View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

Exploiting Branch Constraints without Exhaustive Path Enumeration

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

Abstract

Statically estimating the worst case execution time (WCET) of a program is important for real-time software. This is difficult even in the programming language level 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 of a loop without resorting to exhaustive path enumeration. The efficiency of our approach is demonstrated with a real-life control-intensive program.

1. Introduction

Static analysis of a program for obtaining the Worst Case Execution Time (WCET) is important for hard real-time embedded systems. WCET analysis typically consists of three phases: *flow analysis* to identify loop bounds and infeasible flows through the program; *architectural modeling* to determine the effect of pipeline, cache, branch prediction etc. on the execution time; and finally *estimation* to find an upper bound on the WCET of the program given the results of the flow analysis and the architectural modeling. In this paper, we concentrate on the estimation problem.

There exist mainly three different approaches for WCET estimation: *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") [5]. This method is quite simple and efficient. But it has limitations in exploiting the results returned by flow analysis. In particular, it is difficult to exploit infeasible paths due to branch constraints (dependencies among branch statements) in this approach as the timing rules are local to a program statement. Implicit path enumeration techniques (IPET) [4] 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. Attempts have been made to integrate special flow information in IPET [2]. However, the kind of flow information that can be handled by IPET is inherently limited. This is because the usual ILP formulation introduces formal variables for the execution counts of the nodes and edges in the Control Flow Graph (CFG) of the program. Since the variables denote aggregate execution counts of basic blocks, it is not possible to express certain infeasible path patterns (typically denoting a *sequence* of basic blocks) as constraints on these variables.

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. Thus, path-based techniques can naturally handle the various flow information. Healy et al., in particular, detect and exploit branch constraints within the framework of path-based technique [3]. Originally, path-based techniques were limited to a single loop iteration. However, Stappert et al. [6] have extended it to complex programs with the help of *scope graphs* and *virtual scopes*.

One of the main drawbacks of path-based techniques is that they require the generation of all the paths. In the worst case, this can lead to 2^n paths where n is the number of decisions in the program fragment. In control-intensive programs, we have encountered up to 6.55×10^{16} paths in a single loop iteration (see Table 1). Research by Stappert et al. [6] has sought to avoid this expensive path enumeration by finding (a) the longest program path , (b) checking for the feasibility of , and (c) removing from CFG followed by the search for a new longest path if 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 program paths. Indeed, for our benchmark only a small fraction (less than 0.1%) of the paths are feasible, making this approach quite costly (see Table 1).

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.

ECRTS 2005
5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2007/816

2. WCET Estimation Algorithm

In this section, we present a method for finding the WCET path of a single loop. Once this is obtained, the WCET path of the program can be obtained by composing the WCET paths of individual loops through the well-known timing schema approach [5]. Of course, this will mean that infeasible path information across loops cannot be taken into account. We assume that state-of-the-art WCET analyzers such as aiT[1] can be used to estimate the worst-case execution time of each basic block.

Given a fragment of assembly code corresponding to a loop in a source program, we first construct the directed acyclic graph (DAG) capturing the control flow in the loop body (i.e., the CFG of the loop body without the loop back-edge). We assume that the DAG has a unique source node and a unique sink node. If there is no unique sink node, then we add a dummy sink node. Each path from the source to the sink in the DAG is an **acyclic path** — a possible path in a loop iteration. Our algorithm finds the worst case execution path for a single iteration of the loop, i.e., we find the heaviest acyclic path. 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.

We find the heaviest acyclic path accurately by taking into account infeasible path information and yet we avoid enumerating all the acyclic paths in a loop. Clearly, the infeasible path information that we work with may not always be complete; so the accuracy of our heaviest acyclic path detection depends on the accuracy of the infeasible path information. First, we discuss the infeasible path information used and then explain how it is efficiently exploited in our WCET calculation.

2.1. Infeasible path information

Our infeasible path information consists of two binary relations capturing conflicting pairs of branches/assignments: *AB_conflict* and *BB_conflict*. The relation *AB_Conflict* is a set of (assignment, branch-edge) pairs, that is, if a, e $AB_Conflict$ then a is an assignment instruction and e is an outgoing edge from a conditional branch instruction; on the other hand, the relation $BB_Conflict$ is a set of (branch-edge, branch-edge) pairs.

We do not detect conflicts between arbitrary branches and assignments to avoid an inefficient conflict detection procedure. 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 $variable := constant$. For such assignments and branches we can define pair-wise conflict in a natural way (see [3] for a full discussion). For example, $x := 2$ conflicts with $x > 3$, but not with $x < 3$; similarly $x > 3$ conflicts with $x < 2$ but not with $x > 5$. Now, for such restricted branches and assignments, we put an assignment a conflicting with a conditional branch-edge e into the $AB_Conflict$ relation (i.e., a, e $AB_Conflict$) iff there exists at least one path from a to e which does not contain assignments to the common variable appearing in a, e. Similarly, we put two conflicting conditional branch-edges e , e into the $BB_Conflict$ relation (i.e., e , e $BB_Conflict$) iff there exists at least one path from e to e which does not contain assignments to the common variable appearing in e , e. If e , e $BB_Conflict$, there may be another path from e to e that contains an assignment to the common variable appearing in e , e. Such paths should not be considered as infeasible paths.

