Analyzing Loop Paths for Execution Time Estimation

Abhik Roychoudhury, Tulika Mitra, and Hemendra Singh Negi

School of Computing, National University of Singapore
{abhik,tulika,hemendra}@comp.nus.edu.sg

Abstract. Statically estimating the worst case execution time of a program is important for real-time embedded software. This is difficult even in the programming language level due to the inherent difficulty in detecting infeasible paths in a program's control flow graph. In this paper, we study the problem of accurately bounding the execution time of a program loop. This involves infeasible path detection followed by timing analysis. We employ constraint propagation methods to detect infeasible paths spanning across loop iterations. Our timing analysis is exact modulo the infeasible path information provided. Moreover, the analysis is efficient since it relies on memoization techniques to avoid exhaustive enumeration of all paths through a loop. The precision of our timing analysis is demonstrated on different benchmark programs.

1 Introduction

Statically analyzing the *worst-case execution time* (WCET) of a program is important for real-time embedded software. An embedded system contains processor(s) running specific application programs which communicate with an external environment in a timely fashion. These application programs thus have real-time requirements, that is, there are hard deadlines on the execution time of such software. Therefore, it is important to perform static analysis of embedded software to guarantee the satisfiability of all timing constraints. One of the prominent uses of the WCET estimate of a program is in schedulability analysis.

Due to its inherent importance in embedded system design, timing analysis of embedded software has been extensively studied [6, 8, 12, 13, 17, 18, 20]. Usually this involves (a) a programming language level path analysis to find out infeasible paths in the program's control flow graph, and (b) micro-architectural modeling to take into account the effect of performance enhancing architectural features (such as pipeline, cache and branch prediction). In this paper, we only concentrate on path analysis. Program path analysis for WCET estimation involves solving two related problems (a) detecting infeasible paths and (b) using infeasible path information for timing calculation.

Concretely, the contributions of this paper can be summarized as follows.

– We design and implement an infeasible path detection method based on constraint propagation via weakest pre-condition calculation. The infeasible paths detected by our method can be exploited for WCET analysis as well as other purposes (like reducing test suite sizes, software model checking etc.)

– We provide a programming language level timing analysis algorithm for finding the WCET of a program loop (which is bounded). The algorithm is exact modulo the infeasible path information provided (via our infeasible path detection method). In other words, we can find the longest feasible path through a program loop if the infeasible path information provided is exact. In particular if the detected infeasible path patterns span across at most K loop iterations, we construct a transition system whose nodes denote paths taken in $K-1$ consecutive iterations. This allows us to ensure that no path in the transition system contains any of the infeasible path patterns detected in the first phase; so we can efficiently find the longest path through the program loop. Our technique has been implemented and we show its utility via experimental results on various programs.

We note that different WCET analysis techniques combine the results of path analysis and micro-architectural modeling in different ways. Many of these advocate a *separated* approach (*e.g.* [20]) where the micro-architectural modeling performs a categorization of the program's instructions and this categorization information is fed into path based timing estimation. The WCET analysis technique presented in this paper can also be extended in this fashion; that is, we can augment it to take into account categorization of program instructions based on micro-architectural modeling.

2 Related Work

One of the earliest works on programming language level timing analysis is the *timing schema* approach [18]. 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. Another early work by Puschner and Koza [17] studied the conditions for decidability of WCET analysis and provided some rules for WCET analysis.

Techniques to extend the timing schema approach with infeasible path information have been reported in [15]. In this work, the infeasible path patterns are user-provided, that is, the technique only performs path analysis and not infeasible path detection. Lundqvist and Stenstrom [14] provide an instruction level simulation approach for detection and elimination of infeasible paths. Ermedahl and Gustafson [7] present a static analysis method 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. Altenbernd [1] searches for infeasible paths in a control flow graph via branch-and-bound search.

The components of our WCET analysis mechanism are probably most related to the infeasible path detection technique of [3] and the path analysis technique of [10]. The key idea in [10] is to compute the effect of any assignment or a branch on other branch outcomes; if the effect of an assignment a is to force the outcome of branch b to true, then a path from a to b with the outcome of b being false is an infeasible path. This is certainly a clever and effective way of detecting many commonly occurring infeasible path patterns. However, we note that since our approach is based on constraint propagation, we do not rely on capturing relationship between individual pairs of branches. In general, the outcome of a program branch may be correlated to the outcome of *several* previous branches.

