To appear in Real-Time Systems, 17(2/3):183–207, , 1–27 (1999)

An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution

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

Abstract. Previously published methods for estimation of the worst-case execution time on high-performance processors with complex pipelines and multi-level memory hierarchies result in overestimations owing to insufficient path and/or timing analysis. This does not only give rise to poor utilization of processing resources but also reduces the schedulability in real-time systems.

This paper presents a method that integrates path and timing analysis to accurately predict the worst-case execution time for real-time programs on high-performance processors. The unique feature of the method is that it extends cycle-level architectural simulation techniques to enable symbolic execution with unknown input data values; it uses alternative instruction semantics to handle unknown operands.

We show that the method can exclude many infeasible (or non-executable) program paths and can calculate path information, such as bounds on number of loop iterations, without the need for manual annotations of programs. Moreover, the method is shown to accurately analyze timing properties of complex features in high-performance processors using multiple-issue pipelines and instruction and data caches. The combined path and timing analysis capability is shown to derive **exact** estimates of the worst-case execution time for six out of seven programs in our benchmark suite.

Keywords: Real-time systems, worst-case execution time, timing analysis, path analysis, symbolic execution, multiple-issue processor, caches, architecture simulation.

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. Another reason is to early in the design process determine the processing capacity needed to meet timing constraints of a real-time application.

The actual WCET of a program is defined by the program path that takes the longest time to execute for the entire set of possible input data and initial system states. 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 include program paths that can never be executed regardless of the input data, usually referred to as *infeasible,* or non-executable, program paths. Second, the timing model of the hardware platform may introduce overestimations of the WCET because of simplifying timing assumptions. Both

of these sources to overestimations result in poor resource utilization at run-time which can considerably increase the system cost and/or reduce the schedulability of real-time programs.

The first problem can be addressed by requiring that the programmer provides path annotations (Puschner and Koza, 1989) which clearly requires a considerable programming effort and is error-prone. A more attractive method is to automatically detect infeasible paths through static *path analysis methods* (Chapman et al., 1994; Ermedahl and Gustafsson, 1997; Altenbernd, 1996). 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 (Li et al., 1995; Li et al., 1996; Ottosson and Sjödin, 1997; Kim et al., 1996; Lim et al., 1998; Healy et al., 1995; White et al., 1997; Theiling and Ferdinand, 1998). Unfortunately, none of the methods have managed to successfully integrate timing analysis for a high-performance processor with accurate path analysis to identify and exclude infeasible paths from the analysis automatically.

In this paper we present a new approach to static estimation of the WCET that integrates path analysis with accurate timing analysis. Our approach is to use architectural simulation techniques (Magnusson et al., 1998; Pai et al., 1997) that simulate each path on cycle-level timing models of the hardware platform. We extend such techniques to enable symbolic execution of programs in absence of knowledge of input data by augmentation of the instruction semantics to handle also unknown input data values. This results in the automatic derivation of statically known path information such as loop bounds from arbitrarily complex expressions and exclusion of many infeasible program paths.

A practical limitation of the method in its basic form is that the number of paths to simulate can easily become prohibitive, especially in loop constructs. We have therefore extended the symbolic execution 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 makes the approach useful for realistically sized programs. We demonstrate how the method can be applied to multiple-issue pipelined processors with instruction and data caches by presenting how the merging strategy is implemented to encounter the worst-case effects of these features. We have implemented the method and used it to estimate the WCET of programs run on a simulated dual-issue pipelined processor with instruction and data caches. We have found that the implementation of the method is capable of deriving **exact** estimates of the WCET for six out of seven programs in our benchmark suite.

The next three sections are devoted to describing our method focusing on the basic symbolic execution 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 in the context of a case study of a dual-issue processor with caches and WCET estimation of seven programs is presented in Section 5. In Section 6, we discuss the properties of the method beyond the limitations of our particular implementation. Finally, we relate our contribution to the work by others in Section 7 before we conclude in Section 8.

2. Cycle-Level Symbolic Execution

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

Cycle-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 (Magnusson et al., 1998; Pai et al., 1997). 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 for a given set of input data. Second, and presumably more importantly, when a given path through the program is simulated, all static (i.e., input data independent) information about this path is automatically extracted.

A cycle-level architectural simulator can be seen as an instruction-level simulator connected to a clock-cycle accurate architectural timing model. In this section, we concentrate on instruction-level simulators connected to a timing model that assumes fixed instruction execution-times. First, we present how it can be extended to estimate WCET in Section 2.1. In Section 2.2 we show how the path analysis can be automated using this approach. Finally, in Section 2.3 we discuss implementation issues of the basic approach and how they are addressed.

2.1. The approach

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 symbolically execute the program, which conceptually means that all paths through the program are simulated and in this process infeasible paths are excluded. To do this, we have extended traditional instruction-level simulation techniques with the capability to handle unknown data and with an extended semantics for each data-manipulating instruction to correctly perform arithmetics with the unknown data values as follows.

Each data type is extended with an element denoted *unknown*. In general, the semantics of all arithmetics and logical operations must be redefined to correctly calculate the result if any of the source operands have an *unknown* value. Examples of the extended semantics for some common instruction types can be seen in Table 1. Consider for example an add instruction, ADD T, A, B, that operates on 32-bit unsigned integers, $Z := \{0 \ldots 2^{32} \Leftrightarrow 1\}$ and is defined as $T \leftarrow A + B$. In the extended

semantics, it is instead defined on $\overline{Z} := Z \cup \{unknown\}$ with the semantics seen in Table 1.

Table 1. Extended semantics of instructions.

Instruction type	Example	Semantics
ALU	ADD T,A,B	$T \leftarrow \begin{cases} unknown & \text{if } A = unknown \text{ or } B = unknown \\ A + B & \text{otherwise} \end{cases}$
	AND T,A,B	$T \leftarrow \begin{cases} unknown & \text{if } A = unknown \text{ and } B \neq 0 \\ & \quad \text{or } B = unknown \text{ and } A \neq 0 \\ 0 & \text{if } A = 0 \text{ or } B = 0 \\ A \text{ and } B & \text{otherwise} \end{cases}$
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 determine whether a branch is taken. 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 2.3.)

