Symbolic Simulation on Complicated Loops for WCET Path Analysis

Duc-Hiep Chu*
hiepcd@comp.nus.edu.sg
National University of Singapore

Joxan Jaffar
joxan@comp.nus.edu.sg
National University of Singapore

ABSTRACT

We address the Worst-Case Execution Time (WCET) *Path Analysis* problem for bounded programs, formalized as discovering a tight upper bound of a resource variable. A key challenge is posed by complicated loops whose iterations exhibit non-uniform behavior. We adopt a brute-force strategy by simply *unrolling* them, and show how to make this scalable while preserving accuracy.

Our algorithm performs *symbolic simulation* of the program. It maintains accuracy because it preserves, at critical points, *path-sensitivity*. In other words, the simulation detects infeasible paths. Scalability, on the other hand, is dealt with by using *summarizations*, compact representations of the analyses of loop iterations. They are obtained by a judicious use of abstraction which preserves critical information flowing from one iteration to another. These summarizations can be *compounded* in order for the simulation to have *linear complexity*: the symbolic execution can in fact be *asymptotically shorter* than a concrete execution. Finally, we present a comprehensive experimental evaluation using a standard benchmark suite. We show that our algorithm is fast, and importantly, we often obtain not just accurate but *exact* results.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis; B.2.2 [**Performance Analysis and Design Aids**]: Verification,Worst-case analysis

General Terms

Reliability, Verification, Algorithms

Keywords

WCET, Path Analysis, Interpolation, Summarization

1. INTRODUCTION

Programs use limited physical resources. Thus determining an upper bound on resource usage by a program is often a critical need. *Static* estimation of the Worst-Case Execution Time (WCET) has traditionally been important in the design of real-time systems. Static methods emphasize *safety* by producing bounds on the execution time, guaranteeing that the execution time will not exceed these bounds.

A main issue in WCET analysis is to *avoid pessimism* while being *safe* in timing evaluation. Ideally, WCET estimation method should, given an input program, produce a *tight estimate* of the upper-bound of the actual WCET. But first, we need a *timing model* of the hardware platform. Indeed, such micro-architecture modeling for low-level analysis is non-trivial and it is almost impossible to achieve exact WCET estimates in CPU cycles. Second, it is crucial to estimate accurately bounds for loops and eliminate *infeasible* paths from bound calculation, especially in the presence of nested loops. This can be partially addressed by requiring user-provided path annotations and loop bound information. Apart from considerable effort and error-proneness, sometimes the user may not actually know such information. A more attractive solution is to automatically detect infeasible paths and derive loop bounds through *static path analysis* methods [2, 12, 16, 17].

Path analysis in general is performed separately from low-level analysis. Theiling et al. [35], though their path analysis is not fully automated, emphasize that precise WCET prediction can be achieved by doing low-level analysis and path analysis separately. As a matter of fact, our path analysis is performed separately from low-level analysis. It is intended to be combined with some low-level analysis (e.g. [35]), which gives a worst-case timing for each basic block.

When path analysis is performed separately from low-level analysis, a key issue is the *aggregation phase*, lifting basic block timings (returned by some low-level analysis) to the global timing. At this phase, the information about infeasible paths and loop bounds is crucial because it allows us to exclude certain accumulations of basic block timings which do not correspond to valid paths. This paper adopts *symbolic simulation* with *loop unrolling* for *automatic* and *precise* detection of infeasible paths and loop bounds.

Infeasible path detection concerns path-sensitivity: without it, accuracy is seriously hampered; but with it, how do we make any algorithm scale given the subsequent explosion in the search space of the symbolic execution? For instance, in Fig. 1(e), the WCET of a piece of code depends on the values of its input variables. The fact of whether an analyzer can capture no/partial/full information about the input variables might heavily affect its timing prediction. Similarly, in Fig. 1(f), the paths (a,c) and (b,d) are mutually exclusive. Excluding those paths from bound calculation might increase the analysis precision significantly [2].

We next discuss the inherent difficulties posed by complicated loops. Scalability is discussed in the later sections. Here we simply point out some technical aspects of programs that *exacerbate* the already difficult problem.

*This author is supported by NUS Graduate School for Integrative Sciences and Engineering.

EMSOFT'11, October 9–14, 2011, Taipei, Taiwan.

```
for (i = 0; i < n-1; i++)
    for (j = 0; j < n-1-i; j++) {
        /* test_and_swap */
    }
```
(a)

```
for (i = 0; i < n; i++)
    for (j = 0; j < n-i; j++) {
        /* do_something */
        i++;
    }
```
(b)

```
for (i = 0; i < 10; i++) {
    if (i==4) { /* a */ }
    /* b */
}
```
(c)

```
while (n > 1) {
    if (n % 2 == 0)
        n = n / 2;
    else n = 3 * n + 1;
}
```
(d)

```
if (input == 0) { /* do_a */ }
else { /* do_b */ }
```
(e)

```
if (E < 0) { condition = 0; }     /* a */
else { condition = 1; }           /* b */
if (condition) result = x / y;    /* c */
else result = y;                  /* d */
```
(f)

Figure 1: Challenging program patterns

• *Non-rectangular loops*: we often see triangular loops in sorting algorithms. Fig. 1(a) shows bubblesort program. The number of iterations of the inner loop is dependent on the specific iteration of the outer loop. In bounding the total number of the inner loop iterations in this program, general techniques on a parametric bound would happily accept n^2 as a good bound. Nonetheless, we target the exact bound $n(n-1)/2$ for each known value of n.

• *Amortized loops* [15]: in Fig. 1(b), the outer loop counter being manipulated inside the inner loop makes it hard to give a tight bound (*linear* instead of *quadratic*).

• *Down-sampling code*: predicting accurately the loop timing is hard if one part of its body is executed less often than the rest of the body (Fig. 1(c)). When the timing for /* a */ is significantly larger than the timing for /* b */, the amount of overestimation might become unacceptable.

• *Closed-form is not always possible*: a WCET analysis can produce symbolic expressions which are solved (closed-form) by using off-the-shelf Computational Algebraic Systems (CAS). However, to obtain a closed-form can be unrealistic [37], as the loop counter can be manipulated nondeterministically in each iteration. An extreme example is the famous Collatz problem in Fig. 1(d) [1]. It is desirable that a WCET analyzer still return something *safe* for a terminating program (e.g. Collatz problem with a known value of n), even when its closed-form cannot be deduced.

1.1 Our Contributions and Related Work

To the best of our knowledge, our work is the first fully automated general path analysis method which attempts path-sensitivity and is able to *discover* and *prove* tight upper bound of a resource variable, even in the presence of complicated patterns such as non-rectangular and amortized loops, and down-sampling code even when a closed-form cannot be obtained by traditional CAS. By *prove* here we mean that all infeasible paths detected and used in our analysis are checked by the underlying theorem prover. In the end, we produce not only a bound but also a proof tree so that a third party verifier can certify that the result is *safe*.

Our method is *brute-force* as loops are unrolled. It is different from traditional abstract interpretation (AI) [6] methods dealing with bounds in a way that it never attempts to discover invariants for loops. Instead, we ensure *constraints which are not modified in divergent ways* can be propagated and preserved through loops. I.e *variant effects* caused by the loop bodies are abstracted and summarized using a *polyhedral domain* [7]. It turns out that this approach is very successful in maintaining flow information stretching across loop-nesting levels and between different loops. The reason is that, though a loop can be complicated, variant effects from different paths in the loop body to variables affecting the control flow of the program, usually *agree* upon *one* abstract value. Thus abstraction is not lossy and crucial flow information can be captured precisely. Experimental results show that, very often, we can come up with not only the exact timing for a benchmark, but also its exact ending context (or its best approximation wrt. the abstract domain used).

A significant work on WCET analysis employing symbolic simulation is done by Lundqvist et al. [24]. There low-level analysis and path analysis are combined in one integrated phase. However, that approach has several problems. First, it can only cope with a very simple abstract domain. This leads to limitations in detection of infeasible paths. Second, for the same reason, the approach has a termination issue with some common programming patterns (see the discussion in [24]). Finally, the analysis time is always at least proportional to the actual execution time of the input program. "It leads to a very long analysis since simulation is typically orders of magnitudes slower than native execution" [39].

Indeed, exhaustive symbolic execution is very expensive because of both the breadth and depth of the resulting tree. To address the breadth issue, one requires a notion of merge. The analysis precision is then heavily affected by the power of the employed abstract domain. E.g. the works by Gustafsson et al. [12, 16, 17] employ a form of abstract execution, essentially a combination of symbolic execution and abstract interpretation, using an interval domain. By using the most accurate setting in its AI framework, the method performs full path enumeration and does not scale. To make it practical, similar to [24], path-merging is introduced at different levels. We now briefly mention our merits in avoiding full path enumeration while attempting path sensitivity.

