Integrating Path and Timing Analysis using Instruction-Level Simulation Techniques

Thomas Lundqvist and Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{thomasl,pers}@ce.chalmers.se

Abstract. Previously published methods for estimation of the worst-case execution time on contemporary processors with complex pipelines and multi-level memory hierarchies result in overestimations owing to insufficient path and/or timing analysis. This paper presents a new method that integrates path and timing analysis to address these limitations. First, it is based on instruction-level architecture simulation techniques and thus has a potential to perform arbitrarily detailed timing analysis of hardware platforms. Second, by extending the simulation technique with the capability of handling unknown input data values, it is possible to exclude infeasible (or false) program paths in many cases, and also calculate path information, such as bounds on number of loop iterations, without the need for annotating the programs. Finally, in order to keep the number of program paths to be analyzed at a manageable level, we have extended the simulator with a path-merging strategy. This paper presents the method and particularly evaluates its capability to exclude infeasible paths based on seven benchmark programs.

1 Introduction

Static estimation of the worst-case execution time (WCET) has been identified as an important problem in the design of systems for time-critical applications. One reason is that most task scheduling techniques assume that such estimations are known before run-time to schedule the application tasks so that imposed timing constraints, e.g. deadlines, are met at run-time.

The WCET of a program is defined by the program path that takes the longest time to execute regardless of input data and initial system state. Ideally, a WCET estimation method should take as input a program and estimate a tight upper-bound on the actual WCET for a given hardware platform. There are two important sources to overestimations of the WCET. First, the estimation technique may encounter program paths that can never be executed regardless of the input data, usually referred to as *infeasible* (or false) program paths. Second, the timing model of the hardware platform may introduce pessimism because of simplifying timing assumptions.

The first problem can be addressed by requiring that the programmer provides path annotations [14] which clearly is both time-consuming and error-prone. A more attractive method is to automatically detect infeasible paths through static *path analysis methods* [1, 3]. As for the second problem, several *timing analysis* approaches have been proposed that statically estimate the execution time of a given path by taking into account the effects of, e.g., pipeline stalls and cache misses [5, 7–10, 13, 16]. Unfortunately, since these timing analysis approaches typically charge fixed penalties caused by cache misses and pipeline hazards, they have difficulties to take into account the timing effect of dynamic interactions such as resource contention in the hardware platforms caused by buffering of instructions and memory requests. More seriously, no study has shown how timing analysis methods can be integrated with path analysis methods to relieve the programmer from the burden of identifying and excluding infeasible paths from the analysis.

In this paper we present a new approach to WCET estimation that integrates path analysis with accurate timing analysis. Our WCET estimation method achieves this goal by (1) using instruction-level simulation techniques [12, 17] that conceptually simulate *all* feasible paths on arbitrarily detailed timing models of the hardware platform; (2) by reducing the number of infeasible paths by extending architectural simulation techniques to handle unknown input data; and (3) by calculating, e.g., bounds on number of loop iterations with no need for making annotations for statically known bounds.

A practical limitation of the method in its basic form is that the number of paths to simulate can easily be prohibitive, especially in loop constructs. We have therefore extended the architecture simulation method with a path-merging approach that manages to bound the number of simulated paths in a loop to just a few paths. This merging strategy reduces the number of simulated paths drastically and makes the approach useful for realistically sized programs. We outline how the method can be applied to pipelined processors with instruction and data caches and present how the merging strategy is implemented to encounter the worst-case effects of these features.

The next three sections are devoted to describing our method focusing on the basic simulation approach in Section 2, the WCET algorithm in Section 3 and how detailed timing models of architectural features are integrated into the method in Section 4. An evaluation of the method focusing on the path analysis of seven test programs is presented in Section 5. In Section 6, we discuss the potential of the method along with its weaknesses in the context of previous work in this area before we conclude in Section 7.