The load and stores need special treatment, since the reference address used may be *unknown*. For loads, this results in an *unknown* value 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 used by store instructions are statically known. However, we will discuss efficient solutions to overcome this limitation later in Section 2.3.

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. On the other hand, when the branch condition is known, the extended simulation technique will exclude paths that can never be taken. Let's review this path analysis capability in some more detail in the next section.

2.2. Path analysis

To understand how the cycle-level symbolic execution technique can automate path analysis, consider the program in Figure 1 which calculates the sum of all the values in the upper-right triangle of matrix *b*. For simplicity, we reason about the program

```
  int fun(int b[4][4])
  {
1   int i, j, sum = 0;
2
3   for (i = 0 ; i < 4 ; i++)
4     if (b[i][i] > 0)
5       for (j = i+1 ; j < 4 ; j++)
6         if (m[i][j])
7           sum += b[i][j];
8   return sum;
  }
```

Figure 1. Example program.

in a high-level language, even if the symbolic execution of course is done at the instruction level.

In the beginning of the symbolic execution, 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. When analyzing this program, the conditional branch on line 4 will be the only branch depending on unknown values. Consequently, the two possible execution paths originating from this conditional branch have to be simulated. One path continues through lines 3,4, the other one through lines 5,6...3,4, until they hit the same unknown conditional branch again during the second iteration of the outer loop. Continuing this way, $2^4 = 16$ paths will reach the end of the program. WCET is the longest of these paths. The infeasible paths originating from the conditional branch at line 6 are automatically eliminated since the branch condition is determined during simulation. Also, no loop bound annotation for the inner loop is needed since the iteration count only depends on variable i which is also determined during the simulation.

An important advantage of the symbolic execution approach is that all conditions that depend on data values that are known statically will be computed during the simulation. For example, input data independent bounds on the number of loop iterations that are expressed with arbitrarily complex functions are computed automatically. This is not handled by known compiler-based analysis methods, such as used by (Healy et al., 1998). Interestingly, our method also applies to recursive functions provided that the recursion depth is independent of input data. Moreover, many infeasible paths, paths that cannot be executed regardless of the input data, will be excluded from the analysis and will therefore not affect the WCET estimation. Unlike existing compiler approaches, however, our symbolic execution does not terminate unless loop bounds are statically known.

2.3. 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. This can overestimate WCET because we only distinguish between known and unknown values. However, our approach does not hinder us from extending the domain of values to, e.g., intervals. This would handle the mutually excluding paths above. However, our implementation of the method, which we evaluate in this paper, uses the simpler domain. Nevertheless, we will discuss the implications of using other domains in Section 6.

As mentioned above, store instructions with an unknown reference address need special treatment. To efficiently handle these unpredictable accesses, our method identifies all data structures that are accessed with unpredictable memory instructions as *unpredictable data structures*. These are mapped by the linker into a memory area which can only return unknown values. This means that unpredictable accesses can be safely ignored. Predictable stores which access unpredictable data structures are not permitted to change the memory and predictable loads from unpredictable data structures will always return *unknown*, thus assuring a safe 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 manually 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 further discussion about predictable and unpredictable data accesses can be found in (Lundqvist and Stenström, 1999).

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 during 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 is 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 4, two paths can be merged. When merging, the long path, i.e., 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 system 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 system-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, system states of the merged paths are unioned. The union operation between values is defined as:

$$a \cup b \Leftrightarrow \begin{cases} \mathit{unknown} & \text{if } a = \mathit{unknown} \text{ or } b = \mathit{unknown} \\ \mathit{unknown} & \text{if } a \neq b \\ a & \text{otherwise} \end{cases}$$

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 $R[i]$ **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 all the paths should be simulated. Consider a loop with two feasible paths in each iteration. In order to merge these paths, they must have been simulated the same number of iterations. To accomplish this, the WCET algorithm needs 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*. 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 the simulation is continued. The progress of a path is a record of how many times the simulation of that path has passed each loop header and entered each function, as well as how far the simulation has proceeded in the current loop iteration, as dictated by the program counter.

By always selecting the path that has made least progress, the algorithm makes it certain that all paths in a loop iteration are 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. Time-complexity considerations

The time complexity of this method depends on how many paths need to be simulated, how often merging is done and how fast the actual merge operation is. How often the merging should be done involves an important tradeoff. If merging is done too often, it is likely that there will be fewer paths to simulate. However, when

Algorithm 2 WCET algorithm handling merge.

```
{A is set of active paths, C completed paths, ∅ = empty set, \ = set minus}
A ← ∅, C ← ∅

p ← starting null path
Simulate(p)
if p reached end of program then
  C ← C ∪ {p}
else {p reached an unknown conditional branch}
  A ← A ∪ {p}
end if

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

wcet ← max_{p∈C} p.wcet
```

doing a merge, information can be lost because variables whose values differ are assigned *unknown*. This may result in infeasible program paths not being eliminated which in turn increases the number of simulated paths. Thus, merging less often can actually lead to a fewer number of simulated paths.

The algorithm we have implemented 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. However, it is possible to speed up this operation considerably by utilizing the fact that paths that are to be merged, have 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, it is possible to quickly identify the parts of memory 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 chunks (say 512 bytes each) and each path only keeps the accessed chunks in its system state. In this way, only a few chunks of memory need to be compared during a merge operation.

4. Timing Analysis

The WCET algorithm in the previous section can estimate WCET for hardware platforms with fixed instruction execution times. Thus, an instruction-level simulation model extended to symbolically execute a program with unknown input data suffices. In this section, we extend this method to perform cycle-level symbolic execution in order to model the timing of high-performance processors employing multiple-issue instruction execution and instruction and data caching.

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 (Hennessy and Patterson, 1996) on the execution time. With this capability, it is possible to make an arbitrarily accurate estimation of the WCET of a given path through the program.