Our method first addresses the breadth issue using *compounded summarization*. For a loop-free program, we guarantee to produce the *exact* bound while avoiding full path enumeration. For programs with loops, we introduce path-merging at the end of each loop body. However, we employ a more powerful abstract domain, i.e. the polyhedral domain. This obviously results in tighter loop bounds and better detection of infeasible paths. Consequently, our path analysis will be more precise than path analysis performed by [24, 12, 16, 17]. Another enhancement due to the use of the polyhedral domain is that we do not have any termination issue with common programming practices.

Now consider the depth issue, and this is most affected by loop unrolling. Clearly, analysis must be at least proportional to a concrete execution trace of the program [39]. E.g. the number of states visited by simulating a single-path[1] *quadratic* program will be at least of *quadratic* complexity. [24] essentially symbolically executes a fixed number of paths. Thus its performance is mainly determined by the length of the longest path. Even so, that technique does not scale. Similarly, [12, 16, 17] do not address the depth issue. In contrast, we address the depth challenge, yet again, using *vertically* compounded summarizations. This results in a behavior which we call *depth-wise loop compression*. This is *innovative*. It gives rise for the simulation to be reduced to *linear* complexity for some highly nested loops, even though those loops' complexities are of much higher order. E.g. we can derive the exact bound of bubblesort, a *quadratic* pro-

[1]Every conditional branch is deterministic.

gram, in a *linear* number of steps. For further discussion later, we make the following definitions.

DEFINITION 1. *For each program such that its asymptotic time complexity can be expressed in terms of a single variable, indicating the size of the program instance, that single variable is called the* size parameter *of the program*[2].

DEFINITION 2. *We say that our path analysis on program P is* reduced to linear complexity *if, in terms of a parameter size n, (a) the number of states symbolically executed in the analysis is O(n), whereas (b) the time complexity of P is worse than O(n).*

Our method naturally supports compositional reasoning, which makes it scale well. Large programs can now be easily split up into a number of smaller programs and the analyzing process can be done in a pipelined manner. In the case that continuation/ending context of a program fragment is captured precisely, we do not compromise the accuracy of the analyses for subsequent fragments.

Unlike recent methods [15, 5], we do not infer parametric bounds for programs. In fact, the outputs we produce are constant bounds and our method only successfully returns a bound for program on which the symbolic execution terminates. However, by sticking to constant bounds, we have the opportunities to discover *tighter* (often *exact*) bounds.

Finally, we mention the work [22] from which some conceptual ideas in this paper were originated. There the authors address the *resource-constrained shortest path* (*RCSP*) problem, which is simpler (though NP-hard) than WCET. In RCSP, the cost of traveling from one node to another in a weighted graph, subject to path feasibility determined by some bounds on the resources consumed while traveling, is minimized. The paper introduces the use of interpolation and witnesses for the RCSP problem, but is limited to *loop-free* programs. Furthermore, in RCSP setting, witness path testing can simply be done by recording the amount of resources consumed by the witness, and checking that the adding of the amount to the current consumption does not result in bound violation. In this paper, the corresponding problem is far harder.

2. OVERVIEW

We formulate the WCET path analysis of a program over a symbolic execution tree where each path of the tree is a succession of nodes, each associated with a program point in the program. *In practice, each program point here is replaced by a distinct basic block in the extended CFGs.* Each node i contains a conjunction $\Psi_i = \psi_0 \wedge \psi_{i_1} \wedge ... \wedge \psi_{i_k}$, referred to either as the incoming context for that node or its prefix path formula, symbolically representing a set of states. The edges are labeled with statements executed in the program. ψ_0 is the context of the root/entry node (represents the knowledge about the input of the program) and $\psi_{i_1}, ..., \psi_{i_k}$ are constraints generated from statements executed by the path from the root to the node i. Due to conditional branches and loops, multiple prefix paths in the tree may come to a same program point but with different sets of states, i.e. different contexts. Our method performs depth-first traversal, terminating each path Ψ_i whenever we are at an endpoint, or when Ψ_i is unsatisfiable (i.e. infeasible path detected). In either case, the algorithm records certain information about Ψ_i and backtracks to the next path. Multiple contexts allow us to tighten our WCET estimation and prune out unnecessary traversal due to the exclusion of infeasible paths. Unfortunately, a simple enumeration of all contexts is *exponential*. In the presence of loops and nested loops, it is even worse (e.g. a simple unnested loop of 100 iterations with just *one* conditional branch in its body results in 2^{100} contexts at the end). Our algorithm possesses three key features to mitigate this problem:

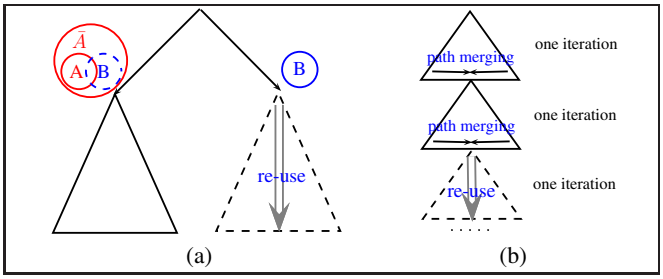

Figure 2: Interpolation and iteration abstraction

Compounded Summarization with Interpolation: The summarization of a subtree reduces the likelihood of fully considering other sub-trees with less general incoming contexts. A summarization contains a *timing solution*, a binary relation called *abstract transformer* to help produce new continuation contexts when reused, and a *condition* under which it is reused. For each subtree at node i, reuse condition is generated by weakening or generalizing the prefix path formula Ψ_i by using a well-known concept called *interpolant* [8]. Essentially, we generalize Ψ_i as long as we preserve the unsatisfiability of all the infeasible paths appeared in the analyzed subtree. The algorithm backtracks and compounds the summarizations computed by the child states and propagates to ancestors for memoing and reuse.

In Fig. 2(a) we assume that A and B are contexts associated to two sibling subtrees, i.e. the nodes associate to a same program point. For brevity, we will refer to these subtrees as subtree A and subtree B. W.l.o.g assume that we have finished analyzing subtree A. In general the two subtrees possess lots of similarities and we want to opportunistically avoid full exploration of B. In Fig. 2(a), context B is not subsumed by context A. However, using the concept of interpolation, context B is subsumed/covered by interpolant $\bar{A}$, a generalization of context A. It means that solutions computed in subtree A can be safely reused in B. We gain performance since, in general, reusing is less costly than fully exploring subtree B.

Iteration Abstraction: Every iteration of a loop is analyzed as a separate subtree, where end points of the loop body are treated as terminating points of paths. Similar to [24, 16], we reduce the breadth of the symbolic tree by merging paths (we use polyhedral domain). This produces only *one* continuation context for analysis of subsequent program fragment. Furthermore, upon finishing the symbolic subtree of an iteration, we compute its summarization. In case reuse happens, analyses for subsequent iterations with similar behaviors can be quickly deduced (Fig. 2(b)).

Witness Path: The use of summarization with interpolation to avoid full path enumeration is *sound*, since to-be-avoided subtrees do not contradict the worst-case path already computed for the original (to-be-reused) subtree. However, the original subtree may contain far more paths than the (to-be-avoided) subtree with a less general context. That is, the worst-case path estimated so far may be infeasible in the less general context. Therefore, though sound, the algorithm may not guarantee the accuracy level we desire.

To remedy the above, we introduce the concept of *witness path*. Assume that we analyze the subtree rooted at node i with the incoming context $\Psi_i = \psi_0 \wedge \psi_{i_1} \wedge ... \wedge \psi_{i_k}$ and

[2]In sorting algorithms, it is the size of the input array.

find out the worst-case path with the path formula $\Psi = \Psi_i \wedge \psi_{i_{k+1}} \wedge ... \wedge \psi_{i_l}$. Our algorithm keeps track for the suffix worst-case path originated from node i, the formula $\omega_i = \psi_{i_{k+1}} \wedge ... \wedge \psi_{i_l}$. We will call ω_i a witness path wrt. the computed timing solution of the subtree rooted at node i under the incoming context Ψ_i. A new node j such that i and j associate to the same program point will not be further expanded if: (a) its incoming context Ψ_j is less general than a previously computed interpolant $\bar{\Psi}_i$, i.e. $\Psi_j \models \bar{\Psi}_i$, and (b) the new context demonstrates that the witness path holds, i.e. $\Psi_j \wedge \omega_i$ is satisfiable. Otherwise, we say that node j cannot be *covered* and a new expansion for that node is required. In a loop-free program, witness path ensures that we achieve *exact* WCET. However, in presence of loops, due to path merging, we do not necessarily get the exact WCET.

