

Path-Sensitive Timing Analysis

JOXAN JAFFAR, JORGE NAVAS, AND ANDREW E. SANTOSA

National University of Singapore
{joxan,navas,andrews}@comp.nus.edu.sg

Abstract. We address the problem of estimating the Worst-Case Execution Time (WCET) for loop-bounded programs. It is well known that for efficiency reasons, current techniques take little or no account of infeasible paths in the symbolic execution tree, thus limiting their accuracy. To consider all infeasible paths, ie. to be path-sensitive, would generally entail an impractical full path enumeration of the tree. In this paper, we present an algorithm that is *systematically* path-sensitive. That is, the algorithm detects all infeasible paths within the unsurmountable limitations of (a) being able to generate a sufficient loop-invariant, and (b) having the power of the underlying theorem-prover to decide feasibility. Thus, eg. for loop-free programs whose infeasible paths are decidable, our algorithm is perfectly accurate.

There are two main features: *interpolation* and *witness paths*. The algorithm traverses the symbolic execution tree, in a post-order manner, and discovers an interpolant which generalizes the execution context of the tree. Trees satisfying the more general context contain no more paths than the original. While the interpolant potentially enables the *reuse* of the longest path estimate obtained from one subtree in order to cover another subtree, a complimentary problem is that infeasible paths in the subsumed subtree but feasible in the original subtree are not taken into account. We deal with this by storing in each traversed subtree a representative “witness” path which defines the longest path estimate in the subtree. Thus covering of subtrees now depend on both the interpolant and the witness path of the traversed subtree. In summary, we provide what is essentially a *dynamic programming* algorithm which can avoid full enumeration. We finally present empirical data with real programs to demonstrate practicality.

1 Introduction

Determining an upper bound on resource usage by a program is often a critical need, and a most important resource is execution time. Unfortunately, this problem undecidable in general. An almost universal approach to this is to statically bound the number of loop iterations. Even so, the problem of estimating the execution time is still of huge complexity [26]. Since all possible input values must be considered, the number of possible program paths increases exponentially with the number of control flow branches. Thus algorithms which enumerate all executable paths cannot be scalable.

Worst-Case Execution Time (WCET) analysis aims to provide the worst possible execution time of a program used in a system and it is usually performed at two different levels [24]. The *low-level*, which is done on binary code, provides the execution time of basic blocks considering the effects of hardware level features such as cache, pipelining,

branch prediction, etc. [13, 8]. *High-level* analysis is performed on source code, and it focuses on characterizing possible executable paths. A big advantage of performing high level analysis is that the hardware model can be replaced independently of the rest of the tool, making a tool much easier to retarget [27].

A main issue in WCET analysis is to avoid overestimation in timing evaluation by providing *tight* bounds. One part of the overestimation is due to the presence of some hardware features that affect the execution time of instructions. On the other hand, to achieve a tight estimation we need information about the program behavior such as infeasible paths and maximum number of loop iterations [7]. Furthermore, WCET estimates must be also *safe*, i.e., guaranteed not to underestimate the execution time, in order to be valid for use in hard real-time systems. To be able to guarantee the correctness, we also need a proof of the WCET estimate [7, 10, 23].

In this paper, we consider programs where each loop is provided with a constant bound on its number of iterations. We then present a WCET algorithm which has a *systematic* detection of infeasible paths, or in program analysis jargon, it is *path-sensitive*. We say systematic here to mean that infeasibility is systematically checked, but within two fundamental limitations:

- the timing of a loop is computed by *aggregation*, that is, it is given as the WCET of the loop-body multiplied by the (given) bound on the number of iterations of that loop. The loop-body, in turn, is analysed with a context being a certain *loop-invariant* which is generated by the algorithm.
- the power of theorem-prover which determines if a given path is infeasible. Clearly no prover exists for all kinds of paths, for example, it is impossible to always decide if paths involving nonlinear integer predicates are feasible or not. Thus we tacitly assume that infeasibility testing is really only applicable to predicates for which a reasonable theorem-prover exists, and these of course must be *decidable* predicates.

Thus for a loop-free program which whose symbolic paths can be solved by a theorem-prover, our method computes the *exact* WCET. In this paper, our experiments are conducted with a custom loop-invariant discovery method, and our theorem-prover deals with linear arithmetic formulas over integer variables and array elements. The key result is that the algorithm does not necessitate full enumeration of paths, and indeed, has good empirical behavior as we show below on some substantial benchmark programs.

Now, considering infeasible paths to increase the accuracy of the WCET analyses has attracted a lot of attention in recent years. However, all previous works either perform *partial* detection of infeasible paths (e.g., based on computing conflict sets) or enumerate all executable paths. We say more about this below.

We formulate the problem of estimating WCET over a symbolic decision tree where each path of the tree is a succession of nodes associated with each program point of the program. Moreover, each node contains a conjunction of formulas $\Psi \equiv \psi_1 \wedge \dots \wedge \psi_i$, symbolically representing a set of states. The edges contain the statements executed in the program. Therefore, ψ_1, \dots, ψ_i are constraints generated from each statement in the path from the root to the node i . A main feature of this tree is that multiple paths may contain the same set of program points but with a different set of states, i.e. different *contexts*. Then, our method performs depth-first traversal, terminating each

path Ψ whenever we are at the endpoint, or when Ψ is unsatisfiable (i.e., infeasible path). In either case, the algorithm records certain information about Ψ and backtracks to the next path. Multiple contexts allow us to tighten our WCET estimation due to the exclusion of infeasible paths but unfortunately, a simple enumeration of all contexts is exponential. Our algorithm possesses two key features to mitigate this problem:

Interpolation. This first feature is “abstraction learning”. We weaken or generalize the path formula Ψ by using a notion of “interpolant”. Essentially, we generalize as long as we preserve the satisfiability/unsatisfiability of Ψ . This generalization reduces the likelihood of considering other paths with a less general context. To operate at any level in the tree, the algorithm must propagate during the backtracking process the interpolants computed by the children states to ancestors.

