Certified WCET Estimation with User Assertions

Joxan Jaffar, Andrew E. Santosa and Răzvan Voicu

School of Computing
National University of Singapore
Republic of Singapore 117543
{joxan,andrews,razvan}@comp.nus.edu.sg

Abstract

We consider the problem of determining tight conservative estimations of the worst case execution time. This problem is in general difficult even in the presence of simplifying assumptions, hence most methods rely on an interplay between various methods of abstraction and user provided (but not validated) annotations to achieve efficient and precise estimations.

We present an algorithm which enumerates symbolic traces of a program. A key feature is the use of dynamic summarizations *to capture abstract, yet context-dependent, input-output relationships that can be subsequently reused. This allows, in general, a reduction in the number of traces that need be considered. The algorithm is designed for user-provided input in the form of symbolic constraints. The important point here is that this information is eventually* validated, *finally resulting in a guaranteed resource estimate. Finally, the algorithm considers programs that are hierarchically structured. Besides leading to a* compositional *approach, this facilitates the specification of user assertions and guides the processes of summarization, re-use, and assertion validation, and allows the user to explore trade-offs between efficiency and precision.*

1 Introduction

Static estimation of the worst case execution time (WCET) of a program has been identified as an important problem in the design of safety-critical systems. In the general case, computing the exact value of WCET is equivalent to the halting problem and is thus undecidable. A practical solution would be to sacrifice generality in favor of decidability, by operating under the simplifying assumptions that programs have no recursive calls, and that all program loops are bounded. Even under these assumptions, computing exact WCET values is potentially intractable, since in the worst case a program has an exponential number of execution paths. To cope with this difficulty, current methods and tools for timing analysis trade off precision for efficiency. The loss of precision has two major sources: a *low-level one*, in the form of the abstraction of hardware (i.e. simplified modeling of micro-architecture), and a *high-level* one, in the form of abstraction of contexts in the path enumeration process. Both sources lead to gross over-estimations when employed in fully automated methods, and major research efforts have been put into making such estimations tighter.

In order to overcome the high-level loss of precision in WCET estimation methods, a key idea is to allow the user to augment the WCET estimation model with information that could not be derived automatically from the program. Such information is typically in the form of linear constraints between frequency counts of execution path segments, and leads to the elimination of certain spurious paths from the path enumeration process. While this approach may lead to very tight estimations, it has the major drawback that the user-provided information is not validated, and as such, the estimation derived by say, a WCET estimation tool is not certified in the formal sense. The estimation is certified only on the proviso that the user-provided information is correct.

In this paper we present a method for producing tight and certified WCET estimations efficiently, for a high-level, architecture independent execution model. The distinguishing feature of our method is that it employs an abstract path enumeration process, and allows user specified loop invariant templates, in order to guarantee that our estimate is correct. This process is based on a postcondition propagation operator, and is in general efficient due to a novel concept of *dynamic summarization*. The key idea of this concept is to construct reasonably tight abstractions of the input-output relationships of selected program fragments on the fly. Whenever a summarized program fragment is about to be re-traversed in the path enumeration process, the summarization is inspected and possibly used as a shortcut, speed-

ing up the process. In fact, the summarization is only used if the summarized entry context *subsumes* the actual entry context available upon re-entry. However, if that is not the case, the program fragment at hand will be re-traversed, generating an alternative summarization, in a manner that is similar to polyvariant procedural analysis.

The summarization process is guided by *user-specified program blocks*, whose boundaries act as abstraction points that trigger the construction of an input-output abstract relation. An important aspect of our method is that when a block spans over a loop's body, a *user-specified template*, in the form of a linear constraint, can be used to discover an invariant. This *validates* the user-specified template which, alongside with using a postcondition propagation operator, contributes to proving the correctness of the method's WCET estimate. Obviously, the quality of the templates and abstract domain will affect the tightness of the estimate.

Last but not least, our method is *compositional*. For a given program, the system of user-specified blocks is hierarchic (i.e. a block may contain smaller blocks), so that disjoint blocks can be summarized independently. This makes it possible to construct a summarization for a block from the summarizations of its components, making the entire process incremental and modular.

1.1 Related Work

Worst case execution time analysis is a widely explored topic. For a survey and discussions see [13, 12, 19]

WCET usually consists of two phases: a *low-level* and a *high-level* analysis [13, 17]. Low level analysis is performed on the object code, and depends on the hardware (micro-architecture) model. It is complicated for modern architectures since it requires the consideration of caches and pipelines [10], branch prediction, interrupts, out-of-order instruction execution [8], etc.

In contrast, *high-level* WCET analysis is applied to an architecture independent description of the system and has the task to compute *path information* [13]. State of the art approaches use abstract interpretation [4, 15], path enumeration techniques [9, 10], and hierarchical structuring of programs into trees [1, 3, 2].

Implicit path enumeration techniques (IPET) [9] concentrate on avoiding traversing the exponential space of concrete paths by using abstraction. The most common approach is to represent a program by its control flow graph (CFG), attach frequency counts to its arcs, and then express the conservation of flow at every node as a linear constraint. Then, integer linear programming (ILP) can be used to derive a conservative estimate of WCET. Since the CFG contains no contextual information, this model will allow the representation of spurious paths, typically resulting in a gross over-estimation of the WCET. To overcome this drawback, IPET methods allow user annotations, in the form of linear constraints, as a means of eliminating some of the spurious paths in the model. Since the model is still an ILP problem, this approach appears to be practical and convenient. However, as argued in [19], this approach does not scale. Moreover, since the user-provided information is not validated, the WCET estimation is not certified to be correct.

Tree-based methods [1, 3, 2] hierarchically decompose programs according to their syntactic structure. They may represent program blocks as (possibly non-linear) abstract input-output relations, and may allow user-specified annotations as a means to improve their precision. This approach leads to a hierarchical, modular approach, that may result in more precise estimations for a wider range of input data, with less restrictions on the program.

In this paper, we are concerned with a high-level WCET analysis. While we allow the user to structure the program in order to obtain a modular and incremental approach, we are not restricted to structured programs. Moreover, while we believe that user-provided information is crucial for obtaining a tight estimate of the resource usage, we also believe that the information should be validated, in order to guarantee that the derived estimate is conservative.

