Efficient Detection and Exploitation of Infeasible Paths for Software Timing Analysis

Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen
Department of Computer Science, National University of Singapore
{vivy, tulika, abhik, chent}@comp.nus.edu.sg

ABSTRACT

Accurate estimation of the worst-case execution time (WCET) of a program is important for real-time embedded software. Static WCET estimation involves program path analysis and architectural modeling. Path analysis is complex due to the inherent difficulty in detecting and exploiting infeasible paths in a program's control flow graph. In this paper, we propose an efficient method to exploit infeasible path information for WCET estimation without resorting to exhaustive path enumeration. We demonstrate the efficiency of our approach for some real-life control-intensive applications.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Measurement, Performance

Keywords

WCET analysis, infeasible path detection

1. INTRODUCTION

Estimating the *Worst Case Execution Time* (WCET) of a program is an important problem in embedded system design. WCET analysis computes an upper bound on the program's execution time on a particular processor for all possible inputs. The immediate motivation of this problem lies in schedulability analysis of real-time embedded systems. Many embedded systems are safety critical (*e.g.*, automotive electronics) and have timing constraints. These timing constraints impose hard deadlines on the execution time of embedded software. WCET analysis of a program can guarantee that such deadlines are met.

Static WCET analysis of a program typically consists of three phases: *flow analysis* to identify loop bounds and infeasible flows through the program; *architectural modeling* to determine the effects of pipeline, cache, branch prediction etc. on the execution time; and finally *calculation* to find an upper bound on the WCET given the results of the flow analysis and the architectural modeling. In this paper, we concentrate on the infeasible path detection and WCET calculation. We note that different WCET analysis techniques typically combine the results of program flow analysis and micro-architectural modeling via a *separated* approach. In this approach, the micro-architectural modeling is used to get an estimate of the WCET of each basic block of the program. These basic block WCET estimates are combined with flow analysis results to produce the program's WCET estimate. Our WCET analysis can also be similarly integrated with micro-architectural modeling.

There exist mainly three different approaches for WCET calculation — *tree-based*, *path-based*, and *implicit path enumeration*. The tree-based approach estimates the WCET of a program through a bottom-up traversal of its syntax tree and applying different timing rules at the nodes (called "timing schema") [12]. This method is quite simple and efficient. But it is difficult to exploit infeasible paths in this approach as the timing rules are local to a program statement. Implicit path enumeration techniques [10] represent the program flows as linear equations or constraints and attempt to maximize the execution time of the entire program under these constraints. This is done via an Integer Linear Programming (ILP) solver. Path-based techniques estimate the WCET by computing execution time for the feasible paths in the program and then searching for the one with the longest execution time. Naturally, they can handle various flow information, but they enumerate a huge number of paths. Recent works [15] have sought to reduce this expensive path enumeration by removing infeasible paths from the flow graph.

In this paper, we present a technique for finding the WCET of a program in the presence of infeasible paths without performing exhaustive path enumeration. We focus on the path-based technique because it generates the worst-case execution path, which is required for various compiler optimization techniques that attempt to reduce the WCET (*e.g.*, [18, 16]). Our technique traverses the control flow graph to find the longest path, but avoids the high cost of path enumerations by keeping track of more information than just the heaviest path during traversal. We pre-compute possible "conflict relations" or sources of infeasibility and maintain the heaviest path for each such source of infeasibility. As a result, even when we find that the "heaviest" path is infea-

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

sible, we do not backtrack to find alternative paths. This indeed is the key idea of the approach. In the rest of the paper, we outline our definition of conflict relations, how this information is used in the WCET calculation, and the experimental results.

2. RELATED WORK

One of the earliest works on programming language level timing analysis is the *timing schema* approach [14]. It is a bottom-up compositional technique which finds the worst-case execution time of a program fragment without considering the contexts in which it is executed. Techniques to extend the timing schema approach with infeasible path information have been reported in [12]. In this work, the infeasible path patterns are user-provided.

Among other works in WCET calculation, [2] searches for infeasible paths in a control flow graph via branch-and-bound search. Lundqvist and Stenstrom [11] provide an extended simulation method (which can proceed in the presence of unknown data values and allows path merging) for detection/elimination of infeasible paths. Infeasible path detection/exploitation in ILP-based WCET calculation has also been investigated. Recently, many diverse kinds of flow information have been successfully integrated into ILP [4, 13], making ILP-based WCET calculation quite popular.

