta(\mathtt{a}) + \theta(\mathtt{b}) = 2\kappa_1 + \underline{a} - \kappa_2$. Note that $\theta_1\langle\theta_2(\mathtt{a})\rangle$ returns the value of `a` after a code with effect θ_1 followed by a code with effect θ_2. For example, if $\theta_1(\mathtt{a}) = 2\underline{a}+1$ represents the effect of assignment `a:=2*a+1` and $\theta_2(\mathtt{a}) = \underline{a}-2$ the effect of `a:=a-2`, then $\theta_1\langle\theta_2(\mathtt{a})\rangle = \theta_1\langle\underline{a} - 2\rangle = (2\underline{a} + 1) - 2$ represents the effect of the two assignments in the sequence. We apply the notation $\theta\langle\varphi\rangle$ and $\varphi[x/\psi]$ with analogous meanings also to formulae φ built over symbolic expressions.

Given a path in a flowgraph leading from its begin node, the *path condition* is a formula over symbols satisfied exactly by all program inputs for which the program execution follows the path. A path condition is constructed during symbolic execution of the path. Initially, the path condition is set to *true* and it can only be updated when an $\mathtt{assume}(\gamma)$ is executed. For example, if a symbolic execution reaches `assume(a>5)` with a path condition φ and a symbolic memory $\theta(\mathtt{a}) = 2\underline{a} - 1$, then it updates the path condition to $\varphi \wedge (2\underline{a} - 1 > 5)$.

Upper bound An *upper bound* for an edge e in a flowgraph P is a $\boldsymbol{\kappa}$-free symbolic expression ρ such that whenever P is executed on any input, the instruction on edge e is executed at most ρ' times, where ρ' is the expression that we get by replacing each variable symbol by the input value of the corresponding variable.

3 The Algorithm

Recall that every program run follows some backbone and the run can diverge from the backbone only to loops along the backbone. The algorithm first detects all backbones. For each backbone π_i and each edge e, it computes a set of upper bounds $\beta_i(e)$ on the number of visits of the edge by any run following the considered backbone. As all these bounds are valid, the set $\beta_i(e)$ can be

interpreted as a single bound $\mathbf{min}\, \beta_i(e)$ on visits of edge e by any run along π_i. At the end, the overall upper bound for an edge e can be computed as the maximum of these bounds over all backbones, i.e. $\mathbf{max}\{\mathbf{min}\, \beta_i(e) \mid \pi_i \text{ is a backbone}\}$.

The algorithm consists of the following four procedures:

`executeProgram` is the starting procedure of the whole algorithm. It gets a flowgraph and computes all its backbones. Then it symbolically executes each backbone and computes for each edge a set of upper bounds on the number of visits of the edge by a run following the backbone. Whenever the symbolic execution enters a loop entry node, the procedure `processLoop` is called to get upper bounds on visits during loop execution.

`processLoop` gets a loop represented by the program induced by the loop. Note that each run of the induced program corresponds to one iteration of the loop and it follows some backbone of the induced program (the backbones are called *loop paths* in this context). The procedure then symbolically executes each loop path by recursive call of `executeProgram` (the nesting of recursive calls thus directly corresponds to the nesting of loops in the program). The recursive call of `executeProgram` produces, for each loop path, a symbolic memory and a path condition capturing the effect of a single iteration along the loop path. The procedure `processLoop` then calls `computeSummary`, which takes the symbolic memories after single loop iterations, assigns to each loop path a unique path counter κ_i, and computes a *parametrized symbolic memory* $\theta^{\boldsymbol{\kappa}}$ describing the effect of an arbitrary number of loop iterations. This symbolic memory is parametrized by path counters $\boldsymbol{\kappa} = (\kappa_1, \ldots, \kappa_k)$ representing the numbers of iterations along the corresponding loop paths. From the parametrized symbolic memory and from the path conditions corresponding to single loop iterations (received from the recursive call of `executeProgram`), we derive a *parametrized necessary condition* for each loop path, which is a formula over symbols and path counters $\boldsymbol{\kappa}$ that has to be satisfied when another loop iteration along the corresponding loop path can be performed after $\boldsymbol{\kappa}$ loop iterations. Finally, `processLoop` infers upper bounds from these parametrized necessary conditions with the help of the procedure `computeBounds`.

`computeSummary` is a subroutine of `processLoop` that gets symbolic memories corresponding to single loop iterations along each loop path and it produces the parametrized symbolic memory $\theta^{\boldsymbol{\kappa}}$ after an arbitrary number of loop iterations (as mentioned above).

`computeBounds` is another subroutine of `processLoop`. It gets a set I of loop paths and the corresponding parametrized necessary conditions, and derives upper bounds on the number of loop iterations along loop paths from I.

We describe the four procedures in the following four subsections. The procedure `processLoop` is described as the last one as it calls the other three procedures. We demonstrate the procedures and finally the whole algorithm on the programs of Figure 1. Descriptions of symbolic memories related to these programs omit the variables `n` and `A`: these variables are never changed and hence the value of `n` and `A` is always $\underline{n}$ and $\underline{A}$, respectively.