2 The Basic Idea

In this section we present the gist of our method by means of two simple examples. First, consider the program fragment given in figure 2. Our method is in fact based on instrumenting the program of interest with *time variables* that are initialized to zero at the beginning of the program, and then incremented at every program point by an amount equal to the duration of the corresponding instruction. The instrumented program is then analyzed with the purpose of determining the maximal value of the instrumented variables. The instrumentation process can of course be performed automatically; however, for purposes of clarity, the program fragment in figure 2 has been instrumented by hand. The angle-bracketed numbers represent program points, and $\texttt{instr}_1 \cdots \texttt{instr}_6$ are instructions whose execution does not affect the values of the boolean variables `b1` and `b2`. We assume that instructions $\texttt{instr}_1$, $\texttt{instr}_4$, and $\texttt{instr}_6$ take 1 time unit to execute, whereas instructions $\texttt{instr}_2$, $\texttt{instr}_3$, and $\texttt{instr}_5$ take 2 time units.

At first sight, in may appear that our program yields as many as $2^3 = 8$ possible computation paths. On a closer look however, at least 4 paths are infeasible since the last two `if` conditions are correlated. When analyzing this program, we shall represent symbolic states as constraints over instances of program variables. For example, assume that $\texttt{instr}_1$ were the assignment `x = x + y` and that the program fragment was entered (via program point $\langle 0 \rangle$) in a

```
⟨0⟩ if ( b1 ) { instr1 ; t = t + 1 ; }
    else { instr2 ; t = t + 2 ; }
⟨1⟩ if ( b2 ) { instr3 ; t = t + 2 ; }
    else { instr4 ; t = t + 1 ; }
⟨2⟩ if ( !  b2 ) { instr5 ; t = t + 2 ; }
    else { instr6 ; t = t + 1 ; }  ⟨3⟩
```

Figure 1. Exponential number of paths

context where b1 were true. Then, the symbolic state at program point $\langle 1 \rangle$ is represented by the constraint

$$b1, X_1 = X_0 + Y_0, T_1 = T_0 + 1.$$

Here, X_0, T_0, and X_1, T_1, represent the values of x and t at program points $\langle 0 \rangle$, and $\langle 1 \rangle$, respectively.

Again, for reasons of clarity, we shall assume that constraints resulting from the analysis of $\texttt{instr}_1, \ldots, \texttt{instr}_6$ are abstracted away (the details of how abstraction works will be provided later). Assume now that the path enumeration process starts at program point $\langle 0 \rangle$ with the context *true* (i.e. the set of all possible states), and traverses the computation tree of this program in a depth-first manner. This process is depicted in figure 2, where node (a) represents the initial symbolic state at program point $\langle 0 \rangle$. The first step is to apply a strongest postcondition operator to the constraints at node (a), propagating them through the *then* branch of the first if statement. This produces node (b), where the constraints corresponding to $\texttt{instr}_1$ are abstracted away for clarity. The process continues as long as the constraints in the current symbolic state are satisfiable, generating nodes(c) and (d). At this point we notice that the constraints corresponding to state (d) are unsatisfiable, due to the presence of both b_2 and $\neg b_2$. Thus, the path (abcd) is infeasible. Following the depth first strategy, we backtrack to node (c), and then produce node (e) by propagating the constraints of node (c) through the *else* branch of the last if statement. State (e) is an *exit* state, since it belongs to the exit program point of the current *program block*. Since the constraints at (e) are satisfiable, they produce a valid binding for T_3, leading to a possible value of the execution time of our program fragment.

Upon backtracking, the current computation subtree is summarized, in the sense that an abstract input-output relation is computed for that subtree. To explain summarization, let us first notice that, given a subtree, there exist certain constraints that are present at every node. For instance, the constraint $T_1 = T_0 + 1$ appears in all the nodes of the subtree rooted at (b). Assume now that this constraint is removed from all the nodes of the subtree. Would the remanining subtree still be a valid computation subtree? If yes, then the constraint at hand is redundant and can be abstracted away, as it is the case with $T_1 = T_0 + 1$. On the other hand, b_2 is not redundant for the subtree rooted at (c), since its removal would make node (d) satisfiable and thus not preserve the infeasability of path (abcd). Once all the redundant constraints have been removed[1] from the current subtree, we can derive the summarization[2] as an abstract input-output relationship mapping the root of the abstract tree to the disjunction of the constraints appearing at the frontier.

Having obtained a summarization $[In \mapsto Out]$, where *In* and *Out* are constraints, we can use it in the following way. Assume that the path enumeration process revisits the program point for which this summarization was obtained, this time under a constraint $In \wedge C$ which is subsumed by *In*. Then, we can specialize the summarization as $In \wedge C \mapsto Out \wedge C$, and derive $Out \wedge C$ as a shortcut for the answer. In attaining this very convenient property, it is important that the summarized tree has the same "shape" as the original tree, that is, the removal of constraints *does not open* infeasible paths[3].

Assume now that the tree rooted at (b) has been completely traversed. As we move to node (i), we note that the constraints at this node are *subsumed* by the root of the summarization of node (b) (indeed, the implication $\neg b_1, T_1 = T_0 + 2 \rightarrow true$ holds). At this point, we can use the summarization of (b) as a shortcut, in order to avoid the traversal of the subtree rooted at (i). The frontier of this subtree can be computed as $\neg b_1, T_1 = T_0 + 2, T_3 = T_1 + 3$. The disjunction of the answers for nodes (b) and (i) produces an answer for node (a), which indicates $T_0 + 5$ as an upper bound for T_3. Thus, we can use 5 as an estimate for the WCET of our program fragment. We note at this point that, due to summarization, an important part of the actual computation tree is not traversed, resulting in a substantial reduction of the search space. Importantly, the path realizing the WCET, made up of $\texttt{instr}_2$, $\texttt{instr}_3$, and $\texttt{instr}_5$, is never traversed. The segment $\texttt{instr}_3$, $\texttt{instr}_5$ is traversed and summarized as part of the path $\texttt{instr}_1$, $\texttt{instr}_3$, and $\texttt{instr}_5$. Then, after the traversal of $\texttt{instr}_2$, the summarization of the segment $\texttt{instr}_3$, $\texttt{instr}_5$ is used to produce the WCET estimate.

Our second introductory example shows informally how to hierarchically decompose a program into blocks[4]and provide user assertions in order to produce tight WCET estimations. Consider the bubblesort program shown in figure 3. We divide this program into single-entry-single-exit blocks

[1] Later, the constraint removal operation will be replaced by a more general abstraction operation.

[2] The process is recursive: the summarization of a node can be derived from the summarizations of the children, as shown in Section 4.