2 The Approach

Consider programs for which the WCET is statically decidable; i.e., all possible execution paths in the program are finite in length. For example, bounds on number of iterations in loops are known before run-time, although they might be difficult to determine due to limitations in compiler analysis methods. Second, for the time being, we will consider processors with fixed instruction execution times although our method is applicable to arbitrary complex processor architectures with associated memory hierarchies which we show in Section 4. Given these assumptions, WCET could be conceptually determined by identifying the feasible path through the program with the longest execution time.

Instruction-level architectural simulation techniques have now matured so that the execution time of a program run on complex processor architectures can be accurately determined with a reasonable simulation efficiency [12, 17]. The advantages of using architectural simulation techniques are twofold. First, it is possible to make arbitrarily accurate estimations of the execution time of a program, given a combination of input data. Second, and presumably more importantly, when a given path through the program has been simulated, all input data independent dependencies will be resolved.

We first describe how we have extended traditional instruction-level simulation to automate the path analysis in Section 2.1. Then, in Section 2.2 we discuss some performance issues with the basic approach and how they are addressed.

2.1 Path analysis using instruction-level simulation techniques

Instruction-level simulation techniques assume that input data is known and therefore only analyze a single path through the program associated with this input data. To find the WCET of a program using this approach, however, the program would have to be run with all possible combinations of input data which is clearly not feasible. Our approach is instead to simulate all paths through the program and in this process exclude the paths that are not possible regardless of input data. To do this, we have extended traditional instruction-level simulation techniques with the capability to handle unknown data, using an element denoted *unknown*. Then the semantics for each data-manipulating instruction is extended to correctly perform arithmetics with the unknown data values. Examples of the extended semantics for some common instruction types can be seen in Table 1.

The load and stores need special treatment, since the reference address used may be *unknown*. For loads, this results in an *unknown* being loaded into a register. For stores, however, an unknown address can modify an arbitrary memory location. Therefore, the correct action would be to assign the value *unknown* to all memory locations to capture the worst-case situation. This is of course a major limitation. For the time being, we will assume that all addresses are statically known. However, we will discuss efficient solutions to overcome this limitation later in Section 2.2.

The semantics for a conditional branch is also special. When a conditional branch whose branch condition is *unknown* is encountered, both paths must be simulated. To understand how this is used to automate path analysis, consider the program in Figure 1 which sums values in the upper-right triangle of matrix b. In the beginning of the simulation, the data values in matrix b are treated as unknown input and all elements are assigned the value *unknown*. The boolean values in matrix m are considered known.

Table 1. Extended semantics of instructions.

Instruction type	Example	Semantics
ALU	ADD T,A,B	$T \leftarrow \begin{cases} \mathit{unknown} \text{ if } A = \mathit{unknown} \vee B = \mathit{unknown} \\ A + B \quad \text{otherwise} \end{cases}$
Compare	CMP A,B	$A - B$ and update the condition code (cc) register. May set bits in the cc-register to *unknown*.
Conditional branch	BEQ L1	Test bit in cc-register to see if we should branch. If bit is *unknown* simulate both paths.
Load	LD R,A	Copy data from memory at address A to register R (the data can be *unknown*). If address is *unknown*, set R to *unknown*.
Store	ST R,A	Copy data from register R to memory at address A (the data can be *unknown*). If address is *unknown*, all memory locations are assigned *unknown* (a more efficient solution is discussed in Section 2.2).

```
1   for (i = 0 ; i < 4 ; i++)
2     if (b[i][i] > 0)
3       for (j = i+1 ; j < 4 ; j++)
4         if (m[i][j])
5           sum += b[i][j];
```

Fig. 1. Example program.

When simulating this program, the conditional branch on line 2 will be the only branch depending on unknown values and the total number of simulated (and feasible) paths will be $2^4 = 16$. The infeasible paths that the conditional branch on line 4 could create are automatically eliminated since the branch condition is known when simulating. Also, no loop bound is needed for the inner loop since the iteration count only depends on variable i which is known during the simulation. Thus, we see that an important advantage of the instruction-level simulation approach is that all conditions that depend on data values that are known statically will be computed during the simulation. This eliminates, in this example, the need for program annotations.