We also note that our constraint propagation methods differs substantially from the propagation method of [3]. This work relies on inferring simple invariant properties (which hold for all visits to a specific control location) in order to detect infeasible paths in the control flow graph. The propagation is stopped at basic block b if the propagated constraint c at basic block b can be be proved to hold for *all* executions of basic block b. Note that if this condition holds then we have found an infeasible path: a path from n to b that cannot make the branch constraint of b false when b is reached. [3] uses some simple sufficient conditions to check whether a constraint c holds for all visits to basic block n (such as n containing an assignment statement whose effect constraint implies c).

```
1  sum = 0;
2  for (j=1; j<= limit; j++) {
3      if (j % 2 == 0) {
4                        sum +=j;}
5  }
6  return sum;
```

Fig. 1. Sum of even numbers

To see the difficulties of the approach of [3], let us consider the program in Figure 1 (taken from [2]) which adds up even numbers. Suppose we want to find out the infeasible paths ending in the branch at line 3. A backward propagation of the branch constraint will revisit the branch on line 3 (the previous iterations). In fact, if we start at the branch on line 3 with the constraint *j is even*, we will propagate this constraint backwards and visit the branch at line 3 with the constraint *j is odd*. Note that in this program, the strongest invariant on j that holds for *all* executions of line 3 is $1 \leq j \leq limit$. From this constraint it is not possible to infer that line 4 cannot be executed/skipped in consecutive iterations (which says that both j and $j+1$ cannot be even/odd). Hence the infeasible path detection technique of [3] will fail to infer this information. In the next section, we will demonstrate how this information can be inferred in our infeasible path detection method.

Finally, we note that our infeasible path detection technique is inspired by the recent progress in abstraction refinement based software model checking of invariants(*e.g.* see [11]). These works search through an abstract model of the program to generate a counter-example trace and then show that the given counter-example trace is an infeasible path in the program's control flow graph.

The proof of infeasibility can be done via a backward (or forward) constraint propagation *along the counter-example trace*. In our work, we start the propagation from a program branch and backwards propagate the branch constraint to *all* paths leading into the branch. Consequently we need to consider issues like termination/speed-up of propagation. These issues are not so important in checking of counter-examples where the propagation is restricted to one finite (and typically short) counter-example trace.

3 Detecting Infeasible Paths

In this section we concentrate on the problem of *detecting* infeasible paths. First we define the notion of an execution trace.

Definition 1 (Execution Trace) *Given a program P with an initial control location l_{start} and feasible inputs drawn from a (potentially infinite) set I, an execution trace of the program is the sequence of basic blocks traversed by starting from l_{start} with some input $i \in I$.*

In practice, we are always dealing with programs where the length of every execution trace is bounded, i.e., the loops are bounded. Indeed for timing analysis of programs, we cannot work with programs having unbounded loops. Hence we consider programs with bounded execution traces. In the rest of this paper, whenever we refer to an infeasible path, we mean the following.

Definition 2 (Infeasible Path) *Given a program P with feasible inputs drawn from a (potentially infinite) set I, an infeasible path π is a finite sequence of basic blocks which does not appear in any execution trace of P (*i.e. *π is not contained in the execution trace of P for any input $i \in I$).*

Our approach for infeasible path detection is based on constraint propagation. In general, to detect infeasible paths ending at a given branch b, we need to propagate backwards the constraint of b to all its immediate predecessors (who in turn propagate it to their immediate predecessors and so on). This essentially amounts to weakest pre-condition computation along the various paths coming into b [5]. In other words, let $\varphi_b(\overline{X})$ be the branch constraint for b where $\overline{X}$ denotes the program data variables. Let $stmt_1$ and $stmt_2$ be two statements which may be executed immediately before branch b. We can capture the effect of any program statement as a constraint relating the program variable values before and after the execution of the statement.[1] Let the effect constraint of $stmt_1$ and $stmt_2$ be $\psi_1(\overline{X}, \overline{X}')$ and $\psi_2(\overline{X}, \overline{X}')$ respectively, where $\overline{X}'$ denotes the values of $\overline{X}$ after the statement execution. Then one step of the weakest