The use of interpolants to avoid traversal is *sound* in the sense that avoided subtrees do not contradict the longest path already computed for the original subtree. However, the original subtree may contain far more paths than the subtree with a less general context. That is, the longest path estimated so far may be infeasible in the less general context. Therefore, though sound, the algorithm may not preserve *accuracy*.

Witness Paths. To remedy this, our algorithm keeps track for the longest path, a formula ω representing a conjunction of constraints defined along with that path. We will call ω a *witness path*. Then, a new path will not be considered if: (a) its context Ψ' is less general than another previously computed context Ψ (i.e., $\Psi' \models \Psi$) for that path, and (b) the new context demonstrates that the witness path holds, i.e., $\Psi' \wedge \omega$ is satisfiable. Otherwise, the new path cannot be covered and a new traversal for that path is required.

In summary, our method is based on (1) interpolation for the ability to reuse, and on (2) witness paths for the ability to define what is reused. Our method thus can be viewed as an opportunistic method for the application of *dynamic programming*. Finally, our experimental evaluation with real programs demonstrates the practicality of our approach.

1.1 Related Work

There are three main categories of high level WCET analyses: *tree-based*, *IPET (Implicit Path Enumeration Technique)*, and *path-based*. The tree-based approach [25], estimates the WCET by a bottom-up traversal of the CFG of the program, using rules defined for each type of compound program statement. IPET [19] methods model the program flow using integer linear programming. Recently IPET is popularized because complex flows [9] (e.g., to exclude infeasible paths) can be defined. Finally, path-based methods [26] estimate the WCET by calculating times for different paths in a program and searching for the path with the longest execution time.

Park [23] and Altenbernd [4] use symbolic execution to improve estimates by excluding infeasible paths but endure full path enumeration. Stappert et.al. [27] present an algorithm that finds the longest path and checks its feasibility. Finding one feasible path, it returns the length of that path. Otherwise, it removes the path and searches for a new longest path. Although simple and elegant, this algorithm also endures full

path enumeration. Ermedahl, Gustafsson, and Lisper [10, 12] use abstract interpretation to approximate all possible variable values in order to determine infeasibility. They allow tuning so that in the most precise setting, their “path-sensitiveness” would be comparable to our algorithm. However, in this precise setting, they would essentially perform full path enumeration. Healy and Whalley [14] and Suhendra et. al. [28] maintain/exploit pairwise conflicts between branches and assignments. Although these two approaches often work really well in practice since they avoid the full enumeration of feasible paths, the detection of infeasible paths is limited.

We also mention the use of model checking for WCET analysis by verifying time bounds using symbolic model checker [22]. The WCET is computed from the verified timing bounds using binary search. This work can be considered path-sensitive due to the nature of model checking, which does not include traverse infeasible states, however, the user has to guess the initial timing bound to be verified, since model checkers do not *discover*.

Interpolation has been used in verification to refine an abstract domain whenever a spurious counterexample is found [15], and to improve the completeness of SAT-based bounded model checking [21] among others. Our use to discover generalization of program states is novel.

The most important related work is [18] which originated the core conceptual ideas in this paper. There the problem was the *resource-constrained shortest path (RCSP)* problem, a substantially simpler problem (though NP-hard) than WCET. There, 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 (and only the perfect solution will do). This paper [18] introduced the use of interpolation and witnesses for the RCSP problem, but was limited to loop-free programs over a finite domain of discourse. Further, in this 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 bounds violation. In this paper, the corresponding problem is far harder. In summary, the present paper advances conceptually by considering loops and general formulas over integer variables and array elements. But the key advance is to demonstrate that our implementation of interpolation and witnesses addresses a representative set of real benchmark programs.

2 Preliminaries

We shall model computations by first considering sets of n system variables x_1, \dots, x_n , denoted \tilde{x} , in their domains, and a variable k ranging over program points.

Definition 1 (Transition and Transition system). A transition is a tuple $\langle k, \rho(\tilde{x}, \tilde{y}), l \rangle$ ρ is a constraint over two sets of system variables \tilde{x} and \tilde{y} , and possibly some additional variables. ρ is induced by the statement between program points k and l . A transition system is a finite set of transitions.

Clearly 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 then represent a C program as a set of transitions, one for each function of the program.

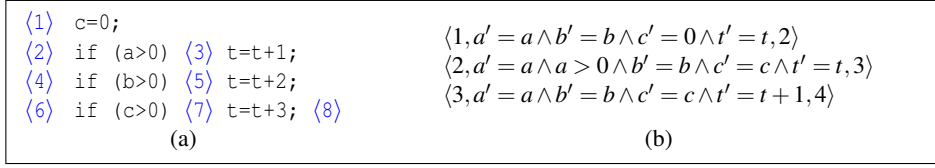


Fig. 1: A Program Fragment and Its Transition System

Example 1 (Translation into Transition System). Consider the C program fragment in Fig. 1(a). The program points are enclosed in angle brackets. Some of the transitions are shown in Fig. 1(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 1 to 2 and the constraint denotes the reset of c to 0. The primed versions express the system variables after the corresponding statement's execution. Here, the variable of interest is t that models the execution time. Note that this variable is always initialized to 0 and the only operation allowed upon it is a constant increment, and the variable is not used in any other way.