In summary, our method is based on (1) dynamic compounded summarization with interpolation for reuse, on (2) iteration abstraction to limit the number of contexts generated by loops, and on (3) witness path to avoid pessimism. This can be viewed as an opportunistic method for the application of *dynamic programming*. Though the concepts of interpolation and summarization have already been well studied, we believe that having them to work with the semantics of exhaustive loop unrolling and path merging while attempting path-sensitivity is a significant contribution.

3. PRELIMINARIES

We consider sets of n system variables $x_1, \ldots, x_n$, denoted $\tilde{x}$, and a variable k ranging over program points.

DEFINITION 3. *A* transition *is a tuple* $\langle k, \rho(\tilde{x}, \tilde{x}'), l\rangle$. ρ *is a constraint over two sets of system variables* $\tilde{x}$ *and* $\tilde{x}'$ *and the timing variable; and* ρ *is induced by the statement between program points* k *and* l. *A* transition system *is a finite set of transitions.*

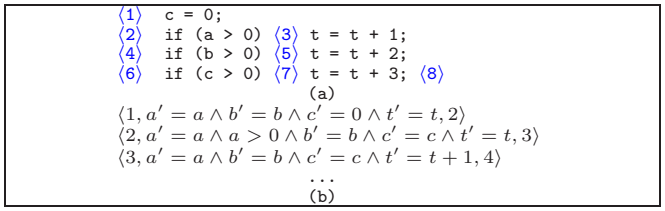

Figure 3: A program fragment and its transition system

The variables in a transition may be renamed freely because their scope is local to the transition. We thus say that a transition is a variant of another if one is identical to the other under renaming substitution. We represent an input program as a set of transition systems, one for each function of the program. One advantage of representing a program as a set of transition systems is that it can be executed symbolically in a simple manner. Secondly, as this representation is general enough, retargeting (e.g. to different types of resource bound analyses) is just the matter of compilation to the designated transition systems. However, translation from binaries into transition systems, similar to the CFG reconstruction problem [34], is a *non-trivial* task. Fortunately, the task becomes trivial by making use of the work by Theiling [34]. Consequently, for clarity and simplicity, in the rest of the paper, our path analysis will be presented on C program and its transition systems.

EXAMPLE 1: Consider the program fragment in Fig. 3(a). The program points are enclosed in angle brackets. Some of the transitions are shown in Fig. 3(b). For instance, the transition $\langle 1, a' = a \wedge b' = b \wedge c' = 0 \wedge t' = t, 2\rangle$ represents that the system state switches from program point $\langle 1\rangle$ to $\langle 2\rangle$ and the constraint denotes the reset of c to 0. The primed versions express the system variables after the corresponding statement execution. Here, the variable of interest t models the execution time. Note that this variable is always initialized to 0 and the only operation allowed upon it is a constant increment. In practice, our method works on basic blocks and the amount of increment at each point will be given by some *low-level analysis* module (e.g. [35]). The variable t is not used in any other way. For the purpose of simplicity, in some later examples we just assume every transition uniformly increments t by 1.

DEFINITION 4. *A* symbolic state *or simply* state $\mathcal{G}$ *is of the form:* $\langle k, \tilde{x}, \phi(\tilde{x})\rangle$ *where* k *is a program point,* $\tilde{x}$ *is a set of system variables, and* ϕ *is a constraint over some or all of the variables* $\tilde{x}$ *and the timing variable.*

DEFINITION 5. *Let there be a transition system, and let* $\mathcal{G} = \langle m, \tilde{x}, \phi(\tilde{x})\rangle$ *be a (symbolic) state. Given a transition* $\langle m, \rho(\tilde{x}, \tilde{x}'), n\rangle$ *in the transition system, a* transition step *gives us a new state* $\langle n, \tilde{x}', \phi(\tilde{x}) \wedge \rho(\tilde{x}, \tilde{x}')\rangle$. *We say that this new state is* infeasible *if the constraint* $\phi(\tilde{x}) \wedge \rho(\tilde{x}, \tilde{x}')$ *is unsatisfiable.*

A transition path *is a sequence of symbolic states s.t. two adjacent states are related by a transition step. An* execution tree *is defined from paths in the obvious way.*

The construction of correct summarizations (briefly mentioned earlier) requires the concept of *interpolants* [8].

DEFINITION 6. [Interpolant]. *If* F *and* G *are formulae such that* $F \models G$, *then there exists an* interpolant H, *denoted as* $\text{int}(F, G)$, *which is a formula such that* $F \models H$ *and* $H \models G$, *and each variable of* H *is a variable of both* F *and* G.

DEFINITION 7. [Summarization of a Block]. *Assume that we analyze a block* B *from entry point* $\langle n\rangle$ *to exit point* $\langle m\rangle$ *wrt. an incoming context* Ψ. *Let* Θ *be the weakest condition such that if we examine* B *with* Θ *as the incoming context, all infeasible paths discovered by previous analysis are preserved. The summarization of* B *wrt.* Ψ *is defined as a tuple* $[\langle n\rangle, \langle m\rangle, WCET, \Delta, \phi, \omega]$, *where* $WCET$ *is the worst case timing of that block and the corresponding path is witnessed by* ω, abstract transformer Δ *is a binary input-output relation between state variables at* $\langle n\rangle$ *and* $\langle m\rangle$, *and* interpolant ϕ *is computed as* $\text{int}(\Psi, \Theta)$.

By definition, the abstract transformer Δ will be the abstraction of all feasible paths from $\langle n\rangle$ to $\langle m\rangle$ (wrt. the incoming context). In general, abstract transformer is *not* a functional relation. We note here that this concept of *abstract transformer* is different from the concept of *abstract transition* developed in [27]. Here, our abstract transformer is a safe approximation for the input-output relationship of a finite tree, whereas in [27], an abstract transition approximates a path (possibly infinite due to the construction of the closure from the transition relation).

DEFINITION 8. [Summarization of a Program Point]. *A summarization of a program point* $\langle n\rangle$ *is the summarization of the block from* $\langle n\rangle$ *to* $\langle m\rangle$ *(wrt. the same context), where* $\langle m\rangle$ *is the nearest subsequent program point s.t.* $\langle m\rangle$ *is of the same nesting level as* $\langle n\rangle$ *and either is (1) an ending point of the program, (2) an ending point of a function, or (3) an ending point of some loop body. The information about* $\langle m\rangle$ *is then often omitted.*

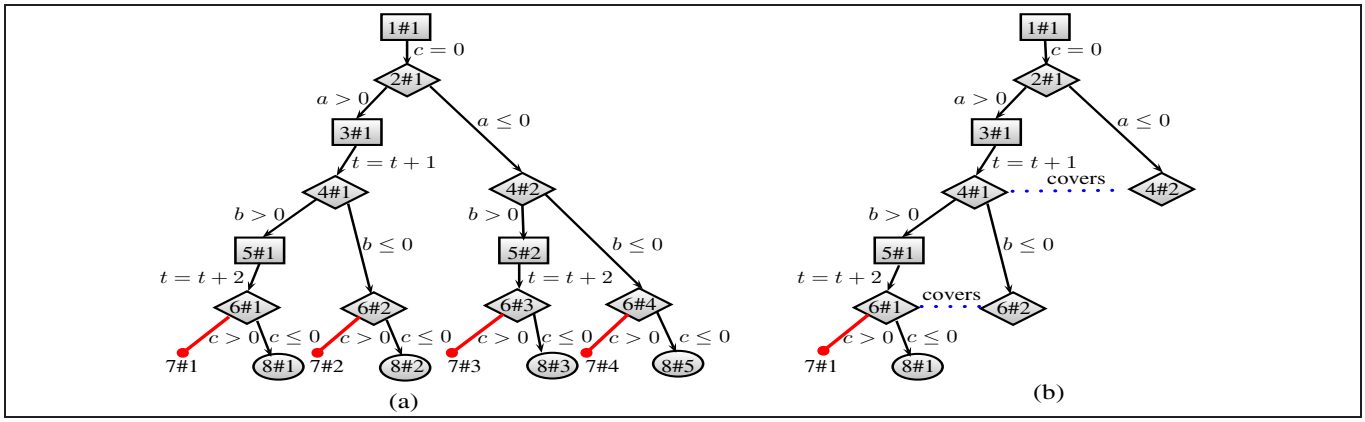

Figure 4: Infeasible paths in analyses: without interpolation (a) with interpolation (b)