The computation of the $AB_Conflict$ and $BB_Conflict$ relations can be accomplished in $O((|V| + |E|)$ $|E|)$ time where $|V|, |E|$ are the number of nodes and the number of edges in the control flow DAG; this is because for each branch-edge we need to perform a depth-first like search to find conflicting branch-edges and/or assignments.

Example: A loop-free program fragment (which can be the body of a loop) and its control flow DAG are shown in Figure 1. In this example, the relation $AB_Conflict$ contains only one pair – the assignment at basic block $B6$ (which sets x to 1) and the branch-edge $B7$ $B9$ (which stands for x 2). The relation $BB_Conflict$ contains the branch-edge pair $B1$ $B2, B7$ $B8$ that captures the conditions $x > 3$ and $x < 2$.

2.2. WCET calculation

We now present our WCET estimation algorithm for finding the heaviest feasible path in an iteration of a loop. We *do not* enumerate the possible paths in an iteration and then find the heaviest. At the same time, we do not consider all paths in the loop's control flow graph to be feasible – we consider the infeasible path information captured by $AB_Conflict$ and $BB_Conflict$ relations.

Our algorithm traverses the loop's control flow DAG from sink to source. However, to take into account the infeasible path information, we cannot afford to remember only the "heaviest path so far" as we traverse the DAG. This is because the heaviest path may have conflicts with earlier branch-edges or assignment instructions resulting in costly backtracking. Instead, at a basic block v, we maintain a set of paths $paths(v)$ where each p $paths(v)$ is a path from v to the sink node. $paths(v)$ contains only those paths which when extended up to the source node can potentially become the WCET path. For each path p $paths(v)$ we also maintain a "*conflict list*". The conflict list contains

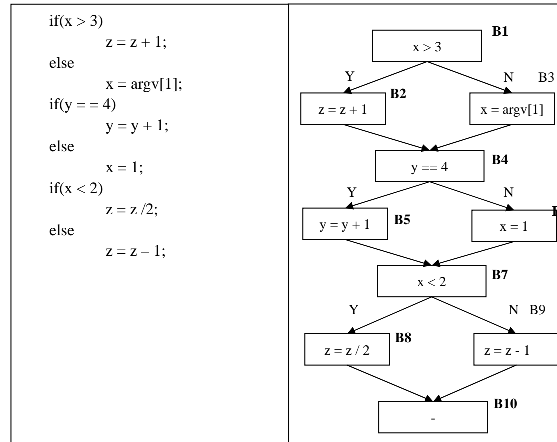

Figure 1: *A loop body with its Control Flow Graph*

the branch-edges of p that participate in conflict with ancestor nodes and edges of v.

During our traversal from sink to source, consider a single step traversal from v to u along the edge $u \to v$ in the control flow DAG. We first construct $paths(u)$ from $paths(v)$ by adding the edge $u \to v$ at the beginning of the paths in $paths(v)$. Also, for each path $p \in paths(u)$ we update its conflict list to contain exactly those edges in p which have conflicts with branch-edges/assignments "prior to" u (in topological order). At this stage we may add the branch-edge $u \to v$ to p's conflict list or we may remove an edge e from p's conflict list if all branch-edges/assignments conflicting with e appear "after" u (in the topological order). If as a result, we have identical conflict lists for two paths $p, p' \in paths(u)$, then we maintain the heavier path among p and p'. Finally, if the conflict list 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 $paths(u)$. Details of the algorithm are omitted for space considerations.

In the worst case, the complexity of our algorithm is exponential in $|V|$, the number of nodes in the loop's control flow DAG. This is because the number of paths in $paths(v)$ for some block v may be $O(2^{|V|})$ due to different decisions in the branches following v. In practice, this exponential blow-up is not encountered because (a) branch-edges which do not conflict with any assignment/branch-edge do not need to be kept track of, and (b) a branch-edge which conflicts with other branch-edges/assignments need not be remembered after we encounter those conflicting branch-edges/assignments during the traversal.

Illustration We demonstrate our WCET calculation method by employing it on the control flow DAG of Figure 1. As mentioned, we traverse the DAG from sink to source and maintain a set of paths $paths(v)$ at each visited node v. For each path $p \in paths(v)$ we also maintain the *conflict list*, a set of branch decisions drawn from branch decisions made so far. Thus each path p in $paths(v)$ is written in the form $p_{conflict\ list}$.