[1] For example, the assignment statement `x:= x+1` can thus be represented as $x' = x + 1 \wedge \forall y \in \overline{X} - \{x\}\ y' = y$ where the primed variables denote the value of the corresponding program variables after the statement is executed. Effect of branch statements can also be captured as a constraint representing the branch condition.

pre-condition computation (for computing infeasible paths ending at branch b involves computing

$$wp_i(\overline{X}) \overset{\text{def}}{=} \forall \overline{X}' \; \psi_i(\overline{X}, \overline{X}') \Rightarrow \varphi_b(\overline{X}') \quad i = 1, 2$$

for the two incoming edges from $stmt_1$ and $stmt_2$ into branch b in the control flow graph.

Clearly such a constraint propagation based approach can detect whether the outcome of a branch b can be deduced from the constraints for several other branches. Termination of the propagation is guaranteed since we only consider bounded loops. However, we still face the practical problem of the constraint propagation amounting to an exhaustive enumeration of paths ending at a branch b. Thus, we need to incorporate mechanisms for speeding up the constraint propagation. In the following, we give our technique for infeasible path detection and illustrate it via an example.

3.1 Technique

We now elaborate our technique for detecting infeasible paths. For simplicity of exposition, let us first consider a single program loop. Let us consider a bounded loop L with k branches inside the loop. Depending on the structure of the control flow within L, the possible number of paths within each iteration can vary from $k + 1$ to 2^k (not all of these paths may be feasible though). To find the infeasible path patterns which (potentially) span across iterations, we first define k propositions $p_1, \ldots, p_k$ corresponding to the conditions in the k branches inside the loop. Let us suppose that the basic blocks which capture control flow within the loop are $B_1, \ldots, B_n$. Then, the infeasible paths detected will be sequences over the alphabet $\{B_1, \ldots, B_n\}$.

Our constraint propagation algorithm proceeds by backwards traversal. Each visit of a basic block B_i is annotated with

- a constraint c_i over the program variables $\overline{X}$.
- a boolean formula b_i over $p_1, \ldots, p_k$.

The constraint propagation terminates if B_i was earlier visited with the same boolean formula b_i, or if c_i is unsatisfiable. If the constraint propagation does not terminate at this visit of B_i, then for each immediate predecessor B_{i_j} of B_i we do the following.

- the constraint c_{i_j} of B_{i_j} is computed by a weakest pre-condition of c_i w.r.t. the statements in B_{i_j}.
- the boolean formula b_{i_j} is the strongest boolean formula over $p_1, \ldots, p_k$ which is implied by c_{i_j}.

Thus, b_{i_j} and c_{i_j} become the annotations of the corresponding visit of B_{i_j}.

We can see that the annotations b_i and c_i for a visit of a basic block B_i serve two different purposes. The boolean formula b_i serves as an approximation of the constraint store c_i. Since the number of distinct boolean formula over a fixed finite set of atomic propositions is bounded, this ensures that the number of visits to any basic block is bounded (thereby ensuring termination).[2] The check for unsatisfiability of c_i allows us to terminate the detection along certain paths earlier. In other words, we maintain the concrete constraint store c_i to accurately detect infeasible paths. We also maintain b_i, a boolean abstraction of the constraint store c_i, to guarantee termination of constraint propagation.

So far we have outlined the termination condition and each step of the constraint propagation. We have not specified the initial condition. In practice, we run the constraint propagation algorithm $2k$ times, corresponding to the true and false outcomes of the k branches within the loop. This will find out all infeasible paths terminating at any of k branches.