2.2 Discussion

Ideally, for each combination of input data, the goal is to eliminate all infeasible program paths. The domain of values we use, however, makes it sometimes impossible to correctly analyze mutually exclusive paths. Consider for example the statements below where b is *unknown*.

```
if (b < 100) fun1()
if (b > 200) fun2()
```

Variable b will be *unknown* in both conditions, forcing the simulation of four paths even if `fun1()` and `fun2()` are mutually exclusive and the number of feasible paths are three. The effect of this will be a potential overestimation of the WCET which is caused by the fact that we only distinguish between known and unknown values. However, there is nothing in the approach that would hinder us from extending the domain of values to, e.g., ranges of values. By doing this, it would be possible to handle cases like the one above. The simulation technique we have implemented, and which we evaluate later in the paper, uses the simpler domain. However, we will discuss the implications of using ranges in Section 6.

As mentioned above, store instructions with an unknown reference address need special treatment. To efficiently handle these accesses, we identify all data structures that are accessed with unknown store instructions as *unpredictable data structures*. These are mapped by the linker into a memory area which can only respond with unknown values. This means that stores to this area can be ignored and loads will always return *unknown* as a result, thus assuring us of a correct estimation of the WCET with no added cost in simulation time.

We identify the unpredictable data structures by letting the simulator output a list of all unknown stores it encounters. With the help of a source-code debugger, it is possible to connect each store instruction to a data structure which is marked as unpredictable. Eventually, the linking phase is redone to properly map the marked data structures to the 'unknown' area of memory. After this step, a correct estimation may be done. The approach mentioned above can be used for statically allocated data structures only. For unpredictable dynamically allocated data structures we do not yet have any solution.

A key problem with the simulation approach in this section is the explosion in number of paths to be simulated. If a loop iteration has n feasible paths and the number of loop iterations is k, the number of paths to simulate is n^k. Fortunately, good heuristics exist to drastically reduce the number of paths we need to simulate. We have used a path merging strategy, which forms the basis of our WCET method to be presented in the next section.

3 The WCET Method

To reduce the number of paths that have to be explored in the path analysis, we apply a path-merge strategy. This reduces the number of paths to be explored from n^k down to n for a loop containing n feasible paths in each iteration and doing k iterations. In each loop iteration, all n paths are explored but in the beginning of the next iteration all these n paths are merged into one. Thus, the number of simulated paths are less than or equal to n. We first describe how the merging operation is performed in Section 3.1. This operation is used in the WCET algorithm which we present in Section 3.2. Finally, in Section 3.3 we discuss how we have implemented the method with a reasonable time complexity.

3.1 Merging algorithm

In order to understand how the merging of two paths is done, consider again the example program in Figure 1. In the second iteration of the outer loop, when the simulator encounters the unknown conditional branch on line 2, two paths can be merged. When merging, the long path (the one with the highest WCET) is kept and the short path is discarded. However, to make a valid estimation of the worst-case execution path throughout the program execution, the impact of the short path on the total WCET must be taken into account. For example, variable `sum` can be assigned different values in the two paths. Therefore, all variables whose values differ, must be assigned the value *unknown* in the resulting path of the merge operation.

Formally, the algorithm views a path, p_A, as consisting of a WCET for the path, $p_A.wcet$, and the system state at the end of the path, $p_A.state$. The state of a path is the partial result of the execution including, e.g., the content of all memory locations, registers, and status bits that can affect the WCET in the future. In order to compute the state of the path resulting from the merging of several paths, the states of the merged paths are compared and *unknown* values are assigned to locations whose values differ. We denote this operation as the *union* operation of path states.

The merging algorithm is described in Algorithm 1. It creates a new path p_C from two paths p_A and p_B. The program counter, which must not differ in the two paths merged, is copied to the new path. The new WCET is the maximum of the WCET of the two original paths. Finally, states of the merged paths are unioned. The union operation between values is defined as:

$$a \cup b \Leftrightarrow \begin{cases} \mathit{unknown} \text{ if } a = \mathit{unknown} \vee b = \mathit{unknown} \\ \mathit{unknown} \text{ if } a \neq b \\ a \qquad\quad \text{otherwise} \end{cases}$$

Algorithm 1 Merging two paths p_A and p_B creating p_C.