Algorithm 1: `executeProgram`

Input:
P // *a flowgraph*
Output:
$\{(\pi_1, \theta_1, \varphi_1), \dots, (\pi_k, \theta_k, \varphi_k)\}$ // *backbones* π_i *(with symbolic memory* θ_i *and*
// *path condition* φ_i *after execution along* π_i*)*
β // *for each edge* e *of* P*,* $\beta(e)$ *is a set of upper bounds for* e

1 `states` $\longleftarrow \emptyset$
2 Compute the set of backbones $\{\pi_1, \dots, \pi_k\}$ of P.
3 **foreach** $i = 1, \dots, k$ **do**
4 Initialize θ_i to return $\underline{a}$ for each (scalar or array) variable `a`.
5 $\varphi_i \longleftarrow$ *true*
6 Initialize β_i to return $\{0\}$ for each edge.
7 Let $\pi_i = v_1 \dots v_n$.
8 **foreach** $j = 1, \dots, n-1$ **do**
9 **if** v_j *is a loop entry* **then**
10 Let C be the loop with the loop entry v_j along π_i.
11 $(\beta_{loop}, \theta_i) \longleftarrow$ `processLoop(`$P(C, v_j), \theta_i, \varphi_i$`)`
12 **foreach** *edge* e *of* $P(C, v_j)$ **do**
13 $\beta_i(e) \longleftarrow \{\rho_1 + \rho_2 \mid \rho_1 \in \beta_i(e),\ \rho_2 \in \beta_{loop}(e)\}$
14 **if** $\iota((v_j, v_{j+1}))$ *has the form* `assume(`ψ`)` *and* $\theta_i(\psi)$ *contains no* $\star$ **then**
15 $\varphi_i \longleftarrow \varphi_i \wedge \theta_i(\psi)$
16 **if** $\iota((v_j, v_{j+1}))$ *has the form* `a` $:=$ *expr* **then**
17 $\theta_i(\mathtt{a}) \longleftarrow \theta_i(expr)$
18 $\beta_i((v_j, v_{j+1})) \longleftarrow \{\rho + 1 \mid \rho \in \beta_i((v_j, v_{j+1}))\}$
19 Insert $(\pi_i, \theta_i, \varphi_i)$ into `states`.
20 **foreach** *edge* e *of* P **do**
21 $\beta(e) \longleftarrow \{\mathbf{max}\{\rho_1, \dots, \rho_k\} \mid \rho_1 \in \beta_1(e), \dots,\ \rho_k \in \beta_k(e)\}$
22 **return** $(\mathtt{states}, \beta)$

3.1 Algorithm `executeProgram`

The procedure `executeProgram` of Algorithm 1 takes a flowgraph as input, determines its backbones, and symbolically executes each backbone separately. For a backbone π_i, symbolic execution computes symbolic memory θ_i, path condition φ_i, and *bound function* β_i assigning to each edge e a set of symbolic expressions that are valid upper bounds on the number of visits of edge e during any single run along the backbone. Each such a set $\beta_i(e)$ of bounds could be replaced by a single bound $\mathbf{min}\ \beta_i(e)$, but we prefer to keep it as a set of simpler expressions to increase the success rate of expression matching in the procedure `processLoop` (we point out the reason in Section 3.4).

The symbolic execution proceeds in the standard way until we enter a loop entry (line 9). Then we call procedure `processLoop` on the loop, current symbolic memory and path condition. The procedure returns function β_{loop} of upper bounds on visits of loop edges during execution of the loop, and a symbolic memory after execution of the loop. We add these bounds and the former bounds in

the **foreach** loop at line 12 and continue the execution along the backbone. If the `processLoop` procedure cannot determine the value of some variable after the loop, it simply uses the symbol $\star$ (unknown).

Another difference from the standard symbolic execution is at line 14 where we suppress insertion of predicates containing $\star$ to the path condition. As a consequence, a path condition of our approach is no longer a necessary and sufficient condition on input values to lead the program execution along the corresponding path (which is the case in the standard symbolic execution), but it is only a necessary condition on input values of a run to follow the backbone.

After processing an edge of the backbone, we increase the corresponding bounds by one (line 18). At the end of the procedure, the resulting bounds for each edge are computed as the maximum of previously computed bounds for the edge over all backbones (see the **foreach** loop at line 20). Besides these bounds, the procedure also returns each backbone with the symbolic memory and path condition after its execution.