One of the advantages of representing a C program as a set of transitions is that it can be also executed symbolically in a simple manner.

Definition 2 (Symbolic State). 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 possibly some additional variables.

Definition 3 (Transition Step, Path and Tree). Let there be a transition system, and let $\mathcal{G} \equiv \langle m, \tilde{x}, \phi(\tilde{x}) \rangle$ 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.

A path may end in a *final* state from which no transition step can be taken. We call the system variables of the final state the *final* variables.

3 Motivating Examples

Example 2 (Infeasible Paths and Interpolation/Reuse). Consider the transition system in Fig. 1(b) and its (full) symbolic execution tree in Fig. 2(a). Nodes are labeled $\mathbb{P}\#\mathbb{C}$ where \mathbb{P} is the program point and \mathbb{C} the context. Edges are labeled by the instruction corresponding to its endpoints. We represent conditional statements with *diamond* nodes, basic block of statements with *box* nodes, and terminal nodes with *ellipses*. *Feasible* transitions are denoted by arrowed edges, and *infeasible* transitions by edges with a dotted head. Without path-sensitivity, the longest path of that transition system would be 6¹ since the two branches of each **if-then-else** may be executed. However, Fig. 2(a) shows that the statement at program point $\langle 7 \rangle$ is not executable since c is never greater than 0. Therefore, we can infer a tighter bound of 3. So far, we have illustrated a well-understood benefit of detecting infeasible paths for accurately inferring WCET.

¹ Recall that longest path is computed by counting increments of the special variable t .

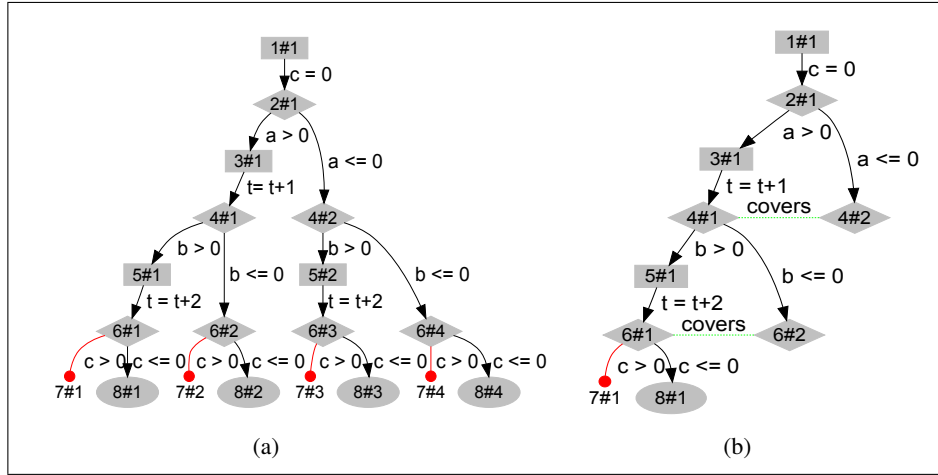


Fig. 2: Infeasible Paths and Interpolation/Reuse.

Fig 2(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 reusing previously computed solutions (by its siblings), thus avoiding 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 $c = 0 \wedge a > 0 \wedge b \leq 0 \models c \leq 0$). We denote covered transitions by dashed edges without arrow head and labeled with "covers".

Example 3 (Witness Paths). While covering a node may save search space while preserving correctness, it does not necessarily preserve *accuracy* of the analysis.

Consider Fig. 3(a) and Fig. 3(b). A possible WCET estimate is 5 by considering the path: 1#1, 4#1, 5#1, 6#1, and 7#1. The estimate is calculated by adding 2 from the transition 4#1 to 5#1 and 3 from transition 7#1 to 8#1. Note here that 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.

Next suppose that 6#2 is not covered by 6#1. The tree is shown in Fig.3(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 the real WCET (4) from the path 1#1, 2#1, 3#1, 6#1, 7#1, and 8#1.

The program in Fig. 3(a) illustrates the need to strengthen the condition for coverage. This is done by storing at each subtree, a *witness path formula* ω which concretely represents the WCET path for the subtree. This witness is then used (in conjunction with the context given by interpolation) to determine coverage.

