Timing Anomalies in Dynamically Scheduled Microprocessors

Thomas Lundqvist Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

Abstract

Previous timing analysis methods have assumed that the worst-case instruction execution time necessarily corresponds to the worst-case behavior. We show that this assumption is wrong in dynamically scheduled processors. A cache miss, for example, can in some cases result in a shorter execution time than a cache hit. Many examples of such timing anomalies are provided.

We first provide necessary conditions when timing anomalies can show up and identify what architectural features that may cause such anomalies. We also show that analyzing the effect of these anomalies with known techniques results in prohibitive computational complexities. Instead, we propose some simple code modification techniques to make it impossible for any anomalies to occur. These modifications make it possible to estimate WCET by known techniques. Our evaluation shows that the pessimism imposed by these techniques is fairly limited; it is less than 27 % for the programs in our benchmark suite.

1. Introduction

Estimation of an upper bound on the execution time, called worst-case execution time (WCET), is important for highly dependable real-time systems. Because of pessimistic timing assumptions, WCET is often grossly overestimated which results in poor resource utilization, especially in real-time systems using high-performance processors with advanced pipelining and caching techniques.

WCET is typically estimated as tight as possible by analyzing the WCET of each path in the program—often in combination with heuristics to prune the number of paths to analyze. Moreover, this analysis often proceeds from the first to the last instruction in each path. In doing this, one must take into account that the execution time (latency) of each instruction is not fixed; it can take one of many discrete values depending on input data. The way known methods deal with this problem [2, 3, 4, 5, 6, 7, 8] is to assume the longest instruction latency because the intuition is that this will always result in a conservative estimate of the WCET. For example, if the outcome of a cache access is unknown, a cache miss is assumed.

We show in this paper that this intuition is simply wrong for many processors using dynamic instruction scheduling. Because the instruction schedule depends on the execution time of each individual instruction, the scheduling of future instructions can actually cause a counter-intuitive increase or decrease in the execution time of the rest of the execution path. We will show many examples of such timing anomalies in the paper.

To find a safe estimate of the WCET in the presence of such anomalies, one would have to analyze the effect of all possible schedules resulting from a variable-latency instruction to find the instruction latency that leads to the longest overall execution time. In general, if we have n variable-latency instructions along a path in the program, where each instruction may lead to k different future schedules, then, in the worst case, one must analyze k^n different schedules. We show that previously published analysis methods for cache and pipeline analysis [2, 3, 4, 6, 8] would result in prohibitive computational complexity to analyze these anomalies.

This paper first identifies necessary conditions for when timing anomalies can show up in dynamically scheduled processors and what architectural features may cause them. We then propose some simple code modification techniques that eliminate the existence of timing anomalies, thus enabling known analysis methods to estimate WCET. The main idea exploited is to make program modifications that will guarantee that a future instruction schedule is not affected by a variable-latency instruction. We evaluate the amount of pessimism introduced on a number of benchmark programs by instruction-level simulation and a model of a dynamically scheduled processor. Our main conclusion is that the pessimism introduced by the modifications is fairly limited; it is less than 27 % for the programs in our benchmarks suite.

The rest of the paper is organized as follows. In Sec-

tion 2, we first consider when and how timing anomalies show up in dynamically scheduled processors. In Section 3, we show why previous methods fail to handle these anomalies. The rest of the paper is devoted to our approach to handle the anomalies. We introduce the idea of program modifications in Section 4 which we evaluate experimentally in Section 5. Finally, we discuss our approach and also point out future directions of research in this area in Section 6, before we conclude in Section 7.

2. Timing Anomalies in Processors

In this section, we will give examples of the timing anomalies present in dynamically scheduled processors. But first, we define necessary conditions that can lead to such anomalies. The term *dynamically scheduled processors* is often used to describe a processor for which instructions execute out-of-program-order. In the next section, a first contribution is that we show that it is not the out-of-order execution that is the central issue here. Rather, it is the order in which resources are allocated in the processor.

