A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis

Moritz Sinn, Florian Zuleger, and Helmut Veith
TU Vienna

Abstract. We present the first scalable bound analysis that achieves amortized complexity analysis. In contrast to earlier work, our bound analysis is not based on general purpose reasoners such as abstract interpreters, software model checkers or computer algebra tools. Rather, we derive bounds directly from abstract program models, which we obtain from programs by comparatively simple invariant generation and symbolic execution techniques. As a result, we obtain an analysis that is more predictable and more scalable than earlier approaches. Our experiments demonstrate that our analysis is fast and at the same time able to compute bounds for challenging loops in a large real-world benchmark. Technically, our approach is based on lossy vector addition systems (VASS). Our bound analysis first computes a lexicographic ranking function that proves the termination of a VASS, and then derives a bound from this ranking function. Our methodology achieves amortized analysis based on a new insight how lexicographic ranking functions can be used for bound analysis.

1 Introduction

Automated methods for computing bounds on the resource consumption of programs are an active area of research [16,13,14,21,5,15,3]. We present the first scalable bound analysis for imperative programs that achieves amortized complexity analysis. Our techniques can be applied for deriving upper bounds on how often loops can be iterated as well as on how often a single or several control locations can be visited in terms of the program input.

The majority of earlier work on bound analysis has focused on mathematically intriguing frameworks for bound analysis. These analyses commonly employ general purpose reasoners such as abstract interpreters, software model checkers or computer algebra tools (see related work below). The cited approaches were either evaluated only on a small benchmark [10,13,14,5,3] or rely on elaborate heuristics to work in practice [11,14]. In this paper we take an orthogonal approach which complements previous research. We propose a bound analysis based on a simple abstract program model, namely lossy vector addition systems with states. We present a static analysis with four well-defined analysis phases that are executed one after each other: program abstraction, control-flow abstraction, generation of a lexicographic ranking function and bound computation. We experimentally evaluate our approach on a large real-world benchmark of C programs. We demonstrate that our approach is very fast in practice and
at the same time able to compute bounds for challenging loops: our current tool
computes more loop bounds than our earlier tool \cite{21} while at the same time
dramatically improving the performance!

Our technical key contribution is a new insight how lexicographic ranking
functions can be used for bound analysis. Earlier approaches such as \cite{4} simply
count the number of elements in the image of the lexicographic ranking func-
tion in order to determine an upper bound on the possible program steps. The
same idea implicitly underlies the bound analyses \cite{3,10,13,11,14,21}. However,
this reasoning misses arithmetic dependencies between the components of the
lexicographic ranking function (see Section \ref{sec:related}). In contrast, our analysis calculates
how much a lexicographic ranking function component is increased when
another component is decreased. This enables amortized analysis.

\textit{Related Work}. An interesting line of research studies the \textit{amortized analysis}
of first-order functional programs (e.g. \cite{16,15}) formulated as type rules over a
template potential function with unknown coefficients; these coefficients are then
found by linear programming. It is not clear how to transfer this approach to an
imperative setting. Promising first steps for the amortized analysis of imperative
programs are reported in \cite{3}. Quantifier elimination is applied for simplifying a
constraint system over template cost functions. Since quantifier elimination is
expensive, the technique does not yet scale to larger programs.

\textit{Lexicographic ranking functions} in automated termination analysis have been
pioneered by Bradley et al. (see \cite{6} and follow-up papers) who employ an elabo-
rate constraint solving technique. A recent paper experimentally compares ter-
mination analysis by lexicographic ranking and transition invariants \cite{8} imple-
mented on top of a software model checker. \cite{4} iteratively constructs a lexi-
ographic ranking function by solving linear constraint systems. \cite{7} is a hybrid
of the approaches \cite{8} and \cite{4}. \cite{6,8} and \cite{7} compute a lexicographic ranking
function for a \textit{single} control location (i.e., one loop header) at a time, while
the application of bound analysis requires to find a common lexicographic ranking
function for \textit{all} control locations. \cite{4} computes such a ranking function, is
however limited to fairly small programs. Our approach complements the cited
approaches as it represents a simple and scalable construction of a lexicographic
ranking function for all control locations.

\textit{Bound Analysis}. The COSTA project (e.g. \cite{3}) studies the extraction of cost
recurrence relations from Java bytecode programs and proposes new methods
for solving them with the help of computer algebra systems. \cite{10} proposes to ex-
tend the polyhedra abstract domain with max- and non-linear expressions. \cite{13}
introduces multiple counters and exploits their dependencies such that upper
bounds have to be computed only for restricted program parts. \cite{11} proposes
an abstract interpretation-guided program transformation that separates the
different loop phases such that bounds can be computed for each phase in isolation.
\cite{14} employs proof-rules for bound computation combined with disjunctive
abstract domains for summarizing inner loops. \cite{21} proposes a bound analysis
based on the size-change abstract domain. The cited approaches to bound analy-
sis employ sophisticated reasoning engines such as computer algebra tools \cite{3}.
Example 1.

```c
void SingleLinkCluster(uint n) {
    int a = n, b = 0;
    l1: while (a > 0) {
        a--; b++;
    } l2: while (b > 0) {
        b--;
    } l3: for (int i = a-1; i > 0; i--) {
        if (?) {
            a--; b++;
        }
    } l4: }
```

Fig. 1. Example 1 shows our running example, '?' denotes non-determinism (arising from a condition not modeled in the analysis). On the right we state the lossy VASS obtained by abstraction from Example 1. \( Id \) denotes \( a' \leq a \), \( b' \leq b \), \( i' \leq i \).

complex abstract domains [10], powerset abstract domains [14,21] and program transformations combined with abstract interpretation techniques [13,11]. We complement the cited approaches by a simpler methodology that is more scalable and also more predictable, because we do not need to rely on elaborate heuristics such as widening.

## 2 Motivation and Overview

Example 1 presented in Figure 1 (encountered during our experiments) is challenging for an automated bound analysis: (C1) There are loops whose loop counter is modified by an inner loop: the innermost loop modifies the counter variables \( a \) and \( b \) of the two outer loops; thus it cannot be sliced away during the analysis of the two outer loops. (C2) The dependencies between the counter variables need to be tracked on the level of individual loop transitions: computing the linear loop bound \( n \) of the middle loop requires to track how often the if-branch of the innermost loop is executed (at most \( n \) times) and not only how often the innermost loop can be executed (at most \( n^2 \) times). (C3) Computing the linear loop bound \( n \) of the middle loop requires to recognize that \( b \) is incremented, when \( a \) is decremented. Current bound analysis techniques cannot model such increments of \( b \) and instead need to assume that \( b \) is reset to some value between 0 and \( n \). For this reason no bound analysis from the literature is able to compute the linear loop bound \( n \) for the middle loop. We now illustrate the main steps of our analysis:

1. **Program Abstraction:** First, our analysis abstracts the program to the VASS depicted in Figure 1. We introduce VASSs in Section 3. In this paper we are using parameterized VASSs, where we allow increments that are symbolic but constant throughout the program (such as \( n-1 \)). We extract lossy VASSs from C programs using simple invariant generation and symbolic execution techniques (described in Section 7).