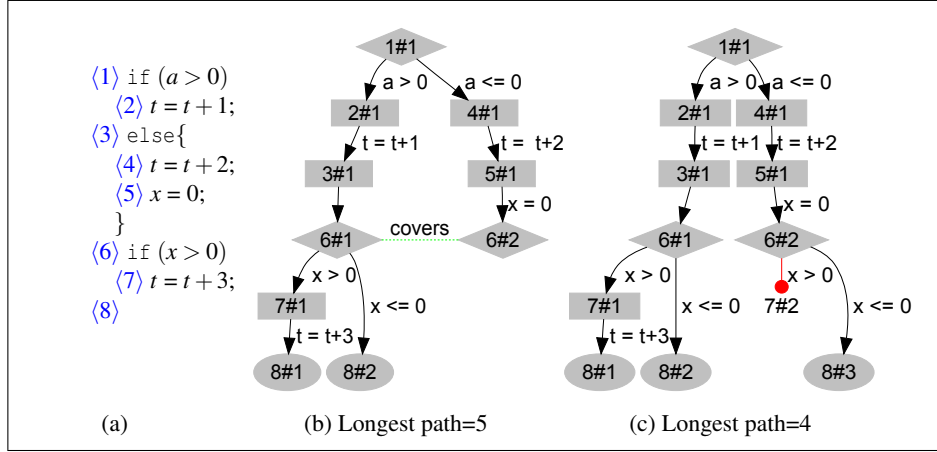


Fig. 3: Witness Paths

For example, a node n with context ϕ_n is covered by k with interpolant ϕ'_k and longest path estimate V if $\phi_n \models \phi'_k$ (as before), and its witness path formula ω holds with the new context ϕ_n . That is, $\omega \wedge \phi_n$ is satisfiable. Coming back to the tree in Fig. 3(c), the context at node $6\#2$ is $\phi_{6\#2} \equiv a \leq 0 \wedge x = 0$. The interpolant at $6\#1$ is $\phi'_{6\#1} \equiv \text{true}$. It is straightforward to see that $\phi_{6\#2} \models \phi'_{6\#1}$. In addition, we test if the witness still holds. That is, $(\omega \equiv x > 0) \wedge \phi_{6\#2}$ is satisfiable. Since it is unsatisfiable, the algorithm must explore the node $6\#2$ obtaining the most precise estimate.

4 Path-Sensitive Timing Algorithm

Our path-sensitive algorithm manipulates a *memo table* which is a set of tuples of the form $\langle \mathcal{G}, WCET, \omega(\vec{x}) \rangle$, where \mathcal{G} is a symbolic state as in Def. 2 over the variables \vec{x} , $WCET$ a non negative number, and $\omega(\vec{x})$ a witness formula. Two operations are provided to manipulate the global memo table *Table*:

- Given a symbolic state $\mathcal{G} \equiv \langle m, \vec{x}, \phi(\vec{x}) \rangle$ and let \mathcal{G}' be another state $\langle m, \vec{x}, \phi'(\vec{x}) \rangle$, then the function `memoed(\mathcal{G} , Table)` tests if there is a tuple $S \equiv \langle \mathcal{G}', WCET, \omega(\vec{x}) \rangle$ in *Table* such that $\phi(\vec{x}) \models \phi'(\vec{x})$ and $\omega(\vec{x}) \wedge \phi(\vec{x})$ is satisfiable. If yes, we say that \mathcal{G} is *covered* by \mathcal{G}' , and it returns S . Otherwise, *false*.
- The function `memoize(S , Table)` inserts a new tuple S in *Table*.

We also assume several essential procedures related to abstraction and interpolation:

- `invariant(\mathcal{G}, \mathcal{P})` automatically generates a loop invariant in the form of a symbolic state with a more general constraint.
- `$\overline{wp}(\phi(\vec{x}), \rho(\vec{x}, \vec{x}'), \phi'(\vec{x}'))$` produces an interpolant $Itp(\vec{x})$ such that $\phi(\vec{x}) \models Itp(\vec{x})$ and $Itp(\vec{x}) \wedge \rho(\vec{x}, \vec{x}') \models \phi'(\vec{x}')$. This interpolant under-approximates the *weakest precondition* of the postcondition $\phi'(\vec{x}')$ wrt the transition relation $\rho(\vec{x}, \vec{x}')$. That is, the formula $\rho(\vec{x}, \vec{x}') \models \phi'(\vec{x}')$ [5].

```

function PST( $\bar{G}$ , LoopSet,  $\mathcal{P}$ )
  Let  $\mathcal{G}$  be  $\langle m, \bar{x}, \phi(\bar{x}) \rangle$ 
  (1) if ( $\phi(\bar{x}) \equiv \text{false}$ ) then return  $\langle \langle m, \bar{x}, \text{false} \rangle, -\infty, \text{false} \rangle$  endif
  (2) if ( $\text{outgoing}(m, \mathcal{P}) = \emptyset$ ) then return  $\langle \langle m, \bar{x}, \text{true} \rangle, 0, \text{true} \rangle$  endif
  (3)  $S := \text{memoed}(\bar{G}, \text{Table})$ 
  (4) if ( $S \neq \text{false}$ ) then return  $S$  endif
  (5) if ( $\text{loop}(m, \mathcal{P})$ ) then
  (6)   if ( $\langle m, \bar{x}, \phi'(\bar{x}) \rangle \in \text{LoopSet}$ ) then
  (7)     return  $\langle \langle m, \bar{x}, \phi'(\bar{x}) \rangle, 0, \text{true} \rangle$ 
  (8)   else
  (9)      $\bar{G} := \text{invariant}(\bar{G}, \mathcal{P})$ 
  (10)     $\langle \bar{G}_1, \text{WCET}_1, \omega_1(\bar{x}) \rangle := \text{TransStep}(\bar{G}, \{\bar{G}\} \cup \text{LoopSet}, \mathcal{P}, \text{entry}(m, \mathcal{P}))$ 
  (11)     $\langle \bar{G}_2, \text{WCET}_2, \omega_2(\bar{x}) \rangle := \text{TransStep}(\bar{G}, \text{LoopSet}, \mathcal{P}, \text{exit}(m, \mathcal{P}))$ 
  (12)    return  $\langle \bar{G}_1, \text{WCET}_1 \times \text{iter}(m, \mathcal{P}) + \text{WCET}_2, \text{true} \rangle$ 
  (13)  endif
  (14) endif
  (15) return  $\text{TransStep}(\bar{G}, \text{LoopSet}, \mathcal{P}, \text{outgoing}(m, \mathcal{P}))$ 
end function