2.1. Definitions and conditions

The execution time of an instruction can take one of many discrete values depending on input data. One example is a load instruction whose execution time depends on whether the address hits or misses in the cache. Another example is an arithmetic instruction whose execution time may depend on the operands. A common assumption is that if the worst-case instruction execution time is assumed, the WCET estimation will be safe. Throughout this paper, we define a *timing anomaly* as a situation when such assumptions do not hold. For clarity reasons, we will use the term *latency* meaning the instruction execution time. When we use the term *execution time* it will mean the overall execution time of the program.

Consider the execution of a sequence of instructions. Let us study two different cases where the latency of the first instruction is modified. In the first case, the latency is increased by i clock cycles. In the second case, the latency is decreased by d cycles. Let C be the future change in execution time resulting from the increase or decrease of the latency. Then:

Definition 1 *A timing anomaly is a situation where, in the first case, $C > i$ or $C < 0$, or in the second case, $C < -d$ or $C > 0$.*

That is, if C is guaranteed to be in the interval: $0 \leq C \leq i$ in the first case or $-d \leq C \leq 0$ in the second case, we have no timing anomalies.

To model the instruction execution in a pipelined processor, one often uses a resource model. In this model, whenever an instruction that proceeds through a pipeline gets stalled, it is due to resource contention with another instruction that accesses a common resource or operand. Typical examples of resources are functional units and registers, but also buses, read and write ports, and buffers should be treated as resources if they can cause instructions to stall.

The resources that an instruction can use can be divided into *in-order* and *out-of-order resources*. In-order resources can only be allocated in program order to instructions. Out-of-order resources can be allocated to instructions dynamically, i.e., a new instruction can use a resource before an older instruction uses it according to some dynamic scheduling decision. Typical out-of-order resources are functional units that service instructions dynamically (out-of-order initiation). Examples of in-order resources are such registers that must be reserved in-order to guarantee that data dependencies in the program are not violated. Given this definition, it is now possible to state exactly a sufficient condition when a processor is free from anomalies:

Condition 1 *If a processor only contains in-order resources no timing anomalies can occur.*

To see why this condition is sufficient, consider a processor that only contains in-order resources. This means that two instructions can only use a resource in program order. If the completion of an instruction is postponed by i cycles, later instructions will also be postponed since they cannot allocate the resource before the first instruction. However, it is possible that future instructions will be postponed by less than i cycles if the new schedule becomes more compact, i.e., containing less idle time. The amount postponed cannot be less than 0 cycles however. Thus, C will be less than or equal to i and greater than 0. The same principle apply if an instruction is completed d cycles earlier. To conclude, if all resources are in-order no timing anomalies may occur.

2.2. Timing anomaly examples

If out-of-order resources are present, timing anomalies may occur. To see how, we will now study an architecture containing out-of-order resources and give examples of how timing anomalies may occur.

The focus of our study will be the model of an architecture seen in Figure 1 based on a *simplified* PowerPC architecture containing no floating point units. A more realistic model is expected to contain more features that would result in out-of-order resource allocation. Our point is then that even for this simplified architecture, timing anomalies show up.

The architecture consists of a multiple-issue pipeline, capable of dispatching two instructions each clock cycle, and