{PCT = program counter, CC = condition code}
Require: $p_A.state.PCT = p_B.state.PCT$
$p_C.state.PCT \leftarrow p_A.state.PCT$

$p_C.wcet \leftarrow \max(p_A.wcet, p_B.wcet)$

$p_C.state.CC \leftarrow p_A.state.CC \cup p_B.state.CC$
for all registers i **do**
  $p_C.state.R[i] \leftarrow p_A.state.R[i] \cup p_B.state.R[i]$
end for
for all memory positions a **do**
  $p_C.state.MEM[a] \leftarrow p_A.state.MEM[a] \cup p_B.state.MEM[a]$
end for

3.2 WCET algorithm

In order to implement the WCET simulation technique and the merging algorithm, one important issue is in which order paths should be simulated. Consider a loop with two feasible paths in each iteration. In order to merge these two paths, they must have been simulated the same number of iterations. To accomplish this, the WCET algorithm uses loop information from the control flow graph of the program.

The algorithm (see Algorithm 2) starts the simulation from the beginning of the program. Whenever an unknown conditional branch is found, the simulation is stopped and the algorithm selects, as the next path to simulate, the path that has made the least progress, *a minimum progress path*, in terms of loop iterations. If this path is not unique, all paths that have made the same progress and are at the same position in the program, *equal progress paths*, are merged into one before simulation is continued.

By always selecting the path that has made least progress, the algorithm makes it certain that all paths in a loop iteration is simulated before a new iteration begins. In fact, merging will occur every time two paths that have made equal progress meet at the same position in the program. This makes an exponential growth of the number of paths impossible.

3.3 Performance considerations

The performance of this method depends on how many paths we need to simulate, how often merging is done and how fast the actual merge operation is. How often the merging should be done is a complex question. If we merge often, it is likely that we will have fewer paths to simulate. However, when doing a merge we can lose information in a way that makes us fail to eliminate infeasible paths. Thus, merging too often can also create more infeasible paths and in the end lead to a larger number of simulated paths.

The algorithm we have evaluated merges as often as possible. This makes an exponential growth of the number of paths impossible, e.g. in loops. Typically, each unknown branch found during the simulation would lead to the creation of yet another path, which later would result in an additional merge operation. The total number of paths simulated, as well as the number of merge operations, can in some cases grow in proportion to the number of loop iterations done in the program. One example is a loop with one unknown exit condition. The simulation of this loop would produce one new path (the exit path) to simulate each iteration. All these paths would, unless they reach the end of the program, be merged resulting in an equal number of merge operations.

For each merge operation, one must union the content of all registers and memory locations. This might be a quite slow process if the amount of memory is large. We can speed up this operation considerably by utilizing the fact that paths that are to be merged, often shared a long history of execution before they got split up. By only recording changes made to the system state since the time where the two paths were created, we can quickly identify the parts of memory

Algorithm 2 WCET algorithm handling merge.

{A is set of active paths, C completed paths, $\emptyset$ = empty set, $\setminus$ = set minus}
$A \leftarrow \emptyset$, $C \leftarrow \emptyset$

$p \leftarrow$ starting null path
Simulate(p)
if p reached end of program **then**
  $C \leftarrow C \cup \{p\}$
else {p reached an unknown conditional branch}
  $A \leftarrow A \cup \{p\}$
end if

while A not empty **do**
  $p \leftarrow$ minimal progress path in A
  $A \leftarrow A \setminus \{p\}$
  for all paths q with equal progress as p **do**
    $A \leftarrow A \setminus \{q\}$
    $p \leftarrow$ merge(p, q)
  end for
  {Path p ends with a branch forcing a split}
  for each possible branch target i **do**
    $p_i \leftarrow$ copy of p
    Simulate(p_i) along target i
    if p_i reached end of program **then**
      $C \leftarrow C \cup \{p_i\}$
    else {p_i reached an unknown conditional branch}
      $A \leftarrow A \cup \{p_i\}$
    end if
  end for
end while

$wcet \leftarrow \max_{p \in C} p.wcet$

where the two system states differ. As an example, suppose that the system we model contains 1 Mbyte of main memory. Then one can divide this memory into small fixed size pages (say 512 bytes each) and each path only keeps the accessed pages in its system state. In this way, only a few pages of memory need to be compared during merging.