2. **Control Flow Abstraction:** We propose a new abstraction for bound analysis, which we call control flow abstraction (CA) (described in Section 4).
abstracts the VASS from Figure 1 into a transition system with four transitions:
\begin{align*}
\rho_1 &\equiv a' \leq a - 1 \land b' \leq b + 1 \land i' \leq i, \\
\rho_2 &\equiv a' \leq a \land b' \leq b - 1 \land i' \leq i + (n - 1), \\
\rho_3 &\equiv a' \leq a \land b' \leq b \land i' \leq i - 1, \\
\rho_4 &\equiv a' \leq a - 1 \land b' \leq b + 1 \land i' \leq i - 1.
\end{align*}
CA effectively merges loops at different control locations into a single loop creating one transition for every cyclic path of every loop (without unwinding inner loops). This significantly simplifies the design of the later analysis phases.

3. Ranking Function Generation: Our ranking function generation (Algorithm 2 stated in Section 5) finds an order on the transitions resulting from CA such that there is a variable for every transition, which decreases on that transition and does not increase on the transitions that are lower in the order. This results in the lexicographic ranking function \( l = \langle a, a, b, i \rangle \) for the transitions \( \rho_1, \rho_4, \rho_2, \rho_3 \) in that order. Our soundness theorem (Theorem 1) guarantees that \( l \) proves the termination of Example 1.

4. Bound Analysis: Our bound analysis (Algorithm 3 stated in Section 6) computes a bound for every transition \( \rho \) by adding for every other transition \( \tau \) how often \( \tau \) increases the variable of \( \rho \) and by how much. In this way, our bound analysis computes the bound \( n \) for \( \rho_2 \), because \( \rho_2 \) can be incremented by \( \rho_1 \) and \( \rho_4 \), but this can only happen \( n \) times, due to the initial value \( n \) of \( a \). Further, our bound analysis computes the bound \( n \ast (n - 1) \) for \( \rho_3 \) from the fact that only \( \rho_2 \) can increase the counter \( i \) by \( n - 1 \) and that \( \rho_2 \) has the already computed transition-bound \( n \). Our soundness result (Theorem 2) guarantees that the bound \( n \) obtained for \( \rho_2 \) is indeed a bound on how often the middle loop of Example 1 can be executed.

Our bound analysis solves the challenges (C1)-(C3): CA allows us to analyze all loops at once (C1) creating one transition for every loop path (C2). The abstract model of lossy VASS is precise enough to capture arithmetic dependencies and our bound analysis is able to exploit these dependencies (C3).

2.1 Amortized Complexity Analysis

In his influential paper [20] Tarjan introduces amortized complexity analysis using the example of a stack, which supports two operations push (which puts an element on the stack) and popMany (which removes several elements from the stack). He assumes that the cost of push is 1 and the cost of popMany is the number of removed elements. We use his example to discuss how our bound analysis achieves amortized analysis:

Example 2.
```
void main(int m) {
    int i=m, n = 0; //stack = emptyStack();
    while (i > 0) {
        i--;
        if (?) //push
            n++; //stack.push(element);
        else //popMany
            while (n > 0 \&\& ?)
                n--; //element = stack.pop();
    }
} 
```
Our analysis first abstracts the program to a VASS and then applies CA. This results in the three transitions $\rho_1 \equiv i' = i - 1 \wedge n' = n + 1, \rho_2 \equiv i' = i - 1 \wedge n' = n - 1$ (the first two transitions come from the outer loop, the last transition from the inner loop). Algorithm 2 then computes the lexicographic ranking function $\langle i, i, n \rangle$ for the transitions $\rho_1, \rho_2, \rho_3$ in that order. Our bound analysis (Algorithm 3) then computes the joint bound $m$ for the transitions $\rho_1$ and $\rho_2$. Our bound analysis further computes the bound $m$ for transition $\rho_3$ from the fact that only $\rho_1$ can increase the counter $n$ by 1 and that $\rho_1$ has the already computed bound $m$. Adding these two bounds gives the amortized complexity bound $2m$ for Example 2. We highlight that our analysis has actually used the variable $n$ of transition $\rho_3$ as a potential function (see [20] for a definition)! A lexicographic ranking function $\langle x_1, \ldots, x_n \rangle$ can be seen as a multidimensional potential function. Consider, for example, the ranking function $\langle a, a, b, i \rangle$ for the transitions $\rho_1, \rho_4, \rho_2, \rho_3$ of Example 1. The potential of $\rho_3$ can be increased by $\rho_2$ whose potential in turn can be increased by $\rho_1$ and $\rho_4$.

3 Lossy VASSs and Basic Definitions

In this section we define the notion of a lossy VASS and state definitions that we need later on. In the following we often drop the ‘lossy’ in front of ‘VASS’ because we do not introduce non-lossy VASSs in this paper and there is no danger of confusion.

Definition 1 (Lossy Vector Addition System with States (VASS)). We fix some finite set of variables $\text{Var} = \{x_1, \ldots, x_n\}$. A lossy vector addition system with states (VASS) is a tuple $P = (L, E)$, where $L$ is a finite set of locations, and $E \subseteq L \times \mathbb{Z}_n \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{d} l_2$ to denote an edge $(l_1, d, l_2)$ for some vector $d \in \mathbb{Z}^n$. We often specify the vector $d \in \mathbb{Z}^n$ by predicates $x'_i \leq x_i + d_i$ with $d_i \in \mathbb{Z}$.

A path of $P$ is a sequence $l_0 \xrightarrow{d_0} l_1 \xrightarrow{d_1} \cdots \xrightarrow{d_i} l_{i+1} \in E$ for all $i$. A path is cyclic, if it has the same start- and end-location. A path is simple, if it does not visit a location twice except for start- and end-location. We write $\pi = \pi_1 \cdot \pi_2$ for the concatenation of two paths $\pi_1$ and $\pi_2$, where the end-location of $\pi_1$ is the start-location of $\pi_2$. We say $\pi'$ is a subpath of a path $\pi$, if there are paths $\pi_1$ and $\pi_2$ with $\pi = \pi_1 \cdot \pi' \cdot \pi_2$.