[3] In this sense, summarization is opportunistic. Potentially, we could produce more general summarizations that do open up infeasible paths, to allow for a wider range of nodes to be subsequently subsumed. However, that would be more expensive, and with uncertain benefits at point of summarization.

[4] Formal definitions are provided in Section 3.

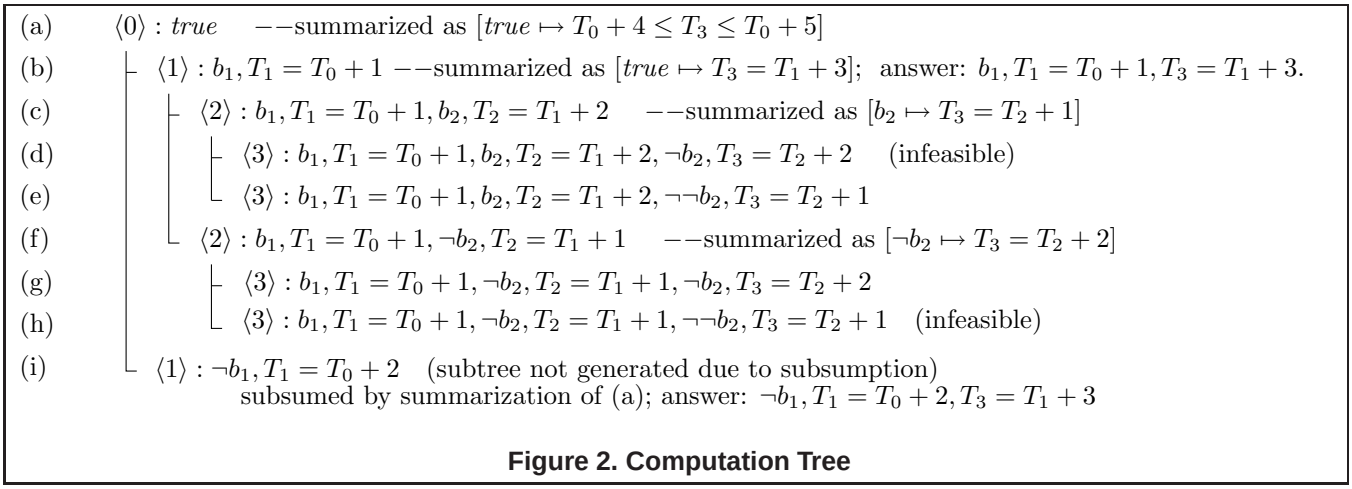

Figure 2. Computation Tree

that can be summarized separately, in the following way. Block *r*, which is the innermost one, consists of the program fragment between program points $\langle 4\rangle$ and $\langle 6\rangle$. Block *q* contains *r*, and consists of the program fragment between program points $\langle 2\rangle$ and $\langle 8\rangle$. Finally, block *p* consists of the entire program, and contains *q*. Each block has its own instrumented variable $\texttt{save\_t}_k$, which saves the current value of the time variable right before the block is entered. Here `p` is the program point where the block starts.

Block *r* represents a straight line program fragment, and its analysis is similar to the one performed in the previous example (which could be considered as being made up of a single block). The resulting summarization will be of the form $[true \mapsto T_6 \leq T_3 + 2, J_6 = J_3 + 1$, which leads to the estimation $WCET_r = 2$. The summarization of block *r* can now be used in the analysis of block *q*, in order to shortcut all transitions between program points $\langle 4\rangle$ and $\langle 6\rangle$. Block *q* however, has a loop that needs to be broken. To that effect, we employ abstraction on the first looping program point that appears on a computation path, in an attempt to discover an invariant. For block *q*, the looping program point is $\langle 4\rangle$. Taking advantage of the loop counter `j`, we produce here the invariant $T_4 = \texttt{save\_t2} + 2 * J_4$[5]. Propagating this invariant outside the loop we obtain at program point $\langle 7\rangle$ that $T_7 = \texttt{save\_t2} + 2 * (N - I_7)$. This constraint contributes to the following summarization for block *q*.

$$[true \mapsto T_8 = \texttt{save\_t2} + 2 * (N - I_8 + 1), I_8 = I_2 + 1$$

where for simplicity, we have abstracted away the value of `j`, and have taken advantage of the fact that the value of `N` is constant. Now, in the analysis of block *p*, the summarization of block *q* can be used to shortcut all transitions between program points $\langle 2\rangle$ and $\langle 8\rangle$. Block *p* also has a loop, and the first looping point occuring on a computation path is $\langle 2\rangle$. Assume first that the user has not specified an invariant for this program point. Then, our algorithm will again take advantage of the loop counter in producing the following invariant for program point $\langle 2\rangle$.

$$T_2 = \texttt{save\_t0} + WCET_q * I_2$$

This invariant can be propagated to program point $\langle 9\rangle$ using block *q*'s summarization, resulting in the following invariant.

$$T_9 = \texttt{save\_t0} + 2 * (N+1) * N$$

This results in an estimation of block *p*'s WCET of $2 * N * (N+1)$, which is an over-approximation by a factor of 2.

Suppose now that the user did provide the following invariant for program point $\langle 2\rangle$.

$$T_2 = \texttt{save}_\texttt{t}\texttt{0} + (2 * N - I_2) * (I_2 + 1)$$

Using block *q*'s summarization, we can propagate this invariant to program point $\langle 8\rangle$, and then, using the transition $\langle 8\rangle \rightarrow \langle 2\rangle$, back to program point $\langle 2\rangle$. We can now verify that the propagated constraint is *subsumed* by the original invariant, thus *validating* it. By propagating the user-specified invariant to program point $\langle 9\rangle$, we obtain the *exact* and *certified* estimate $WCET_p = N * (N+1)$.