4 Timing Analysis

The WCET algorithm in the previous section can estimate WCET for hardware platforms with fixed instruction execution times. In this section, we extend it to model the timing of pipelined processors with caches. To demonstrate the approach, we particularly focus on systems with separate, direct-mapped instruction and data caches and a single-issue pipelined execution unit. However, the approach extends to more sophisticated architectures.

In order to update the WCET properly during simulation, the simulator must of course be extended to model the timing of caches and pipelines. In the context of caches, the simulator must model the impact of cache misses on the execution time. And in the context of pipelines, the simulator must account for the impact of structural, data, and control hazards [6] on the execution time. With this capability, it is possible to make a safe estimation of the execution time of a given path through the program.

A critical issue is how to carry out a merge operation. To do this, one must be able to estimate the impact of the system state on the future execution time. Such state information is exemplified by the content of the tag memory in the caches (which affects future misses) and resources occupied by an instruction, such as data-path components and registers (which affect future structural, data and control hazards). The merging operation introduced in Section 3 must be extended to handle such state information, which we will refer to as *timing state*.

To merge the timing state, we could use the same general principle as used when merging the content of memory locations; for all locations where the timing state differs we assert a pessimistic value, such as *unknown* in the merged timing state. For example, in the case of most caches *unknown* means that a cache block is invalid, and the unioning of two cache timing states makes all cache blocks whose tags differ invalid. We call this method the *pessimistic merge*, since it can incur a severe pessimism in the estimation.

Fortunately, it is possible to reduce the pessimism when merging. If it was known in advance which path belongs to the worst case path through the program, one could update the worst-case execution time with the execution time of that path and also choose the timing state of that path when merging and discard the timing state of the other path. Then, no pessimism would be incurred on a merge operation. While it is not possible to know in advance which of the two paths belongs to the worst-case path, a good guess would be that the longer of the two belongs to the worst-case path. If we also estimate how big effect the timing state of the shorter path has on the future execution time, we can make sure whether it is correct to use the WCET of the longer path along with its timing state when merging two paths. This approach is formulated in the following algorithm where we assume that the worst-case execution times of the long and the short paths are $WCET_L$ and $WCET_S$, respectively.

1. Estimate the *worst-case penalty* ($WCET_P$) that the short path would incur on the future execution time that will not be incurred by the long execution path.
2. If $WCET_L \geq WCET_S + WCET_P$ then use the timing state and WCET of the long path in the merge operation and discard the timing state of the short path.
3. Otherwise, the pessimistic merge approach must be used.

In order to make this algorithm useful, we must clearly define what we mean with timing state and worst-case penalty, and how to perform the pessimistic merge. This is done in the next two sections in the context of caches and pipelining.

4.1 Instruction and data cache analysis

Both instruction and data caches can be described and treated using the same principles. Therefore, we make no distinction between them. Consider the timing state of two caches, C_L and C_S, belonging to the long and short path, respectively. The timing state of a cache is represented by an array of tags showing how blocks are currently mapped in the cache. To calculate the worst-case penalty, we must consider all cases where C_S may lead to a greater number of future cache misses than C_L would. The worst case is found if we imagine that all cache blocks resident in C_L but not found in C_S will be needed in the future. We would then lose all these potential misses if we discard C_S. To find the worst-case penalty, we go through all cache blocks and compare the tag in C_S with the tag in C_L and sum up the cache miss penalties caused by all differences found.