```

Fig. 4: PST

In addition, we assume the following helper procedures used in the algorithm:

- $\text{outgoing}(m, \mathcal{P})$ returns the set of all transitions $\langle k, \rho, l \rangle$ such that $k = m$.
- $\text{loop}(m, \mathcal{P})$ returns *true* if m is a starting program point of a loop. Otherwise, it returns *false*. (We assume that all loops are structured while loops.)
- $\text{iter}(m, \mathcal{P})$ returns the number of iterations of the loop as specified by the user or any other automatic tool that can infer loop bounds [11, 8, 6].
- $\text{entry}(m, \mathcal{P})$ returns the set of all transitions from the loop start point m that leads to a program point in the loop body.
- $\text{exit}(m, \mathcal{P})$ returns the set of all transitions from the loop start point m that leads to a program point not in the loop body. Note that if $\text{loop}(m, \mathcal{P})$ is *true* then $\text{outgoing}(m, \mathcal{P}) = \text{entry}(m, \mathcal{P}) \cup \text{exit}(m, \mathcal{P})$ and $\text{entry}(m, \mathcal{P}) \cap \text{exit}(m, \mathcal{P}) = \emptyset$.

Our algorithm is shown in Fig. 4 and 5. The algorithm consists mainly of two functions: PST, the main procedure (Fig. 4), and TransStep (Fig. 5) which is called from PST and it makes a transition step to the next program point.

The input of the algorithm is a symbolic state \bar{G} denoting the possible initial states of the original C program, the transition system of the original program \mathcal{P} (obtained as in Fig. 1), and a variable *LoopSet* initialized to empty. We also assume that the global *Table* is also initialized to empty. The algorithm then performs a depth-first traversal of the execution tree of the program rooted at \bar{G} and collects the WCET estimate and witness information in a post-order manner. It returns a tuple $\langle \bar{G}, \text{WCET}, \omega \rangle$, where \bar{G} is a generalization of \mathcal{G} , *WCET* the estimate of the length of the longest path, and finally, ω the constraints along the path that gives rise to *WCET*, i.e., the witness path.

This algorithm is most naturally implemented recursively. The function PST handles three base cases. First, when the context $\phi(\bar{x})$ carried by \bar{G} is unsatisfiable (line

1). From this point on, no execution need to be considered. The algorithm returns a *false* interpolant that generalizes the fact that \mathcal{G} is unsatisfiable. Moreover, it also returns the WCET estimate $-\infty$ and its witness *false*. Note that it is here where the path-sensitiveness plays a role since only executable paths will be considered. Second, the algorithm checks if \mathcal{G} is a final state (line 2). In this case, it returns the interpolant *true* with a WCET estimate 0 (at this point, it takes no time to reach the end of the program), and its witness path is *true* (anything reaches the end point). In the third base case, described in lines 3-4, the state \mathcal{G} has been already visited with a more general context and its witness holds. The test is done by memoed. The algorithm returns the tuple returned by memoed.

```

function TransStep( $\mathcal{G}, LoopSet, \mathcal{P}, TransSet$ )
  Let  $\mathcal{G}$  be  $\langle m, \bar{x}, \phi(\bar{x}) \rangle$ 
  (13)  $\langle \bar{\phi}(\bar{x}), WCET, \omega(\bar{x}) \rangle := \langle true, 0, true \rangle$ 
  (14) foreach  $\langle \langle m, \rho(\bar{x}, \bar{x}'), n \rangle \in TransSet \wedge$ 
       $\rho(\bar{x}, \bar{x}') \rightarrow t' = t + \alpha$  do
  (15)  $\langle \langle n, \bar{x}', \phi'(\bar{x}') \rangle, WCET', \omega'(\bar{x}') \rangle :=$ 
      PST( $\langle n, \bar{x}', \phi(\bar{x}) \wedge \rho(\bar{x}, \bar{x}') \rangle, LoopSet, \mathcal{P}$ )
  (16)  $\bar{\phi}(\bar{x}) := \bar{\phi}(\bar{x}) \wedge \overline{wp}(\phi(\bar{x}), \rho(\bar{x}, \bar{x}'), \phi'(\bar{x}'))$ 
  (17) if  $(WCET' + \alpha > WCET)$  then
  (18)    $WCET := WCET' + \alpha$ 
  (19)    $\omega(\bar{x}) := \rho(\bar{x}, \bar{x}') \wedge \omega'(\bar{x}')$ 
  endif
  endfor
  (20)  $\bar{\mathcal{G}} := \langle \langle m, \bar{x}, \bar{\phi}(\bar{x}) \rangle, WCET, \omega(\bar{x}) \rangle$ 
  (21)  $Table := memoize(\bar{\mathcal{G}}, Table)$ 
  (22) return  $\bar{\mathcal{G}}$ 
end function

```

Fig. 5: TransStep

loop is approximated to be $WCET_1 \times iter(m, \mathcal{P}) + WCET_2$. Extensions to sequences of loop or cascading loops are obvious. To ensure safe approximation, the state at the end of the loop should entail the state at the start of the loop, hence a loop invariant is needed. When reaching $\langle m \rangle$ for the first time, the algorithm generalizes the state using an automatically computed loop invariant.