Starting from node $B10$ in Figure 1, our traversal is routine till we reach node $B7$ ($\emptyset$ denotes empty set).

$$\begin{aligned} paths(B10) &= \{\langle B10\rangle_\emptyset\} \\ paths(B9) &= \{\langle B9, B10\rangle_\emptyset\} \\ paths(B8) &= \{\langle B8, B10\rangle_\emptyset\} \end{aligned}$$

The outgoing edges from node $B7$ appear in conflict relations capturing infeasible path information. Consequently, our method maintains two paths in $paths(B7)$ — the heaviest path starting with $B7 \to B8$ and the heaviest path starting with $B7 \to B9$.

$$paths(B7) = \{\langle B7, B8, B10\rangle_{\{B7 \to B8\}}, \langle B7, B9, B10\rangle_{\{B7 \to B9\}}\}$$

Now, from node $B7$ we traverse to nodes $B5$ and $B6$. The assignment in node $B6$ conflicts with $B7 \to B9$. Therefore, we do not consider any path in $paths(B7)$ which contains $B7 \to B9$ in its conflict list. This is how infeasible path information is accounted for in our WCET calculation. Thus we have

$$paths(B6) = \{\langle B6, B7, B8, B10\rangle_\emptyset\}$$

We drop $B7 \to B8$ from the conflict list of $\langle B6, B7, B8, B10\rangle$ as we have encountered an assignment to program variable x in $B6$. The assignment implies that the conflict between $B7 \to B8$ and $B1 \to B2$ does not hold along any extension of the partial path $\langle B6, B7, B8, B10\rangle$.

At node $B5$, we first add $B5$ to the two partial paths from $B7$. Then, we notice that the edge $B7 \to B9$ is involved only in a conflict with $B6$ and we have already traversed $B6$. Therefore, we can drop this edge from the conflict list of the partial path $\langle B5, B7, B9, B10\rangle$ and this path now becomes completely conflict free. Assuming that

Function	Basic Blocks	Total Paths	Feasible paths	BB-Conflicts	AB-Conflicts	Enumerated Paths
statemate	334	6.55x10^{16}	1.09x10^{13}	74	15	121,831
statemate1	44	19,440	7,440	6	0	15
statemate2	73	902	36	26	0	14
statemate3	161	1,459,364	69,867	15	0	40
statemate4	17	10	10	0	0	1
statemate5	43	256	58	2	0	4

Table 1: *Efficiency of our WCET calculation method.*

Function	WCET Estimation (cycles)	
	w/o infeasibility	with infeasibility
statemate	44,800	41,520
statemate1	29,400	28,960
statemate2	2,750	2,270
statemate3	7,300	7,000
statemate4	1,070	1,070
statemate5	2,370	2,090

Table 2: *Accuracy of WCET estimation with and without considering infeasibility.*

$B5, B7, B9, B10$ is heavier than $B5, B7, B8, B10$, we have

$$paths(B5) = \{ B5, B7, B9, B10 \}$$

On the other hand, if $B5, B7, B8, B10$ is heavier, we still need to maintain both the partial paths (as the heavier path may become infeasible later). 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 three branches and $2^3 = 8$ paths. However, when we visit any basic block v of the control flow DAG, $paths(v)$ contains at most two paths (*i.e.*, the exponential blow-up is avoided here in practice).

3. Experiments

In this section, we present preliminary experiment results for the proposed method. The benchmark used in our experiment is a car window lift control program taken from the C-Lab benchmark suite. It is automatically generated by the Statemate tool based on a state-chart specification. We report results for the entire program (`statemate`) as well as program fragments corresponding to all the five functions (`statemate1` to `statemate5`).

An enumeration-based WCET estimation method typically examines each possible path, filters out the infeasible paths and selects the feasible path with the maximum execution time. Table 1 shows that the total number of paths through a single iteration of the loop body can be quite large (6.55×10^{16} possible paths out of which at most 1.09×10^{13} are feasible for `statemate`). In fact, a naive WCET calculation method, which enumerates all these possible paths for one loop iteration and chooses the longest feasible one, runs out of memory even on a PC with 1 GB main memory. The column *Enumerated Paths* shows the maximum number of paths that need to be maintained by our estimation technique at any point of time. The results are quite encouraging; even for the entire `statemate` program, we only need to keep at most $121,831$ paths at any point of time during the estimation. As a result, our estimating technique requires less than 1 minute for the entire `statemate` program on a Pentium4 1.7Ghz platform with 1GB memory. Finally, Table 2 shows that, as expected, our method produces more accurate estimation compared to a method that does not take infeasibility information into account. The only exception is `statemate4`, which does not have any infeasible path.

4. Discussion

In this paper, we have reported preliminary results on exploiting (limited) infeasible path information during WCET estimation of a loop without resorting to path enumeration. The efficacy of our technique has been demonstrated on a substantial real-life car window control benchmark. In near future, we will develop WCET estimation methods that can take into account infeasible path patterns of arbitrary length without compromising efficiency.

References

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

[2] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2003.

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

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

[5] C. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-time Systems*, 5(1), 1993.

[6] F. Stappert, A. Ermedhal, and J. Engblom. Efficient longest execution path search for programs with compelx flows and pipeline effects. In *CASES*, 2001.