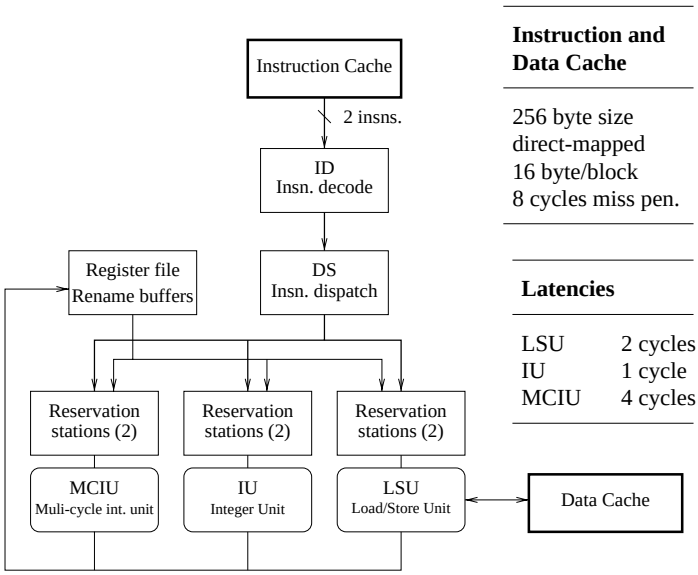
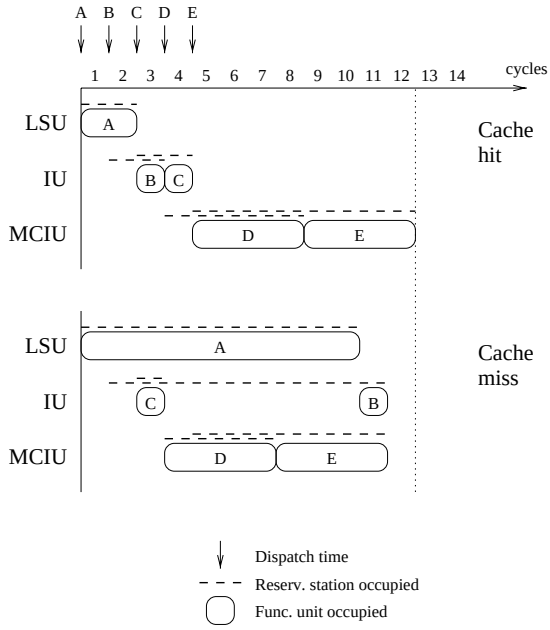


Figure 1. A simplified, yet timing-anomalous, PowerPC architecture.



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	ADD	r5, r4, r4
C	3	ADD	r11, r10, r10
D	4	MUL	r12, r11, r11
E	5	MUL	r13, r12, r12

Figure 2. An example when a cache hit causes a longer execution time than a cache miss.

separate instruction and data caches. To implement out-of-order execution of instructions, each functional unit has two reservation stations. These can hold dispatched instructions before their operands are available. Register renaming is used to avoid unnecessary data hazards. Also needed, but not shown, is a completion unit with a reorder buffer, which completes instructions in-order by updating the register file from the renaming buffers.

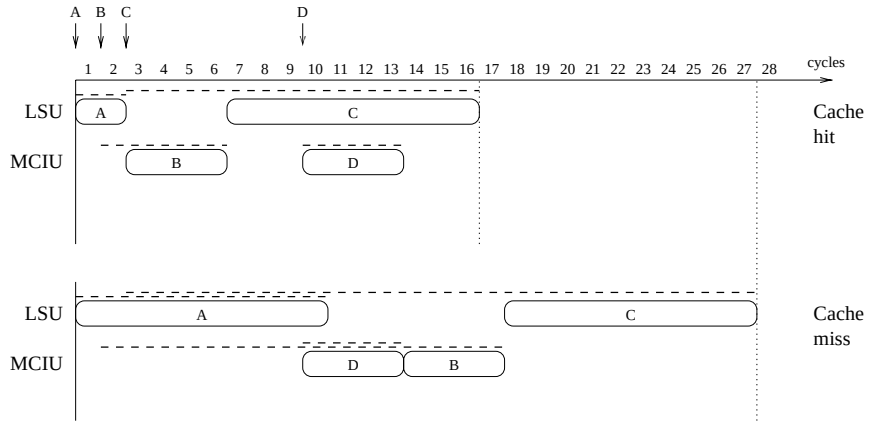
All resources in the modeled processor are considered to be in-order resources except the integer unit (IU) and the multiple-cycle integer unit (MCIU) which are out-of-order resources. The load/store unit (LSU) often initiate execution in-order to preserve ordering of memory accesses so we also treat it as an in-order resource here. The out-of-order resources, IU and MCIU, make timing anomalies possible as we will demonstrate in three examples: one showing that a cache hit may be worse than a cache miss, another showing that the miss penalty can be greater than expected, and a third showing a possible domino effect when executing loops.