3 Constraint Transition Systems

We start by defining a language of first-order formulas. Let $\mathcal{V}$ denote an infinite set of variables, each of which has a type in the domains $\mathcal{D}_1, \cdots, \mathcal{D}_n$, let Σ denote a set of *functors*, and Π denote a set of *predicate symbols*. We assume that each predicate symbols is given a numeric rank. A *term* is either a constant (0-ary functor) in Σ or of the form $f(t_1, \cdots, t_m)$, $m \geq 1$, where $f \in \Sigma$ and each t_i is a term, $1 \leq i \leq m$. Similarly, an *atom* is of the form $p(t_1, \cdots, t_m)$, $m \geq 1$, where $p \in \Pi$ and each t_i is a term, $1 \leq i \leq m$. A

[5] In Section 4, we show that such invariants can be validated using block *r*'s summarization

```
⟨0⟩  i=0; save_t0=t ;
⟨1⟩  while (i<N) {
⟨2⟩       j=0; save_t2=t ;
⟨3⟩       while (j<N-i) {
⟨4⟩            if (a[j+1]<a[j])
                    { swap(a,j,j+1); t+=2; }
               else t++;
⟨5⟩            j++
⟨6⟩       }
⟨7⟩       i++
⟨8⟩  } ⟨9⟩
```

Figure 3. Bubble Sort

$$
\begin{array}{l}
p(0,\tilde{\chi}) \mapsto p(1,i',j,a,t,save\_t_0',save\_t_2), i'=0, save\_t_0'=t \\
p(1,\tilde{\chi}) \mapsto p(2,\tilde{\chi}), i<N \\
p(1,\tilde{\chi}) \mapsto p(9,\tilde{\chi}), i \geq N \\
p(2,\tilde{\chi}) \mapsto q(2,\tilde{\chi},\tilde{\chi}'), p(8,\tilde{\chi}') \\
p(8,\tilde{\chi}) \mapsto p(2,\tilde{\chi}), i<N \\
p(8,\tilde{\chi}) \mapsto p(9,\tilde{\chi}), i \geq N, t=0 \\
\\
q(2,\tilde{\chi},\tilde{\chi}') \mapsto q(3,i,j',a,t,save\_t_0,save\_t_2',\tilde{\chi}'), \\
\quad j'=0, save\_t_2'=t \\
q(3,\tilde{\chi},\tilde{\chi}') \mapsto q(4,\tilde{\chi},\tilde{\chi}'), j<N-i \\
q(3,\tilde{\chi},\tilde{\chi}') \mapsto q(7,\tilde{\chi},\tilde{\chi}'), j \geq N-i \\
q(4,\tilde{\chi},\tilde{\chi}'') \mapsto r(4,\tilde{\chi},\tilde{\chi}'), q(6,\tilde{\chi}',\tilde{\chi}'') \\
q(6,\tilde{\chi},\tilde{\chi}') \mapsto q(4,\tilde{\chi},\tilde{\chi}'), j<N-i \\
q(6,\tilde{\chi},\tilde{\chi}') \mapsto q(7,\tilde{\chi},\tilde{\chi}'), j \geq N-i \\
q(7,\tilde{\chi},\tilde{\chi}'') \mapsto q(8,i',j,a,t,save\_t_0,save\_t_2,\tilde{\chi}''), i'=i+1 \\
\\
r(4,\tilde{\chi},\tilde{\chi}'') \mapsto r(5,i,j,a',t',save\_t_0,save\_t_2,\tilde{\chi}''), \\
\quad a[j+1]<a[j], a'=swap(a,j+1,j), t'=t+2 \\
r(4,\tilde{\chi},\tilde{\chi}'') \mapsto r(5,i,j,a',t',save\_t_0,save\_t_2,\tilde{\chi}''), \\
\quad a[j+1] \geq a[j], t'=t+1 \\
r(5,\tilde{\chi},\tilde{\chi}'') \mapsto r(6,i,j',a,t,save\_t_0,save\_t_2,\tilde{\chi}''), j'=j+1
\end{array}
$$

Legend: $\tilde{\chi}$ abbreviates the sequence $i,j,a,t,save\_t_0,save\_t_2$
$\tilde{\chi}'$ and $\tilde{\chi}''$ are its primed versions.

Figure 4. CTS of Bubble Sort

primitive constraint is of the form $\phi(t_1,\cdots,t_m)$ where ϕ is a $m-ary$ constraint symbol and each t_i is a term, $1 \leq i \leq m$. A *constraint* is constructed from primitive constraints using logical connectives in the usual manner. Where Ψ is a constraint, we write $\Psi(\tilde{x})$ to denote that Ψ possibly refers to variables in $\tilde{x}$, and we write $\tilde{\exists}\Psi(\tilde{x})$ to denote the existential closure of $\Psi(\tilde{x})$ over variables away from $\tilde{x}$.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. We write $[\tilde{x} \mapsto \tilde{y}]$ to denote such mappings. A *grounding* is a substitution which maps each variable into a value in its domain. Where e is an expression containing a constraint Ψ, $[\![e]\!]$ denotes the set of its instantiations obtained by applying *all possible* groundings which satisfy Ψ.

A program is organized hierarchically using blocks, each of which is represented as a transition system for a predicate symbol p. A nested block is then represented by a transition system for a predicate symbol whose rank is less than that of p. This system can be executed symbolically. The following key definition serves two main purposes. First, it is a high level representation of the operational semantics of p, and in fact, it represents the exact *trace* semantics of p. Second, it is an *executable specification* against which an assertion can be checked.

We shall model computation by considering n *system variables* $\tilde{v} = v_1,\cdots,v_n$ with domains $\mathcal{D}_1,\cdots,\mathcal{D}_n$ respectively, and a program counter k ranging over program points. In this paper, we shall use just two example domains, that of integers, and that of integer arrays.

Definition 1 (States and Transitions) *A* system state *(or simply state) is of the form $(k,d_1,\cdots,d_n)$ where k is a program point and $d_i \in \mathcal{D}_i, 1 \leq i \leq n$, are values for the system variables. A* transition *is a pair of states.* □

In what follows, we use atoms in the form $p(k,\tilde{x},\tilde{y})$ where the number k denotes a program point, and the variables $\tilde{x}$ and $\tilde{y}$ respectively denote the input and output system states.

Definition 2 (Constraint Transition System) *A basic* constraint transition *of p is a formula*

$$p(k,\tilde{x},\tilde{z}) \mapsto \Psi(\tilde{x},\tilde{y}), p(k_1,\tilde{y},\tilde{z})$$

The variables $\tilde{x}$ and $\tilde{z}$ are, respectively, the primary *and* secondary *variables of the transition. The constraint Ψ may possibly contain auxiliary variables in addition to $\tilde{x}$ and $\tilde{z}$.*

A composite *constraint transition is similar:*

$$p(k,\tilde{x},\tilde{z}) \mapsto q(0,\tilde{x},\tilde{y}),\ p(k_1,\tilde{y},\tilde{z})$$

where the rank of q is higher than that of p.