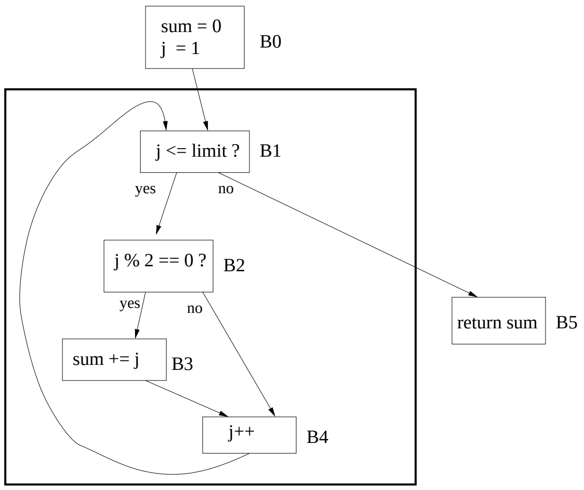

Fig. 2. Control Flow Graph for Example Program in Figure 1

Extensions In the above, we described a method for detecting infeasible paths within a single loop. However, the constraint propagation mechanism in the method is generic, and can analyze arbitrary nestings and sequences of loops. We will then need to run the constraint propagation for all program branches which are not loop branches. We note that our current implementation performs infeasible path detection for each loop separately. This is not due to a limitation of our infeasible path detection technique; rather this is because of the fact that our WCET analysis method analyzes each program loop separately. So even if

[2] One can use a canonical representation of boolean functions such as reduced ordered Binary Decision Diagrams to detect whether a basic block was previously visited with the same boolean formula b_i.

we detect infeasible path patterns spanning across different loops, our current WCET analysis cannot exploit such information. In future, we plan to augment our WCET estimation technique to more accurately analyze complex control flow involving sequences and nesting of loops.

3.2 An Example to show Infeasible Path Detection

We now work out the even number addition example in Figure 1 to detect infeasible paths using our constraint propagation technique. The program in Figure 1 illustrates a class of infeasible paths which are hard to detect statically using current methods. In particular, these paths:

- span across multiple iterations of a loop
- contain branches whose outcome is different in different iterations (the differing outcomes make it impossible to use strong invariants for all executions of the branch).

The control flow graph of the program fragment in Figure 1 is shown in Figure 2. The loop is shown in a bold box. There is only one branch inside the loop, the branch in basic block B2. Thus, the constraint propagation algorithm will be executed twice corresponding to the yes/no outcomes of this branch. We also define only one proposition corresponding to the condition in the only branch inside our loop. Thus, proposition p_1 is defined as $p_1 \equiv$ `j % 2 == 0`. Let us now illustrate the constraint propagation for finding infeasible paths which end at a no outcome at basic block B2. Note that during constraint propagation, for each visit of a basic block we maintain a boolean formula (over the branch propositions) and a constraint (computed via weakest pre-condition analysis). So, we start with

$$B2,\ \neg p_1,\ \mathtt{j}\%2 \neq 0$$

We now propagate backwards and visit B1. This produces

$$B1,\ \neg p_1,\ \mathtt{j} \leq \mathtt{limit} \wedge \mathtt{j}\%2 \neq 0$$

Now, the predecessors of B1 are B0 and B4. Since we are only analyzing the infeasible paths spanning the iterations of a loop (this of course can be relaxed), we only visit B4. This produces

$$B4,\ p_1,\ \mathtt{j}+\mathtt{1} \leq \mathtt{limit} \wedge (\mathtt{j}+1)\%2 \neq 0$$

Note that this involves inferring the truth of p_1 from `(j+1) % 2` $\neq$ 0. This inferencing has to be achieved by an external constraint solver. If this cannot be inferred, then we will visit B4 with the boolean formula *true* instead (i.e., the constraint propagation will anyway proceed). The predecessors of B4 are B2 and B3. When we visit B2, the constraint store implies `(j+1) % 2` $\neq 0 \wedge$ `j` % 2 $\neq$ 0. Since this is false, we can infer that the path B2,B4,B1,B2 cannot end with a no outcome. In other words B2,B4,B1,B2,B4 is an infeasible path. Note that termination of the analysis is guaranteed, since each basic block in this example can be visited at most four times (with the boolean formulae *true*, $p1$, $\neg p1$ and $false$).

4 WCET Analysis

In this section, we present our analysis technique for estimating the WCET of a program loop. We note that if the input program has nested loops or sequences of loops, we perform the analysis for each loop separately and then compose the results. Thus, for nested loops, the inner loop is analyzed first followed by the outer loop.