Anomaly 1: Cache hits can result in worst-case timing

The first example presents a case where a data cache hit causes an overall longer execution time than a data cache miss. Consider the table in Figure 2, which shows a sequence of instructions (A-E) and in which clock cycle they are dispatched. The instructions represent the use of different functional units: the `LD rd,0(ra)` instruction uses the LSU, the `ADD rd,ra,rb` uses the IU, and the `MUL rd,ra,rb` uses the MCIU. Register `rd` is the destination register and `ra` and `rb` are the source registers. The registers create data dependencies and thereby an ordering between instructions. To simplify the discussion of the examples we focus only on the functional units and their reservation stations. We assume that the instructions are dispatched according to the relative times seen in the instruction table in Figure 2 although in reality, on a dual-issue pipeline, additional instructions would be needed to make the instructions dispatch according to the example.

The diagram in Figure 2 shows when each functional unit is busy executing an instruction. Also shown as horizontal dashed lines is when the reservation stations are occupied. At the top, arrows indicate when each instruction is dispatched to the reservation stations. Two cases can be identified, one when the load address hits in the data cache and one when it misses the cache.

If the load address hits in the cache then the `LD` instruction executes for 2 cycles and can forward its result to instruction B which can start executing in cycle 3. Here, we



Label	Disp. cycle	Instruction	
A	1	LD	r4, 0(r3)
B	2	MUL	r5, r4, r4
C	3	LD	r6, 0(r5)
D	10	MUL	r11, r10, r10

Figure 3. An example when the cache miss penalty is higher than expected.

assume that B gets priority over C since B is older. Thus, C must wait for B. On the other hand, if the load address misses in the cache then the LD instruction executes for 10 cycles and the execution of B will be postponed. This means that C can start executing in cycle 3, one cycle earlier than in the cache hit case. This will make D and E execute one cycle earlier as well, leading to an overall reduction of the execution time by 1 cycle in the cache miss case. In this case, the anomaly is made possible due to the IU being an out-of-order resource permitting B and C to execute out-of-order.

Anomaly 2: Miss penalties can be higher than expected

The second example shows that the overall penalty in execution time due to a cache miss can be higher than the normal cache miss penalty. Consider the instruction sequence in Figure 3. The first instruction is a load instruction which can either hit or miss in the cache. We assume that the second load instruction (C) always misses. The first three instructions: A, B, and C, depend on each other and must execute one at a time. In the cache hit case all instructions will execute as soon as possible. The last instruction, D, will not interfere with the execution of the other instructions.

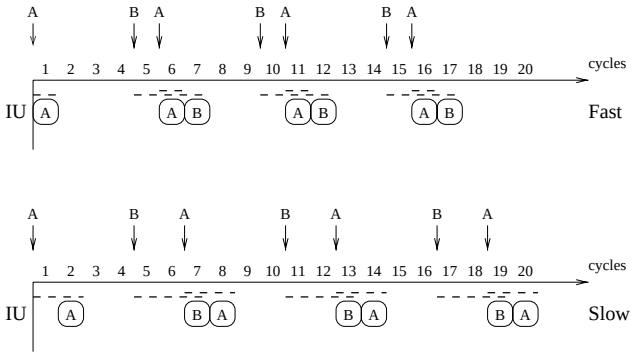
If the first load experiences a cache miss, the execution of B will be postponed. In this unfortunate case, instruction D has already started when B becomes eligible for execution and B will be further postponed. The result of this is that instruction C will finish executing 11 clock cycles later in the cache miss case as compared with the cache hit case. This is greater than the normal cache miss penalty of 8 clock cycles. In this case, the anomaly is due to the MCIU being an out-of-order resource, which allows instruction B and D to execute in arbitrary order.

Anomaly 3: Impact on WCET may not be bounded

We saw in the previous example how the total penalty of a cache miss can be increased due to changes in the instruction schedule. However, it is bounded by a constant value. We will now show an example when the increase is not necessarily limited by a constant value, but can be proportional to the length of the program. This means that a small interference in the beginning of the execution may contribute with an arbitrarily high penalty to the overall execution time.

Consider the instruction sequence in Figure 4. The two instructions A and B constitute the body of a loop doing a number of iterations. The delicate execution scenario shown here demands special requirements on the dispatch and execute cycles. Therefore, the table entry for the dispatch clock cycle and the additional table entry for the execute clock cycle show the dispatch and execute clock cycle relative to a previous instruction. By E_A we mean the clock cycle when A executed in the previous iteration of the loop. By D_A we mean the clock cycle when A was dispatched in the current loop iteration.