A constraint transition system *(CTS) of p is a finite set of constraint transitions of p.* □

The above formulation of program transitions is familiar in the literature for the purpose of defining a set of transitions. What is new, however, is how we use a CTS to define a *symbolic* transition sequences, and thereon, the notion of a proof. Figure 4 exemplifies a representation of the bubblesort program in Figure 3. Note that we have chosen to represent the inner loop as a block q, and the body of this loop as a block r.

By similarity with logic programming, we use the term *rule* to denote a constraint transition, and *goal* to denote a literal that can be subjected to an *unfolding process* in order to infer a logical consequence.

Definition 3 (Goal) *A query or* goal *has the same form as the "body" of a rule, ie*

$$\Psi(\tilde{y},\tilde{z}), p(k_1,\tilde{y},\tilde{z}) \quad or \quad q(0,\tilde{y},\tilde{y}), p(k_1,\tilde{y},\tilde{z})$$

The variables $\tilde{x}$ and $\tilde{z}$ are respectively called the primary and secondary variables of this goal, while any additional variable in Ψ is called an auxiliary *variable of the goal. Where $\mathcal{G}$ is a goal, we write constraint($\mathcal{G}$) to denote the constraint in $\mathcal{G}$, and $\mathcal{G}(\tilde{x},\tilde{z})$ to highlight the primary and secondary variables in $\mathcal{G}$.* □

We say the goal is a *initial goal* for p if $k = 0$, the start program point for the block represented by p. Similarly, a goal is a *final goal* if k is the terminal program point for the block represented by p. We say that a goal $\mathcal{G}$ is *subsumed* by another $\mathcal{G}_1$ if $[\![\mathcal{G}]\!] \subseteq [\![\mathcal{G}_1]\!]$.

Running a initial goal is tantamount to asking the question: which values of $\tilde{x}$ which satisfy $\tilde{\exists}\Psi(\tilde{x})$ will lead to a goal at the final point? The idea is that we successively reduce one goal to another until the resulting goal is final point, and then inspect the results.

Next we define what it means for a CTS to prove a goal.

Definition 4 (Proof Step, Sequence and Tree) *Let there be a CTS for p, and let $\mathcal{G} = \Psi, p(k,\tilde{x},\tilde{z})$ be a goal for this. A* proof step *from $\mathcal{G}$ may be obtained providing Ψ is satisfiable. It may be obtained using a variant $p(k,\tilde{x}',\tilde{z}') \mapsto \Psi', p(k_1,\tilde{y},\tilde{z}')$ of a basic rule in the CTS in which all the variables are fresh. The result is a goal of the form*

$$\Psi, \tilde{x} = \tilde{x}', \tilde{z} = \tilde{z}', \Psi', p(k_1,\tilde{y},\tilde{z}')$$

Note that this new goal is a false goal if the constraint $\Psi, \tilde{x} = \tilde{y}, \Psi_1$ is unsatisfiable. A proof step from $\mathcal{G}$ may also be obtained using a variant of a composite rule $p(k,\tilde{x}',\tilde{z}') \mapsto q(0,\tilde{x}',\tilde{y}'), p(k_1,\tilde{y}',\tilde{z}')$. The result is a goal of the form

$$\Psi, \tilde{x} = \tilde{x}', \tilde{z} = \tilde{z}', q(0,\tilde{x}',\tilde{y}'), p(k_1,\tilde{y}',\tilde{z}')^6$$

A proof sequence *is a finite or infinite sequence of proof steps. A* proof tree *is defined from proof sequences in the obvious way.* □

Our algorithm accommodates the use of predefined or *static* abstraction. Given an arbitrary abstraction function $\mathcal{A}$ which maps a goal $\Psi, p(k,\tilde{x},\tilde{z})$ into a goal $\Psi', p(k,\tilde{x},\tilde{z})$ where $\Psi \models \Psi'$, we say that a proof tree is obtained with static abstraction if the proof tree is obtained as before, except that when a goal $\mathcal{G}$ is obtained via a proof step, we replace $\mathcal{G}$ by the goal obtained by applying the abstraction function $\mathcal{G}$. Note that this allows, in particular, "intermittent" abstraction [7] where abstraction is performed only at selected goals.

[6] We shall not need to describe further proof steps from such a goal.

4 The Algorithm

The main algorithm, in Figure 5, produces a *summarization* of a given goal $\mathcal{G}$. This is a triple $(\bar{\mathcal{G}}, \Psi, \Psi^m)$ where $\bar{\mathcal{G}}$ is a generalization of $\mathcal{G}$, Ψ a disjunction of constraints, and Ψ^m a single constraint which represents one of the disjuncts in Ψ. Implicit in both Ψ and Ψ^m are final variables which represent the values of the primary variables at the final program point of $\mathcal{G}$. The disjunction Ψ describes an approximation of all possible reachable final values from $\bar{\mathcal{G}}$ (not necessarily $\mathcal{G}$), and Ψ^m describes the *maximal* value of the difference between the time variable of $\bar{\mathcal{G}}$ and its corresponding final variable. Thus the main use of a summarization is, as its name suggests, to produce a "closed form", a disjunction of constraints, for a goal $\mathcal{G}$, say $\Psi_0, p(k,\_)$. The summarization is described as a disjunction Ψ for a generalization of $\mathcal{G}$. The closed form for $\mathcal{G}$ itself is then obtained by simply conjoining Ψ_0 with Ψ. Similarly, the optimal time consumption of $\mathcal{G}$ is given by $\Psi_0 wedge \Psi^m$. Note that this means the use of a summarization provides the feature of a "frame rule" allowing certain constraints in an input context to be also output.

The algorithm performs a depth-first construction of a proof tree for $\mathcal{G}$, enumerating its symbolic traces. Termination occurs when there is a finite number of traces, for example, when the loops in the program have statically determined bounds, because summarization provides a form of memo'ing which prevents cycling. Note that this does not mean that the number of program states are finite.