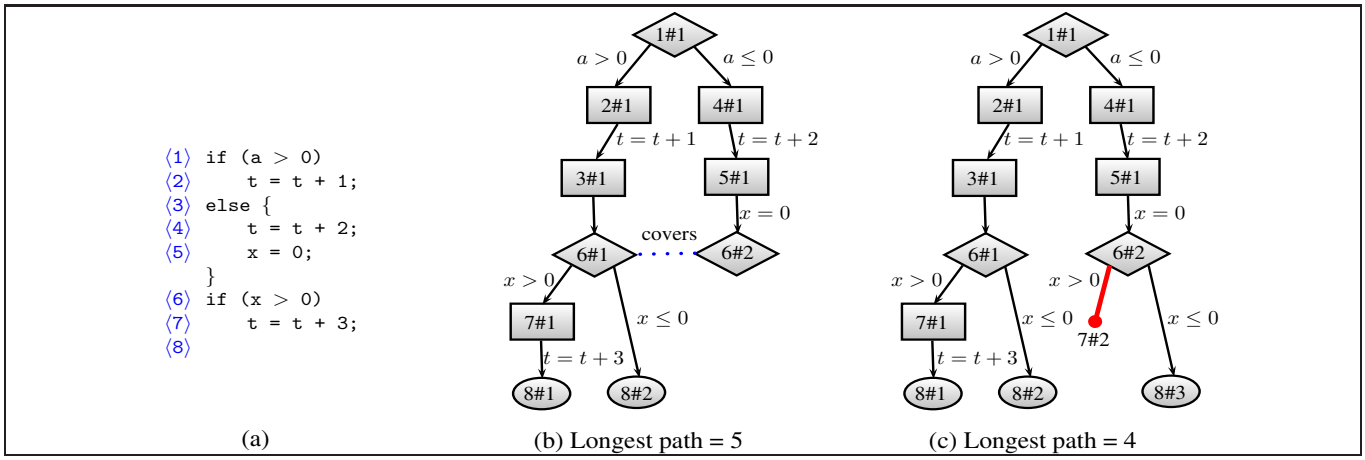

Figure 5: A straight-line program (a) and its analyses: without witness (b) with witness (c)

4. MOTIVATING EXAMPLES

EXAMPLE 2: Consider the transition system in Fig. 3(b) and its (full) symbolic execution tree in Fig. 4(a). Node is labeled `P#C` where `P` is the program point and `C` the context identifier. We label edge by the statement corresponding to its endpoint. We represent conditional statement with *diamond*, basic block of statements with *box*, and terminal node with *ellipse*. *Feasible* transition is denoted by *arrowed* edge, and *infeasible* transition by edge with a *dotted* head.

Without path-sensitivity, the longest path of that transition system would be 6 since the two branches of each **if-then-else** may be executed. However, Fig. 4(a) shows that the statement at program point ⟨7⟩ is not executable since $c \not> 0$. Thus, we infer a tighter bound of 3.

So far, we have illustrated a well-understood benefit of detecting infeasible paths to tighten the estimate. Fig. 4(b) depicts a tree computed by our method. The key idea is to generalize the context of each node (if possible) in order to increase the likelihood for reuse, thus we avoid full path enumeration. In this example, our algorithm enlarges the context of the nodes `4#1` and `6#1` to the formula $c \leq 0$ since this formula is enough to keep the infeasible path detected at `7#1`. Then, whenever their siblings `4#2` and `6#2` are visited with the contexts $c = 0 \wedge a \leq 0$ and $c = 0 \wedge a > 0 \wedge b \leq 0$, respectively, our algorithm tests that `4#2` and `6#2` are covered by their siblings since those new contexts are less general (i.e. $c = 0 \wedge a \leq 0 \models c \leq 0$ and $c = 0 \wedge a > 0 \wedge b \leq 0 \models c \leq 0$). In Fig. 4(b) coverage/reuse is denoted by dashed edge labeled with "covers".

EXAMPLE 3: Though covering a node (using interpolant) may reduce the search space while preserving correctness, it does not necessarily preserve *accuracy* of the analysis. Consider the program in Fig. 5(a) and its possible analysis in Fig. 5(b). The interpolant associated with the subtree rooted at `6#1` is *true* since there are no infeasible paths. Hence `6#1` covers the context of `6#2`. Using the same reasoning as in previous example, a possible WCET estimate is 5 by considering the path: ⟨1⟩ ⟨4⟩ ⟨5⟩ ⟨6⟩ ⟨7⟩ ⟨8⟩ (note that this path is infeasible though). The estimate is calculated by adding 2 from the transition `4#1` to `5#1` and 3 from transition `7#1` to `8#1`.

For better precision, we should expand `6#2`, shown in Fig. 5(c). The key observation is that the new subtree rooted at `6#2` contains an infeasible path if $x \leq 0$. This infeasible path eliminates the potential path from `6#2` to `7#2` which would have provided a longer (5) but *spurious* answer. Thus we are left with a tighter estimate (4) from the path `1#1`, `2#1`, `3#1`, `6#1`, `7#1`, and `8#1`.

The program in Fig. 5(a) illustrates the need to strengthen the condition of coverage for better accuracy. This is done by storing at each subtree, a *witness path formula* ω which concretely represents the WCET path for that very subtree. This witness is then used (in conjunction with the interpolant) to determine coverage/reuse.

In Fig. 5(c), the context at node `6#2` is $\phi_{6\#2} \equiv a \leq 0 \wedge x = 0$. The interpolant at `6#1` is $\phi'_{6\#1} \equiv \mathit{true}$. It is straightforward to see that $\phi_{6\#2} \models \phi'_{6\#1}$. In addition, we test if the witness still holds, i.e. we are testing whether $(\omega_{6\#1} \equiv x > 0) \wedge \phi_{6\#2}$ is satisfiable. Since it is unsatisfiable, the algorithm must explore the node `6#2`, thus obtaining a more precise (actually the exact) bound.

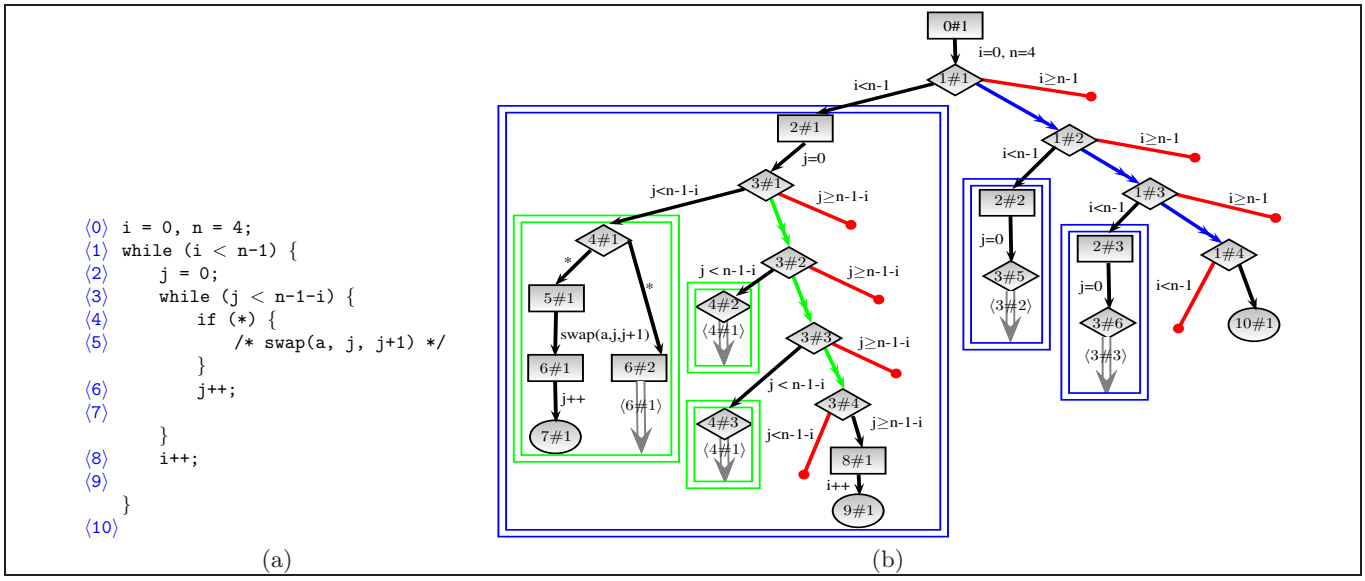

Figure 6: bubblesort program (a) and its analysis (b)

EXAMPLE 4: Consider bubblesort in Fig. 6(a) and its analysis in Fig. 6(b). We represent a separate computation for an iteration as a rectangle with double boundaries. Each when summarized and memoed will be replaced by a single abstract transition denoted as a double-headed arrow. Reuse of summarization is denoted as a double-bodied arrow with the program point and previously encountered context attached. For simplicity, in this example, every (non-abstract) transition increments the timing variable t by 1 (even for `swap` function). Furthermore, when talking about contexts, we use the projected knowledge [20] instead of a long chain of accumulated constraints (due to the presence of loops). Witness paths are also omitted as they do not improve accuracy in this example.