The set of valuations of $\text{Var}$ is the set $V_{\text{Var}} = \text{Var} \rightarrow \mathbb{N}$ of mappings from $\text{Var}$ to the natural numbers. A trace of $P$ is a sequence $(l_0, \sigma_0) \xrightarrow{d_0} (l_1, \sigma_1) \xrightarrow{d_1} \cdots$ such that $l_0 \xrightarrow{d_0} l_1 \xrightarrow{d_1} \cdots$ is a path of $P$, $\sigma_i \in V_{\text{Var}}$ and $\sigma_{i+1} \leq \sigma_i + d_i$ for all $i$. $P$ is terminating, if there is no infinite trace of $P$.

Values of VASS variables are always non-negative. We discuss why this is a realistic modeling assumption in Section 7. The non-negativity of VASS values has two important consequences: (1) Transitions in VASSs contain implicit guards: for example a transition $x' \leq x + c$ can only be taken if $x + c \geq 0$. (2) VASS transitions can be used to model variable increments as well as variable
resets: we replace the assignment $x := k$, where $k \in \mathbb{Z}$, by the VASS transition $x' \leq x + k$ during program abstraction. This only increases the set of possible program traces and thus provides a conservative abstraction.

**Minimal Program Model.** VASSs offer a minimal program model for bound analysis. We use locations in VASSs for modeling the control structure of imperative programs. We use VASS variables for modeling local progress measures on the program state such as loop counters (we discuss this in Section 7). In this paper we focus on the analysis of sequential programs without procedures. We leave the extension to concurrent and recursive programs for future work.

**Parameterized VASSs.** In our implementation we use a slight generalization of lossy VASSs. We allow the increment $n$ in a transition predicate $x' \leq x + n$ to be symbolic but constant; in particular, we require that $n$ does not belong to the set of variables $\text{Var}$. Our bound algorithm works equally well with symbolic increments under the condition that we know the sign of $n$. We call these extended systems parameterized VASSs. See Figure 1 for an example.

In the following we introduce some standard terminology that allows us to precisely speak about loops and related notions.

**Definition 2 (Reducible Graph, Loop Header, Natural Loop, Loop-nest Tree, e.g. [2]).** Let $G = (V, E)$ be a directed graph with a unique entry point such that all nodes are reachable from the entry point. A node $a$ dominates a node $b$, if every path from entry to $b$ includes $a$. An edge $l_1 \rightarrow l_2$ is a back edge, if $l_2$ dominates $l_1$. $G$ is reducible, if $G$ becomes acyclic after removing all back edges. A node is a loop header, if it is the target of a back edge. The (natural) loop of a loop header $h$ in a reducible graph is the maximal set of nodes $L$ such that for all $x \in L$ (1) $h$ dominates $x$ and (2) there is a back edge from some node $n$ to $h$ such that there is path from $x$ to node $n$ that does not contain $h$.

We list some consequence of the above definition: Every natural loop is uniquely defined by its loop header. Two natural loops $A$ and $B$ are either disjoint (i.e., $A \cap B = \emptyset$) or nested inside each other (i.e., $A \subseteq B$ or $B \subseteq A$). Further, in case of $A \subseteq B$, set containment is proper if and only if $A$ and $B$ have different loop headers.

In the rest of this paper we restrict ourselves to VASSs and programs whose control flow graph is reducible. This choice is justified by the fact that irreducible control flow is very rare in practice (e.g. see the study in [19]). For analyzing irreducible programs we propose to use program transformations that make the program reducible; we do not elaborate this idea further due to lack of space.

Next, we define a special case of path, which is an important notion in this paper.

**Definition 3 (Loop-path).** A loop-path $\pi$ is a simple cyclic path, which starts and ends at some loop header $l$, and visits only locations inside the natural loop of $l$. 
**Procedure**: \( \text{CA}(P) \)

**Input**: a reducible VASS \( P \)

**Output**: a transition system \( T \)

\[
T := \emptyset; \\
\text{foreach loop header } l \text{ in } P \text{ do} \\
\quad \text{foreach loop-path } \pi = l \xrightarrow{d_1} l_1 \cdots l_{n-1} \xrightarrow{d_n} l \text{ do} \\
\quad \quad T := T \cup \{d_1 + \cdots + d_n\}; \\
\text{return } T; 
\]

**Algorithm 1**: CA performs control flow abstraction

**Example**: \( l_2 \xrightarrow{\tau_2} l_3 \xrightarrow{Id} l_2 \) is a loop-path for the VASS in Figure 1. However, \( l_2 \xrightarrow{Id} l_1 \xrightarrow{\tau_1} l_2 \) is not a loop-path because it does not stay inside the natural loop of \( l_2 \). \( l_2 \xrightarrow{\tau_2} l_3 \xrightarrow{Id} l_4 \xrightarrow{\tau_1} l_3 \xrightarrow{Id} l_2 \) is not a loop-path, because it is not simple (\( l_3 \) is visited twice).

**Definition 4 (Instance of a loop-path in a path)**. Let \( \pi = \pi_1 \cdot t_1 \cdot C_1 \cdot t_2 \cdots C_{i-1} \cdot t_i \cdot \pi_2 \) be a path, where \( t_i \in E \) are transitions in \( \pi \) and where \( \pi_1, \pi_2 \) and \( C_i \) are subpaths of \( \pi \). We say that the transitions \( t_1, \ldots, t_i \) belong to an instance of a loop-path \( \nu \) with header \( l \), if (i) the \( C_i \) are cyclic and do not contain \( l \), and (ii) we have \( t_1 \cdot t_2 \cdots t_i = \nu \).

The following facts about instances are easy to see: Every transition in a given path belongs to at most one instance of a loop-path. Every transition in a given cyclic path belongs to exactly one instance of a loop-path.

**Example**: There are four instances of loop-paths in the path \( \pi = l_1 \xrightarrow{\tau_1} l_2 \xrightarrow{\tau_2} l_3 \xrightarrow{Id} l_4 \xrightarrow{\tau_3} l_3 \xrightarrow{Id} l_2 \xrightarrow{Id} l_1 \), \( l_2 \xrightarrow{Id} l_2 \xrightarrow{Id} l_1 \) (twice) and \( l_3 \xrightarrow{Id} l_4 \xrightarrow{\tau_3} l_3 \).

**4 Control Flow Abstraction**

In this section we introduce control flow abstraction (CA). CA is a mapping from VASSs to transition systems.