The inputs to our analyzer are the following.

- The loop bound N. The loop bound is computed using offline techniques like [9].
- The set of feasible paths IP, each member of which denotes the possible execution of one iteration of the loop. From now on, we will refer to a path in the set IP as **ipath** to distinguish it from a path through multiple iterations of the loop. Each ipath is associated with its WCET.
- The set of infeasible ipath sequences through the loop called the **infeasible patterns**. Each infeasible pattern is a finite string over the alphabet IP. Let $K + 1$ be the maximum length for any infeasible pattern for the loop. Clearly $1 \leq K \leq N - 1$. Typically, $K << N$.

The basic idea of the technique is based on the following observation. Let the maximum length of any infeasible pattern for the loop be $K + 1$. Therefore, given a partially constructed ipath sequence, we need to look back *at most* K iterations to enumerate the feasible ipaths in the next loop iteration such that the sequence does not contain any infeasible pattern. Therefore, in order to compute the WCET for the entire loop, we only exhaustively enumerate all the legal ipath sequences of length K. As K is quite small in practice, this exhaustive enumeration is quite fast. Note that if there is no infeasible pattern, then the WCET of the loop is simply $(max_{p \in IP}\ wcet(p)) \times N$.

Next, we find out whether an ipath sequence can follow another ipath sequence. This information is represented by a directed graph, called the **transition graph** $G = (V, E)$. Each node $v \in V$ of this graph represents a legal ipath sequence of length K. An edge $u \rightarrow v \in E$ implies that v can follow u. A node v can follow a node u *if and only if* the concatenation of the ipath sequences of u and v does not include any infeasible pattern. Note that the graph can also contain self-edges. Clearly, in the worst case $|V| = |IP|^K$. Each node $v \in V$ is annotated with its WCET, $wcet(v)$, defined as the summation of the WCETs of its K constituent ipaths.

Given the transition graph $G = (V, E)$, we need to find the WCET of the loop. First, let us assume that N is a multiple of K. Then the problem reduces to finding the sequence of N/K nodes (with possibly repeating nodes) of maximum weight through the transition graph G. This problem can be solved through dynamic programming as follows. Let $WCET_v^l$ be the maximum execution time of any sequence of nodes of length l (i.e., a sequence of ipaths of length $l \times K$) ending at node v. We define $WCET_v^l$ recursively as follow. First,

$$WCET_v^1 = wcet(v) \quad \forall v \in V$$

For $l > 1$

$$WCET_v^l = \max_{u \in V,\ u \rightarrow v} \left(WCET_u^{l-1} + wcet(v) \right)$$

Therefore, the WCET of the loop is defined as

$$WCET = \max_{v \in V} \left(WCET_v^{N/K} \right)$$

The complexity of this dynamic programming approach is $O(\frac{N}{K} \times |V|^2) = O(\frac{N}{K} \times |IP|^{2K})$. In practice, both $|IP|$ and K are quite small.

If N is not a multiple of K, then we need to take the remainder iterations $N\%K$ into consideration. First, we enumerate all legal sequences of ipaths of length $N\%K$; the number of such sequences is small since $K << N$. Let these sequences be represented by the set S. Then, the WCET of the loop is defined as

$$WCET = \max_{v \in V, s \in S, feasible(v,s)} \left(WCET_v^{N/K} + wcet(s) \right)$$

where $feasible(v, s)$ is true if and only if the concatenation of the ipath sequence corresponding to v and s does not include any infeasible pattern. A fast but conservative approach can simply use the worst possible ipath for the remainder. That is, the WCET of the loop is

$$\max_{v \in V} \left(WCET_v^{N/K} \right) + (max_{p \in IP}\ wcet(p)) \times (N\%K)$$

Note that the algorithm above works only because the state transition graph is defined in such a way that no path in the graph contains any known infeasible sequence of ipaths (*i.e.* a sequence detected as infeasible in the previous phase of infeasible path detection).