A critical issue is how to carry out a merge operation. To do this, the method must estimate the impact of the system state on the future execution time. Such state information is exemplified by the identity of the blocks contained in the caches, which affects future misses, and the resources occupied by an instruction, such

as data-path components and registers, which affects future structural and data hazards. The merge 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 caches, *unknown* means that a cache block is invalid, and the unioning of two cache timing states makes all cache blocks whose identities (i.e., memory tags) differ invalid. We call this method the *pessimistic merge* since it can incur a severe pessimism in the estimation. Fortunately, a method will be presented in Section 4.1 that avoids this in many cases. We will concretely apply this method to a high-performance processor whose timing model is introduced in Section 4.2. We then explain in detail how the method is applied to model caching and pipelining in Sections 4.3 and 4.4, respectively.

4.1. Optimistic merge approach

Our approach to reduce the pessimism caused by merging, is based on the idea that if it was known in advance which partial path belongs to the worst-case path through the program, one could update the worst-case execution time with the execution time of that partial path and also choose the timing state of that path when merging and then 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 of the long path and $WCET_L$ 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, worst-case penalty, and pessimistic merging in the context of caches and pipelining. We will make these definitions in the context of a high-performance processor that uses many realistic features and which is presented in the next section.

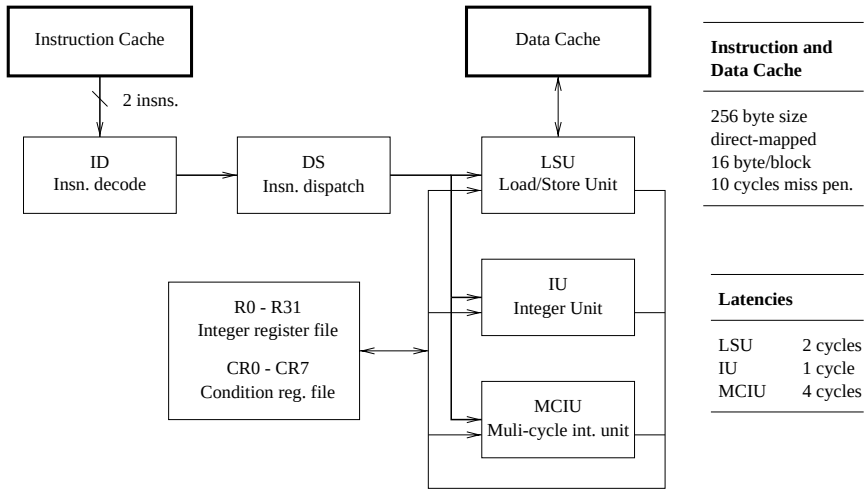

Figure 2. Example architecture.

4.2. Modeled architecture

The architecture used to demonstrate the timing analysis can be seen in Figure 2. It consists of a multiple-issue pipeline, capable of dispatching two instructions each clock cycle, and separate instruction and data caches. While the architecture is a subset of the PowerPC instruction set architecture — floating-point instructions are excluded — it nevertheless contains many of the key features critical for high-performance processors such as pipelining and caching.

Instructions are fetched from the instruction cache and put into the buffers in the *instruction decode* (ID) stage. From the decode stage, instructions are sent to the *dispatch stage* (DS) which in turn dispatches instructions to the three different functional units: the *load/store* unit (LSU), the *integer* unit (IU), and the *multiple-cycle integer* unit (MCIU).

At most two instructions in each cycle can be fetched from the instruction cache and put into the buffers of the ID-stage. A branch is handled ideally by not incurring any penalty, i.e., instructions will be fetched from the correct branch target in zero cycles. Instruction fetching will stall if the buffers of the ID-stage are full or if the fetch causes an instruction cache miss.

In the DS-stage, zero, one, or two instructions are dispatched each cycle in program order. An instruction can not be dispatched if it needs a resource which is currently busy or if an older instruction has not dispatched. Busy resources can be functional units (structural hazards) or registers (data hazards). An instruction reserves its destination registers and if later instructions use this register they are stalled until the first instruction completes and releases its destination register. Instructions in the pipeline are moved forward as soon as possible; if only one instruction is dispatched, instructions still advance so that at least one instruc-

tion is fetched from the instruction cache in order for the pipeline to dispatch two instructions during the next cycle.

All operands needed by an instruction are read from the register files or forwarded from another functional unit at dispatch time. There is one result-bus from each unit handling the write-back of data into the register file and the forwarding of data to another unit.

The load/store unit handles all loads and stores in an equivalent manner as far as timing is concerned. A load or store access that misses the cache causes a new entry (tag and block) to be allocated in the cache. Only one load or store is processed at a time. Normally, all LSU operations have a latency of 2 cycles, but if the access misses the data cache, the LSU unit will be busy during the fetching of the data including the data cache miss penalty.

The integer unit handles all single-cycle ALU operations in addition to the branch instructions. The multiple-cycle integer unit handles long-latency operations such as multiply and divide in addition to instructions involving any special purpose register in the PowerPC instruction set. All multiple-cycle instructions have a latency of 4 cycles. Each instruction thus has a fixed pipeline latency.

To keep the discussion simple, no contention exists when reading and writing in the register file or on the result buses. Also, the timing of all features external to the model (access to main memory) is assumed to be ideal, i.e., no contention exists between fetching data to the instruction cache and to or from the data cache. However, the model introduced accurately accounts for the overlap of simultaneous long-latency operations such as pipeline stalls and cache misses.

4.3. Instruction and data cache analysis

In the following, we first describe how to derive the worst-case penalty as introduced in Section 4.1 for caches. Also, in order to handle the case when it is not possible to discard the cache state of the short path, we need to define how to do a pessimistic merge between two cache states. Both instruction and data caches can be described and treated using the same principles. Therefore, we only treat how to handle instruction caching in this section. We will begin focusing only on direct-mapped caches before extending the method to handle set-associative caches.