While pessimistic merging might be needed if it is not possible to discard C_S, it is not always necessary to invalidate all blocks found different. By invalidating a block in C_L we make it impossible for C_S to cause any additional misses in the future. Thus, for each block we invalidate in C_L we reduce the worst-case penalty. Eventually, it may became small enough to proceed and actually discard C_S.

4.2 Pipeline analysis

A timing model of a pipeline must keep track of the resources (pipeline stages and registers) an instruction uses and when each resource is released in order to resolve structural, data and control hazards. The time when each resource is last released influences the future instructions and the worst-case penalty. Consider the timing state of two pipelines, P_L and P_S, belonging to the long and short path, respectively. We want to estimate the possible future effect on the execution time that P_S may lead to compared to P_L, and must in this case consider all future hazards that P_S can lead to, which P_L cannot lead to. For each resource we determine when it is last released and whether it is released later in P_S compared to P_L. The worst-case penalty is the maximum difference in release time found among all resources.

If we are not allowed to discard P_S, we must do the pessimistic merge instead. However, this can sometimes be avoided using the same principle as for the caches; we change P_L in a way that reduces the possible future effect on the execution time that P_S may lead to compared to P_L. For each possible structural or data hazard resulting from P_S but not from P_L, we can change P_L so that resources are released later. In this way we can gradually reduce the worst-case penalty, until at last, we are allowed to discard P_S.

5 Experimental Results

We have estimated the WCET of seven benchmark programs, running on an idealized architecture with no caches, and where all instructions execute in one clock cycle. Thus, the evaluation focuses on the path analysis aspects of the method.

5.1 Methodology

A WCET simulator has been constructed by extending an existing instruction-level simulator, PSIM [2], which simulates the POWERPC instruction set. The original simulator has been extended with the capability of handling unknown values. Also, the WCET algorithm described in Section 3.2 has been added to control the path exploration and merging. No cache and pipeline simulation was enabled and the execution time is equal to the simulated instruction count.

The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the benchmarks. No optimization was enabled. The simulated run-time environment contains no operating system; consequently, we disabled all calls to system functions such as I/O in the benchmarks.

Benchmarks and metrics An overview of the seven benchmark programs can be seen in Table 2. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *DES*, *jfdctint*, and *compress*.

The two benchmarks *fib* and *compress* both contain a loop where the exit condition depends on unknown input data. In order to bound the number of iterations, we need to add manual annotations. This is not supported in our implementation. Instead, we have added an extra exit condition in the loops. In *fib* we have added the condition: $i \leq 30$ because we know that input data is always in this range. In *compress* we bound an inner loop whose iteration variable is j, using the current iteration count, i, of the outer loop: $j \leq i$. This is a safe but very pessimistic bound, but we found it difficult to prove that a tighter bound could be used.

The true WCETs of all programs have been measured by running the programs in the simulator with the worst-case input data. This works fine for all programs except *compress*, where the worst case input data is hard to find. Instead, a random sequence of 50 bytes has been used as input.

Besides the estimated WCET from the extended simulator, we also did a manual estimate of the structural WCET, i.e., the execution time of the longest structural path in the control flow graph of the program and using fixed bounds on the number of iterations of all loops. This figure would represent a WCET estimation method that eliminates no infeasible paths and uses fixed iteration bounds for loops. The purpose of doing this is to analyze the capability of the method to eliminate infeasible paths.

5.2 Estimation results

Table 2 compares the measured true WCET with the estimated WCET from the simulator and the manually derived structural estimation for the benchmark programs. (WCET is expressed in clock cycles, ratio is the WCET relative to the measured, true WCET.)

For all benchmarks, except *compress*, we find that the method succeeds in finding the true WCET. In *compress*, the overestimation is caused by the inner

Table 2. The estimated and true WCET of benchmark programs.