5 Experimental Results

We have implemented a prototype analyzer to estimate the worst case execution time of a loop using the technique described in the previous sections. Figure 3 shows the framework of out timing analyzer which combines the infeasible path detection and WCET analysis. The input to our analyzer is the binary executable. For this particular implementation, we use executables compiled by modified gcc for Simplescalar [4], an architectural simulation platform. The analyzer first disassembles the binary, identifies the basic blocks, and constructs the control flow graph (CFG) of the entire program. It then separates out the CFGs corresponding to the loops. The analyzer first estimates the WCET of inner loops and then uses these information to estimate the WCET of outer loops. For each loop, the analyzer enumerates all the ipaths in the loop. Each ipath is associated with the corresponding execution time. In the prototype analyzer, we simply assume the execution time of an ipath is equal to the number of instructions in the ipath.

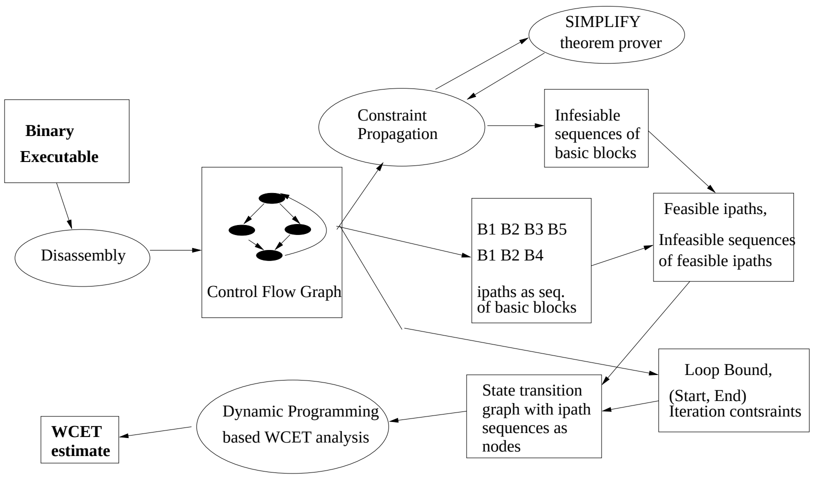
SIMPLIFY theorem prover
Binary Executable
Constraint Propagation
Infesiable sequences of basic blocks
Disassembly
Control Flow Graph
B1 B2 B3 B5
B1 B2 B4
ipaths as seq. of basic blocks
Feasible ipaths,
Infeasible sequences of feasible ipaths
Loop Bound,
(Start, End) Iteration contsraints
State transition graph with ipath sequences as nodes
Dynamic Programming based WCET analysis
WCET estimate

Fig. 3. Design Flow of Timing Analyzer

The core of the analyzer first identifies the infeasible paths using the constraint propagation method. We use the Simplify theorem prover [19] in this phase to check satisfiability of the constraint store in each step of the weakest pre-condition computation. The infeasible path information is used to eliminate some ipaths from further consideration. Moreover, this information is also used to generate the infeasible ipath patterns. Finally, we generate the transition graph over ipath sequences and use it to compute the WCET.

In our experiments, we have used the benchmarks shown in Table 1. Each of these benchmarks contains only one loop. Three of them: `fresnel`, `sprsin` and `expint` are taken from the book *Numerical Recipes in C* [16]; these benchmarks have been used in other works on program path analysis for estimating WCET (e.g. see [10]). The `fresnel` program has a loop which takes different ipaths in odd and even numbered iterations. The loop in `sprsin` avoids the longer ipath when the iteration counter reaches a specific constant value. `Expint` has the reverse characteristic: the longer ipath in a loop is executed only when the loop iteration counter reaches a specific constant value. The `wordcount` program counts the words in a file by detecting spaces; this is done by a loop which executes different ipaths depending on whether (or not) the next character marks the end of a word. This leads to infeasible path patterns spanning across iterations. The programs `SHM` and `check_data` also have iteration spanning infeasible path patterns. In particular, since the loop in `check_data` exits when a negative number is encountered, an ipath corresponding to a negative number input can never be followed by any other ipath.