Consider the timing state of two instruction caches, IC_L and IC_S, associated with the long and short path, respectively. A cache state is represented by an array of block identities, tags, showing how blocks are currently mapped in the cache (see the example in Figure 3a). To estimate the impact on the future execution time of discarding IC_S, we must consider all cases where IC_S may lead to a greater number of future cache misses than IC_L would. The worst case is found if we imagine that all cache blocks resident in IC_L but not found in IC_S will be needed in the future. We would then lose all these potential misses if we discard IC_S. To find the total possible loss of execution time when discarding IC_S, we go through all cache blocks and compare the tag in IC_S with the tag in IC_L and determine the number of entries where they differ. This number is then multiplied by the cache miss penalty to form the worst-case execution penalty.

Formally, the worst-case penalty for a direct-mapped instruction cache, $WCET_{IC}$, can be expressed as:

$$WCET_{IC} = P_{IC} \sum_{i=0}^{n-1} c(i)$$

where P_{IC} is the instruction cache miss penalty, n is the number of cache blocks, and $c(i)$ is defined by:

$$c(i) = \begin{cases} 1 & \text{if } IC_L(i) \neq IC_S(i) \text{ and } IC_L(i) \neq \text{invalid} \\ 0 & \text{otherwise} \end{cases}$$

where $IC_L(i)$ and $IC_S(i)$ is the cache tag for block i in the long and short path, respectively. We get a contribution to the worst case penalty if the tags differ between cache blocks in the long and short path, but not if the cache block in the long path is invalid. Then, it is impossible for the state in the short path to cause any additional misses in the future.

To derive the worst case penalty for set-associative caches, we need to first redefine the timing state of a cache. For an m-way set-associative cache with n sets, the timing state can be represented by a 2-dimensional array, IC, with size $n \times m$. The tag $IC(s, i)$ is then tag number i in set s. If we assume a LRU replacement policy, i.e., the least recently used block is replaced when a cache miss occurs, then the ordering of tags in a set reflects their relative LRU status, i.e., tag $i = 0$ belongs to the most recently used block and $i = m \Leftrightarrow 1$ to the least recently used block, which is the one to replace next.

Consider again the timing state of two instruction caches, IC_L and IC_S, associated with the long and short path, respectively. To estimate the impact on the future execution time of discarding IC_S, we must consider all cases where IC_S may lead to a greater number of future cache misses than IC_L would. This can happen if a cache block only is resident in IC_L and not in IC_S, or if it also is resident in IC_S but could get replaced earlier than in IC_L. A block, b, can be replaced earlier in IC_S than in IC_L if there exists another more recent block in IC_S that does not exist among the younger blocks in IC_L. In other words, all blocks younger than b in IC_S must also exist in IC_L and be younger than b in IC_L, in order for block b to survive in the worst case.

Formally, the worst-case penalty for a m-way set-associative instruction cache, $WCET_{IC}$, can be expressed as:

$$WCET_{IC} = P_{IC} \sum_{s=0}^{n-1} \sum_{i=0}^{m-1} c(s, i)$$

where P_{IC} is the instruction cache miss penalty, n is the number of sets, and $c(s, i)$ is defined by:

X = Invalid

	IC_L		IC_S
0	0	0	0
1	1	1	1
2	2	2	2
3	19	3	3
4	20	4	4
5	21	5	5
6	X	6	6
7	X	7	7
8	X	8	8
9	X	9	9
10	10	10	10
11	11	11	11
12	12	12	12
13	X	13	X
14	30	14	X
15	31	15	X

(a)

Resource	Release Time	Stall Time
ID0	120	0
ID1	120	0
DS0	121	0
DS1	121	0
LSU	123	1
IU	122	0
MCIU	124	2
R0	124	2
R1	122	0
R2	123	1
R3	60	0
...		
R31	100	0
CR0	109	0
CR1	109	0
...		
CR7	109	0

PL_L

Resource	Release Time	Stall Time
ID0	100	0
ID1	100	0
DS0	101	0
DS1	101	0
LSU	113	11
IU	102	0
MCIU	94	0
R0	94	0
R1	102	0
R2	113	11
R3	60	0
...		
R31	100	0
CR0	89	0
CR1	89	0
...		
CR7	89	0

PL_S

(b)

Figure 3. Example of (a) two instruction cache states and (b) two pipeline states.

$$c(s,i) = \begin{cases} 1 & \text{if there exists no block } k \text{ so that } IC_L(s,i) = IC_S(s,k) \\ & \quad \text{and } IC_L(s,i) \neq \text{invalid} \\ 1 & \text{if there exists a block } k \text{ so that } IC_L(s,i) = IC_S(s,k) \\ & \quad \text{and } IC_L(s,i) \neq \text{invalid} \\ & \quad \text{and if there also exists a block } g < k \text{ such that} \\ & \quad\quad IC_S(s,g) \neq IC_L(s,h) \text{ for all } h < i \\ 0 & \text{otherwise} \end{cases}$$

It should be noted that the $WCET_{IC}$ expression above is quite pessimistic. In reality, it may be impossible to replace combinations of blocks only in IC_S without replacing them in IC_L as well. The evaluation of the impact of this pessimism is outside the scope of this paper.

The pessimistic merging is done by invalidating cache blocks in the cache associated with the long path which are not already invalid and contribute to the worst case penalty. In this way, $WCET_{IC}$ can be reduced.

4.4. Pipeline analysis

Because it is well known how to take into account arbitrarily detailed timing models of pipelines in an architectural simulator (Pai et al., 1997), we will here focus on an approach to define the timing state needed to make an optimistic (or pessimistic) merge operation.

The pipeline will incur penalties caused by resource contention (structural hazards) and register dependencies (data hazards). For our pipeline model, resource contention and dependencies can be modeled using pipeline reservation tables, which record when each resource (pipeline stage or register) is released. An example is seen in Figure 3b. The ID- and DS-stages are divided into two sub-stages, ID0, ID1 and DS0, DS1, since each stage can hold two instructions at a time and each pipeline-stage buffer is treated as an individual resource. If two instructions are present in the pipeline stage, then both sub-stages are reserved. If only one instruction is present, then only ID1 (or DS1) is reserved.