**Definition 5 (Transition System)**. A transition system is a set of vectors \( d \in \mathbb{Z}^n \). We often specify a vector \( d \in \mathbb{Z}^n \) by predicates \( x'_i \leq x_i + d_i \) with \( d_i \in \mathbb{Z} \).

Note that transition systems are not meant to be executed. The sole purpose of transition systems is to be used for ranking function generation and bound analysis. CA is based on two main ideas: (1) Given a program \( P \), CA results into one transition for every loop-path of \( l \) for all loop headers \( l \) of \( P \). (2) The control structure is abstracted: effectively, all loops are merged into a single loop. The first idea enables a path-sensitive analysis, which ensures high precision during ranking function generation and bound analysis. The second idea allows to compute a common lexicographic ranking function for all loops later on. Our algorithm for CA is stated as Algorithm 1.
Procedure: Ranking($T$)

Input: a transition system $T$

Output: a lexicographic ranking function $l$, which has one component for every transition $\rho \in T$

$S := T$;

$l := \text{"lexicographic ranking function with no components";}$

while there is a transition $\rho \in S$ and a variable $x$ such that $\rho \models x' < x$ and for all $\rho' \in S$ we have $\rho' \models x' \leq x$ do

$S := S \setminus \rho$;

$l := l.\text{append}(x)$;

if $S = \emptyset$ then return $l$;
else return \"Transitions $S$ may be non-terminating\";

Algorithm 2: Ranking computes a lexicographic ranking function

Loop-path Contraction. Algorithm 1 creates one transition for every loop-path $\pi = l \xrightarrow{d_1} l_1 \cdots l_{n-1} \xrightarrow{d_n} l$. The transition $d_1 + \cdots + d_n$ represents the accumulated effect of all variable increments along the path. The key idea of loop-path contraction is to ignore any inner loop on $\pi$. We will incorporate the effects of the inner loops only later on during the ranking function generation and bound analysis phase. Loop-path contraction is an abstraction in itself since intermediate guards are lost: e.g., the loop-path $\pi = l \xrightarrow{x' \leq x-1} l' \xrightarrow{x' \leq x+1} l$ can only be executed if $x > 0$. By CA we obtain the transition $x' \leq x$. During bound analysis we assume that the transition $x' \leq x$ can be taken if $x \geq 0$!

CA represents our choice of precision in the analysis: CA facilitates a high degree of disjunctiveness in the analysis, where we keep one disjunct for every loop-path. By encapsulating the level of precision in a single analysis phase, we achieve a modular analysis (only during CA we need to deal with the control structure of the VASS). This simplifies the design of the later termination and bound analysis and also allows us to easily adjust the analysis precision (see the discussion on path merging in Appendix C).

5 Ranking Function Generation

In this section we discuss our algorithm for ranking function generation (Algorithm 2).

Algorithm. Algorithm 2 reads in a transition system obtained from CA and returns a lexicographic ranking function that provides a witness for termination. The key idea of the algorithm is to incrementally construct the lexicographic ranking function from local ranking functions. We call a variable $x$ a local ranking function for a transition $\rho$, if $\rho \models x' < x$; $x$ proves that $\rho$ cannot be taken infinitely often without taking other transitions. Algorithm 2 maintains a set of transitions $S$ for which no local ranking function has been added yet. In each
step the algorithm checks if there is a transition $\rho$ in $S$ and a variable $x$ such that (1) $x$ is a local ranking function for $\rho$ and (2) no remaining transition increases the value of $x$, i.e., the condition $\forall \rho' \in S, \rho' \models x' \leq x$ is satisfied. If (1) and (2) are satisfied, $\rho$ is removed from the set of remaining transitions $S$ and $x$ is added as the component for $\rho$ in the lexicographic ranking function $l$. Conditions (1) and (2) ensure that the transition $\rho$ cannot be taken infinitely often if only transitions from $S$ are taken (remember that $x$ is always positive and hence cannot be decreased infinitely often). The algorithm stops, if no further transition can be removed. If $S$ is empty, the lexicographic ranking function $l$ is returned. Otherwise it is reported that the remaining transitions $S$ might lead to non-terminating executions.

Next we state the correctness of the combined application of Algorithm 1 and Algorithm 2. The proof can be found in the appendix.

**Theorem 1.** If Algorithm 2 returns a lexicographic ranking function $l$ for the transition system $T$ obtained from Algorithm 1 then VASS $P$ is terminating.

**Reasons for Failure.** There are two reasons why our ranking function generation algorithm may fail: First, there can be a transition $\rho$, which has no local ranking function, i.e., there is no variable $x$ with $\rho \models x' < x$. Such a transition $\rho$ will never be removed from $S$. Second, there can be a cyclic dependency between local ranking functions. This is the case if we have a local ranking function $x_i$ for all transitions $\rho_i \in S$ but the second condition for all $\rho' \in S$ we have $\rho' \models x'_i \leq x_i$ fails for all $\rho_i \in S$. We found cyclic dependencies to be very rare in practice (we only found 4 instances during our experiments, see Appendix D).

## 6 Bound Computation

In this section we introduce our bound algorithm. We first define our notion of bound.

**Definition 6 (Path-bound).** A bound for a loop-path $\pi$ is an expression $b$ over $\text{Var}$ such that every trace of $P$ contains at most $b$ instances of $\pi$.

Path-bounds have various applications in bound and complexity analysis: the computational complexity of a program can be obtained by adding the bounds of the loop-paths of all loops; a loop bound can be obtained by adding the bounds of all loop-paths of a given loop; the number of visits to a single control location $l$ can be obtained by adding the bounds of the loop-paths that include $l$ (our notion of a path-bound can be seen as a path-sensitive generalization of the notion of a “reachability-bound” [14]); similarly we can compute the number of visits to a set of control locations.