The estimated WCET values for the benchmarks are shown in Table 2. The estimate is given in terms of the number of instructions executed in the loop. The number of iterations for the only loop in each benchmark is shown in the

Benchmark	Description
Wordcount	Counts the number of words in a string of 256 characters
Check_data	Check if the input vector of 100 integers has a negative entry
Fresnel	Computes non-complex Fresnel integrals
Sprsin	Convert 10×10 matrix to row-indexed sparse storage mode
Expint	Computes an exponential integral
SHM	Sequence of variable values repeats in a loop according to Simple Harmonic Motion

Table 1. Description of benchmarks used

Program	# Iterations	Default WCET	Our WCET	Improvement
wordcount	256	9472	8064	14.9%
fresnel	100	5200	5000	3.8%
SHM	100	2200	2002	9%
check_data	100	1900	916	51.8%
sprsin	10	520	476	8.5%
expint	100	185200	6109	96.7%

Table 2. WCET Estimation Results

column *# iterations*. *Default WCET* is simply the execution time of the longest ipath multiplied by the number of iterations. In other words, it does not take into account infeasible path information. The column *Our WCET* shows the result of our WCET analysis which takes into account infeasible path information. The column *Improvement* shows the reduction in WCET estimate using our method.

Running Times On a Pentium IV 2.4 GHz machine, our infeasible path detection phase takes only few seconds for all the benchmarks. The time is primarily spent in the external prover Simplify. We found that the time overheads for using the Simplify prover are tolerable, with each call to Simplify typically taking less than 10 milliseconds. The second phase of our technique (i.e. WCET analysis) takes less than 0.01 second for all the benchmarks.

6 Discussion

Detection of infeasible paths is central for obtaining tight Worst-case Execution Time (WCET) estimates. In this paper, we have developed an infeasible path detection technique based on constraint propagation. We have then exploited these path patterns to develop tight WCET estimates of program loops. Our WCET analysis technique is based on dynamic programming and carefully avoids exhaustive enumeration of feasible path sequences. Experimental results on non-trivial benchmarks show that our technique leads to substantial reduction in WCET estimates.

References

1. P. Altenbernd. On the false path problem in hard real-time programs. In *Euromicro workshop on Real-time Systems*, 1996.
2. T. Ball and J.R. Larus. Programs follow paths. Technical report, Microsoft Research, MSR-TR-99-01, 1999.
3. R. Bodik, R. Gupta, and M. Lou Soffa. Refining data flow information using infeasible paths. In *ESEC/SIGSOFT FSE*, 1997.
4. D. Burger, T. Austin, and S. Bennett. "Evaluating future microprocessors: The simplescalar toolset". Technical Report CS-TR96-1308, University of Wisconsin-Madison, 1996.
5. E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.
6. J. Engblom and B. Jonsson. Processor pipelines and their properties for static WCET analysis. In *Intl. Conf. on Embedded Software (EmSoft), LNCS 2491*, 2002.
7. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *EUROPAR*, 1997.
8. C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *ACM Intl. Workshop on Languages, Compilers and Tools for Real-Time Sys. (LCTRTS)*, 1997.
9. C.A. Healy et al. Supporitng timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2-3), 2000.
10. C.A. Healy and D.B. Whalley. Automatic detection and exploitation of branch constraints for timing analysis. *IEEE Transactions on Software Engineering*, 28(8), 2002.
11. T.A. Henzinger, R. Jhala, R. Majumder, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
12. X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *IEEE Real-time Systems Symposium (RTSS)*, 2004.
13. Y-T. S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *ACM Transactions on Design Automation of Electronic Systems*, 4(3), 1999.
14. T. Lundqvist and P. Stenstrom. Integrating path and timing analysis using instruction-level simulation techniques. In *Intl. Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES)*, 1998.
15. C.Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-time Systems*, 5(1), 1993.
16. W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition,.* Cambridge University Press, 1988.
17. P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Real-time Systems*, 1(2), 1989.
18. A.C. Shaw. Reasoning about time in higher level language software. *IEEE Transactions on Software Engineering*, 1(2), 1989.
19. Simplify. Simplify theorem prover, 1998. `http://www.research.compaq.com/SRC/esc/Simplify.html`.
20. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analysis. *Real Time Systems*, 18(2/3), 2000.