Among path-based WCET calculation methods, the work of Stappert et. al. [15] cuts complexity by (a) finding the longest program path π, (b) checking for the feasibility of π, and (c) removing π from control flow graph followed by the search for a new longest path if π is infeasible. This technique is a substantial improvement over exhaustive path enumeration. However, if the feasible paths in the program have relatively low execution times, then this approach still has to examine many paths. We present experiments to compare [15] with our method (see Section 6).

Ermedahl and Gustafson [5] use dataflow analysis to derive (and exploit) infeasible paths using program semantics; a nice feature of this work is that it also automatically derives minimum and maximum loop bounds in a program. Healy and Whalley detect and exploit branch constraints within the framework of the path-based technique [7]. The key idea here is to compute the effect of any assignment or a branch on other branch outcomes; this is an efficient way of computing many common infeasible path patterns. Thematically, [7] is somewhat related to our work because we also maintain/exploit pairwise conflicts between branches and assignments. However, our WCET calculation includes an in-built machinery to perform an accurate path-sensitive search and yet avoid enumeration of acyclic paths in a loop.

Elimination of infeasible paths has also been studied in the context of software model checking [8]. These works abstract data values via a finite number of propositions and eliminate infeasible paths by invoking an external theorem prover. Our analysis works directly on the control flow graph (data values are not considered at all) but removes the need for external theorem provers by detecting and exploiting a limited notion of infeasibility.

3. INFEASIBLE PATH INFORMATION

In this section, we describe the inferencing of infeasible path information that is exploited in our WCET calculation method. For efficient analysis, we only detect/exploit infeasible path information within a loop body, that is, we do not detect infeasible paths spanning across loop iterations. Thus, in the following, we consider the control flow graph (CFG) to be a *directed acyclic graph* (DAG), representing the body of a loop. Further, we only keep track of pairwise "conflicts" between branches/assignments, which can be Assignment-Branch (AB) conflicts or Branch-Branch (BB) conflicts.

DEFINITION 3.1. *The* effect constraint *of an assignment* $var := expression$ *is* $var == expression$. *The* effect constraint *of a branch-edge* e *in the CFG for a branch condition* c *is* c *(*$\neg c$*) if* e *denotes that the branch is taken (not taken).*

DEFINITION 3.2. *A branch-edge (or assignment)* x *has* BB (AB) conflict *with a subsequent*[1] *branch-edge* e *if and only if*

- *Conjunction of the effect constraints of* x *and* e *is unsatisfiable, and*
- *There exists at least one path from* x *to* e *in the CFG that does not modify the variables appearing in their effect constraints.*

Given a program's CFG, we compute the binary relations $BB\_Conflict$ and $AB\_Conflict$. We represent $BB\_Conflict$ between edges e' and e by the tuple (e', e), and $AB\_Conflict$ between assignment x and edge e by the tuple (u, e) where u is the basic block containing assignment x.

Restrictions. Since our definition of BB and AB conflicts captures only pairwise conflicts, we cannot detect (and exploit) arbitrary infeasible path information. For example,

$$y = 2;\ \ x = y;\ \ if(x > 3)\{...$$

denotes an infeasible path, but it will not be captured in the (restricted) notion of pairwise conflicts. Note that the right-hand-side of an assignment statement can only be a variable or an expression. To avoid the need for expensive data flow analysis, our infeasible path detection technique handles only branch conditions and assignments with constants as their right-hand-side expressions. In other words, the only conditional branches whose edges appear in our $BB\_Conflict$ and $AB\_Conflict$ relations are of the form

$$variable\ relational\_operator\ constant$$

Similarly, the only assignments which appear in $AB\_Conflict$ are of the form $variable := constant$. However, this is not a restriction on the programs we can handle; we simply ignore more complicated branches/assignments during path analysis. Moreover, we observe that this simple definition of conflict relations is sufficient for our purposes as we perform the analysis at assembly language level where each individual assignment/branch condition cannot be complex.