In general, however, termination and indeed, efficiency, would depend on a judicious use of static abstraction. A general methodology is that abstraction is performed at the *final* program points of each block. The idea here is that a block represents an encapsulable program fragment with a defined purpose. Thus its intended purpose should be identifiable with some abstract statement. A typical example is that a block only operates on a certain subset of variables, and so an abstraction function is simply to perform projection of encountered constraints onto these variables. Another important example is that a block is the body of a loop, and therefore abstraction serves to provide a *loop invariant*. We note that in the algorithm in Figure 5, we have provided for the application of an abstraction $\mathcal{A}$ at precisely the final program points of the subject goal $\mathcal{G}$. The idea here is that $\mathcal{A}$ curtails the proliferation of constraints in the construction of a disjunction of constraints. This however does

Solve $\mathcal{G} \equiv \Psi_0, p(k,\tilde{x},\tilde{z})$, returns $(\bar{\mathcal{G}},\ \Psi,\ \Psi^m)$

(0) if $\mathcal{G}$ is subsumed by a previous call
let $\mathcal{G}$ generalize $\mathcal{G}$ and remain subsumed
return $(\bar{\mathcal{G}},\ \mathit{false},\ \mathit{false})$

(1) if there exists a summarization $(\mathcal{G}_s, \Psi_s, \Psi_s^m)$ s.t.
$\mathcal{G}_s$ subsumes $\mathcal{G}$ and $(\mathcal{G}, \Psi_s^m\theta) \not\equiv \mathit{false}$,
where θ renames $\mathcal{G}_s$ into $\mathcal{G}$
return $(\mathcal{G}_s,\ \Psi_s, \Psi_s^m)$

(2) if $\mathcal{G}$ cannot be further reduced
if $\mathcal{G}$ is a final goal: **return** $(\mathcal{A}(\mathcal{G}),\ \mathit{true},\ \mathit{true})$
if $\mathcal{G}$ is *false*: **return** $(\mathit{false},\ \mathit{false},\ \mathit{false})$

(3) if $\mathcal{G}$ has reducts $\mathcal{G}_1, \ldots, \mathcal{G}_p$

For each $1 \leq i \leq p$, let R_i denote the rule used, and ϕ_i denote the constraints obtained from the one-step transition from $\mathcal{G}$ to $\mathcal{G}_i$. Now if R_i is

- a basic rule:
  solve $\mathcal{G}_i$ obtaining $(\bar{\mathcal{G}_i}, \Psi_i, \Psi_i^m)$
  let R_i' be the rule R_i
- a composite rule:
  suppose $\mathcal{G}_i$ is of the form $q(0,\tilde{x},\tilde{y}), p(k_1,\tilde{y},\tilde{z})$
  solve $\Psi_0, q(0,\tilde{x},\tilde{y})$ obtaining $(\_,\ \Psi_q,\ \_)$
  let R_i' be the rule: $p(k,\tilde{x},\tilde{z}) \mapsto \Psi_q, p(k_1,\tilde{y},\tilde{z})$
  solve $\Psi_0, \Psi_q, p(k_1,\tilde{y},\tilde{z})$ obtaining $(\mathcal{G}_i, \Psi_i, \Psi_i^m)$

Let
$\mid \phi_j \wedge \Psi_j^m \mid$ be optimal amongst $1 \leq j \leq p$
$\bar{G} = wp(\bar{\mathcal{G}}_1, R_1') \wedge \cdots \wedge wp(\bar{\mathcal{G}}_p, R_p')$
$\Psi = \mathcal{A}((\phi_1 \wedge \Psi_1) \vee \cdots \vee (\phi_p \wedge \Psi_p))$, and
$\Psi^m \equiv (\phi_j, \Psi_j^m)$

summarize and **return** $(\bar{G},\ \Psi,\ \Psi^m)$.

Figure 5. The Algorithm

not mean that we must abstract (for we can choose a passive identity abstraction function), and this also does limit us to perform abstractions at other program points.

An important point is that there is a standard way to use the algorithm *automatically*, with no user input. Essentially, this means that loops need to be provided with an invariant in a standard way. This can be done because the algorithm, in addition to verifying that certain properties hold, also *discovers* time consumption. In particular, where a block represents a loop body, the algorithm can discover an estimate of its WCET, either as a single number, or as a symbolic expression involving the program variables. This expression, say α, can be used in conjunction with a variable, say i, that behaves as the loop counter, in order to specify the obvious loop invariant: $t \leq t_0 + i * \alpha$ where the variable t_0 has been instrumented to capture the value of t just before the loop executes. Now, in addition to this standard loop invariant, one could also include constraints that are known to be invariant through the loop. More concretely, suppose we encounter a loop with context Ψ. Let Ψ' denote a generalization of Ψ which is invariant. For example, we could retain in Ψ' just those constraints in Ψ that are not modified in the loop. It is easy to see that we can now process the loop using the invariant $\Psi' \wedge t \leq t_0 + i * \alpha$. Note that if the loop were a single block and when it is summarized by an invariant, say Ψ, then it can be used in any context, say Ψ_0, so that the output of the loop is $\Psi_0 \wedge \Psi$. Once again, note that this use of the loop summarization allows the passage of constraints in the input context Ψ_0 to be propagated through the loop.

We now provide a few necessary definitions.

Intuitively, the *weakest precondition* $wp(\mathcal{G}, R)$ of a goal $\mathcal{G}$ when produced by the rule R is the most general goal $\mathcal{G}'$ such that there is a proof step $\mathcal{G}' \mapsto \mathcal{G}$ using the transition R. Formally, the *weakest precondition* of a goal $\mathcal{G} \equiv \Psi_0(\tilde{x},\tilde{y}), p(k_1,\tilde{x},\tilde{y})$ wrt to a rule R of the form $p(k,\tilde{x},\tilde{z}) \mapsto \Psi(\tilde{x},\tilde{y}), p(k_1,\tilde{y},\tilde{z})$ is as follows:

$$wp(\mathcal{G}, R) \equiv \forall \tilde{y}.\Psi(\tilde{x},\tilde{y}) \longrightarrow \Psi_0(\tilde{y},\tilde{z}), p(\tilde{x},\tilde{z})$$

The *time consumption* of a constraint is defined as follows. We shall be applying this concept to a constraint of the form $\phi \wedge \Psi$ where ϕ is obtained from a rule, and Ψ is obtained from a summarization. Thus ϕ has a set of primary variables, one of which, say t, represents the time variable. Similarly, because Ψ is obtained from a (previous) summarization, its has a set of final variables one of which, say t_f, represents the time variable. We now define that the expression $\mid \phi \wedge \Psi \mid$ denotes the value of $t_f - t$. In the algorithm, we shall be comparing resource consumption over pairs of such constraints $\phi \wedge \Psi$. We remark that since the time consumption is, in general, a symbolic expression, that it is not always possible to determine which expression is the larger. We believe however that in practice it is usual that this comparison can be made.