Example 1. When `executeProgram` is called on the flowgraph of Figure 1(b), it finds two backbones $\pi_1 = cdefc'$ and $\pi_2 = cdfc'$. Since there are no loops along these backbones, their symbolic execution easily ends up with the corresponding symbolic memories and path conditions

$$\begin{array}{llll} \pi_1 : & \theta_1(\mathtt{i}) = \underline{i} + 1 & \theta_1(\mathtt{k}) = \underline{k} + 1 & \varphi_1 = \underline{i} < \underline{n} \ \wedge \ \underline{k} < 3 \ \wedge \ \underline{A}(\underline{i}) \neq 0 \\ \pi_2 : & \theta_2(\mathtt{i}) = \underline{i} + 1 & \theta_2(\mathtt{k}) = \underline{k} & \varphi_2 = \underline{i} < \underline{n} \ \wedge \ \underline{k} < 3 \ \wedge \ \underline{A}(\underline{i}) = 0 \end{array}$$

and a bound function β assigning $\{1\}$ to each edge of the flowgraph.

3.2 Algorithm `computeSummary`

The procedure `computeSummary` gets loop paths $\pi_1, \ldots, \pi_l$ together with symbolic memories $\theta_1, \ldots, \theta_l$, where each θ_i represents the effect of a single iteration along π_i. Then it assigns fresh path counters $\boldsymbol{\kappa} = (\kappa_1, \ldots, \kappa_l)$ to the loop paths and computes the parametrized symbolic memory $\theta^{\boldsymbol{\kappa}}$ after $\boldsymbol{\kappa}$ iterations of the loop, i.e. after $\sum_{1 \leq i \leq l} \kappa_i$ iterations where exactly κ_i iterations follow π_i for each i and there is no assumption on the order of iterations along different loop paths. If we do not find the precise value of some variable after $\boldsymbol{\kappa}$ iterations (for example because the value depends on the order of iterations along different loop paths), then $\theta^{\boldsymbol{\kappa}}$ assigns $\star$ (unknown) to this variable.

Due to the limited space, we do not provide any pseudocode or intuitive description of the procedure `computeSummary` here. Both can be found in [15]. It follows the ideas of the procedure of the same name introduced in [14].

Example 2. Assume that `computeSummary` gets symbolic memories θ_1, θ_2 corresponding to loop paths π_1, π_2 as computed in Example 1. It assigns path counters κ_1, κ_2 to π_1, π_2 respectively, and computes the parametrized symbolic memory $\theta^{\boldsymbol{\kappa}}$ describing the values of program variables after $\boldsymbol{\kappa} = (\kappa_1, \kappa_2)$ iterations of the loop that induces the flowgraph of Figure 1(b). Note that $\underline{i}, \underline{k}$ here represent the values of `i`, `k` just before the loop is executed.

$$\theta^{\boldsymbol{\kappa}}(\mathtt{i}) = \underline{i} + \kappa_1 + \kappa_2 \qquad \theta^{\boldsymbol{\kappa}}(\mathtt{k}) = \underline{k} + \kappa_1$$

Algorithm 2: `computeBounds`

Input:
  I *// indices of backbones*
  φ *// a necessary condition to perform an iteration along a backbone*
    // with an index in I after $\boldsymbol{\kappa}$ iterations
Output:
  B *// upper bounds on the number of iterations*
    // along backbones with indices in I

1 **if** $\varphi[\kappa_i/0 \mid i \in I]$ *is not satisfiable* **then return** $\{0\}$
2 $B \longleftarrow \emptyset$
3 **foreach** inequality $\sum_{j \in J \supseteq I} a_j \kappa_j < b$ implied by φ, where each a_j is a positive integer and b is $\boldsymbol{\kappa}$-free **do**
4   $B \longleftarrow B \cup \{\mathbf{max}\{0, \lceil b/\mathbf{min}\{a_i \mid i \in I\}\rceil\}\}$
5 **return** B

3.3 Algorithm `computeBounds`

The procedure `computeBounds` of Algorithm 2 gets a set I of selected loop path indices, and a necessary condition φ to perform an iteration along some loop path with an index in I (we talk about an *iteration along* I for short) after $\boldsymbol{\kappa}$ previous loop iterations. From this information, the procedure infers upper bounds on the number of loop iterations along I.

We would like to find a tight upper bound, i.e. a $\boldsymbol{\kappa}$-free symbolic expression B such that there exist some values of symbols (given by a valuation function v) for which the necessary condition $\varphi[\underline{a}/v(\underline{a}) \mid \underline{a} \text{ is a symbol}]$ to make another iteration along I is satisfiable whenever the number of finished iterations along I is less than $B[\underline{a}/v(\underline{a}) \mid \underline{a} \text{ is a symbol}]$ and the same does not hold for the expression $B + 1$. An effective algorithm computing these tight bounds is an interesting research topic itself.

The presented procedure infers some bounds only for two special cases. Line 1 covers the case when even the first iteration along any loop path in I is not possible: the procedure then returns the bound 0.