EXAMPLE 3.1. *Figure 1 shows a program and its corresponding CFG. Here, branch-edge* $B1 \rightarrow B2$ *and branch-edge* $B7 \rightarrow B8$ *have BB conflict; branch-edge* $B1 \rightarrow B3$ *does not have any conflict with either* $B7 \rightarrow B8$ *or* $B7 \rightarrow B9$. *Similarly, the assignment* $x = 1$ *has AB conflict with the edge* $B7 \rightarrow B9$ *but not with* $B7 \rightarrow B8$.

[1] *Subsequent* in the sense of the topological order of the control flow DAG

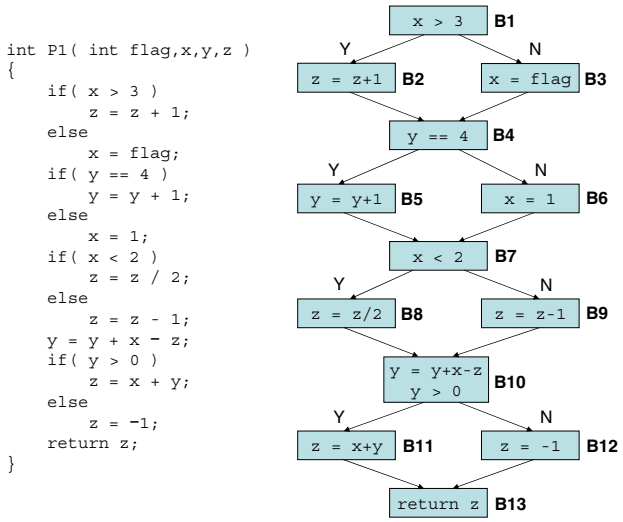

Figure 1: An example program and its CFG

Algorithm 1 describes our technique for deriving infeasible path information in a program. As shown, procedures are treated individually; thus the infeasible path information does not account for procedure call contexts or infeasibility across procedures. Within a procedure, each loop is in turn considered separately, thus not capturing infeasible paths across loops. However, we take note of assignments within the nested loop or called procedure (lines 6–7) that may cancel the effect of earlier assignments or branch-edges, so that we do not falsely identify conflicts. The method essentially takes each branch-edge $e = u \rightarrow v$ (line 8) and performs a backward breadth-first traversal of the CFG starting from node u. The traversal along a path terminates when we encounter either an assignment (conflicting or otherwise) to the variable involved in the effect constraint of e (lines 13–14) or a conflicting branch-edge (lines 16–17).

The computation of the $AB\_Conflict$ and $BB\_Conflict$ relations can be accomplished in $O((|V| + |E|) * |E|)$ time for each procedure where $|V|, |E|$ are the number of nodes and the number of edges in the CFG of the procedure. This is because each branch-edge is tested for conflict against all the ancestor nodes and branch-edges in the worst case.

4. WCET CALCULATION ALGORITHM

In this section, we present our WCET calculation algorithm that exploits the pre-computed conflict relations. In the next section, we illustrate the method with an example. The main feature of our technique is that it avoids enumeration of large number of possible execution paths, which is typical in medium to large control-intensive programs.

Algorithm 2 estimates the WCET of a program given the conflict relations. It calculates the WCET for each individual procedure (lines 1–6) and accounts for this cost at the call sites of the procedure (line 9). We assume that there is no recursive procedure call. Within a procedure, the WCET of each loop is calculated separately and nested loops are analyzed starting with the innermost loop (line 2).

To estimate the WCET of a loop, we find the heaviest *acyclic path* in the loop. An acyclic path is a possible path in a loop iteration, i.e., a path in the loop's control flow DAG from source to sink. If the estimated execution time of the heaviest acyclic path is t and the loop bound is lb, then the loop's estimated WCET is $lb * t$ (line 4 of Algorithm 2). Our algorithm traverses the loop's control flow DAG from sink to source (lines 12–20). This traversal constitutes the heart of our method. For a basic block u in the the loop, we keep $paths(u)$, a subset of possible execution paths in the subgraph rooted at u that may be part of the overall WCET path. To take into account the infeasible path information, we cannot afford to remember only the "heaviest path so far" at the control flow merge points. This is because the heaviest partial path may have conflicts with earlier branch-edges or assignment instructions resulting in costly backtracking. For each path $p \in paths(u)$ we also maintain a $conflictList$, which contains the branch-edges of p that participate in conflict with ancestor nodes and edges of u.