We arrive at `2#1` analyzing the first iteration of the outer loop. From choice point `3#1` we go into the first iteration of the inner loop. The path `4#1 5#1 6#1 7#1` is analyzed normally. The summarizations of program point ⟨6⟩ and program point ⟨5⟩ are computed and stored during backtracking. It is worth to note that the summarization for `6#1` is [⟨6#1⟩, 1, $j' = j+1$, *true*] (it is implicitly understood that $i' = i \wedge n' = n$). In the next visit of ⟨6⟩, which is `6#2`, we obviously can make use of that summarization.

The summarization for `4#1` then is computed as [⟨4#1⟩, 3, $j' = j+1$, *true*]. The WCET is the maximum increment for the timing variable t by considering both paths originated from `4#1`. The interpolant simply is *true* as there are no infeasible paths. The abstract transformer is combined from the two paths, which is $(\texttt{swap}(a, j, j+1) \wedge j' = j+1) \vee (j' = j+1)$. After simplification (here we ignore the effects on array a), it yields just $j' = j+1$. The whole iteration is then replaced by a single transition (double headed arrow) from `3#1` to `3#2`, making use of the abstract transformer $\Delta \equiv j < n-1-i \wedge j' = j+1$ (note that the loop entry condition is added). We continue the analysis of `3#2` with *one* abstract context, $\langle\langle 3\#2\rangle, \tilde{x}, i = 0 \wedge j = 1 \wedge n = 4\rangle$. Similarly, we go into the body of the inner loop at `4#2` and `4#3`, making use of the previous summarization for `4#1` to continue the analysis. At `3#4`, the attempt going into the inner loop body fails as an infeasible path is detected.

When we backtrack, by treating double-headed arrows as normal transitions, we come up a serialization (4 instances) of compounded summarizations for program point ⟨3⟩:

[⟨3#4⟩,3,$i' = i+1, n-1-i \leq j$]
[⟨3#3⟩,7,$i' = i+1 \wedge j' = j+1, n-2-i \leq j < n-1-i$]
[⟨3#2⟩,11,$i' = i+1 \wedge j' = j+2, n-3-i \leq j < n-2-i$]
[⟨3#1⟩,15,$i' = i+1 \wedge j' = j+3, n-4-i \leq j < n-3-i$]

The abstract transformers for those summarizations are computed in a similar manner as how the abstract transformer for `4#1` is computed. On the other hand, the interpolant for `3#4` is the weakest condition which ensures the attempt re-entering the loop body at `3#4` fails, i.e. the corresponding infeasible path is preserved. The interpolant for `3#3` preserves not only such infeasible path but also the infeasible path on the attempt exiting the loop at `3#3`. Similarly, the process goes on for `3#2` and `3#1`. Utilizing compounded summarization saves us from analyzing the inner loop again in the future exploration of subsequent outer loop's iterations. Specifically, at `3#5`, we reuse the summarization of `3#2`; while, at `3#6`, we reuse the summarization of `3#3`. As a result, even though the complexity and the WCET of bubblesort program is *quadratic* to n, the number of the inner loop's iterations explored by our method is just *linear* to n. This behavior remains (for similar programs) even when we introduce more nesting levels. This fact sets us apart from other typical simulation approaches (e.g. [16, 24])

5. SYMBOLIC SIMULATION ALGORITHM

In this section, our presented algorithm (shown in Fig. 7 and 8) only deals with loops. Recursive functions can be treated in a similar manner. Our symbolic simulation algorithm manipulates global memo table *Table*, which is initialized to empty. During analysis, new summarizations of the form $[m, WCET, \Delta(\tilde{x}, \tilde{x}'), \phi(\tilde{x}), \omega(\tilde{x})]$ (as in Def. 8) will be inserted into *Table* (line 15).

In constructing compounded summarization, we rely on two important functions, JoinVertical and JoinHorizontal. Each of them takes in, as inputs, two summarizations S_1 and S_2, respectively summarizing two subtrees T_1 and T_2. T_1 and/or T_2 could well be just a single transition. In fact, even when they are not, we still treat them each as a single *abstract* transition, the abstract transformer plays the role of the transition relation. We first explain these two *crucial* functions. Then, we will discuss our algorithm as a whole. Some implementation details are deferred till section 6.

Compounding Vertically two Summarizations: We achieve this by JoinVertical in Fig. 8. JoinVertical summarizes a compounded subtree T, where T_2 suffixes T_1. In other words, a path π_1 in T_1 followed by a path π_2 in T_2

```
function SS(G, P)
      Let G be ⟨m, x̃, φ(x̃)⟩
⟨1⟩   if (φ(x̃) ≡ false) return [m, −∞, false, false, false]
⟨2⟩   if (outgoing(m, P) = ∅) return [m, 0, Id(x̃, x̃′), true, true]
⟨3⟩   if (endloop(m, P)) return [m, 0, Id(x̃, x̃′), true, true]
⟨4⟩   S := memoed(G, Table)
⟨5⟩   if (S ≠ false) return S endif
⟨6⟩   if (loop(m, P))
⟨7⟩       S1 := [m, WCET, Δ(x̃, x̃′), φ̄(x̃), ω(x̃)]
                := TransStep(G, P, entry(m, P))
⟨8⟩       if (ω(x̃) ≡ false)
⟨9⟩           S̄ := JoinHorizontal(S1, TransStep(G, P, exit(m, P)))
          else
⟨10⟩          G′ := ⟨m, x̃′, φ(x̃) ∧ Δ(x̃, x̃′)⟩
⟨11⟩          Sn−1 := SS(G′, P)
⟨12⟩          Sn :=JoinVertical(S1, Sn−1)
⟨13⟩          S̄ := JoinHorizontal(Sn, TransStep(G, P, exit(m, P)))
          endif
      else
⟨14⟩      S̄ := TransStep(G, P, outgoing(m, P))
      endif
⟨15⟩  Table := Table ∪ {S̄}
⟨16⟩  return S̄
end function
```

Figure 7: Symbolic simulation algorithm

```
function JoinVertical(S1, S2)
      Let S1 be [m, WCET1, Δ1(x̃, x̃′), φ1(x̃), ω1(x̃)]
      Let S2 be [n, WCET2, Δ2(x̃′, x̃′′), φ2(x̃′), ω2(x̃′)]
⟨17⟩  WCET := WCET1 + WCET2
⟨18⟩  Δ(x̃, x̃′) := Δ1(x̃, x̃′) ∧ Δ2(x̃′, x̃′′)
⟨19⟩  φ(x̃) := wp(φ1(x̃), Δ1(x̃, x̃′), φ2(x̃′))
⟨20⟩  ω(x̃) := ω1(x̃) ∧ Δ1(x̃, x̃′) ∧ ω2(x̃′)
⟨21⟩  return [m, WCET, Δ(x̃, x̃′), φ(x̃), ω(x̃)]
end function

function JoinHorizontal(S1, S2)
      Let S1 be [m, WCET1, Δ1(x̃, x̃′), φ1(x̃), ω1(x̃)]
      Let S2 be [m, WCET2, Δ2(x̃, x̃′), φ2(x̃), ω2(x̃)]
⟨22⟩  if (WCET1 ≥ WCET2)
⟨23⟩      WCET := WCET1
⟨24⟩      ω(x̃) := ω1(x̃)
      else
⟨25⟩      WCET := WCET2
⟨26⟩      ω(x̃) := ω2(x̃)
      endif
⟨27⟩  Δ(x̃, x̃′) := Δ1(x̃, x̃′) ∨ Δ2(x̃, x̃′)
⟨28⟩  φ(x̃) := φ1(x̃) ∧ φ2(x̃)
⟨29⟩  return [m, WCET, Δ(x̃, x̃′), φ(x̃), ω(x̃)]
end function

function TransStep(G, P, TransSet)
      Let G be ⟨m, x̃, φ(x̃)⟩
⟨30⟩  S̄ := [m, 0, false, true, true]
⟨31⟩  foreach (⟨m, ρ(x̃, x̃′), n⟩ ∈ TransSet∧
                  ρ(x̃, x̃′) → t′ = t + α) do
⟨32⟩      S1 := SS(⟨n, x̃′, φ(x̃) ∧ ρ(x̃, x̃′)⟩, P)
⟨33⟩      S2 := JoinVertical([m, α, ρ(x̃, x̃′), φ(x̃), true], S1)
⟨34⟩      S̄ := JoinHorizontal(S̄, S2)
      endfor
⟨35⟩  return S̄
end function
```