Now consider loops. Firstly, the algorithm tests (line 6) if this point has been already explored by checking if its entry is in *LoopSet*. If this is the case, then this point is actually a loop end point, and it returns the loop invariant of the corresponding loop as the invariant, 0 as the timing estimate, and *true* as its witness path (line 7). Otherwise, the point m is not in *LoopSet* and the algorithm must compute its loop invariant by generalizing the current goal (line 8). After this, the transitions that follow the point m are considered separately by splitting them into entry and exit transitions.

Lines 5-11 handle the case when the current program point m is a loop start point. Note again that we assume all loops to be in the form of structured while loops. In such case transitions emanating from the loop start point can be classified into two: *entry* transitions and *exit* transitions.

We first briefly overview our loop handling mechanism. Upon encountering a loop, our algorithm computes separately the WCET of the loop body ($WCET_1$) and the WCET of the loop continuation ($WCET_2$), with their corresponding witnesses, as illustrated in Figure 6. The WCET of the loop body is computed between two occurrences of the same program point (loop start point $\langle m \rangle$), while the WCET of the continuation is computed to the end of the program. The WCET of the

The algorithm first calls the procedure `TransStep` to explore recursively the entry transitions (line 9). In this call it adds \mathcal{G} into the *LoopSet* to record the state of the loop start point, which would later trigger the processing described earlier at line 8. This call returns a tuple consisting of the generalization \mathcal{G}_1 of \mathcal{G} , worst-case timing estimate $WCET_1$, and its witness ω_1 . The algorithm again calls the procedure `TransStep` (line 10) to explore the exit transitions, resulting in a tuple that contains the estimate $WCET_2$ for the exit paths. At the end of loop handling section (line 11), the algorithm returns the more general loop invariant \mathcal{G}_1 , together with the WCET estimate for the subtree.

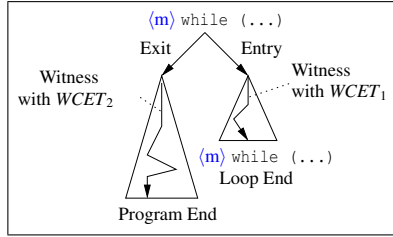


Fig. 6: Loops WCET

The remaining case (line 12) is when transitions can be taken from the current point m , and m is not a looping point. Here we call again the procedure `TransStep` to move recursively to the next transition and pass on its return value.

We next explain the procedure `TransStep` that implements the traversal of transition steps emanating from a program point m by calling `PST` recursively and then combining the tuples returned by each call to `PST` into a tuple of m . The arguments of `TransStep` are a state \mathcal{G} , the loop invariants in *LoopSet* of the loops visited so far, the transition system of the program \mathcal{P} , and the set of outgoing transitions *TransSet* to be explored.

For each transition in *TransSet* the procedure extends the current state with the transition relation $\rho(\tilde{x}, \tilde{x}')$ defined by the transition resulting in a child state, which is then given as an argument in a recursive call to `PST` (line 15). This results in a tuple that contains a generalization of the input state, the WCET estimate of execution rooted at the input state, and the witness formula of that estimate. From the tuples returned by all the calls to `PST`, the algorithm computes the generalization of the state \mathcal{G} , its WCET estimate, and witness. In line 16 the algorithm uses the function $\overline{w\bar{p}}$ to compute the generalization of \mathcal{G} based on the generalization of a child state. The final generalized context of \mathcal{G} is the conjunction of the result of the all calls to $\overline{w\bar{p}}$. The WCET estimate for \mathcal{G} is the maximum value of the WCET of a child added with the execution time α of the state transition from \mathcal{G} to the child. Finally, the witness of \mathcal{G} is the witness of the child whose WCET estimate plus the transition time α is the maximum, extended (in conjunction) with the transition relation $\rho(\tilde{x}, \tilde{x}')$ between \mathcal{G} and the child. These two components are computed in lines 17-19.

4.1 Key Implementation Design Elements

Loop Invariant Generation. The important feature is that we are able to bring to each occurrence of a loop, an accurate context. The constraints in this context hereby become *candidates* for the loop invariant. Choosing amongst them is rather straightforward. We identify the variables U that are updated in a loop body by static analysis. Given a call invariant $(\mathcal{G}, \mathcal{P})$, where \mathcal{G} is $\langle m, \tilde{x}, \phi(\tilde{x}) \rangle$, and $\phi(\tilde{x})$ a conjunction $\bigwedge_{i=1}^n c_i$ of n constraints, we then remove the conjuncts c_i of $\phi(\tilde{x})$ where $var(c_i) \cap U \neq \emptyset$. This results in the

constraint ϕ' . The final loop invariant is a projection of ϕ' onto \tilde{x} , that is $\exists var(\phi) - \tilde{x} . \phi'(\tilde{x})$. Here we employ the projection mechanism of $CLP(\mathcal{R})$ [16].

The main feature, once again, is that by (path-sensitive) propagation, we expose valuable constraints to constitute the invariant. The main advantage is therefore that dynamically created constraints may be propagated *through* loops.