Now let us consider a single step backward traversal from v to u along the edge $u \rightarrow v$ (lines 14–18). We construct $paths(u)$ from partial paths in $paths(v)$ that do not have a conflict with edge $u \rightarrow v$ or an assignment in u (line 16) by adding node u at the beginning of each of these partial paths (line 17). The $conflictList$ of this extended path contains exactly the edges (a) whose conflicts have not "expired" due to assignments and (b) whose corresponding conflicting branch-edges/assignments have not been visited (line 18).

We notice that a partial path $p \in paths(u)$ has no chance of becoming the WCET path if there is another path $p' \in paths(u)$ with strictly greater cost and less potential conflict (that is, its $conflictList$ is subsumed by p's $conflictList$). In that case, p can be removed from the set (lines 19–20). This implies that if the $conflictList$ of a path $p \in paths(u)$ becomes empty and p is the heaviest path in $paths(u)$, we assign the singleton set $\{p\}$ to $paths(u)$.

In the worst case, the complexity of our algorithm is exponential in $|V|$, the number of nodes in the CFG. This is because the size of $paths(u)$ for some node u may be $O(2^{|V|})$ due to different decisions in the branches following u. In practice, this exponential blow-up is not encountered because (a) branch-edges that do not participate in any conflict are not kept track of, and (b) a branch-edge that conflicts with other branch-edges/assignments is not remembered after we encounter those conflicting branch-edges/assignments during traversal.

5. AN EXAMPLE

In this section, we illustrate our WCET calculation by employing it on the control flow DAG of Figure 1. The conflicting pairs detected are

$$BB\_Conflict = \{(B1 \rightarrow B2, B7 \rightarrow B8)\}$$
$$AB\_Conflict = \{(B6, B7 \rightarrow B9)\}$$

We traverse the DAG from sink (node $B13$) to source (node $B1$) and maintain a set of paths $paths(u)$ at each visited node u. For each path $p \in paths(u)$, we maintain $conflictList$ — a subset of branch-edges drawn from branch decisions made so far. Thus each path in $paths(u)$ is written in the form

$$\langle \text{ Sequence of basic blocks starting with } u \rangle_{conflictList}$$

Starting from node $B13$ in Figure 1, our traversal is routine till we reach node $B10$ (ϕ denotes empty set).

```
1  AB_Conflict := ∅;   BB_Conflict := ∅;
2  foreach procedure P ∈ pgm do
3  |  foreach loop L ∈ P do
4  |  |  Let G be the DAG capturing the control flow in L without the back edge;   Detect_Conflicts(G);
   |  /* Process P by treating loops and procedure calls inside P as black boxes */
5  |  Let G' be the DAG capturing the control flow in P;   Detect_Conflicts(G');

   Function Detect_Conflicts(G)
6  Replace each loop L in G by a dummy node containing all assignments occuring in L;
7  Replace each call site of procedure P ∈ G by a dummy node containing all non-local assignments in P;
8  foreach branch-edge (e = u → v) ∈ G do
9  |  Let var be the variable appearing in the effect constraint of e;
10 |  queue := ∅;   enqueue u to queue;
11 |  while queue ≠ ∅ do
12 |  |  dequeue the first node q from queue;
13 |  |  if q contains an assignment to var then
14 |  |  |  if last assignment to var in q conflicts with e then add (q, e) to AB_Conflict;
15 |  |  else
16 |  |  |  foreach predecessor p of q in G do
17 |  |  |  |  if (p → q) conflicts with e then add (p → q, e) to BB_Conflict;   else  enqueue p to queue;
```