**Example:** We have obtained the loop bound of the middle loop in Example 1 from the path-bound $n$ of its single transition $\rho_2$ in Section 2. We have obtained $2n$ as the amortized complexity of Example 2 by adding the path-bounds of all its transitions in Section 2.1.
Algorithm 3: \textbf{Bound} returns a bound for transition $\rho$

Procedure: \textbf{Bound}(\rho)

\textbf{Input}: a transition $\rho$

\textbf{Output}: a bound for transition $\rho$

\textbf{Global}: transition system $\mathcal{T}$, lexicographic ranking function $l$

\begin{algorithmic}
\STATE $x := \text{ranking function component of } \rho \text{ in } l$
\STATE $b := \text{InitialValue}(x)$
\FOR{\text{transition } $\rho' \in \mathcal{T}$ \text{ with } $\rho' \not\mid x' \leq x$}
\STATE Let $k \in \mathbb{N}$ s.t. $x' \leq x + k$ in $\rho'$\;
\STATE $b := b + \text{Bound}(\rho') \cdot k$
\ENDFOR
\STATE Let $k \in \mathbb{N}$ s.t. $x' \leq x - k$ in $\rho$
\STATE \textbf{return} $b = b/k$
\end{algorithmic}

\textit{Algorithm.} Algorithm 3 computes a bound $b$ for a transition $\rho$ of the transition system $\mathcal{T}$. The main idea of Algorithm 3 is to rely only on the components of the lexicographic ranking function $l$ for bound computation. Let $x$ be the component of $\rho$ in $l$. We recall that the termination algorithm has already established that $x$ is a local ranking function for $\rho$ and therefore we have $\rho \mid x > x'$. Thus $\rho$ can be executed at most $\text{InitialValue}(x)$ often unless $x$ is increased by other transitions: Algorithm 3 initializes $b := \text{InitialValue}(x)$ and then checks for every other transition $\rho'$ if it increases $x$, i.e., $\rho' \not\mid x' \leq x$. For every such transition $\rho'$ Algorithm 3 recursively computes a bound, multiplies this bound by the height of the increase $k$ and adds the result to $b$. Finally, we divide $b$ by the decrease $k$ of $x$ on transition $\rho$. We have given an example application of Algorithm 3 in Section 2.

\textit{Termination.} Algorithm 3 terminates because the recursive calls cannot create a cycle. This is because Algorithm 3 uses only the components of $l$ for establishing bounds and the existence of the lexicographic ranking function $l$ precludes cyclic dependencies.

Next we state the correctness of the bound computation algorithm. The proof can be found in the appendix.

\textbf{Theorem 2.} Let $b$ be a bound computed by Algorithm 3 for a transition $\rho$ obtained from a loop-path $\pi$ during CA. Then $b$ is an upper bound on the number of instances of $\pi$ in every trace of $P$.

\textit{Complexity of the Algorithm / Size of Bound Expressions.} For ordinary VASS, the complexity of Algorithm 3 is polynomial in the size of the input with a small exponent (depending on the exact definition of the complexity parameters). Unfortunately, this statement does not hold for parameterized VASSs, for which bound expressions can be exponentially big: We consider $n$ transitions $\rho_1, \ldots, \rho_n$ with the local ranking functions $x_1, \ldots, x_n$ and the lexicographic ranking function $(x_1, \ldots, x_n)$. We assume that transition $\rho_i$ increments $x_j$ by some constant $c_{ij}$ for $i < j$. Then, Algorithm 3 computes the bound stated in the following
formula, which is exponentially big for symbolic coefficients $c_{ij}$:

$$b(\rho_n) = \sum_{k \in [0,n-1]} \prod_{i_1 < \cdots < i_k \in [1,n-1]} \text{InitialValue}(x_{i_1}) c_{i_1i_2} \cdots c_{i_kn}$$

However, in practical examples the variable dependencies are sparse, i.e., most coefficients $c_{ij}$ are zero (confirmed by our experiments). We highlight that Algorithm 3 exploits the sparsity of the dependencies as it does not compute the bound using the explicit formula stated above but rather computes the bound for the current transition $\rho$ using only the bounds of the transitions that actually increase the counter of $\rho$ (i.e., $c_{ij} > 0$). We note that in our experiments the computed bounds are small and the running time of Algorithm 3 is basically linear in the number of transitions. We conclude that in practice one should make use of the fine-grained precision offered by the possibly exponentially-sized bound expressions.

**Preprocessing: Merging Transitions.** Before computing bounds with Algorithm 3 our analysis applies the following preprocessing: Let $\rho_1$ and $\rho_2$ be two transitions that have the same ranking function component $x$ in the lexicographic ranking function $l$ and that contain the same decrement $x' \leq x - k$ for some $k \in \mathbb{N}$. $\rho_1$ and $\rho_2$ are replaced by a new transition $\rho$ that contains the transition predicate $x' \leq x - k$ and the transition predicates $y' \leq y + \max\{k_1, k_2\}$ for variables $y \neq x$ with $y' \leq y + k_1 \in \rho_1$ and $y' \leq y + k_2 \in \rho_2$. It is not difficult to see that merging transitions is sound and always improves the bound computed by Algorithm 3 (we do not give a formal justification here for lack of space).

### 7 Program Abstraction

In this section we describe how a program given in source code can be abstracted to a lossy VASS.

**Definition 7 (Program).** Let $\Sigma$ be a set of states. The set of transition relations $\Gamma = 2^{\Sigma \times \Sigma}$ is the set of relations over $\Sigma$. A program is a tuple $P = (L, E)$, where $L$ is a finite set of locations, and $E \subseteq L \times \Gamma \times L$ is a finite set of transitions. We write $l_1 \xrightarrow{\rho} l_2$ to denote a transition $(l_1, \rho, l_2)$.

We now define the relation of a given program $P$ to the VASS on which our analysis is performed:

**Definition 8 (Norm).** A norm $\text{norm} \in \Sigma \rightarrow \mathbb{N}$ is a function that maps the states to the natural numbers (including 0).

**Definition 9 (Abstraction of a Program).** Let $P = (L, E)$ be a program. A VASS $V = (L, E')$ with variables $\text{Var}$ over a set of norms $N$ is an abstraction of the program $P$ iff for each transition $l_1 \xrightarrow{\rho} l_2 \in E$ there is a transition $l_1 \xrightarrow{d} l_2 \in E'$ s.t. for all $n' \leq n + c \in d$ and for all $(s_1, s_2) \in \rho$ it holds that $n(s_2) \leq n(s_1) + c$. 

7.1 Extracting VASSs from Programs

Extracting Norms. The key idea of Algorithm 2 is to associate a local ranking function to every loop-path of the underlying program. This motivates the following heuristic for extracting norms: We consider any integer-valued expression \( r : \Sigma \rightarrow \mathbb{Z} \) a norm candidate, if \( r \) is a local ranking function on at least one loop-path \( \pi \) of the program. This heuristic is implemented in two steps: (1) Expressions from the conditions on \( \pi \) are extracted, e.g., we extract \( n - i \) from the condition \( n > i \). (2) Symbolic execution is applied to check if the extracted expressions decrease on \( \pi \). Our experiments confirm that this heuristic works well in practice. A norm candidate \( r \) is included in the set of norms \( N \), if \( r \) satisfies the invariant \( r \geq 0 \). This invariant holds for most local ranking functions: An expression \( r \) that is indeed a local ranking function is typically counted down until zero and is not used to store negative values at other parts of the program. Any numeric invariant analysis, e.g., interval or octagon analysis, can be used for establishing the required invariants.