Computing Interpolants. Let us consider the C program fragment $\langle 0 \rangle a=1, b=1, y=-1; \langle 1 \rangle \text{if } (x<0) \langle 2 \rangle y=a; \text{else } \langle 3 \rangle y=b; \langle 4 \rangle \text{if } (y>0) \langle 5 \rangle x=1; \langle 6 \rangle$. There are two infeasible paths of the program (written with constraints):

$$\begin{aligned} \langle 0 \rangle a = 1 \wedge b = 1 \wedge y = -1 \langle 1 \rangle x < 0 \langle 2 \rangle y' = a \langle 4 \rangle y' \leq 0 \langle 6 \rangle \\ \langle 0 \rangle a = 1 \wedge b = 1 \wedge y = -1 \langle 1 \rangle x \geq 0 \langle 3 \rangle y' = b \langle 4 \rangle y' \leq 0 \langle 6 \rangle \end{aligned} \quad (1)$$

By the infeasibility, the state at $\langle 6 \rangle$ for the two paths here is *false*. If we use the notion of weakest precondition 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 $\neg(\exists y' . x \geq 0 \wedge y' = b \wedge y' \leq 0) \equiv x \geq 0 \rightarrow b > 0$ for the second path. Our first issue here is how to approximate the weakest precondition for a path efficiently.

Now, both paths share a prefix $\langle 0 \rangle, \langle 1 \rangle$. The desired weakest precondition for $\langle 1 \rangle$ that 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. Now, the second issue that need to be resolved is how to combine the approximations of various paths efficiently.

To resolve the above two issues, instead of weakest precondition, we use interpolant, which approximates weakest precondition, but easier to compute. Given the paths in (1), 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 agrees on the removal of $y = -1$, hence we can actually 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.

Propagating Witnesses. We refer again to line 19 in Fig. 5. 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 it would be a source of inefficiency to record it. Recall that we use witnesses to test for feasibility of a solution within the memoed function (lines 3-4 of Fig. 4), 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 $\omega(\tilde{x})$ may contain other variables, which we assume are disjoint from the variables of ϕ . Here, $\phi(\tilde{x}) \wedge \omega(\tilde{x})$ is satisfiable if and only if $\phi(\tilde{x}) \wedge (\exists var(\omega) - \tilde{x} . \omega(\tilde{x}))$. Therefore, rather than maintaining $\omega(\tilde{x})$, our algorithm maintains a formula that is equivalent to $\exists var(\omega) - \tilde{x} . \omega(\tilde{x})$. The consequence of this is that in line 19, instead of assigning $\rho(\tilde{x}, \tilde{x}') \wedge \omega'(\tilde{x}')$ to $\omega(\tilde{x})$, we actually assign an equivalent of $\exists x' . \rho(\tilde{x}, \tilde{x}') \wedge \omega'(\tilde{x}')$. Again, $CLP(\mathcal{R})$ projection is useful here.

Program	CLOC	PI			PS w/ interp.+witness			Improvement
		WCET	States	Time (secs)	WCET	States	Time	
adpcm	523	1089	1021	0.4	813	846	2	25%
linpack	956	1372	5546	13	1357	3060	424	1.1%
mpeg	1773	743	3132	2	623	2341	7	16%
statemate	1097	292	790	0.6	260	11097	89	11%
susan_thin	2371	42247	918	0.8	25095	898	15	41%

Table 1. Comparison path-insensitive (PI) vs path-sensitive (PS)

5 Implementation and Results

In order to demonstrate the practicality of our method we have implemented our timing analyzer and performed a number of experiments with real programs. We consider 5 large programs: `adpcm` and `statemate` from the Mälardalen WCET group [20]; `linpack` from [2]; `mpeg` from [1]; finally, `susan_thin` from [3].

We used an Intel Core Duo T2350 @ 1.86Ghz with 1Gb RAM. We used the CLP(\mathcal{R}) [17] system and its native constraint solver, and custom code for reasoning about arrays, thus providing an accurate test for feasibility. Recall that our analysis automatically generates loop invariants (Sec. 4.1). Hence, no manual intervention was used.

Table 1 differentiates a path-insensitive algorithm (PI) and our path-sensitive method (PS). For simplicity, we built the PI algorithm by tuning our PS algorithm turning off the detection of infeasible paths and making obvious optimizations to make PI as fast as possible (e.g., no evaluation of symbolic constraints, no witness paths, etc). The column CLOC tabulates C lines of (uncommented) code, State the number of nodes in the symbolic execution tree, and Improvement shows the reduction in the estimated WCET value when infeasibility is considered.

The results in Table 1 provide a measure of the added accuracy of our algorithm. Since path-sensitivity is well-known to provide accuracy, the main column of interest is in fact Time. That is, the main contribution here is that our algorithm can actually run in reasonable time for significantly large programs, indicating its scalability. For this reason, we do not compare directly the running times against an insensitive or partially sensitive timing analysis (which of course will be faster).