Algorithm 1: Infeasible Path Detection in a Program *pgm*

```
1  foreach  procedure P ∈ pgm by reverse topological order in procedure call graph do
   |  /* process innermost loops first */
2  |  foreach loop L ∈ P in decreasing order of nesting depth do
3  |  |  Let G be the DAG capturing the control flow in L without the back edge;
4  |  |  L.cost := L.loopbound × WCET_Estimate(G);
   |  /* Process P by treating loops and procedure calls inside P as black boxes */
5  |  Let G' be the DAG capturing the control flow in P;
6  |  P.cost := WCET_Estimate(G');
7  Let M be the main procedure in pgm;   return M.cost;

   Function WCET_Estimate(G)
8  Replace each loop L ∈ G by a dummy node with cost L.cost and containing all assignments occuring in L;
9  Replace each call site of procedure P ∈ G by a dummy node with cost P.cost and containing all non-local assignments in P;
10 foreach node u ∈ G do visited(u) := FALSE;
11 paths(G.sink) := {⟨G.sink⟩};   ⟨G.sink⟩.conflictList := ∅;   visited(G.sink) := TRUE;
   /* Traverse from sink to source */
12 foreach node u ∈ (G − G.sink) in reverse topologically sorted order do
13 |  visited(u) := TRUE;   paths(u) := ∅;
14 |  foreach immediate successor v of u do
15 |  |  foreach partial path p in paths(v) do
   |  |  |  /* Augment the partial path p with node u if there is no conflict */
16 |  |  |  if ∄e ∈ p.conflictList s.t. ((u → v, e) ∈ BB_Conflict ∨ (u, e) ∈ AB_Conflict) then
17 |  |  |  |  p' := ⟨u⟩ ∘ p;   p'.cost := p.cost + u.cost;  paths(u) := paths(u) ∪ {p'};
   |  |  |  |  /* Augment conflictList while removing expired elements */
18 |  |  |  |  p'.conflictList := p.conflictList ∪ {u → v | u → v appears in AB_Conflict or BB_Conflict}
   |  |  |  |  − {e | e ∈ p.conflictList and u contains an assignment to the variable appearing in e's effect constraint }
   |  |  |  |  − {e | e ∈ p.conflictList and ∄x s.t. ¬visited(x) ∧ ((x, e) ∈ AB_Conflict ∨ (x → y, e) ∈ BB_Conflict)};
   |  /* remove partial paths that clearly cannot lead to the WCET path */
19 |  foreach partial path p ∈ paths(u) do
20 |  |  if ∃p' ∈ paths(u) s.t. p'.conflictList ⊆ p.conflictList ⋀ p'.cost > p.cost then  paths(u) := paths(u) − {p};
21 Let p ∈ paths(G.source) be the path with maximum cost;   return p.cost;
```

Algorithm 2: Estimating WCET of a program *pgm* given infeasible path information

Table 1: Efficiency of our WCET calculation method

Benchmark	# Basic	# Conflicts			# Paths			Runtime	Stappert's Method	
	Blocks	AB	BB	Traced	Total	Feasible	Traced	(ms)	# Expl. Paths	Runtime
`adpcm`	32	2	5	2	1,536	288	2	0.20	5	0.42 ms
`display`	37	13	0	2	96	42	2	0.21	7	0.61 ms
`statemate`	334	74	15	15	6.55×10^{16}	1.09×10^{13}	738	853.52	> 2000	> 36 mins
`susan_thin`	93	15	13	3	146,189,962	33,820	12	1.06	> 2000	> 24 mins
`compress`	213	9	3	2	110	45	4	1.18	27	3.72 ms

$$paths(B13) = \{\langle B13\rangle_\phi\}$$
$$paths(B12) = \{\langle B12, B13\rangle_\phi\}$$
$$paths(B11) = \{\langle B11, B13\rangle_\phi\}$$

At node $B10$, we have two potential paths. However, all branch-edges in these paths, $B10 \rightarrow B11$ and $B10 \rightarrow B12$, do not participate in any conflict relation, hence both paths have empty $conflictList$. Therefore, we only carry the heaviest of the two paths (assuming $B11.cost \geq B12.cost$).

$$paths(B10) = \{\langle B10, B11, B13\rangle_\phi\}$$
$$paths(B9) = \{\langle B9, B10, B11, B13\rangle_\phi\}$$
$$paths(B8) = \{\langle B8, B10, B11, B13\rangle_\phi\}$$

Node $B7$ again has two potential paths, and both of its outgoing edges appear in conflict relations. Until we visit the corresponding conflicting edges or nodes, we cannot determine the feasibility of the partial paths. Consequently, we maintain both paths along with the potentially conflicting edges in the set $conflictList$ associated with each path.