During the simulation, the reservation table can be updated for each instruction at a time because all resources that the instruction requires are known. First, the instruction and data cache accesses are simulated. Then, the pipeline reservation table is updated to show when each resource is released. The current WCET of the path, i.e., the one used when merging, is updated to reflect the earliest point in time when it can be determined when each resource is released. Since this is the time when the last instruction is fetched we always keep WCET updated to this time. When a path through a program has been simulated the estimated WCET is the execution time of that path, defined as the time when the last resource is released in order to make a safe estimation.

Consider now the timing state of two pipelines, PL_L and PL_S, belonging to the long and short path, respectively (see the example in Figure 3b). We want to estimate the possible future effect on the execution time that PL_S may lead to compared to PL_L, and must in this case consider all future hazards that PL_S can lead to, which PL_L cannot lead to. For each pipeline stage and register we determine whether PL_S can make future instructions to stall for a longer time than PL_L would. The maximum difference in stall delay found is the worst-case penalty.

Formally, the worst case penalty for the pipeline, $WCET_{PL}$, is:

$$WCET_{PL} = \max_{r \in R}(s_S(r) \Leftrightarrow s_L(r))$$

where R is the set of all resources in the pipeline reservation table and $s_S(r)$ and $s_L(r)$ are the future stall times that resource r may cause based on the reservation tables of the short and long path, respectively.

To find the future stall time $s_S(r)$ (or $s_L(r)$) we simply imagine an instruction that uses only resource r, besides the ID and DS stages, and determine if it gets stalled if executed as the next instruction in the pipeline. However, since we model a dual-issue pipeline it is not enough using only one future instruction. To correctly handle the sub-stages of ID and DS, we must imagine two future instructions and choose the maximum stall time of the two instructions. An example of the stall times can be seen in Figure 3b. Using the stall times in the example we get: $WCET_{PL} = 10$.

If we are not allowed to discard PL_S, we must do the pessimistic merge instead. To reduce the worst case penalty for the pipeline we must identify the resources which contribute to $WCET_{PL}$. Then, by increasing the release-time entries in the pipeline reservation table for the long path, we can reduce $WCET_{PL}$.

4.5. Pessimistic merging

It is now possible to describe how the pessimistic merging approach can be done so that the instruction cache, data cache, and pipeline are taken into account altogether, and not as separate entities.

As described in Section 4.1 the merging is determined by the following optimistic merge condition, where $WCET_P$ is the worst case penalty and $WCET_L$ and $WCET_S$ are the WCET of the long and short path, respectively.

$$WCET_L \geq WCET_S + WCET_P$$

If this condition is true we can do an optimistic merge and discard the timing state of the short path and continue the simulation from the timing state of the long path. Otherwise, we must do a pessimistic merge. Before we describe how to do pessimistic merging for the complete system, we must define $WCET_P$, $WCET_L$, and $WCET_S$ for the entire high-performance system.

The worst case penalty, $WCET_P$, is the sum of the penalties for the instruction cache, data cache, and the pipeline:

$$WCET_P = WCET_{IC} + WCET_{DC} + WCET_{PL}$$

This is a safe, although pessimistic, estimate of the worst-case penalty that assumes that no overlap exists between future pipeline stalls, instruction, and data cache misses. Note, however, that this is only used as a merge condition; simulation of each path will accurately account for the overlap of long-latency events.

The pessimistic merge is done by modifying the timing state in order to reduce the worst-case penalty and make the optimistic merge condition true. We have chosen the following procedure:

1. Reduce the worst case penalty of the instruction cache ($WCET_{IC}$).
2. If the merge condition still is false reduce the worst case penalty of the data cache ($WCET_{DC}$).
3. If the merge condition still is false reduce the worst case penalty of the pipeline ($WCET_{PL}$).

When doing the pessimistic merge for the caches or the pipeline, it is not always necessary to reduce the worst-case penalty to zero. For example, when invalidating blocks in the instruction cache we can stop if the merge condition becomes true. In this way, as little pessimism as needed is introduced in the timing state. Fortunately, as we will see in the next section, optimistic merging is done in most cases.

5. Experimental Results

In order to evaluate the analysis accuracy of our cycle-level symbolic execution method, we have estimated the WCET of seven programs run on two different

Table 2. Characteristics of the programs used.

Name	Description
matmult	Multiplies two 50x50 matrices
bsort	Bubblesort of 100 integers
isort	Insertsort of 10 integers
fib	Calculates n:th element of the Fibonacci sequence for $n \leq 30$
DES	Encrypts 64-bit data
jfdctint	Does a discrete cosine transform of an 8x8 pixel image
compress	Compresses 50 bytes of data (downscaled version of compress from SPEC CPU95 benchmark suite)

processor architectures making it possible to evaluate the path and timing analysis capability in isolation. While both architectures support the same instruction set — a subset of PowerPC — the first one assumes that all instructions execute in a single cycle with a zero-cycle memory access time. In contrast, to focus on the accuracy of the timing analysis, the timing model of the second architecture corresponds to the system presented in Section 4. The timing parameters assumed for the caches and the latencies in the pipeline stages and functional units are depicted in Figure 2.

5.1. Methodology: Systems, test programs, and metrics

The WCET simulation method has been implemented by extending an existing instruction-level simulator, PSIM (Cagney, 1994-1996), with the capability of handling unknown values to allow for symbolic execution of all the programs. The WCET algorithm described in Section 3.2 has been added to control the path exploration and merging algorithms.

The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the programs. 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 programs.

An overview of the seven 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*.

In order to make a useful comparison how good estimates our method produces, we need to calculate the actual WCET of each program. The actual WCET has been determined by running the programs on the simulator with the worst case input data. This is straightforward to determine 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.

Another evaluation issue is that two of the programs, *fib* and *compress* actually have a termination condition that depends on input data. Therefore, we need to

Table 3. The estimated WCET using the ideal architecture.

Program	Actual WCET	Estimated WCET	Ratio	Structural WCET	Ratio
matmult	7063912	7063912	1	7063912	1
bsort	292026	292026	1	572920	1.96
isort	2594	2594	1	4430	1.71
fib	697	697	1	697	1
DES	118675	118675	1	119877	1.01
jfdctint	6010	6010	1	6010	1
compress	9380	49046	5.2	161161	17.2

bound the number of iterations to make WCET statically decidable. A common approach is 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 have 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 pessimistic bound, but we have found it difficult to prove that a tighter bound could be used. The inner loop implements a secondary probe in a hash table and the number of iterations depends on unknown input data in a complex manner.