The other special case is the situation when the necessary condition implies an inequality of the form $\sum_{j \in J \supseteq I} a_j \kappa_j < b$, where each a_j is a positive integer and b is $\boldsymbol{\kappa}$-free. To detect these cases, we transform the necessary condition to the conjunctive normal form, look for clauses that contain just one predicate and try to transfer the predicate into this form. Each such inequality implies the following:

$$\sum_{j \in J \supseteq I} a_j \kappa_j < b \implies \sum_{i \in I} a_i \kappa_i < b \implies \mathbf{min}\{a_i \mid i \in I\} \cdot \sum_{i \in I} \kappa_i < b.$$

Hence, $\sum_{i \in I} \kappa_i < \lceil b/\mathbf{min}\{a_i \mid i \in I\}\rceil$ has to be satisfied to perform another iteration along I after $\boldsymbol{\kappa}$ previous iterations including $\sum_{i \in I} \kappa_i$ iterations along I. As all path counters are non-negative integers, we derive the bound $\mathbf{max}\{0, \lceil b/\mathbf{min}\{a_i \mid i \in I\}\rceil\}$ on iterations along I.

Example 3. We call `computeBounds`$(\{1\}, \varphi)$ to get bounds on κ_1 from the condition $\varphi = \kappa_1 + \kappa_2 < \underline{n} \;\wedge\; \kappa_1 < 3 \;\wedge\; \underline{A}(\kappa_1 + \kappa_2) \neq 0$. Since $\varphi[\kappa_1/0]$ is satisfiable, the procedure uses inequalities $\kappa_1 + \kappa_2 < \underline{n}$ and $\kappa_1 < 3$ implied by φ to produce bounds $B = \{\mathbf{max}\{0, \underline{n}\}, \mathbf{max}\{0, 3\}\} = \{\mathbf{max}\{0, \underline{n}\}, 3\}$.

3.4 Algorithm `processLoop`

The procedure `processLoop` of Algorithm 3 gets a flowgraph Q representing the body of a loop, i.e. each run of Q corresponds to one iteration of the original loop. We symbolically execute Q using the recursive call of `executeProgram` at line 2. We obtain all loop paths $\pi_1, \ldots, \pi_k$ of Q and bounds β_{inner} on visits of each edge in the loop during any single iteration of the loop. For each π_i, we also get the symbolic memory θ_i after one iteration along π_i and a necessary condition φ_i to perform this iteration. The procedure `computeSummary` produces the parametrized symbolic memory $\theta^{\boldsymbol{\kappa}}$ after $\boldsymbol{\kappa}$ iterations. Symbols $\underline{a}$ appearing in $\theta^{\boldsymbol{\kappa}}$ refer to variable values before the loop is entered. If we combine $\theta^{\boldsymbol{\kappa}}$ with the symbolic memory before entering the loop θ_{in}, we get the symbolic memory after execution of the code preceding the loop and $\boldsymbol{\kappa}$ iterations of the loop. We use this combination to derive necessary conditions $\varphi_i^{\boldsymbol{\kappa}}$ to perform another iteration along π_i and upper bounds $\beta^{\boldsymbol{\kappa}}$ on visits of loop edges in the next iteration of the loop.

The **foreach** loop at line 6 computes upper bounds for all edges of the processed loop on visits during all its complete iterations (incomplete iterations when a run cycles in some nested loop forever are handled later). We already have the bounds $\beta^{\boldsymbol{\kappa}}$ on visits in a single iteration after $\boldsymbol{\kappa}$ preceding iterations. For each edge e, we compute the set I of all loop path indices such that iterations along these loop paths can visit e. The `computeBounds` procedure at line 8 takes I and a necessary condition to perform an iteration along I after $\boldsymbol{\kappa}$ iterations and computes bounds B_{outer} on the number of iterations along I. If there is 0 among these bounds, e cannot be visited by any complete iteration and the computation for e is over. Otherwise we try to compute some overall bounds for each bound ρ_{inner} on the visits of e during one iteration (after $\boldsymbol{\kappa}$ iterations) separately. If ρ_{inner} is a $\boldsymbol{\kappa}$-free expression (line 13), then it is constant in each iteration and we simply multiply it with every bound on the number of iterations along I. The situation is more difficult if ρ_{inner} contains some path counters. We can handle the frequent case when it has the form $\mathbf{max}\{c, b + \sum_{i=1}^{k} a_i\kappa_i\}$, where $a_1, \ldots, a_k, b, c$ are $\boldsymbol{\kappa}$-free (see line 15 and note that this is the reason for keeping the bounds simple). First we get rid of path counters κ_j that have some influence on this bound (i.e. $a_j \neq 0$), but e cannot be visited by any iteration along loop path π_j. Let J be the set of indices of such path counters (line 16). We try to compute bounds B_J on the number of iterations along J (line 17), which are also the bounds on $\sum_{j \in J} \kappa_j$. Note that if $J = \emptyset$, we call `computeBounds`$(\emptyset, \mathit{false})$, which immediately returns $\{0\}$. If we get some bounds in B_J, we can overapproximate $\sum_{i=1}^{k} a_i\kappa_i$ as follows:

$$\sum_{i=1}^{k} a_i\kappa_i = \sum_{j \in J} a_j\kappa_j + \sum_{i \in I} a_i\kappa_i \;\leq\; \mathbf{max}\{0, a_j \mid j \in J\} \cdot \mathbf{min}\, B_J + \mathbf{max}\{a_i \mid i \in I\} \cdot \sum_{i \in I} \kappa_i$$