Abstracting Transitions. We can directly abstract increments \( x' = x + k \) and resets \( x' = k \) by the transition predicate \( x' \leq x + k \), if \( k \) is an integer or a variable that is constant throughout the program. If \( k \) is not constant, we search for an invariant \( k \leq u \), where \( u \) is an expression that contains only constant variables. This invariant then allows us to obtain the transition predicate \( x' \leq x + u \). We implement invariant analysis by a set of proof rules stated in Appendix B.

Non-linear Local Ranking Functions. In our experiments we only found a few non-linear local ranking functions, which almost always arise from iterated division or multiplication by a constant such as in the transition relation \( x > 1 \land x' = x/2 \). In such a case we introduce a shadow variable \( y \) for the logarithm of \( x \), i.e. \( y = \log x \), and replace the transition relation by \( y > 0 \land y' = y - 1 \) during program abstraction.

Data Structures. We propose to abstract programs with data structures into integer programs before the actual analysis using appropriate norms such as the length of a list or the number of characters in a string. For a detailed description of suitable abstraction techniques we refer the reader to [12,18]. In our implementation (discussed in Section 8) we use a light-weight abstraction based on optimistic aliasing assumptions.

8 Experiments

We implemented the discussed approach as an intraprocedural analysis based on the LLVM [17] compiler framework. As our reasoning engine we employ the Z3 SMT solver [9]. We evaluated the tool on the program and compiler optimization benchmark Collective Benchmark [1](cBench). The cBench contains a total of 1027 different C files (we removed code duplicates in order to get an unbiased result) with 211,892 lines of code.
Analyzed | Outer Dep | Inner Dep | Paths > 1 | Non-Trivial
--- | --- | --- | --- | ---
Loops | | | | |
Bounded | 1278 | 78% | 127 | 744 | 886 |
Overall | 1276 | 58% | 1514 | 832 |
SCCs | | | | |
Bounded | 2833 | 81% | 241 | 542 | 566 |
Overall | 2005 | 50% | 938 | 566 |

Fig. 2. Loop and SCC Statistic of our current implementation for the cBench Benchmark, the results obtained with the implementation of [21] are given in square brackets.

The bounds computed by our tool are integer expressions over the variables available at the entry point of the strongly connected component (SCC) in which the loop under consideration is situated. An SCC consists of a top-level loop (a loop that does not have an outer loop) and all its inner loops. For expressing bounds of loops iterating over arrays or recursive data structures the tool introduces shadow variables representing an appropriate measure such as the length of a list or the size of an array.

**Setting.** Our tool models integers as mathematical integers (not bit-vectors) and makes the following optimistic assumptions which are reported to the user: Pointers do not alias (237 cases), a recursive data structure is acyclic if a loop iterates over it (204 cases), a loop iterating over an array of characters is assumed to be terminating if an inequality check on the string termination character '\0' is found (204 cases), given a loop condition of form $a \neq 0$ we heuristically decide whether $a > 0$ or $a < 0$ can be assumed as loop-invariant (122 cases), given a counter update of form $x = x \times 2$ or $x = x/2$ we assume $x > 0$ (2 cases). We made these assumptions in order to find interesting examples, a manual check on a sample of around 100 loops in the benchmark found the assumptions to be valid with respect to termination. The task of validating an assumption is orthogonal to our approach and can be performed by any standard tool for invariant computation.

**Results.** Our tool computes loop bounds and at the same time analyzes the computational complexity of the SCCs. In Figure 2 we give our results on different loop classes. We recall that our bound analysis is based on an explicitly computed termination proof. We do not list the results of our termination analysis separately, because a bound was computed for 98% of all loops for which termination was proven. A detailed analysis of the cases in which our termination analysis fails is given in Appendix D. In column Analyzed we state the results over all loops in the benchmark. We summarize the results over all loop categories except Analyzed in the column Non-Trivial.

**Challenging Loop Classes.** We defined syntactic categories for loops that deviate from standard for-loops ‘for(i = 0; i < n; i++)’. These categories pose a challenge to bound analysis tools. The categories for the SCCs are the same as for the loops: we define an SCC to be in a certain category if it contains at least...

---

1 This assumption is necessary since the type system of C does not distinguish between an array of characters and a string
one loop which is in that category. The loop-class ‘*Outer Dependent*’ captures all outer loops whose termination behavior is affected by the executions of an inner loop. (E.g., in Example 1 termination of loop $l_1$ depends on loop $l_3$, while in Example 2 termination of loop $l_1$ does not depend on loop $l_2$.) We define an inner loop to be in the set of ‘*Inner Dependent*’ loops if it has a loop counter that is not reset before entering the loop. (E.g., in Example 1 loop counter $i$ of loop $l_3$ is always reset to $a - 1$ before entering the loop, while loop counter $b$ of loop $l_2$ is never reset.) The loop-class ’*Paths > 1*’ contains all loops which have more than 1 path left after program slicing (see Appendix C) was applied. Success ratios of 50% and more in the difficult categories demonstrate that our method is able to handle non-trivial termination and complexity behavior of real world programs!

*Amortization.* For 179 loops out of the 455 loops in the class ‘*Inner Dependent*’, the bound that our tool computed was amortized in the sense that it is asymptotically smaller than one would expect from the loop-nesting depth of the loop. In 11 cases the amortization was caused by incrementing a counter of the inner loop in the outer loop as in Examples 1 and 2. For these loops a precise bound cannot be computed by any other tool for real-world software (including our earlier tool [21])!

*Performance.* The discussed results were obtained on a Linux machine with a 3.2 Ghz dual core processor and 8 GB Ram. Our implementation assumes reducible control flow as discussed in Section 3. 92 loops of the 4302 loops in our benchmark are located in 44 SCCs with an irreducible control flow. We thus analyzed 4210 loops. The total runtime of our tool on the benchmark (more than 200,000 LoC) did not exceed 40 minutes! The time out limit of maximal 420 seconds computation time per SCC was not reached. There were only 27 out of 2832 SCCs (174 out of 4210 loops) on which the analysis spent more than 10 seconds. The only complexity source of our approach is its path-sensitivity. Thanks to the simplicity and modularity of our method, we can tackle the path explosion problem by two simple path reduction techniques (see Appendix C).