$$paths(B7) = \{ \quad \langle B7, B8, B10, B11, B13\rangle_{\{B7 \rightarrow B8\}}, \quad \langle B7, B9, B10, B11, B13\rangle_{\{B7 \rightarrow B9\}} \ \}$$

Moving on to node $B6$, we find that the assignment in node $B6$ conflicts with $B7 \rightarrow B9$ rendering the path

$$\langle B6, B7, B9, B10, B11, B13\rangle$$

infeasible. Thus we only extend one path leading to

$$paths(B6) = \{\langle B6, B7, B8, B10, B11, B13\rangle_\phi\}$$

We drop $B7 \rightarrow B8$ from the $conflictList$ as we have encountered an assignment to program variable x in $B6$. The assignment implies that the conflict between $B7 \rightarrow B8$ and $B1 \rightarrow B2$ has "expired" along this partial path.

Indeed, these last two steps show the key source of efficiency in our method. Since we have kept track of both possibilities in which branch at node $B7$ can be resolved, we do not need to backtrack when we find that the branch decision $B7 \rightarrow B9$ can lead to infeasibility. Also, we do not store paths corresponding to the decision of every branch, but only those involved in conflicts. Furthermore, once we have encountered an assignment to the variable involved in a conflict, we need not keep track of that conflict any further.

Continuing in this way we reach node $B1$; we omit the details for the rest of the traversal. Note that the control flow DAG of Figure 1 has four branches and $2^4 = 16$ paths. However, when we visit any basic block u of the control flow DAG, $paths(u)$ contains *at most two paths* (*i.e.*, exponential blow-up is avoided in this example).

6. EXPERIMENTS

We apply the WCET estimation method on several benchmark programs. `adpcm` is the ADPCM coder taken from Mediabench [9]. `display` is an image dithering kernel taken from MPEG-2. `compress` is a data compression program, while `statemate` is a car window controller automatically generated from a statechart specification; both are taken from C-Lab [17]. `susan_thin` from MiBench's automotive application suite is a kernel performing edge thinning [6].

We use SimpleScalar toolset [3] for the experiments. The programs are compiled using gcc 2.7.2.3 targeted for SimpleScalar. As our focus is on infeasible program paths, we assume a simple embedded processor with single-issue in-order pipeline, perfect instruction cache and branch prediction. The execution time corresponding to each basic block is thus easily estimated for this simple architecture. For more complex micro-architectures, we can use state-of-the-art WCET estimation tools such as aiT [1]. We assume that loop bounds required for WCET calculation are provided via manual annotation. All experiments are performed on 3.0GHz P4 CPU with 1MB cache and 2GB memory.

The total number of basic blocks, BB conflicts and AB conflicts detected for each benchmark are shown in Table 1. The column *# Conflicts Traced* gives the maximum length of $conflictList$ maintained during computation, which is very few in all cases. The next two columns give the total number of paths and the number of feasible paths for each benchmark. `statemate` and `susan_thin` in particular have huge numbers of infeasible paths, despite the limited conflict detection applied. The `statemate` code, being automatically generated from a statechart, contains a lot of repetitive checks which give rise to many infeasible paths. `susan_thin` applies different computations based on a single value that is checked at multiple points; thus the entrance of a computation block renders all partial paths within the other computation blocks infeasible. A general observation is that unoptimized programming practices contribute substantially to infeasible program paths.

The column *# Paths Traced* in Table 1 shows the maximum number of paths maintained by our technique at any point of time. It is encouraging to note that we only need to keep track of at most 738 partial paths at any time for our benchmarks. This figure depends heavily on the number of conflicting pairs and the distance between the pairwise conflicts. Most of the conflicting pairs are localized; thus they expire quickly and need not be kept track of further. In `statemate`, some conflicting pairs have long "conflict windows", that is, the assignment/branch conditions of a conflicting pair appear far apart in the CFG; this makes it necessary to maintain more partial paths at each node. The number of paths maintained in turn affects the runtime of the algorithm. The *Runtime* column in Table 1 shows that our technique requires less than 1 second for any benchmark. Even for programs with long "conflict windows", our algorithm performs far better than maintaining a single heaviest path throughout the CFG traversal (and backtracking when this path turns out to be infeasible).