Figure 8: JoinVertical - JoinHorizontal - TransStep

corresponds a path π (possibly infeasible) in T. The WCET of T is computed intuitively (line 17) whereas T's abstract transformer is computed as the conjunction of the abstract transformers of T_1 and T_2 (line 18). Similar for the case of T's witness path (line 20). The only difference is that, T_1's witness and T_2's witness are related by the abstract transformer Δ_1 of T_1. By treating T_1 as an abstract transition, computing the interpolant for T relies on the function $\overline{wp}(\phi_1(\tilde{x}), \Delta_1(\tilde{x}, \tilde{x}'), \phi_2(\tilde{x}'))$ to produce an interpolant $Itp(\tilde{x})$ such that $\phi_1(\tilde{x}) \models Itp(\tilde{x})$ and $Itp(\tilde{x}) \wedge \Delta_1(\tilde{x}, \tilde{x}') \models \phi'(\tilde{x}')$. This interpolant under-approximates the *weakest precondition* of the postcondition $\phi_2(\tilde{x}')$ wrt. the transition relation $\Delta_1(\tilde{x}, \tilde{x}')$. That is, the formula $\Delta_1(\tilde{x}, \tilde{x}') \models \phi_2(\tilde{x}')$ [4].

Compounding Horizontally two Summarizations: We achieve this by JoinHorizontal in Fig. 8. JoinHorizontal summarizes a compounded subtree T, where T_1 and T_2 are siblings. This is the `join` operation that we often see in other techniques [24, 12, 16]. The compounded WCET and witness are computed intuitively (lines 22-26). Preserving all infeasible paths in T requires preserving infeasible paths in both T_1 and T_2 (line 28). The input-output relationship of T is safely abstracted as the disjunction of the input-output relationships of T_1 and T_2 respectively (line 27).

Inputs and Output: The inputs of our algorithm include a symbolic state $\mathcal{G}$ denoting a possible initial context of the original program $\mathcal{P}$ and the transition system of $\mathcal{P}$.

The algorithm performs a depth-first traversal of the execution tree of the program rooted at $\mathcal{G}$; summarizations are collected in a post-order manner. It finally returns a summarization $[m, WCET, \Delta(\tilde{x}, \tilde{x}'), \overline{\phi}(\tilde{x}), \omega(\tilde{x})]$, representing the whole analyzed program.

When to Reuse: Function memoed checks whether a summarization has already been in the *Table* and can be reused. Specifically, given a symbolic state $\mathcal{G} \equiv \langle m, \tilde{x}, \phi(\tilde{x})\rangle$, we use memoed($\mathcal{G}$, *Table*) to test if there is a tuple $S \equiv [m, WCET, \Delta(\tilde{x}, \tilde{x}'), \overline{\phi}(\tilde{x}), \omega(\tilde{x})]$ in *Table* such that $\phi(\tilde{x}) \models \overline{\phi}(\tilde{x})$ and $\omega(\tilde{x}) \wedge \phi(\tilde{x})$ is satisfiable. If yes, we say that $\mathcal{G}$ is *reused* and return S. Otherwise, *false* is returned.

Base Cases: Our algorithm is most naturally implemented recursively. The function SS handles four base cases. First, when the context $\phi(\tilde{x})$ carried by $\mathcal{G}$ is unsatisfiable (line 1), no execution needs to be considered. Note that here the path-sensitivity plays a role since only (provably) executable paths will be considered. Second, the algorithm checks if $\mathcal{G}$ is a final state (line 2). Here $Id(\tilde{x}, \tilde{x}') \equiv \forall i \in \{1, ..., |\tilde{x}|\} \,.\, \tilde{x}'[i] = \tilde{x}[i]$, i.e. it represents the transition relation for `skip` statement. Ending point of a loop is treated similarly in the third base case (line 3). The last base case, lines 4-5, is the case that a summarization can be reused.

Expanding to next Program Points: Line 14 depicts the case when transitions can be taken from the current program point m, and m is not a loop starting point. Here we call TransStep to move recursively to next program points. The returned value is then passed on. TransStep implements the traversal of transition steps emanating from m by calling SS recursively and then compounds the returned summarizations into a summarization of m. The arguments of TransStep are state $\mathcal{G}$, the transition system $\mathcal{P}$, and a set of outgoing transitions *TransSet* to be explored.

For each transition in *TransSet*, TransStep extends the current state with the transition relation $\rho(\tilde{x}, \tilde{x}')$. Resulting child state is then given as an argument in a recursive call to SS (line 32). From the summarizations returned by all the calls to SS, the algorithm computes the compounded summarization, using JoinVertical and JoinHorizontal. Here, the first argument to JoinVertical (line 33) indeed represents a non-abstract transition.

Loop Handling with Compounded Summarization: Lines 7-13 handle the case when the current program point is a loop starting point. We assume all loops to be in the form of structured while loops. In such case, transitions emanating from a loop starting point can be classified into two: *entry* transitions and *exit* transitions.

Upon encountering a loop, our algorithm attempts to unroll it once by calling procedure TransStep to explore the entry transitions (line 7). When the returned witness is *false*, it understands that we cannot go into the loop body anymore, thus proceeds to exit branches. The returned summarization is compounded (using JoinHorizontal) with the summarization of previous unrolling attempt (line 9). On the contrary, if some feasible paths found by going into the loop body, we use the returned abstract transformer to produce a new con-

text. From this context, we recursively call SS to do the rest of the unrolling process. The returned information is then compounded (using JoinVertical) with the first unrolling attempt and later compounded (using JoinHorizontal) with the analysis of the exit branches (line 10-13). Our algorithm can be *reduced to linear complexity* because these compounded summarizations of the inner loop(s) can be reused in later iteration of the outer loop.

THEOREM 1. [Safe WCET Estimate]. *Our symbolic simulation algorithm always produces* safe *WCET estimates.*

6. IMPLEMENTATION DETAILS

6.1 Propagating Witnesses

We refer to line 20 in Fig. 8. As shown, witnesses ($\omega(\tilde{x})$) are constructed from the constraints along the path that gives rise to WCET. Such path can be very long and naively record it would be a source of inefficiency. Recall that we use witnesses to test for feasibility of a solution within memoed function. That is, given a state $\langle m, \tilde{x}, \phi(\tilde{x})\rangle$ and a witness $\omega(\tilde{x})$, we test if $\phi(\tilde{x}) \wedge \omega(\tilde{x})$ is satisfiable. In general, the witness $\omega(\tilde{x})$ contains other variables, which are disjoint from the variables of ϕ. $\phi(\tilde{x}) \wedge \omega(\tilde{x})$ is satisfiable `iff` $\phi(\tilde{x}) \wedge (\exists var(\omega) - \tilde{x} \;.\; \omega(\tilde{x}))$. Therefore, rather than maintaining $\omega(\tilde{x})$, we maintain a formula that is equivalent to $\exists var(\omega) - \tilde{x} \;.\; \omega(\tilde{x})$. CLP($\mathcal{R}$) projection [20] is useful here.

6.2 Computing the Abstract Transformer

Let us again refer to Fig. 8. The operation in line 18 is similar to the manipulation of witness paths and we deal with it similarly (by projection). However, operation in line 27 requires more attention. In fact, we make use of the *polyhedral library* [7, 36] to handle this disjunction, computed as the *convex hull* of its components. As a result, we only capture linear input-output relationships of system variables. Input-output relationships are in general non-linear. Fortunately, transformations of system variables which affect the flow of the program are very often just linear and are captured precisely by the polyhedral domain.

6.3 Computing the Interpolants

Let us consider the following program fragment:

```
⟨0⟩  a=1,b=1,y=-1;
⟨1⟩  if (x<0) ⟨2⟩ y=a; else ⟨3⟩ y=b;
⟨4⟩  if (y>0) ⟨5⟩ x=1; ⟨6⟩
```

There are 2 infeasible paths of the program:
$\langle 0\rangle a = 1 \wedge b = 1 \wedge y = -1 \langle 1\rangle x < 0 \langle 2\rangle y' = a \langle 4\rangle y' \leq 0 \langle 6\rangle$
$\langle 0\rangle a = 1 \wedge b = 1 \wedge y = -1 \langle 1\rangle x \geq 0 \langle 3\rangle y' = b \langle 4\rangle y' \leq 0 \langle 6\rangle$
By infeasibility, the state at $\langle 6\rangle$ for the two paths here is *false*. If we use the notion of weakest precondition [9] to generalize preceding states for the first path we get the weakest precondition $\neg(\exists y' \;.\; x < 0 \wedge y' = a \wedge y' \leq 0) \equiv x < 0 \rightarrow a > 0$ at $\langle 1\rangle$ for the first path, and $\neg(\exists y' \;.\; x >= 0 \wedge y' = b \wedge y' \leq 0) \equiv x \geq 0 \rightarrow b > 0$ for the second path. Our issue is how to approximate the weakest precondition for a path efficiently. Both paths share a prefix $\langle 0\rangle\ \langle 1\rangle$. The desired weakest precondition for $\langle 1\rangle$, which would maintain the infeasibility of both paths, is the conjunction of the weakest preconditions of both paths: $(x < 0 \rightarrow a > 0) \wedge (x \geq 0 \rightarrow b > 0)$ which is a complex formula involving conjunction and disjunction. Combining the approximations of various paths efficiently is another issue. There are two techniques in our system.