Algorithm 3: `processLoop`

Input:
- Q // *a flowgraph induced by a loop*
- θ_{in} // *a symbolic memory when entering the loop*
- φ_{in} // *a path condition when entering the loop*

Output:
- β_{loop} // *upper bounds for all edges in the loop*
- θ_{out} // *symbolic memory after the loop*

1. Initialize β_{loop} to return $\emptyset$ for each edge e of Q.
2. $(\{(\pi_1, \theta_1, \varphi_1), \ldots, (\pi_k, \theta_k, \varphi_k)\}, \beta_{inner}) \longleftarrow$ `executeProgram(`Q`)`
3. $\theta^{\boldsymbol{\kappa}} \longleftarrow$ `computeSummary(`$\{(\pi_1, \theta_1), \ldots, (\pi_k, \theta_k)\}$`)`
4. $\varphi_i^{\boldsymbol{\kappa}} \longleftarrow \varphi_{in} \wedge \theta_{in}\langle\theta^{\boldsymbol{\kappa}}\langle\varphi_i\rangle\rangle$ for each $i \in \{1, \ldots, k\}$
5. $\beta^{\boldsymbol{\kappa}}(e) \longleftarrow \{\theta_{in}\langle\theta^{\boldsymbol{\kappa}}\langle\rho\rangle\rangle \mid \rho \in \beta_{inner}(e)\}$ for each edge e of Q
6. **foreach** edge e of Q **do**
7. $I \longleftarrow \{i \mid e$ is on π_i or on a loop along $\pi_i\}$
8. $B_{outer} \longleftarrow$ `computeBounds(`$I, \bigvee_{i \in I} \varphi_i^{\boldsymbol{\kappa}}$`)`
9.   **if** $0 \in B_{outer}$ **then**
10. $\beta_{loop}(e) \longleftarrow \{0\}$
11.   **else**
12.   **foreach** $\rho_{inner} \in \beta^{\boldsymbol{\kappa}}(e)$ **do**
13.   **if** $\rho_{inner} \equiv c$ where c is $\boldsymbol{\kappa}$-free **then**
14. $\beta_{loop}(e) \longleftarrow \beta_{loop}(e) \cup \{c \cdot \rho_{outer} \mid \rho_{outer} \in B_{outer}\}$
15.   **else if** $\rho_{inner} \equiv \mathbf{max}\{c, b + \sum_{i=1}^{k} a_i \kappa_i\}$ where c, b and all a_i are $\boldsymbol{\kappa}$-free **then**
16. $J \longleftarrow \{j \mid j \notin I \wedge a_j \neq 0\}$
17. $B_J \longleftarrow$ `computeBounds(`$J, \bigvee_{j \in J} \varphi_j^{\boldsymbol{\kappa}}$`)`
18.   **if** $B_J \neq \emptyset$ **then**
19. $b' \longleftarrow b + \mathbf{max}\{0, a_j \mid j \in J\} \cdot \mathbf{min}\, B_J$
20. $a \longleftarrow \mathbf{max}\{a_i \mid i \in I\}$
21.   **foreach** $\rho_{outer} \in B_{outer}$ **do**
22. $\beta_{loop}(e) \longleftarrow \beta_{loop}(e) \cup \{\sum_{K=0}^{\rho_{outer}-1} \mathbf{max}\{c, b' + a \cdot K\}\}$
23. **foreach** edge e of Q **do**
24.   **if** an edge e' of Q such that $\beta^{\boldsymbol{\kappa}}(e') = \emptyset$ is reachable from e in Q **then**
25. $\beta_{loop}(e) \longleftarrow \{\rho_1 + \rho_2 \mid \rho_1 \in \beta_{loop}(e),\ \rho_2 \in \beta^{\boldsymbol{\kappa}}(e)$, and ρ_2 is $\boldsymbol{\kappa}$-free$\}$
26. $\theta_{out}(\mathtt{a}) \longleftarrow \theta_{in}\langle\theta^{\boldsymbol{\kappa}}(\mathtt{a})\rangle$ for each variable `a`
27. Eliminate $\boldsymbol{\kappa}$ from θ_{out}.
28. **return** $(\beta_{loop}, \theta_{out})$

Using the definitions of b' and a at lines 19–20, we overapproximate the bound ρ_{inner} on visits of e during one iteration along I after $\boldsymbol{\kappa}$ loop iterations by

$$\rho_{inner} = \mathbf{max}\{c, b + \sum_{i=1}^{k} a_i \kappa_i\} \ \leq \ \mathbf{max}\{c, b' + a \cdot \sum_{i \in I} \kappa_i\}.$$

As K-th iteration along I is preceded by $K-1$ iterations along I, the edge e can be visited at most $\mathbf{max}\{c, b' + a \cdot (K-1)\}$ times during K-th iteration. For each

bound ρ_{outer} on the iterations along I, we can now compute the total bound on visits of e as $\sum_{K=0}^{\rho_{outer}-1} \mathbf{max}\{c, b' + a \cdot K\}$.