Program	Description	Measured true WCET	Estimated WCET	Ratio	Estimated structural WCET	Ratio
matmult	Multiplies 2 10x10 matrices	7063912	7063912	1	7063912	1
bsort	Bubblesort of 100 integers	292026	292026	1	572920	1.96
isort	Insertsort of 10 integers	2594	2594	1	4430	1.71
fib	Calculate n:th element of the Fibonacci sequence	697	697	1	697	1
DES	Encrypts 64-bit data	118675	118675	1	119877	1.01
jfdctint	Discrete cosine transform of an 8x8 image	6010	6010	1	6010	1
compress	Compresses 50 bytes of data	9380	49046	5.2	161161	17.2

loop. As mentioned previously, we bound this loop using the very pessimistic condition $j \leq i$, but when measuring the true WCET, we found that this inner loop is actually only doing one single iteration. For *compress*, we do not know if the measured WCET is actually the true WCET. The true WCET is probably higher than the measured one.

Two of the benchmarks, *matmult* and *jfdctint*, have no infeasible paths at all, and only one path was simulated. In *DES*, however, there exist infeasible paths caused by data dependencies between different functions. These infeasible paths were eliminated and only one path was simulated. In *bsort* and *isort*, all infeasible paths were not eliminated. Still, this did not lead to any over-estimation, since all simulated infeasible paths were shorter than the worst-case path found.

If we take a look at the estimated structural WCET of the programs, we see that the WCET is grossly over-estimated for *bsort*, *isort*, and *compress*. In *bsort* and *isort* it depends entirely on using a fixed iteration count for an inner loop which is normally bounded by the outer loops current iteration count. This leads to an over-estimation of a factor of two for the loop and influences *bsort* more than *isort* because of the greater number of iterations done in *bsort*. In *compress* there is a similar inner loop which is forced to have a fixed iteration bound again causing an over-estimation of a factor of two. In addition, there exists a very long infeasible path that extends the structural estimate. This path is eliminated when estimating with our simulation method. The tiny over-estimation in *DES* results from infeasible paths that are not possible to eliminate.

One strength of doing the analysis on the instruction level can be seen when looking at *DES*. In the source code, one can find several conditional expressions which should indicate several possible feasible paths through the program. However, the compiler (gcc with no optimization enabled) automatically generates code without any branches for these conditional expressions and the resulting program has only a single feasible path. In summary, the method in this paper appears promising in eliminating infeasible paths.

6 Discussion and Related Work

As the results indicate, to get a tight estimation of the WCET of a program, it is crucial to eliminate infeasible paths, especially in the presence of loops with dynamic bounds. Of equal importance is an accurate timing analysis. This has not been evaluated in this study, but the potential can be seen when looking at *DES*, *matmult*, and *jfdctint*, where the simulator only needs to simulate one path through the program. This path can be simulated with an arbitrary detailed timing model and will always give us the tightest possible WCET. Thus, by eliminating infeasible paths we can concentrate on the feasible ones, and make a more accurate timing analysis.

A big advantage of integrating the path and timing analysis can be seen when comparing with solutions where the path and timing analyses are kept separated. If the path analysis is done first, we would need a way to represent the path information generated from the path analysis, and the timing analysis phase must be able to utilize this information. On the other hand, if the timing analysis is done first, we would be forced to work with fixed WCETs for blocks of statements when doing the path and WCET calculation. These problems are not present in our method, which does the path and timing analysis simultaneously.

Our method is related to the *path analysis* methods presented in [1, 3]. The method of Ermedahl and Gustafsson operate on the source-program level. It establishes upper bounds on the number of loop iterations and also identifies infeasible paths, but makes no timing analysis. Altenbernd's method uses precalculated execution times on each basic code block. These times are used to prune paths during the path exploration. However, in some cases, the method suffers from complexity problems. We use a path-merging strategy instead, which guarantees a manageable number of paths, and also makes it possible to integrate it with a detailed timing analysis. In [15], Stappert and Altenbernd present a new method which handles caches and pipelines by first making a timing analysis for each basic block and then searching for the longest feasible path. The method only handles programs without loops. In [4], a way to automatically derive loop bounds by means of a syntactical analysis has been proposed. They successfully use this technique together with timing analysis in order to reduce the work for the user, but they do no general path analysis in order to identify infeasible paths. In a recent work, Liu and Gomez [11] construct time-bound functions for high-level languages, using a technique related to ours. However, no concern is made for doing accurate low-level timing analysis.