*Experimental Comparison.* We compared our tool against our previous work [21], which to the best of our knowledge represents the only other experimental evaluation of a bound analysis on a large publicly-available benchmark of C programs. For the purpose of a realistic comparison, we ran the tool of [21] on the same machine with an equal time out limit of 420 second. The results are given in square brackets in Figure 2. Note the significant increase in the number of loops bounded in each of the challenging categories. The execution of the tool [21] took an order of magnitude longer (nearly 13 hours) and we got 78 time outs. The main reason for the drastic performance increase is our new reasoning on inner loops: The approach of [21] handles inner loops by inserting the transitive hull of an inner loop on a given path of the outer loop. This can blow up the number of paths exponentially. We avoid this exponential blow-up thanks to CA: CA allows us to analyze inner and outer loops at the same time and thus eliminates the need for transitive hull computation.
References

1. http://ctuning.org/wiki/index.php/CTools:CBench
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. Elvira Albert, Samir Genaim, and Abu Naser Masud. On the inference of resource usage upper and lower bounds. *ACM Transactions on Computational Logic*, 2013. To appear.
4. Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonmand. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In SAS, pages 117–133, 2010.
5. Diego Esteban Alonso-Blas and Samir Genaim. On the limits of the classical approach to cost analysis. In SAS, pages 405–421, 2012.
6. Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. Linear ranking with reachability. In CAV, pages 491–504, 2005.
7. Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In CAV, pages 413–429, 2013.
8. Byron Cook, Abigail See, and Florian Zuleger. Ramsey vs. lexicographic termination proving. In TACAS, pages 47–61, 2013.
9. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In TACAS, pages 337–340, 2008.
10. Bhargav S. Gulavani and Sumit Gulwani. A numerical abstract domain based on expression abstraction and max operator with application in timing analysis. In CAV, pages 370–384, 2008.
11. Sumit Gulwani, Sagar Jain, and Eric Koskinen. Control-flow refinement and progress invariants for bound analysis. In PLDI, pages 375–385, 2009.
12. Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In POPL, pages 239–251, 2009.
13. Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. Speed: precise and efficient static estimation of program computational complexity. In POPL, pages 127–139, 2009.
14. Sumit Gulwani and Florian Zuleger. The reachability-bound problem. In PLDI, pages 292–304, 2010.
15. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14, 2012.
16. Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In POPL, pages 185–197, 2003.
17. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In CGO, pages 75–88, 2004.
18. Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In POPL, pages 211–222, 2010.
19. James Stanier and Des Watson. A study of irreducibility in c programs. *Softw., Pract. Exper.*, 42(1):117–130, 2012.
20. R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, April 1985.
21. Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In SAS, pages 280–297, 2011.
A Proofs

Proof (of Theorem 1). Assume Algorithm 2 returns a lexicographic ranking function $l$. For the sake of contradiction we assume that there is an infinite trace $(l_0, \sigma_0) \xrightarrow{d_0} (l_1, \sigma_1) \xrightarrow{d_1} \cdots$ of $P$. Thus there is at least one loop-path with infinitely many instances in the path $\pi = l_0 \xrightarrow{d_0} l_1 \xrightarrow{d_1} \cdots$. By definition of CA $\mathcal{T}$ contains one transition per loop-path of $P$. Given a transition $\rho$, we denote by $\phi(\rho)$ the loop-path from which $\rho$ has been obtained. Now, we fix a transition $\rho$ that is the minimal transition in $l$ such that $\phi(\rho)$ has infinitely many instances in $\pi$. Let $k$ be an index in $\pi$ such that all loop-path that have only finitely many instances appear before $k$. Let $x$ be the ranking function component of $\rho$ in $l$. We denote by $x' \leq x + c_i$ the transition predicate of $x$ in $d_i$. Let $k \leq k_1 < k_2 < \cdots$ be the start indices of the infinitely many instances of $\rho$ in $\pi$. We consider the accumulated effect of $x$ between locations $k_i$ and $k_{i+1}$. We have $x_i+1 \leq x_i + c_{k_i} + c_{k_{i+1}} + \cdots + c_{k_{i+1}}$ ($\ast$). We note that the path $l_{k_i} \xrightarrow{d_{k_i}} \cdots \xrightarrow{d_{k_{i+1}-1}} l_{k_{i+1}}$ is cyclic and therefore can be completely decomposed into instances of loop-paths. We now reorder the sum ($\ast$) in order to aggregate the summands which belong to instances of the same loop-path (this is justified by the fact that plus is commutative): $x_{i+1} \geq x_i + n_{\rho'} \cdot c_{\rho'} + n_{\rho''} \cdot c_{\rho''} + \cdots$ ($\ast\ast$), where $c_{\rho*}$ denotes the total of the summands of the loop-path $\phi(\rho*)$ and $n_{\rho*}$ denotes the number of instances of the loop-path $\phi(\rho*)$. Because $\rho$ has been chosen minimal we have that no transition $\rho*$ appears before $\rho$ in $l$, and we get $\rho* \models x' \leq x$. Therefore, $c_{\rho*} \leq 0$ for all $\rho*$. We know that $\rho$ appears in ($\ast\ast$) and that $c_{\rho} < 0$ because $x$ is the local ranking function of $\rho$ in $l$. Thus we get $x_{i+1} < x_i$ from ($\ast\ast$). Because that holds for all $i$ we get an infinitely decreasing chain. This is impossible because all $x_i$ are non-negative by assumption.

Proof (of Theorem 2). Our soundness result rests on the technical condition that $P$ is an SCC with a unique entry point, that is also the unique exit point of the SCC. We can always ensure this condition by a program transformation that encloses $P$ in a dummy while-loop

\begin{verbatim}
while(y > 0) { P; y-- },
\end{verbatim}

where $y$ is a fresh variable with $\text{InitialValue}(y) = 1$.

By definition of CA $\mathcal{T}$ contains one transition per loop-path of $P$. Given a transition $\rho$, we denote by $\phi(\rho)$ the loop-path from which $\rho$ has been obtained. Let $\rho$ be a transition of the VASS $\mathcal{T}$. Let $x$ be the ranking function component of $\rho$ in $l$. Let $(l_0, \sigma_0) \xrightarrow{d_0} (l_1, \sigma_1) \xrightarrow{d_1} \cdots$ be a trace of $P$ that starts and ends at the unique entry point of $P$. We denote by $x' \leq x + c_i$ the update of $x$ in $d_i$. By definition of a VASS, variables take values only in the non-negative numbers. In particular, the final value of $x$ is non-negative, i.e., $\text{InitialValue}(x)+c_0+c_1\cdots\geq 0$ ($\ast$). Because $l_0 \xrightarrow{d_0} l_1 \xrightarrow{d_1} \cdots$ is cyclic the sequence ($\ast$) can be completely decomposed into instances of loop-paths. Because plus is commutative we can reorder the sequence ($\ast$) and aggregate the summands that belong to instances of the same loop-path: $\text{InitialValue}(x)+n_\rho \cdot c_\rho+n_\rho' \cdot c_\rho'+n_\rho'' \cdot c_\rho''+\cdots\geq 0$ ($\ast\ast$), where
$c_\rho$ denotes the total of the summands of the loop path $\phi(\rho*)$ and $n_\rho*$ denotes the number of instances of the loop path $\phi(\rho*)$. Algorithm 2 has established $c_\rho < 0$ and $c_\rho* \leq 0$ for all transitions $\rho*$ that appear after $\rho$ in $l$.