Until now we have considered visits of loop edges during *complete* iterations. However, it may also happen that an iteration is started, but never finished because the execution keeps looping forever in some nested loop. For example, in the program `while(x>0){x:=x-1;while(true){}}`, we easily compute bound 0 on the number of complete iterations of the outer loop and thus we assign bound 0 to all loop edges at line 10. However, some edges of the loop are visited. These incomplete iterations are treated by the **foreach** loop at line 23. Whenever an edge e can be visited by an incomplete iteration (which is detected by existence of some subsequent edge e' without any bound and thus potentially lying on an infinite nested loop), we add the ($\boldsymbol{\kappa}$-free) bounds on visits of e during one iteration to the total bounds for e. If there is no such κ-free bound, we leave e unbounded to be on the safe side.

Finally, the lines 26 and 27 combine the symbolic memory before the loop with the effect of the loop and eliminate loop counters from the resulting symbolic memory θ_{out}. Roughly speaking, the elimination replaces every expression that is not $\boldsymbol{\kappa}$-free by $\star$. In fact, the elimination can be done in a smarter way. For example, after the loop in the program `i:=0;while(i<n){i:=i+1}`, the elimination can replace κ by $\mathbf{max}\{0, \theta_{out}(\mathtt{n})\}$.

Example 4. We demonstrate the whole algorithm on the program of Figure 1(a). We follow calls to individual procedures and we present the current state of the computation in terms of variables of the procedure at the top of the call stack.

The execution starts by calling `executeProgram` with the flowgraph at Figure 1(a). The flowgraph has only one backbone $\pi_1 = abcg$. The node c is the loop entry to the loop $\{c, d, e, f\}$ along the backbone. Symbolic execution of π_1 up to c is straightforward and leads to the symbolic memory $\theta_1(\mathtt{k}) = \theta_1(\mathtt{i}) = 0$, the path condition $\varphi_1 = \mathit{true}$, and the bound function β_1 maps each edge to $\{0\}$ except $\beta_1((a, b)) = \beta_1((b, c)) = \{1\}$. At the entry node c we build an induced flowgraph $P(\{c, d, e, f\}, c)$ depicted in Figure 1(b). Then we call `processLoop(`$P(\{c, d, e, f\}, c)$, θ_1, φ_1`)`.

`processLoop` calls `executeProgram` with the flowgraph at Figure 1(b), as we did in Example 1. Recall that `processLoop` receives the following

$$\begin{array}{llll} \pi_1 = cdefc' & \theta_1(\mathtt{i}) = \underline{i} + 1 & \theta_1(\mathtt{k}) = \underline{k} + 1 & \varphi_1 = \underline{i} < \underline{n} \wedge \underline{k} < 3 \wedge \underline{A}(\underline{i}) \neq 0 \\ \pi_2 = cdfc' & \theta_2(\mathtt{i}) = \underline{i} + 1 & \theta_2(\mathtt{k}) = \underline{k} & \varphi_2 = \underline{i} < \underline{n} \wedge \underline{k} < 3 \wedge \underline{A}(\underline{i}) = 0 \end{array}$$

and a bound function β_{inner} assigning $\{1\}$ to each edge of the flowgraph. Now we call `computeSummary` for the symbolic memories θ_1 and θ_2 and we get the parametrized symbolic memory $\theta^{\boldsymbol{\kappa}}$ described in Example 2:

$$\theta^{\boldsymbol{\kappa}}(\mathtt{i}) = \underline{i} + \kappa_1 + \kappa_2 \qquad \theta^{\boldsymbol{\kappa}}(\mathtt{k}) = \underline{k} + \kappa_1$$

Next, at line 4 of `processLoop` we compute necessary conditions to perform another iteration along backbones π_1 and π_2 respectively:

$$\begin{array}{lcl} \varphi_1^{\boldsymbol{\kappa}} & = & \kappa_1 + \kappa_2 < \underline{n} \ \wedge \ \kappa_1 < 3 \ \wedge \ \underline{A}(\kappa_1 + \kappa_2) \neq 0 \\ \varphi_2^{\boldsymbol{\kappa}} & = & \kappa_1 + \kappa_2 < \underline{n} \ \wedge \ \kappa_1 < 3 \ \wedge \ \underline{A}(\kappa_1 + \kappa_2) = 0 \end{array}$$

The next line produces bound function $\beta^{\boldsymbol{\kappa}}$ which is the same as β_{inner}, in this case. Now we have all data we need to start the computation of resulting bounds for all five edges of the passed flowgraph.