Using Constraint Deletion: Given the paths as before, we remove all constraints that are not necessary to ensure infeasibility. To ensure the infeasibility of the first path, we may remove $b = 1$, $y = -1$, and $x < 0$. For the second path, we may remove $a = 1$, $y = -1$ and $x \geq 0$. Here, both paths share the prefix $\langle 0\rangle\ \langle 1\rangle$which contains $a = 1$, $b = 1$, and $y = -1$. Both paths agree on the removal of $y = -1$, hence we remove it, obtaining the state $a = 1 \wedge b = 1$ at $\langle 1\rangle$ which generalizes the original state $a = 1 \wedge b = 1 \wedge y = -1$, yet not as complex as the weakest precondition mentioned above.

Using Polyhedral Library: Given a transition relation $R(\tilde{x}, \tilde{x}')$ on variables $\tilde{x}$ and $\tilde{x}'$, where $\tilde{x}$ represents the program variables before the transition and $\tilde{x}'$ represents the program variables after the transition, and a postcondition $Post(\tilde{x}')$. A weakest precondition is the formula:

$$\begin{aligned} wp(R(\tilde{x}, \tilde{x}'), Post(\tilde{x}')) &\equiv \forall \tilde{x}' \;.\; R(\tilde{x}, \tilde{x}') \rightarrow Post(\tilde{x}') \\ &\equiv \neg(\neg(\forall \tilde{x}' \;.\; R(\tilde{x}, \tilde{x}') \rightarrow Post(\tilde{x}'))) \\ &\equiv \neg(\exists \tilde{x}' \;.\; R(\tilde{x}, \tilde{x}') \wedge \neg Post(\tilde{x}')) \end{aligned}$$

which now can be estimated using projection (pre-image computation). Here we are only allowed to narrow, but not to widen. We make use of the polyhedral library to ease this computation. The reason is that the polyhedron library allows us to represent a Disjunctive Normal Form (DNF) formula as a union of respective polyhedra. And all the needed operations are closed under this representation (our native CLP($\mathcal{R}$) system do not allow us to represent and manipulate disjuctive formula directly). The projection to eliminate those variables $\tilde{x}'$ may be an overestimation. However, this is safe as the negation of it will be an underestimation of the weakest precondition.

Return to the same example, the weakest precondition for the first path is $(x \geq 0 \vee a > 0)$. However, in getting a conjunctive formula as the interpolant, we decide just to keep $a > 0$. Similarly, what we will keep for the second path is just $b > 0$. As a result, the final interpolant at $\langle 1\rangle$ will be $(a > 0 \wedge b > 0)$.

6.4 Determining Exactness of the Results

In WCET path analysis, it is important to be able to automatically determine whether the returned bound is *exact*. In our approach, we only lose some path-sensitivity due to path merging at the end of each loop iteration. Obviously, our method produces the *exact* bound for a single-path program. For a loop-free program, our method also computes the *exact* bound. For multi-path programs with loops, our method has an advantage compared to others that we can easily incorporate the techniques in [33] into our algorithm. In short, we initially perform data-flow analysis to determine those control flow merges (called "Destructive Merges" [33]) which may cause loss in the analysis precision. Then our algorithm can automatically conclude that the returned bound is *exact* if the input program contains no destructive merges.

7. EXPERIMENTAL EVALUATION

We have selected most difficult benchmark programs from the Mälardalen WCET group [25], namely bubblesort, expint, fft1, fir, insertsort, janne_complex, ns, nsichneu (part of it), ud. In addition, tcas, a real life implementation of a safety critical embedded system, is used to illustrate the performance of our method for the case of big loop-free programs. We also introduce 3 academic programs, namely amortized, two_shapes, non_deter to stress more on complicated behaviors of loops (as in section 1). Benchmark descriptions and sizes are briefly summarized in Table 1.

We used an Intel Core 2 Duo @ 2.93Ghz with 2Gb RAM and built our system upon the CLP($\mathcal{R}$) [21] and its native constraint solver, and custom code for reasoning about arrays, thus providing an accurate test for feasibility. Since

Benchmark	Size Parameter (SP)	Actual WCET	Complexity (wrt. SP)	Symbolic Simulation (SS)					State of The Art (IA)		
				States	Time (ms)	WCET	Exact? Manual	Exact? Auto	States	Time (ms)	WCET
bubblesort	n = 25	1648	$O(n^2)$	135	233	1648	Y	N	2873	3087	1648
	n = 50	6423		260	701	6423	Y	N	11373	20363	6423
	n = 100	25348		510	2438	25348	Y	N	45248	178268	25348
expint	NA	859	-	519	8247	859	Y	Y	1009	13842	859
fft1	n = 8	181	$O(nlogn)$	111	446	181	Y	Y	218	539	181
	n = 16	379		176	927	379	Y	Y	461	1313	379
	n = 32	791		287	2197	791	Y	Y	970	3764	791
	n = 64	1661		495	6818	1661	Y	Y	2049	14829	1661
fir	NA	760	-	108	387	760	Y	Y	986	5036	760
insertsort	n = 25	1120	$O(n^2)$	159	387	1120	Y	N	2861	4847	1120
	n = 50	4120		309	1504	4120	Y	N	10736	45873	4120
	n = 100	15745		609	7542	15745	Y	N	-	timeout	-
janne_complex	NA	133	-	165	491	534	N	N	*	*	*
ns	n = 5	2655	$O(n^4)$	63	59	2655	Y	Y	5936	4359	2655
	n = 10	35555		103	116	35555	Y	Y	86666	104392	35555
	n = 20	522105		183	344	522105	Y	Y	-	timeout	-
nsichneu	NA	281	-	334	15542	281	Y	N	-	timeout	-
ud	NA	819	-	487	1137	819	Y	Y	992	1802	819
amortized	n = 50	394	$O(n)$	95	287	394	Y	Y	760	649	394
	n = 100	792		186	1035	792	Y	Y	1551	2312	792
	n = 200	1590		339	4057	1590	Y	Y	3142	10539	1590
two_shapes	n = 50	2199	$O(n^2)$	259	797	2199	Y	Y	2874	5068	2199
	n = 100	8149		509	3235	8149	Y	Y	10749	42092	8149
	n = 200	31299		1009	19839	31299	Y	Y	-	timeout	-
non_deter	n = 25	3904	$O(n^2)$	129	509	3904	Y	Y	8255	17941	3904
	n = 50	15304		242	1876	15304	Y	Y	-	timeout	-
	n = 100	60604		467	9253	60604	Y	Y	-	timeout	-
tcas	NA	99	-	6020	15925	99	Y	Y	-	timeout	-

Table 2: Experimental results

Benchmark	Description	#LC
bubblesort	Bubble sort program	128
expint	Series expansion for computing an exponential integral function	157
fft1	Fast Fourier Transform using the Cooly-Turkey algorithm	219
fir	Finite impulse response filter (signal processing algorithms)	276
insertsort	Insertion sort program	92
janne_complex	Nested loop program with complex flow	64
ns	Search in a multi-dimensional array	535
nsichneu	Automatically generated code containing large amounts of if-statements	2000
ud	LU decomposition algorithm	147
amortized	A program with amortized loop	41
two_shapes	A nested loop where the inner loop is executed only on even-th iteration of outer	20
non_deter	A nested loop having inner loop's counter incremented nondeterministically in each iteration (simpler version of Collatz)	20
tcas	A traffic collision avoidance system, a real life safety critical embedded system	400

Table 1: Benchmark programs

the benchmark programs are of small and moderate sizes, timeout is set at 300 seconds.

As mentioned earlier, the methods in [24, 12, 16] are similar to our method with only the iteration abstraction feature (modulo the abstract domain). To have a better comparison, both the performances of our full symbolic simulation method (SS) and its "Iteration Abstraction only" version (IA) are reported in Table 2. IA closely mimics the performance of the methods described in [24, 12, 16], especially in term of its complexity wrt. the size parameter. In Table 2 we refer to IA as the current state-of-the-art.