Table 2: Comparison of observed WCET, WCET estimation with and without infeasibility information

Benchmark	Est. WCET (cycles)		Improvement	Obs. WCET	Estimated/Observed	
	with infeas.	w/o infeas.		(cycles)	with infeas.	w/o infeas.
`adpcm`	896,286	907,286	1.21%	717,201	1.25	1.27
`display`	244,187,271	257,556,615	5.19%	229,755,271	1.06	1.12
`statemate`	41,578	44,938	7.48%	31,636	1.31	1.42
`susan_thin`	293,989,241	485,328,185	39.42%	173,769,229	1.69	2.79
`compress`	312,904	383,329	18.37%	25,819	12.12	14.85

We compare the performance of our method with Stappert's approach [15] which iteratively searches for a longest path, tests its feasibility, and removes the path if it is infeasible. We provide both methods with the same execution time for basic blocks and infeasibility information (see Section 3); thus both yield the same WCET path for each benchmark. The last two columns of Table 1 give the number of paths examined by Stappert's method and the runtime of the method. We observe that the huge number of infeasible paths in `statemate` and `susan_thin` are the heaviest paths as well. So, in these two benchmarks, we have to terminate the run of Stappert's algorithm after examining 2000 paths without finding a feasible path. The overestimation of the last examined infeasible path (by Stappert's method) compared to the feasible WCET value obtained by our approach is as much as 60% for `susan_thin`.

Table 2 gives the results of our WCET estimation algorithm on the benchmarks, taking into account infeasibility information. These are compared with the results of WCET estimation on the same benchmarks by assuming all paths are feasible. The *Improvement* column shows the reduction in the estimated WCET value when infeasibility is considered. As expected, the WCET estimation yields a tighter value when infeasibility is taken into account. Finally, the column *Estimated/Observed* of Table 2 shows the ratio of the estimated WCET values, with and without infeasibility detection, to the observed WCET values obtained via simulation: the closer the ratio to 1, the tighter the estimation. Our analysis gives tight estimates in all the benchmarks except `compress`; the result for compress can only be improved by considering infeasibility across loop iterations.

7. DISCUSSION

In this paper, we have proposed and evaluated a method for estimating the WCET of a program. We accurately estimate the WCET by taking into account (limited) infeasible path information without resorting to exhaustive path enumeration. The utility of our technique has been demonstrated on a number of benchmarks.

We envision that our approach will be useful for WCET analysis of control-intensive applications. Oftentimes the program code corresponding to these applications are automatically generated from high-level specifications, such as Statecharts. Such controller applications from automotive industry (see `statemate` in Table 1) often have very large number of possible execution paths. Our approach maintains only a handful of these paths and completes WCET calculation in less than 1 second.

Acknowledgments

This work was supported by NUS project R252-000-171-112 and an ICITI project.

8. REFERENCES

[1] AbsInt. aiT: Worst case execution time analyzer, 2004. `http://www.absint.com/ait/`.

[2] P. Altenbernd. On the false path problem in hard real-time programs. In *ECRTS*, 1996.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2), 2002.

[4] A. Ermedahl and J. Engblom. Modeling complex flows for worst-case execution time analysis. In *RTSS*, 2000.

[5] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *EUROPAR*, 1997.

[6] M. R. Guthaus et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE Workshop on Workload Characterization*, 2001.

[7] C. Healy and D. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Trans. on Software Engineering*, 28(8), 2002.

[8] T. Henzinger, R. Jhala, R. Majumder, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[9] C. Lee et al. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *MICRO*, 1997.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES*, 1995.

[11] T. Lundqvist and P. Stenstrom. Integrating path and timing analysis using instruction-level simulation techniques. In *LCTES*, 1998.

[12] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real Time Systems*, 5(1):31–62, 1993.

[13] P. Puschner and A. Schedl. Computing maximum task execution times - a graph based approach. *Real Time Systems*, 13(1), 1997.

[14] A. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 1(2), 1989.

[15] F. Stappert, A. Ermedahl, and J. Engblom. Efficient longest executable path search for programs with complex flows and pipeline effects. In *CASES*, 2001.

[16] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS*, 2005.

[17] WCET benchmarks, 2004. `http://www.c-lab.de/home/en/download.html`.

[18] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET code positioning. In *RTSS*, 2004.