The main part of this computation is performed in the loop at line 6. We show the computation for the edge (e, f). First we call `computeBounds(`$\{1\}, \varphi_1^{\boldsymbol{\kappa}}$`)`. As shown in Example 3, we obtain the set $B_{outer} = \{\mathbf{max}\{0, \underline{n}\}, 3\}$. Since $0 \notin B_{outer}$ and $\beta^{\boldsymbol{\kappa}}((e, f)) = \{1\}$, we get to the line 14 in `processLoop`, where we receive $\beta_{loop}((e, f)) = \{\mathbf{max}\{0, \underline{n}\}, 3\}$. The computation proceeds similarly for other edges, but for (c, d), (d, f), (f, c) it produces only one bound $\{\mathbf{max}\{0, \underline{n}\}\}$. The difference originates in the calls of `computeBounds`. For (c, d) and (f, c), we call `computeBounds(`$\{1, 2\}, \varphi_1^{\boldsymbol{\kappa}} \vee \varphi_2^{\boldsymbol{\kappa}}$`)` and get only the bound $B_{outer} = \{\mathbf{max}\{0, \underline{n}\}\}$. For (d, f), we call `computeBounds(`$\{2\}, \varphi_2^{\boldsymbol{\kappa}}$`)` and get the same single bound. Since, the condition at line 24 is false for all edges, the resulting β_{loop} returns $\{\mathbf{max}\{0, \underline{n}\}, 3\}$ for (d, e) and (e, f), and $\{\mathbf{max}\{0, \underline{n}\}\}$ for the others. The resulting symbolic memory θ_{out} assigns $\star$ to `i` and `k`.

The control-flow then returns back to `executeProgram` where we update β_1 according to received β_{loop}. Then we symbolically execute the remaining edge (c, g). The computation in the loop at line 20 computes maximum over all bounds for a considered edge. The algorithm then terminates with the bound function β assigning $\{1\}$ to edges (a, b), (b, c), (c, g), the set $\{\mathbf{max}\{0, \underline{n}\}\}$ to edges (c, d), (d, f), (f, c), and the set $\{\mathbf{max}\{0, \underline{n}\}, 3\}$ to (d, e) and (e, f).

We can conclude for the flowgraph at Figure 1(a) that the loop can be executed only if the program is called with some positive integer $\underline{n}$ for the parameter `n`. In that case the loop is executed at most $\mathbf{max}\{0, \underline{n}\}$ times (according to $\beta((c, d))$), but the path following the `if` branch can be executed at most $\mathbf{min}\{\mathbf{max}\{0, \underline{n}\}, 3\}$ times. So the asymptotic complexity for the program is $\mathcal{O}(\underline{n})$, but $\mathcal{O}(1)$ for the `if` branch inside the loop.

4 Experimental Evaluation

We implemented our algorithm in an experimental program analysis tool called Looperman. It is built on top of the symbolic execution package Bugst [17] and it intensively uses the SMT solver Z3 [21].

We compared Looperman with state-of-the-art loop bound analysis tools Loopus [12], KoAT [5], PUBS [1], and Rank [3] on 199 simple C programs used in previous comparisons of loop bound analysis tools [19,20]. We focused on two kinds of bounds: asymptotic complexity bounds for whole programs and *exact* (meaning non-asymptotic) bounds for individual program loops. The comparison of asymptotic complexity bounds and other details about our experimental evaluation can be found in [15]. Here we present only the comparison of exact bounds, which was restricted to Looperman and Loopus as the other tools use input in a different format and (as far as we know) they do not provide any mapping of their bounds to the original C code. Note that Loopus is a strong competitor as it achieved the best results in the asymptotic complexity bounds.

Table 1. Comparison of loop bounds inferred by LOOPERMAN and LOOPUS.

	LOOPERMAN	LOOPUS
correctly bounded loops	227	267
incorrectly bounded loops	0	3
loops with no bound found	86	43
bounded loops, not bounded by the other	11	51
asymptotically tighter bounds	16	11
tighter bounds, but not asymptotically	44	2

The presented experiments run on a machine with 8GB of RAM and Intel i5 CPU clocked at 2.5GHz. We apply the 60 seconds time limit to the analysis of one program by one tool. The LOOPERMAN tool (both sources and Windows binaries), the 199 benchmarks, and all measured data are available here [18].

The 199 benchmarks contain 313 loops. Table 1 provides for both tools the numbers of correctly and incorrectly bounded loops, and the number of loops for which no bound is inferred. The second part of the table compares the inferred loop bounds. It presents the number of loops where one tool produces a correct loop bound while the other does not, the number of loops where one tool provides an asymptotically tighter loop bound than the other, and the number of loops where one tool infers a tighter bound than the other tool, but the difference is not asymptotic (e.g. n versus $2n$). To complete the presented data, let us note that both tools inferred exactly the same bound for 143 loops.

The results show that LOOPUS can infer bounds for slightly more loops than LOOPERMAN. However, there are also loops bounded by LOOPERMAN and not by LOOPUS. The biggest advantage of LOOPERMAN is definitely the tightness of its bounds: LOOPERMAN found a tighter bound for 28% of 216 loops bounded by both tools, while LOOPUS found a tighter bound only for 6% of these loops.