Finally, we remark that this algorithm can in fact be used to prove general safety properties. For example, we could instrument the program so that a certain unsafe program point is goto'd just when the desired safety property fails to hold at a desired program point. We would then provide an abstraction function for each loop body which provides a constraint that is true for every iteration of the loop, ie a loop invariant. Then, by applying the algorithm to a goal corresponding to the start state of the program, we could verify that no goal involving the unsafe program point is ever encountered. Both the termination of the algorithm and the determination of safety is guaranteed if the provided loop invariants are correct and sufficiently precise in the sense of traditional Hoare-style program verification.

5 Examples

5.1 IPET

In this section we demonstrate how we may perform timing reasoning similar to those of *implicit path enumeration* (*IPET*) technique [9]. Here we use an example of [9]. The original program, its CFG, and the instrumented program is shown in Figure 6. Here we assume that each statement costs 1 t.u. In the CFG, $\langle \mathrm{p} \rangle$ is denoted as a box with variable x_p. Here, x_p denotes a counting variable that is incremented when $\langle \mathrm{p} \rangle$ is visited. In the instrumented program we provide a loop bound as loop condition. The IPET method also requires the user to provide the same information.

The steps of timing reasoning using our algorithm is as follows:

1. $p(0,0,x_2,0,0,0), 0 \leq x_2$
2. $p(0,1,x_2,0,0,1), 0 \leq x_2$
3. $p(2,1,x_2+1,0,0,2), 0 \leq x_2$
4. $p(4,1,10,9,1,21)$
5. $p(5,1,10,9,1,22)$

Steps 1 to 3 and 4 to 5 are obvious, however, here there is a jump from step 3 to 4. This is because we specify the while loop in the program as a block, which is analyzed separately.

We note that we can also write goal 3 as

$$p(2,1,x_2,0,0,t), 1 \leq x_2, t = t_0 = 2.$$

This we can abstract into the following goal, using standard loop invariant:

$$p(2,1,x_2,x_3,0,t), 1 \leq x_2 \leq 10, t - t_0 \leq 2x_2.$$

The coefficient 2 in the last constraint can be discovered, but here we assume it is given.

The analysis of the loop itself is shown below:

a. $q(2,1,x_2,x_3,0,t)$,
$1 \leq x_2 \leq 10, x_2 = x_3 + 1, t - t_0 \leq 2x_2$
b. $q(3,1,x_2,x_3+1,0,t+1)$,
$1 \leq x_2 \leq 10, x_2 = x_3 + 1, t - t_0 \leq 2x_2$
c. $q(2,1,x_2+1,x_3+1,0,t+1)$,
$1 \leq x_2 \leq 10, x_2 = x_3 + 1, t - t_0 \leq 2x_2$
d. $q(4,1,x_2,x_3,1,t+1)$,
$x_2 = 10, x_3 = 9, t - t_0 \leq 2x_2$

Here, the analysis of a branches into b and d. B is further unfolded to c. We note here that c is subsumed by a.

At d, we are able to produce the final goal
$q(4,1,x_2,x_3,1,t), x_2 = 10, x_3 = 9, t - t_0 \leq 21$

The produced summarization by the analysis of the loop alone is therefore:
$q(2,1,x_2,x_3,0,t,x_1',x_2',x_3',x_4'), 1 \leq x_2 \leq 10, x_2 = x_3 + 1 \mapsto$
$x_1' = 1, x2' = 10, x_3' = 9, x_4' = 1, t' - t \leq 20$

Applying this summarization to 3 we get 4.

For comparison, given the same CFG, the IPET method would produce the constraints
$x_1 = 1, x_2 = x_1 + x_3 = x_3 + x_4, x_2 \leq 10$
where the last constraint is given by the user to bound the number of loop iterations. The IPET method proceeds by using an ILP solver to maximize $x_1 + x_2 + x_3 + x_4$, which would result in 22, which is the same answer as ours. When IPET uses ILP solver, we use search method to analyze WCET with the same accuracy by using counting variables.

We note here that when we only have counting variables in the program, the size of the analysis is always *linear* to the number of statements in the program (nodes in the CFG). Intuitively, we perform longest path computation using *dynamic programming*.

The advantage of IPET approach well-touted in the literature is the user's ability to specify ad-hoc constraints on the counting variables, such as demonstrated by the bounding of the variable x_2 above. Other such constraints include *mutual exclusive paths* and *loop down sampling*. In our method it is also always easy to specify similar constraints in the CTS transitions.

5.2 Cache Must-Analysis

We briefly demonstrate our algorithm on a well-known abstract interpretation approach to modelling caches [18, 19], using "must-analysis". Consider Figure 3 again and suppose that the inner loop's time depends on cache behavior, say 5 for a miss and 1 for a hit.

Here we add a list data structure into our CTS representation, and we model the cache state as a quadruple $[i_1, i_2, i_3, i_4]$, where i_1 is the youngest instruction, and i_4 the oldest instruction in the cache. In a must analysis, an element in the quadruple must either be a number denoting instruction or $\emptyset$, denoting unknown instruction.

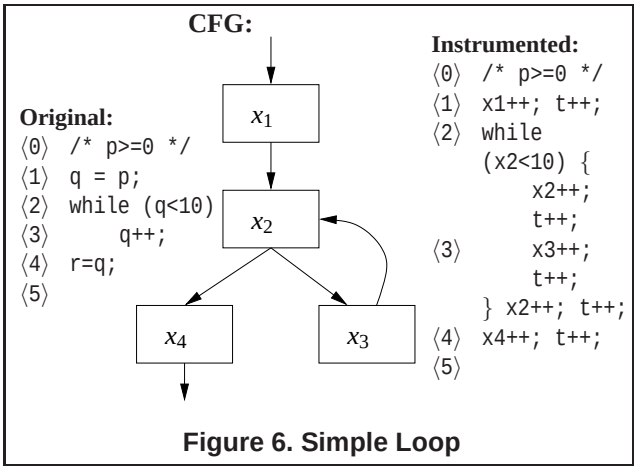

Figure 6. Simple Loop

We then split each CTS rule into two: one which models the cache hit, and another modeling cache miss. Here we use the inner loop of the bubble sort (Figure 3) as an example. Here, the statement at $\langle 3 \rangle$ is re-modeled by the following 2 transitions:

$$q(3,i,j,c,t) \mapsto 1 \in c, j < 99-i, \\ q(4,i,j,lrupeek(1,c),t+1)$$
$$q(3,i,j,c,t) \mapsto 1 \notin c, j < 99-i, \\ q(4,i,j,lrupush(1,c),t+5)$$

We note that the argument c is the list representing the cache state. Note also that we use the basic functions *lrupush* and *lrupeek* to model the possible updates on the cache. *Lrupush* pushes an intruction into the cache as the youngest, and flushes the oldest instruction. *Lrupeek* sets a particular instruction to be the youngest and increments the age of all of the previously younger instructions.