The program `susan_thin` shows the highest accuracy improvement (41%). Here, some input values are fixed and then the evaluation of many if statements depend on those values. Therefore, there are many infeasible paths. In `statemate`, there is also a huge number of infeasible paths but the impact on WCET is not so meaningful as in `susan_thin` but still important. However, the execution tree is the biggest in our collection (11097 nodes). The reason is that there are many different contexts. This program demonstrates the ability of our interpolation-based method to cover less general subtrees in order to reduce the search space. In fact, running *without* interpolation runs beyond one hour. Finally, in the `linpack` program the gains are very small (1%) and the running time is the longest one (> 400 seconds). While it is often hard to predict how interpolation can affect the size of symbolic tree, we have noticed in `linpack` that the depth of the symbolic paths are by far the longest in comparison with the other programs. This feature would explain why our algorithm takes more time since basic operations such as forward propagation of constraints during the symbolic execution

Program	PS w/ witnesses					PS w/o witnesses			
	WCET	States	Time	Covered	Unsat Witness	WCET	States	Time	Covered
adpcm	813	846	2	29	1	813	845	1.2	29
linpack	1357	3060	424	587	24	1357	3036	409	587
mpeg	623	2341	7	325	44	623	2304	5	319
statemate	260	11097	89	6486	396	262	9869	47	5694
susan_thin	25095	898	15	180	1	25095	897	2	180

Table 2. Witness Path Information

and generation of interpolants depend on the path depth. On the other hand, this program contains a huge number of non-linear constraints. This results in a low detection of infeasible paths since our theorem prover cannot prove unsatisfiability for this class of constraints.

In our second experiment, shown in Table 2, we provide a more detailed information about the impact of witness paths. Here, we compare our algorithm described in Figs. 4 and 5, Sec. 4 (PS w/ witnesses) with the same algorithm assuming that witness paths are not considered. The column Covered shows the number of times that a state with a less general context was covered by another more general, and column Unsat Witness denotes the number of times that a witness path did not hold (i.e., it was unsatisfiable wrt to the current context).

As we explained, the role of witness paths is vital to our method in order to be *systematically* path-sensitive. This essentially means that feasibility tests are accurately performed. Without witnesses, our algorithm would detect infeasible paths only partially and it would be hard to predict under what conditions. That is, our accuracy would be ad-hoc. Table 2 exhibits also that the consideration of witness paths, while considerably expensive, is not a significant practical limitation to our algorithm.

The maximum overhead appears in the *statemate* program (2x the time of the version w/o witnesses). The reason is that the search space is bigger since now some subtrees are not covered and hence, our algorithm needs to explore new states. On the other hand, the WCET estimate is more precise since no relevant infeasible paths are hidden under covered nodes. That is, the undesirable scenario shown in Ex. 3 is not possible. For the rest of programs, the same behavior is observed but without differences in the WCET estimates since the number of witnesses that do not hold is small.

6 Concluding Remarks

We developed a path-sensitive WCET algorithm which systematically checks infeasible and therefore is highly accurate. The main contribution is to show that this algorithm is practical, and we do this on real benchmarks.

References

1. High level synthesis benchmark suite. <http://mesl.ucsd.edu/spark/benchmarks.shtml>.
2. Linpack - the netlib. www.netlib.org/benchmark/linpackc.
3. Mibench version 1.0: a free, commercially representative embedded benchmark suite. <http://www.eecs.umich.edu/mibench/>.

4. P. Altenbernd. On the false path problem in hard real-time programs. In *In Proceedings of the 8th Euromicro Workshop on Real-time Systems*, pages 102–107, 1996.
5. N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *TCS*, 173(1):49–87, February 1997.
6. M. Rustagi C. A. Healy, M. Sjödin and D. Whalley. Bounding loop iterations for timing analysis. In *RTAS '98*, page 12. IEEE Computer Society, 1998.
7. A. Colin and G. Bernat. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *14th ECRTS*, page 50. IEEE Computer Society, 2002.
8. A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Syst.*, 18(2-3):249–274, 2000.
9. J. Engblom and A. Ermedahl. Modeling complex flows for worst-case execution time analysis. *Real-Time Systems Symposium, IEEE International*, 0:163, 2000.
10. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *3rd Euro-Par'97*, pages 1298–1307. Springer-Verlag, 1997.
11. S. Gulwani, K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL'09*, 2009.
12. J. Gustafsson, A. Ermedahl, and Björn Lisper. Algorithms for infeasible path calculation. In *WCET*, 2006.
13. C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.
14. C. A. Healy and D.B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. Softw. Eng.*, 2002.
15. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *31st POPL*, pages 232–244. ACM Press, 2004.
16. J. Jaffar, M. Maher, P. Stuckey, and R. Yap. Projecting CLP(\mathcal{R}) constraints. In *New Generation Computing*, volume 11, pages 449–469. Ohmsha and Springer-Verlag, 1993.
17. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(\mathcal{R}) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
18. J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *23rd AAI*, pages 297–303. AAAI Press, 2008.
19. Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *SIGPLAN Not.*, 30(11):88–98, 1995.
20. Mälardalen WCET research group benchmarks. URL <http://www.mrtc.mdh.se/projects/wcet/benchmarks.html>, 2006.
21. K. L. McMillan. Interpolation and SAT-based model checking. In *15th CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
22. A. Metzner. Why model checking can improve WCET analysis. In R. Alur and D. A. Peled, editors, *16th CAV*, volume 3114 of *LNCS*. Springer, 2004.
23. C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Syst.*, 5(1):31–62, 1993.
24. P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Journal of Real-Time Systems*, 18(2/3):115–128, 2000.
25. P. Puschner and C. Koza. Calculating the maximum, execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.
26. F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *J. Syst. Archit.*, 46(4):339–355, 2000.
27. F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES '01*, pages 132–140. ACM, 2001.
28. V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. In *DAC '06*, 2006.