5.2. Path analysis results

In this section we evaluate to what extent our method manages to extract path information such as loop bounds and to what extent it manages to exclude infeasible paths from the analysis. We do this by using the idealized PowerPC architecture where each instruction executes in a single cycle.

In Table 3, we show three WCET numbers for each program: actual, estimated, and structural. The actual WCET is our measurement of the actual WCET as explained in Section 5.1. The estimated WCET is the WCET determined by the method and the structural WCET is the execution time of the longest structural path, including all infeasible paths, in the control flow graph of the program when using fixed bounds on the number of iterations of all loops. This number represents what a WCET method would estimate that does not eliminate any infeasible paths and uses fixed iteration bounds for all loops. The purpose of doing this is to analyze the capability of the method to eliminate infeasible paths. In the table, WCET is expressed in clock cycles and ratio is the estimated (or structural) WCET divided by the actual WCET.

For all benchmarks, except *compress*, we can see that the method succeeds in finding the actual WCET. In *compress*, the overestimation is caused by the inner loop. As mentioned previously, we bound this loop using the pessimistic condition $j \leq i$, but during a normal run, we have found that this inner loop is actually only

doing one single iteration. It should be mentioned that for *compress*, we do not know if the WCET we determined is the actual WCET. We suspect that the actual WCET we use is lower than the real one.

Two of the programs, *matmult* and *jfdctint*, have no infeasible paths at all, and only one path is simulated. In *DES*, however, there exist infeasible paths caused by data dependencies between different functions. These infeasible paths are eliminated by our method and only one path is simulated. In *bsort* and *isort*, all infeasible paths were not eliminated. Still, this does not lead to any overestimation, since all simulated infeasible paths are shorter than the worst-case path that the method finds.

If we take a look at the estimated structural WCET of the programs, we see that the WCET is grossly overestimated 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 bound by the current iteration count of the outer loop. This leads to an overestimation of a factor of two of the execution time for the loop and affects *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 overestimation of a factor of two. In addition, there exists a very long infeasible path that extends the structural estimate. As for *DES*, the tiny overestimation results from infeasible paths. As shown in Table 3, our method successfully manages to extract the loop bounds and eliminates the infeasible paths automatically.

A strength of doing the analysis on the instruction level has been revealed in *DES*. In the source code, one can find several conditional expressions which seem to 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. This is detected by our method.

5.3. Timing analysis results

In this section, we analyze how well our integrated path and timing analysis method manages to estimate the WCET of each program. We now assume the detailed timing model of the PowerPC architecture discussed in Section 4.2 with the timing parameters according to Figure 2. Table 4 shows three WCET numbers: actual, estimated, and conservative. Again, actual is the actual WCET we have determined whereas estimated WCET is the WCET determined by our method. Finally, conservative WCET corresponds to the estimated WCET when caches are turned off and each instruction proceeds through the pipeline one at a time. Finally, the two ratios shown correspond to estimated and conservative WCETs divided by the actual WCET, respectively.

Starting with all applications except *DES* and *compress* we note that our method manages to make an **exact** estimate of the actual WCET. This is somewhat unexpected even if these same programs were perfectly analyzed with respect to path properties. The expected added complication now stems from two sources: pes-

Table 4. The estimated WCET with caching and pipelining enabled. The ratios for the cache-all versions of *DES* and *compress* include the slowdown resulting from not caching accesses to unpredictable data structures.

Program	Actual WCET	D-cache Miss-rate	Estimated WCET	Ratio	Conservative WCET	Ratio
matmult	9715029	21.2 %	9715029	1	89741658	9.2
bsort	387331	3.0 %	387331	1	3474229	9.0
isort	3614	2.1 %	3614	1	38208	10.6
fib	1367	3.5 %	1367	1	12072	8.8
DES (cache pred)	323898	15.6 %	323898	1	1810204	5.6
DES (cache all)	323276	15.5 %		1.002		5.6
jfdctint	15276	8.0 %	15276	1	97213	6.4
compress (cache pred)	32235	53.4 %	103901	3.22	638265	19.8
compress (cache all)	27345	32.0 %		3.80		23.3

simistic merging of the timing state and the use of a data cache. We will now take a closer look on these sources in order to understand the reason for the good result.

The first reason for the good result when doing timing analysis is that no pessimistic timing merge was performed at all during the analysis. In *matmult*, *DES*, and *jfdctint*, no merge at all was needed since only one feasible path was simulated. In *fib*, all paths reached the end of the program before any merge was needed. Finally, in *bsort*, *isort*, and *compress*, only optimistic merges were needed, i.e., two paths that were to be merged always differed enough in length making it possible to discard the timing state of the short path and continue the simulation with the timing state of the long path.

The second issue which can cause overestimation is data caching. If the address of a data reference depends on unknown input data, this reference may in the worst case result in a miss which forces another block to be evicted from the cache. Thus, a safe estimate would be to charge two miss penalties to an unknown data reference. However, all applications, except *DES* and *compress*, contain only predictable data structures meaning that all data references are independent of input data. Thus, data caching is predictable and can be perfectly analyzed by our method.

DES and *compress* contain unpredictable data structures as defined in Section 2.3. In our method, we avoid the pessimism of charging two misses by using an alternative approach. Since unpredictable data structures are mapped into a special area of the memory, as discussed in Section 2.3, we also mark this area as non-cacheable. This is supported by most processors, since memory mapped I/O locations are in general not cached. During analysis we then know that an unknown reference will at most cause a single cache miss. Further discussion about unpredictable data accesses can be found in (Lundqvist and Stenström, 1999). For comparison purposes we have included two versions of *DES* and *compress*: one where only accesses going to predictable data structures are cached — called cache-predictable version — and one where all accesses are cached — called cache-all version. WCET estimation

has only been done for the cache-predictable versions. These numbers are shown in Table 4. The ratios shown for the cache-all versions of *DES* and *compress* are the estimated and conservative WCET for the cache-predictable versions divided by the WCET for the cache-all versions. It should also be mentioned that when data caching is enabled it is hard to find the worst case input data for the cache-all versions of *DES* and *compress*; the data addresses going to the data cache sometimes depend on unknown input data. We have not made any effort to address this problem.