5 Related Work

Techniques based on recurrence equations attempt to infer a system of recurrence equations from a loop (or a whole program) and to solve it. PUBS [1] focuses primarily on solving of the system generated by another tool, e.g. [2]. R-TUBOUND [11] builds a system of recurrence equations by rewriting multi-path loops into single-path ones using SMT reasoning. The system is then solved by a pattern-based algorithm. In ABC [4], inner loops are instrumented by iteration counters (one counter for a whole loop). Recurrence equations are then constructed over program variables and counters. SPEED [7] instruments counters into the program (one counter for each back-edge) as artificial variables. Then it computes their upper bounds by a linear invariant generation tool. In our approach, we use recurrence equations and counters to summarise loops. We compute upper bounds from necessary conditions for executing backbones. In contrast to [4,7], we introduce a counter for each loop path and counters are not instrumented.

Rank [3] applies an approach based on ranking functions. It reuses results from the termination analysis of a given program (i.e. a ranking function) to get an asymptotic upper bound on the length of all program executions. KoAT [5] uses ranking functions of already processed loops to compute bounds on values of program variables, which are then used to improve ranking functions of subsequent loops. Loopus [16] uses several heuristics to transform a program in particular locations so that variables appearing there represent ranking functions. Program loops are then summarised per individual paths through them. The approach was further improved by merging nested loops [12] and by computation of maximal values of variables [13]. Our algorithm does not use ranking functions. However, the passing of information from a preceding to a subsequent loop we see in [5] or [13] happens also in our approach, through symbolic execution. The loop summarisation per individual loop paths presented in [16] is similar to ours. However, while [16] computes summary as a transitive hull expressed in the domain of a size-change abstraction, we compute precise symbolic values of variables after loops. In contrast to [12], we do not merge nested loops.

There are other important techniques computing upper bounds, which are, however, less related to our work. For instance, SWEET [9] uses abstract interpretation to derive bounds on values of program variables and a pattern matching of loops of predefined structure. In [8], a program is transformed with respect to a given location: preserving reachability from the location back to itself. Loops are summarised into disjunctive invariants from which upper bounds are computed using a technique based on proof-rules. `WISE` [6] symbolically executes all paths up to a given length in order to infer a branching policy for longer paths. Then it symbolically executes all paths satisfying the policy. The longest path represents the worst-case execution time of the program.

6 Conclusion

We presented an algorithm computing upper bounds for execution counts of individual instructions of an analysed program during any program run. The algorithm is based on symbolic execution and the concept of path counters. The upper bounds are parametrized by input values of the analysed program. Evaluation of our experimental tool Looperman shows that our approach often infers loop bounds that are tighter than these found by leading loop bound analysis tools. This may be a crucial advantage in some applications including the worst case execution time (WCET) analysis.

Acknowledgement P. Čadek has been supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF) and by the Vienna Science and Technology Fund (WWTF) through grant ICT12-059, J. Strejček by the Czech Science Foundation grant GBP202/12/G061, and M. Trtík by the QBOBF project funded by DGCIS/DGA.

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *SAS*, volume 5079 of *LNCS*, pages 221–237. Springer, 2008.
2. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. In *ESOP*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.
3. C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.
4. R. Blanc, T. A. Henzinger, T. Hottelier, and L. Kovács. ABC: Algebraic bound computation for loops. In *LPAR*, volume 6355 of *LNCS*, pages 103–118. Springer, 2010.
5. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, volume 8413 of *LNCS*, pages 140–155. Springer, 2014.
6. J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473. IEEE, 2009.
7. S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139. ACM, 2009.
8. S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, pages 292–304. ACM, 2010.
9. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *RTSS*, pages 57–66. IEEE, 2006.
10. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
11. J. Knoop, L. Kovács, and J. Zwirchmayr. Symbolic loop bound computation for WCET analysis. In *PSI*, volume 7162 of *LNCS*, pages 227–242. Springer, 2012.
12. M. Sinn, F. Zuleger, and H. Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, volume 8559 of *LNCS*, pages 745–761. Springer, 2014.
13. M. Sinn, F. Zuleger, and H. Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *FMCAD*, pages 144–151. IEEE, 2015.
14. J. Strejček and M. Trtík. Abstracting path conditions. In *ISSTA*, pages 155–165. ACM, 2012.
15. P. Čadek, J. Strejček, and M. Trtík. Tighter loop bound analysis (technical report). *CoRR*, abs/1605.03636, 2016.
16. F. Zuleger, S. Gulwani, M. Sinn, and H. Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS*, volume 6887 of *LNCS*, pages 280–297. Springer, 2011.
17. `Bugst`. `http://sourceforge.net/projects/bugst/`.
18. `Looperman, benchmarks, and evaluation`. `https://sourceforge.net/projects/bugst/files/Looperman/1.0.0/`.
19. `http://aprove.informatik.rwth-aachen.de/eval/IntegerComplexity`.
20. `http://forsyte.at/static/people/sinn/loopus/CAV14/index.html`.
21. `Z3`. `https://github.com/Z3Prover/z3`.