In our method, the simulator uses a very simple domain where values can be either known or unknown. This domain performs quite well compared to a more complex domain, e.g., based on intervals of values which is used in [1, 3]. However, as we saw in Section 2.2, overestimations may sometimes arise for mutually exclusive paths. There is no inherent problem in extending our method to a more powerful domain. The result would be a slower simulator needing more memory. Our choice of domain results in an additional 1 bit of memory for each 32-bit word of memory to hold the known/unknown status. An interval representation would need 2 extra words for each word of memory. Also, a more complex semantics would be needed, which would result in a slower execution of each instruction. On the other hand, the more complex domain might be preferable for some applications, if it manages to cut more infeasible paths and thereby gain speed and accuracy compared to our simple domain.

A more serious problem with our simple domain is that if all exit conditions of a loop is input data dependent, we get a completely unknown upper bound on the number of iterations in the loop, and our WCET algorithm will not terminate. This can be detected by using some heuristic or by user interaction. For cases like this, we must add a manual annotation or add a known exit condition by modifying the loop condition in the program. For example, the loop in the program below, where b is unknown input data, will never do more than 100 iterations regardless of b. This fact cannot be represented with our simple domain, and during the simulation the loop will get a completely unknown exit condition, forcing us to add annotations or modify the program.

```
if (b < 100)
  for (i = 0 ; i < b ; i++)
    sum = sum + i;
```

In the example above the simple domain was causing the problem. A similar problem can also arise when merging. The union operation used when merging may cause information needed to bound a loop to be lost. If this happens, we are also forced to annotate or change the program.

7 Conclusions

In this paper we have presented a new method for estimating the WCET of a program. This method integrates path and timing analysis and thereby has the potential to do tight estimations by eliminating infeasible paths and concentrating the timing analysis on the feasible ones. A study of seven benchmark programs, focusing on the path analysis aspects, show that many infeasible paths were indeed eliminated by the method, and the true WCET was found for almost all programs. However, it remains to be shown how well the method performs when considering the timing analysis as well. This is the topic of our current and future work.

Acknowledgments

We are deeply indebted to Dr. Jan Jonsson of Chalmers for his constructive comments on previous versions of this manuscript. This research is supported by a grant from Swedish Research Council on Engineering Science under contract number 221-96-214.

References

1. P. Altenbernd. On the false path problem in hard real-time programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pages 102–107, June 1996.
2. A. Cagney. PSIM, PowerPC simulator. ftp://ftp.ci.com.au/pub/psim/index.html
3. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *Proceedings of EUROPAR'97*, pages 1298–1307, August 1997.
4. C. Healy, M. Sjödin, V. Rustagi, and D. Whalley. Bounding Loop Iterations for Timing Analysis. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, June 1998. To appear.
5. C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.
6. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 2ed.* Morgan Kaufmann, 1996.
7. S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pages 230–240, June 1996.
8. Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 298–307, December 1995.
9. Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 254–263, December 1996.
10. S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium*, pages 97–108, December 1994.
11. Y. A. Liu and G. Gomez. Automatic Accurate Time-Bound Analysis for High-Level Languages. *Dept. of Computer Science, Indiana University, Technical Report TR-508*, April 1998.
12. P. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. Simics/sun4m: A virtual workstation. In *Proceedings of USENIX98*, 1998. To appear.
13. G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1997.
14. P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *The Journal of Real-Time Systems*, pages 159–176, 1989.
15. F. Stappert and P. Altenbernd. Complete Worst-Case Execution Time Analysis of Straight-line Hard Real-Time Programs. *C-LAB Report 27/97*, Paderborn, Germany, December 1997.
16. R. T. White, F. Mueller, C. A. Healy, D. B. Whalley, and M. G. Harmon. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3nd IEEE Real-Time Technology and Applications Symposium*, pages 192–202, June 1997.
17. E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Proceedings of ACM SIGMETRICS '96*, pages 68–79, 1996.