As can be seen from Table 4, an almost perfect estimate of the WCET of *DES* is determined by our method in spite of unpredictable data structures. The numbers for the cache-all versions of *compress* and *DES* reveal to what extent the unpredictable data structures affect the estimation. In *DES*, only 0.6 % of all data accesses are directed to unpredictable data structures resulting in a slight overestimation. On the other hand, in *compress* 34 % of all accesses are directed to unpredictable data structures. By not caching these accesses we increase the execution time (and the overestimation) by 18 %.

The overestimation by a factor of 3.22 in *compress* is due to the inner loop as mentioned Section 5.2. This is lower than the factor of 5.2 previously found for the idealized architecture. The reason for this is that cache misses in the initialization part of the program makes the loop in the middle of the program a little less significant for the total program execution time.

To fully realize the importance of doing timing analysis, we can take a look at the conservative WCET in Table 4. We see that when treating all cache accesses as misses and permitting no pipelined execution we increase the overestimation of the WCET by a factor of between 5.6 and 10.6. In *compress*, for example, we find that the additional overestimation when doing no timing analysis is a factor of 6.1. Additionally, as we saw in the previous section, the overestimation when doing no path analysis (the structural WCET) is a factor of 3.3. Together, we get a factor of 20 in additional overestimation for *compress* when ignoring both path and timing analysis which clearly shows the importance of integrating accurate path and timing analysis.

6. Discussion

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 nested loops where a loop bound depends on the loop iteration variable of an outer loop. Of equal importance is an accurate timing analysis. The potential of detailed timing analysis is especially emphasized when optimistic merge is the common case. Then, as we saw in the previous section, it is possible to derive an exact estimate of the actual WCET. Also, as was illustrated by *DES*, *matmult*, and *jfdctint*, where there is only a single path through the program, this path can be simulated with an arbitrarily detailed timing model and will always give us an estimated WCET with no overestimation. 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 approaches where the path and timing analyses are kept separated. If an automatic 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 automatic path analysis and WCET calculation. These problems are not present in our method, which does the path and timing analysis simultaneously.