At the start of the loop, the reachable states can be abstracted into the following single goal which represents the loop invariant:

$$q(1,i,j,[4,\emptyset,2,1],t), 1 \le j \le 99-i, \\ i < 99, t-t_0 \le 8(j-1).$$

6 Experiments on Subsumption

In this section we provide experimental runs to gauge the effectiveness of dynamic summarization in increasing the chance for subsumption (case (1) in Figure 5) to reduce the size of timing analysis. In these experiments we consider all programs to be consisting only of a single block, and we therefore unroll all loops.

We implement a prototype which analyzes timing and performs optimization using summarization when given a code fragment. The implementation is a pure CLP($\mathcal{R}$) program [6]. We augment the CLP($\mathcal{R}$) system with a memoization mechanism, storing the best WCET result of each encountered goal, in order to perform subsumption checking. We also implement an efficient algorithm based on constraint deletion to check redundant constraints at every false and subsumed node.

First consider the bubble sort program above. We consider two variations, in order to demonstrate the handling of complex contexts: (a) first is without microarchitecture considerations and where we simply count the number of swaps, and (b) where underlying data cache is modeled. In (b), a cache miss results in additional 1 t.u. to the array element comparison.

Results are shown in Table 1, obtained using Linux 2.4.22 OS on a Pentium 4 2.8GHz processor with 512Mb RAM. Note that the number of nodes is linear in the square of array size (which in turn is linear to the maximal path length), in both versions[7]. The cached version introduced more constraints, as expected, leading to a more complex analysis. Nevertheless, our algorithm remained linear.

Next we ran a few random programs such as the ADPCM encoder and decoder [14]. Here we simply assumed that each C statement consumes 1 time unit. For both programs, we also coded versions with instruction cache, in the same spirit as Section 5.2. The cache stores up to 8 statements, where a cache miss costs 10 time units. Here we obtain the exact answer for the encoder (w/o cache) which is 37, whereas it is 39 if infeasible paths were included. We also experimented with an iterative square root algorithm [16] and janne_complex [5, 11] examples. The results are shown in Table 2.

In all examples, our dynamic summarization produced significant improvements.

7 Conclusion

We have presented a general algorithm for analyzing the traces of a program, with emphasis on timing constraints. Its main feature is that it can use, and then prove, user provided assertions in the form of abstraction functions. Importantly, it is able to operate automatically by resorting to abstraction functions that are oblivious to everything except timing constraints.

The main technical feature is that the algorithm produces, dynamically, a summarization given a program fragment and its context. A summarization in fact describes the behavior of the program fragment in a context more general than the original. Its key advantage is thus to provide an opportunity of avoiding the same program fragment in different contexts. We have shown for a small benchmark suite that this opportunity indeed occurs frequently in practice.

Finally, the algorithm considers programs that are hierarchically structured. Besides leading to a *compositional*

[7] Standard benchmarks [16, 11] usually fix the array values and consider analysis only on a single execution path.

Problem	Array Size	No Summarization **Nodes**	(Time)	W/ Summarization **Nodes**	(Time)	Derived Answer
Normal	5	**1606**	(9.98)	**58**	(0.06)	10
	15	∞		**478**	(10.71)	105
	25	∞		**1298**	(134.48)	300
	35	∞		**2518**	(824.72)	595
Cached	5	**2233**	(20.46)	**88**	(0.20)	16
	10	∞		**336**	(5.03)	66
	15	∞		**798**	(45.02)	149
	20	∞		**1410**	(216.60)	266

Table 1. Bubble Sort

Problem	No Summarization **Nodes**	(Time)	W/ Summarization **Nodes**	(Time)	Derived Answer
Encoder	**494**	(1.22)	**266**	(0.69)	37
Decoder	**344**	(0.46)	**164**	(0.30)	22
Encoder (Cached)	**494**	(1.63)	**266**	(0.95)	82
Decoder (Cached)	**344**	(0.56)	**164**	(0.39)	48
Square Root	**923**	(5.96)	**253**	(1.91)	140
Janne_complex	**1517**	(24.25)	**683**	(5.8)	81

Table 2. Some Random Programs

approach, this facilitates the specification of user assertions and guides the processes of summarization, re-use, and assertion validation, and allows the user to explore trade-offs between efficiency and precision.

References

[1] A. Betts and G. Bernat. Tree-based WCET analysis on instrumentation point graphs. In *ISORC '06, 558–565, 2006.*

[2] A. Colin and G. Bernet. Scope-tree: A program representation for symbolic worst-case execution time analysis. In *ECRTS 2002*, pages 50–60, 2002.

[3] A. Colin and I. Puaut. A modular and retargetable framework for tree-based WCET analysis, 2001.

[4] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. [5], pages 1298–1307.

[5] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. *3rd Euro-Par*, *LNCS 1300*, 1298–1307, 1987.

[6] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) lang and system. *TOPLAS*, 14(3):339–395, 1992.

[7] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP method for compositional and intermittent predicate abstraction. *7th VMCAI*, volume 3855 of *LNCS*, 17–32. Springer, 2006.

[8] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. 92–103.

[9] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *2nd LCT-RTS*, 88–98, 1995.

[10] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. volume 17, 183–207, 1999.

[11] Mälardalen WCET research group benchmarks. URL `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[12] P. Puschner. Is WCET analysis a non-problem?–towards new software and hardware architectures, WCET2002.

[13] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *J.RTS*, 18(2/3):115–128, May 2000.

[14] R. Richey. *Adaptive Differential Pulse Code Modulation Using PICmicro Microcontrollers*. Microchip Tech Inc., 1997.

[15] Tobias Schuele and Klaus Schneider. Abstraction of assembler programs for symbolic worst case execution time analysis. In *DAC '04*, 107–112, 2004.

[16] SNU real-time benchmarks. URL `http://archi.snu.ac.kr/realtime/benchmark/.`

[17] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Efficient detection and exploitation of infeasible paths for software timing analysis. 358–363.

[18] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by seperate cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, May 2000.

[19] R. Wilhelm. Why AI+ILP is good for WCET, but MC is not, nor ILP alone. *5th VMCAI*, LNCS 2937, 309–322, 2004.