In fact, if IA ever returns, its bound will be at least as good as the bound returned by SS. Due to the employment of more accurate abstract domain, IA detects more infeasible paths compared to [24, 12, 16] and therefore its bounds will be tighter than those computed by [24, 12, 16]. For each benchmark, IA also visits less states compared to [24, 12, 16]. There are certain programs that cannot be handled by [24, 12, 16] due to their limited abstract domains, but will be well handled by our IA (see discussion in [24]). Of course, it is expensive to maintain a more accurate abstract domain. In particular, we expect that, IA version might take longer running time compared to [24, 12, 16] due to calls to the polyhedral library and the underlying theorem prover for checking feasibility.

For a loop-free program, SS guarantees to produce the *exact bound*[3]. For this kind of program, very importantly, we demonstrate that by using interpolation, we do not necessitate full enumeration of paths. The performances of SS vs. IA for tcas illustrate the point.

As shown in Table 2, except for janne_complex, SS achieves the exact timing for each of the benchmarks - indicated by column Manual. Some of those, current non-brute-force technique [28] cannot achieve the exact bounds even for certain loops alone. Except for bubblesort, insersort, nsichneu, janne_complex, not only SS produces the exact bounds but also it can *automatically conclude* that it has computed the exact upper bounds - indicated by column Auto - based on the technique elaborated in section 6.4.

The program janne_complex, firstly introduced in [12], was designed in such a way that, as long as path merging is applied at the end of the outer loop body, we overestimate its WCET. As expected, SS does not produce an exact timing for this benchmark, however, the result is still comparable with the method introduced in [12, 16] (the number of inner loop iterations is estimated at 66). IA, expected to produce a similar result, however fails because of overflow during a call to the polyhedral library.

On the other hand, nsichneu is a (multi-path) program with a single loop having a very large body with lots of conditional branches. Its purpose is to test the scalability of an analyzer. Our algorithm does not finish on full nsichneu program (about 4000 LOC) due to the heavy workload on the solver for checking infeasible paths. However, on the attempt to reduce the size of nsichneu, i.e. by reducing the body of the loop by half, our algorithm then not only runs in good time, it also computes the *exact* bound.

SS finishes in less than 20 seconds for every benchmark

[3]In theory, this is limited by the power of the theorem prover, since the problem of detecting all infeasible paths is *incomplete*. However, in practice with CLP($\mathcal{R}$) , we have encountered no problems regarding this matter.

program. It significantly outperforms IA in *all* benchmarks. More importantly, for programs of which a size parameter exists and can be easily modified as an input variable, the complexity of our SS is *reduced to linear* (wrt. the size parameter). In most cases, the number of states visited by SS is even smaller than the "Actual WCET", which corresponds to the maximum number of states in a concrete execution of the program. IA, therefore methods in [24, 12, 16], clearly do not possess such properties.

8. OTHER RELATED WORK

WCET path analysis has been the subject of much research, and substantial progress has been made in the area (see [29, 39] for surveys). Implicit Path Enumeration Technique (IPET) [23] and its extensions (e.g. [11, 13, 5]) have been widely used due to its efficiency and simplicity. However, pure IPET methods have problems with infeasible paths and flow information stretching across loop-nesting levels. However, complex flow facts can be expressed using user-defined constraints [11, 13], but the complexity of solving the resulting problem is potentially exponential, since the program is completely unrolled and all flow information is lifted to a global level. Moreover, the correctness of those constraints is not verified and the predicted WCET may be *untight* or, worse, *unsafe*. Consequently, in fully automated techniques [12, 16, 17, 24] where flow information can be captured precisely, loops are resorted to unrolling.

Considering *infeasible paths* to increase the accuracy of WCET path analysis has attracted a lot of attention in recent years. Nonetheless, all previous works either perform partial detection of infeasible paths (e.g. using conflict sets) [18, 32] or suffer from full path enumeration [26, 2, 31].

SATURN [10] and [30] are general techniques in program analysis. Ours is related to them since all are summary-based. However, those works are path-insensitive ("where precise means meet-over-all-paths" [30]) and cannot be applied to problems which require a high-level of accuracy such as WCET prediction. In contrast, our work attempts path-sensitivity while doing loop summarization. The problem we address is fundamentally different, and much harder.

Our approach poses a commonality with recent CEGAR-based model checking approaches [3, 19] in using interpolation concept to eliminate irrelevant facts and optimize the search space. In CEGAR, when coverage happens, a sub-tree can be safely pruned. However, in our case, it only means that the previously analyzed sub-tree (to be exact, its summarization) can be reused under a new context. The difference is due to the fact that here we address a *discovery* and *optimization* problem whereas CEGAR works on decision problem. For instance, as a simpler version of WCET, RCSP (resource-constrained shortest path) by no means can be easily addressed using CEGAR. Indeed, it has been argued before that model checking techniques cannot efficiently deal with WCET analysis [38, 14].

9. CONCLUDING REMARKS

We presented a brute-force path analysis method for inferring and proving tight resource bounds by symbolically simulating loops. The main novelty is first, the use of *interpolation* which allows abstract reasoning which in turn makes the search space manageable; second, the use of *witness paths* to curtail the use of the said abstration in cases where accuracy is likely to be affected; and finally, the use of *compounded summarizations* on loop iterations in such a way that the state space explored by our symbolic simulation can even be smaller than the number of states in a concrete execution. Using well-known WCET benchmarks, we showed that our method performs not just well, but often it obtains *exact* results.

We have briefly mentioned earlier that our method naturally supports compositional analysis. However, as shown in Section 7, even without it our algorithm still performs well with benchmarks under the real-time system domain. Exploring the usefulness of compositionality property for larger programs is left as our future work.

10. REFERENCES

[1] On the 3x+1 problem: http://www.ericr.nl/wondrous.
[2] P. Altenbernd. On the false path problem in hard real-time programs. In *Euromicro Workshop RTS '96*.
[3] T. Ball et al. Automatic predicate abstraction of C programs. In *PLDI '01*.
[4] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. In *TCS '97*.
[5] S. Bygde, A. Ermedahl, and B. Lisper. An efficient algorithm for parametric WCET calculation. In *RTCSA '09*.
[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis. In *POPL '77*.
[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*,.
[8] W. Craig. Three uses of Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Sym. Comp.*, 1955.
[9] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. In *Commun. ACM*, 1975.
[10] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08*.
[11] J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. In *RTSS '00*.
[12] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Euro-Par '97*.
[13] A. Ermedahl, F. Stappert, and J. Engblom. Clustered calculation of worst-case execution times. In *CASES '03*.
[14] Lv et al. Performance comparison of techniques on static path analysis of WCET. In *EUC '08*.
[15] S. Gulwani et al. The reachability-bound problem. In *PLDI '10*.
[16] J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *WORDS '05*.
[17] J. Gustafsson, A. Ermedahl, and B. Lisper. Algorithms for infeasible path calculation. In *WCET '06*.
[18] C. A. Healy et al. Automatic detection and exploitation of branch constraints for timing analysis. In *IEEE TSE '02*.
[19] T. A. Henzinger et al. Lazy abstraction. In *POPL '02*.
[20] J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP($\mathcal{R}$) constraints. In *New Generation Computing*, 1993.
[21] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. In *TOPLAS '92*.
[22] J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI '08*.
[23] Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC '95*.
[24] T. Lundqvist et al. An integrated path and timing analysis method based on cycle-level symbolic execution. In *RTS '99*.
[25] URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.
[26] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. In *RTS '93*.
[27] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL '05*.
[28] A. Prantl et al. Constraint solving for high-level WCET analysis. In *WLPE 08'*.
[29] P. Puschner et al. A review of WCET analysis. In *RTS '00*.
[30] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*.
[31] F. Stappert et al. Efficient longest executable path search for programs with complex flows and pipeline effects. *CASES '01*.
[32] V. Suhendra et al. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC '06*.
[33] A. Thakur and R. Govindarajan. Comprehensive path-sensitive data-flow analysis. In *CGO*, 2008.
[34] H. Theiling. CFGs for real-time systems analysis. PhD Thesis.
[35] H. Theiling et al. Fast and precise WCET prediction by separated cache and path analyses. In *RTS '00*.
[36] H. Le Verge. A note on chernikova's algorithm. TR, 1994.
[37] E. Vivancos et al. Parametric timing analysis. In *LCTES '01*.
[38] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI '04*.
[39] R. Wilhelm et al. The WCET Problem – Overview of Methods and Survey of Tools. *Trans. on Emb. Comp. Syst. '08*.