Our cycle-level symbolic execution approach uses a domain where values can be either known or unknown. This simple domain performs remarkably well compared to more complex domains, e.g., based on intervals of values which is used in (Ermedahl and Gustafsson, 1997; Altenbernd, 1996). However, as we saw in Section 2.3, 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 simulation that needs 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 heuristics or by user interaction. In these cases, 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;
```

The problem is caused by the simple domain of values. 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. Related Work

We will finally put our method in context of recent work on path and timing analysis methods from the open literature.

Historically, one of the first approaches to provide path information was to require it from the programmer in terms of annotations (Puschner and Koza, 1989). This, of course, requires a considerable programming effort and is error-prone. In terms of automated approaches to path analysis some methods were recently proposed that rely on symbolic execution. Ermedahl and Gustafsson (Ermedahl and Gustafsson, 1997) use a path analysis approach that associates ranges with unknown input data and can use this domain to exclude infeasible paths automatically at the source-code level. Similar techniques that also use symbolic execution to perform timing analysis were proposed by (Chapman et al., 1994; Altenbernd, 1996). They estimate the WCET of programs with fixed execution times for basic code blocks. Like our method, they use symbolic execution but do not apply it on the cycle level. This prevents them from deriving useful estimates of WCET for high-performance processors where instruction execution-times depend on complex interactions between low-level long-latency events such as pipeline stalls and cache misses. The method presented by Altenbernd also suffers from complexity problems in some cases. We use a path merging strategy instead, which guarantees a manageable number of paths.

An alternative method for doing path analysis is to use partial evaluation. This is done by Nirkhe and Pugh (Nirkhe and Pugh, 1993), who use partial evaluation in order to transform a program into a less complex one that may contain fewer infeasible paths and simpler loop bounds. However, they do not attack the problem of doing detailed timing analysis.

Timing analysis approaches for high-performance processors has gained a lot of attention recently (Li et al., 1995; Li et al., 1996; Ottosson and Sjödin, 1997; Lim et al., 1995; Kim et al. 1996; Lim et al. 1998; Healy et al., 1995; White et al., 1997; Alt et al., 1996; Ferdinand and Wilhelm, 1998). Three quite different techniques are used in these approaches.

Integer linear programming and constraint solving is one approach (Li et al., 1995; Li et al., 1996; Ottosson and Sjödin, 1997; Theiling and Ferdinand, 1998). It expresses WCET as a function of constraints on number of times specific paths are executed and the number of long-latency events that result from the execution of each path. Fixed-cycle penalties of cache misses and pipeline stalls are then associated with such events to derive an estimate of the WCET. Such methods provide accurate estimates only if exact path information is provided to express the constraints and if the execution time is a linear function in the number of long-latency operations. Unfortunately, accurate estimates have only been demonstrated for pipelined processors with caches where the overlap of long-latency events is not critical to the accuracy. Moreover, they have not yet demonstrated how to integrate it with an automated approach to provide path information.

The approach taken by (Lim et al., 1995; Kim et al. 1996; Lim et al. 1998) is to extend Shaw's timing schema (Shaw, 1989) to account for long-latency events

in each basic code block. Unfortunately, unlike our method, their extension only approximates the overlap of long-latency events across basic blocks. In addition, since they do not integrate the method with automated path analysis methods, it can result in considerable overestimations due to infeasible paths (Kim et al., 1996).

Abstract interpretation and dataflow analysis is another approach taken (Healy et al., 1995; White et al., 1997; Alt et al., 1996; Ferdinand and Wilhelm, 1998; Theiling and Ferdinand, 1998). Especially in the modeling of caches, dataflow analysis techniques have shown powerful to estimate a tight upper bound on the worst-case number of cache misses (Mueller and Whalley, 1995; Alt et al., 1996; Ferdinand and Wilhelm, 1998; Theiling and Ferdinand, 1998). Moreover, in combination with a reservation-table driven approach to model pipeline hazards, it is possible to model overlap between e.g. pipeline stalls and cache misses (Healy et al., 1995). Unfortunately, these methods also suffer from overestimations of WCET as they lack integration with automatic path analysis methods. In contrast, while we have shown that our cycle-level symbolic execution approach can make as accurate timing analysis as these methods, it also manages to extract important path information automatically. In this paper, we have demonstrated that it can model as aggressive multiple-issue processors as in (Lim et al. 1998) and as fancy cache organizations as in (White et al., 1997). In addition, we do not see any inherent limitation of the method to model multi-level caches as in (Mueller, 1997).

To the best of our knowledge, our method is the first that combines accurate path and timing analysis in one holistic approach.

8. Conclusions

In this paper we have presented a new method for estimating the WCET of a program. This method uses cycle-level symbolic execution to integrate path and timing analysis and thereby has the potential to do tight estimations by eliminating certain classes of infeasible paths and concentrating accurate timing analysis on the feasible ones.

Our results concerning the path analysis aspects show that many infeasible paths indeed are eliminated in the tested programs. In fact, for six out of the seven programs we managed to do a perfect estimation. We also studied how the impact of performance enhancing features in high-performance processors, such as pipelined multiple-issue execution, instruction and data caching, could be incorporated in the symbolic execution method. The critical issue to address is to formulate conditions for the merge operation. We demonstrated how this is done for pipelining and caching. In evaluating the accuracy of the timing analysis we found that we could make an exact estimate of the WCET, again for six out of the seven programs we tested. The overall effect of integrating path and timing analysis was shown to improve the WCET estimation a factor of twenty in some cases over a conservative method.

Acknowledgments

We are deeply indebted to all the people that have provided valuable feedback on earlier versions of this paper: Dr. Jan Jonsson of Chalmers, Prof. Reinhard Wilhelm of the University of Saarland, and all anonymous referees. We also acknowledge the helpful comments given by people involved in the Swedish WCET Network (see `http://www.docs.uu.se/artes/wcet/`). This research is supported by a grant from Swedish Research Council on Engineering Science under contract number 221-96-214.

References

Alt, M., Ferdinand, C., Martin, F., and Wilhelm, R. 1996. Cache behavior prediction by abstract interpretation. In *Proceedings of SAS'96, Static Analysis Symposium*, pp. 52–66.

Altenbernd, P. 1996. On the false path problem in hard real-time programs. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*, pp. 102–107.

Cagney, A. 1994-1996. PSIM, a PowerPC simulator. http://sourceware.cygnus.com/psim/.

Chapman, R., Burns, A., and Wellings, A. 1994 Integrated program proof and worst-case timing analysis of SPARK Ada. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. K1–K11.

Ermedahl, A., and Gustafsson, J. 1997. Deriving annotations for tight calculation of execution time. In *Proceedings of EUROPAR'97*, pp. 1298–1307.

Ferdinand, C., and Wilhelm, R. 1998. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, LNCS 1474, pp. 16–30.

Healy, C., Whalley, D., and Harmon, M. 1995. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 288–297.

Healy, C., Sjödin, M., Rustagi, V., and Whalley, D. 1998. Bounding loop iterations for timing analysis. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium*, pp. 12–21.

Hennessy, J., and Patterson, D. 1996. *Computer Architecture: A Quantitative Approach, 2ed.* Morgan Kaufmann.

Kim, S.-K., Min, S., and Ha, R. 1996. Efficient worst case timing analysis of data caching. In *Proceedings of the 2nd IEEE Real-Time Technology and Applications Symposium*, pp. 230–240.

Li, Y.-T., Malik, S., and Wolfe, A. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pp. 298–307.

Li, Y.-T., Malik, S., and Wolfe, A. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pp. 254–263.

Lim, S.-S., Bae, Y., Jang, G., Rhee, B.-D., Min, S., Park, C., Shin, H., Park, K., and Kim, C. 1995. An accurate worst case timing analysis for RISC processors. In *IEEE Transactions on Software Engineering*, 21(7), pp. 593–604.

Lim, S.-S., Han, J., Kim, J., and Min, S. 1998. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 334–345.

Lundqvist, T., and Stenström, P. 1999. Empirical bounds on data caching in high-performance real-time systems. Technical Report 99–4, Dept. of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden.

Magnusson, P., Dahlgren, F., Grahn, H., Karlsson, M., Larsson, F., Moestedt, A., Nilsson, J., Stenström, P., and Werner, B. 1998. Simics/sun4m: A virtual workstation. In *Proceedings of USENIX98*, pp. 119–130.

Mueller, F. 1997. Timing predictions for multi-level caches. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. 29–36.

Mueller, F., and Whalley, D. 1995. Fast instruction cache analysis via static cache simulation. In *Proceedings of 28th Annual Simulation Symposium*, pp. 105–114.

Nirkhe, V., and Pugh, W. 1993. A partial evaluator for the Maruti hard real-time system. In *Real-Time Systems*, 5(1), pp. 13–30.

Ottosson, G., and Sjödin, M. 1997. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pp. 47–55.

Pai, V., Ranganathan, P., and Adve, S. 1997. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *IEEE Technical Committee on Computer Architecture newsletter*, pp. 32–38.

Puschner, P., and Koza, C. 1989. Calculating the maximum execution time of real-time programs. In *Real-Time Systems*, 1(2), pp. 159–176.

Shaw, A. 1989. Reasoning about time in higher-level language software. In *IEEE Transactions on Software Engineering*, 15(7), pp. 875–889.

Theiling, H., and Ferdinand, C. 1998. Combining abstract interpretation and ILP for microarchitecture modelling and program path analysis. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pp. 144–153.

White, R., Mueller, F., Healy, C., Whalley, D., and Harmon, M. 1997. Timing analysis for data caches and set-associative caches. In *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium*, pp. 192–202.