We now prove the claim by induction on the position of $\rho$ in the lexicographic ranking function $l$. Base case: $\rho$ is first and we get $\text{InitialValue}(x) + n_\rho \cdot c_\rho \geq 0$ from (**), and thus $n_\rho \leq \text{InitialValue}(x)$. Step case: by induction assumption we have $n_\rho* \leq \text{Bound}(\rho*)$ for all transitions $\rho*$ that appear before $\rho$ in $l$. We get $\text{InitialValue}(x) + n_\rho \cdot c_\rho + \text{Bound}(\rho') \cdot c_\rho' + \text{Bound}(\rho'') \cdot c_\rho'' + \cdots \geq 0$ from (**), and thus $n_\rho \leq (\text{InitialValue}(x) + \text{Bound}(\rho') \cdot c_\rho' + \text{Bound}(\rho'') \cdot c_\rho'' + \cdots)/c_\rho$.

### B Proof Rules for Transition Abstraction

**Example 3.**

```c
void main(int n, int m) {
  int a = m; int i = 0;
  l1: while(i < n) {
    j = a;
  }
  l2: while(j > 0) {
    j--;
    a = a + 4; i++;
  }
}
```

We implement the invariant analysis needed for abstracting transitions by four proof rules:

1. If there is a condition $k \leq n$ dominating the increment $x' = x + k$ resp. reset $x' = k$ of $x$ such that $n$ is constant throughout the program, we obtain the invariant $k \leq n$.

2. If $k$ is never increased throughout the program, we obtain the invariant $k \leq \text{InitialValue}(k)$. For example, we consider the assignment $i = (a - 1)$ in Example 1. Since $a$ is never increased and $\text{InitialValue}(a) = n$, we model this assignment by the transition predicate $i' \leq i + (n - 1)$.

3. If there is a transition on which $k$ is incremented, we recursively apply our bound analysis to establish how often the increment can be executed. We explain this rule on Example 3. Counter $j$ of the inner loop is reset on the outer loop to $a$. Proof rule 2 fails, because variable $a$ is increased on the outer loop. Proof rule 3 now applies our bound algorithm for establishing the transition bound $n$ for the transition on which $a$ is incremented. This gives us the invariant $a \leq \text{InitialValue}(a) + n + 4$, where $\text{InitialValue}(a) = m$.

4. If $k = x - y$ is a composed linear expression, we recursively compute an upper bound for $x$ and a lower bound for $y$. We employ symmetrical rules for the computation of lower bound invariants.

We found the described proof rules to be sufficient for most of the programs we encountered during our experiments.
C Path Reduction

| # paths | 1  | 2  | 3 - 9 | 10 - 99 | 100 - 299 | 300 - 999 | 2000 - 4999 | ≥ 5000 |
|---------|----|----|-------|---------|-----------|-----------|-------------|-------|
| unsliced | 1174 | 578 | 616   | 66      | 44        | 11        | 34          |       |
| sliced  | 1623 | 512 | 424   | 183     | 37        | 30        | 8           | 16    |
| merged  | 1766 | 429 | 415   | 186     | 24        | 10        | 2           |       |
| refined | 1766 | 429 | 414   | 187     | 24        | 10        | 1           | 2     |

Fig. 3. Number of paths with respective number of SCCs

First we apply program slicing with regard to the loop exit conditions, i.e. we delete all program behavior that cannot affect loop termination. In the next step we exclude path doubles through syntactic comparison. On loops with more than 250 paths left, we apply what we call path merging: For each path $p$ the conjunction $c_p$ over all its predicates is built. Paths which assign syntactically identical expressions to the loop counters are grouped. The paths in the same group $G$ are substituted by a new path with the single predicate $\bigvee_{p \in G} c_p$ (we simplify this predicate by standard techniques from propositional logic). The merged path overapproximates all paths in $G$. Though some path sensitivity is lost by this technique, we can still bound 81 (31%) of the 259 loops to which path merging is applied. After the second path reduction step we apply standard control flow refinement techniques such as loop unrolling. Figure 3 states the number of SCCs with the respective number of paths in the original program (first row), in the sliced program (second row), after deleting path duplicates and applying path merging (third row) and after applying control flow refinement (last row). We state the paths per SCC because in our implementation all loops in one SCC are processed at once. Our statistics (Figure 3) demonstrate that slicing and merging significantly reduce the number of paths while control flow refinement does not lead to any problematic increase in the path count.

D Limitations

For a total of 897 loops our implementation failed to prove termination. For 35 loops our tool proved termination but was not able to infer a bound. We distinguish the two reasons for failure of our termination analysis discussed in Section 5:

- No local ranking function. For 833 loops our analysis failed because there was a loop-path with no local ranking function. We analyzed on a random sample of 50 loops out of the 833 loops the reasons for failure. We first report on the loops that could be handled by our analysis, if we improved the modeling features of our implementation: bitwise operations (not modeled, 5 cases), function inlining (applied very restrictive, 4 cases), unsigned integers (modeled as integers, 2 cases), external functions (not modeled, 4 cases). In 5 cases termination is conditional and cannot be proven (e.g., a character stream must contain the line break character). In 7 cases function pointers need to be resolved. In 27 cases our invariant analysis is insufficient (7 array invariants, 20 arithmetic invariants).
*Cyclic dependencies.* For a total of 64 loops, our analysis found a local ranking function for every loop-path but was not able to compute a lexicographic ranking function, because of cyclic dependencies. However, a manual inspection of the 64 loops revealed that only 4 cases are indeed instances of a cyclic dependency failure. For the other 60 loops the real reason for failure is that our tool was not able to compute the right local ranking function for certain loop-paths (for the same reasons discussed above); the reported cyclic dependency was caused by variables that were actually not relevant for the termination of the loop. In the 4 remaining cases control flow refinement by contextualization \cite{21} would resolve the cyclic dependency.