The two different scenarios shown in Figure 4 are the result of dispatching and executing the two instructions A and B repeatedly according to the dispatch and execute cycle rules starting from two different executions of the first A instruction. In the fast case, instruction A in the first iteration executes immediately when it is dispatched. In the slow case, we imagine that it gets delayed one clock cycle because of a dependency with an earlier instruction. This delay in the beginning is enough to cause a domino effect that will delay the execution of A by one clock cycle in each iteration. The total penalty on the execution time, caused by the small delay of A in the beginning, will be k clock cycles if the loop does k iterations. In the slow case, we assume that the old B instruction gets priority over the new A in-



Label	Disp. cycle	Execute cycle	Instruction	
A	$E_A + 5$	Immediate	ADD	r4, r3, r3
B	$D_A + 4$	$D_A + 6$	ADD	r11, r10, r10

Figure 4. Example of domino effects.

struction in each iteration.

In summary, we have shown three examples when timing anomalies may show up in dynamically scheduled processors. These anomalies were possible due to the presence of out-of-order resources. The first two examples show that worst-case instruction execution assumptions may result in optimistic estimates of the WCET if the future scheduling is not taken into account. It is not difficult to construct other instruction sequences where similar anomalies appear. While the last example shows a presumably rare event, it emphasizes that it may not be safe to make assumptions regarding timing on the instruction level.

3. Limitations of Previous Methods

In the previous section we have seen that timing anomalies may occur in dynamically scheduled processors. To correctly estimate the WCET, one would have to consider the effect all variations in instruction execution times have on the possible instruction schedules. We will now consider the problems that arise if we want to perform accurate pipeline analysis for dynamically scheduled processors and how previous methods fail to handle these problems. To simplify the discussion, we will use the following definitions:

Definition 2 *The current pipeline state is the current state of the pipeline timing model. It describes which instructions are currently executing in the pipeline and the current resource allocations.*

Definition 3 *The current cache state is the current content of the cache timing model. It consists of the cache tag memory, i.e., the identification tags of the current blocks in the cache.*

Consider first a program containing only a single feasible path. The WCET is then the longest execution time of the instruction sequence along this path. Assume that the sequence contains n variable-latency instructions with unknown latencies, but we know that each instruction can have k different latencies. Then, we must for each variable-latency instruction find the latency that causes the longest overall execution time. To be safe, we must examine k^n instruction schedules because the execution of each variable-latency instruction may cause k schedules of all succeeding instructions.

In general, analyzing all k^n combinations is not feasible and another approach is needed. Normally, timing analysis methods rely on the possibility of making safe decisions locally at the instruction or basic block level. That is, a pessimistic choice is always made at this level. Unfortunately, due to the anomalies, we cannot make a local safe decision. Consider a partial sequence of instructions, e.g., a basic block, containing a variable-latency instruction. When simulating the execution of this partial sequence in the pipeline we may end up with k different pipeline states. To be safe, we must then choose the pipeline state that will give us the longest overall execution time. But this is impossible without knowledge of the whole instruction sequence.

All previously presented methods for doing cache and pipeline analysis [2, 3, 4, 5, 6, 7, 8] perform the pipeline analysis by first looking at each instruction or basic block and then combining the WCET of all these entities into a total WCET for the whole program. While none of these methods are designed to handle dynamically scheduled processors, they nevertheless rely on a capability to make local safe decisions when regarding variable-latency instructions. For example, in [2, 8] the cache analysis is done first and then later used in a pipeline analysis step. Whenever it is not possible to classify a cache access as a hit or a miss, it is conservatively assumed to be a miss. This may lead to a too optimistic estimation as we have seen in the first anomaly example according to Figure 2.

Consider next a program containing several feasible paths. The WCET is then the maximum WCET found among all paths and in order to find the WCET we would have to examine all paths in the program. This is, in general, not feasible and timing analysis methods again rely on the possibility of making local safe decisions to reduce the complexity. When analyzing a small section of the program, e.g., a loop, the longest path in this section is chosen before doing the analysis of the rest of the program. Unfortunately, due to the anomalies, it is not possible to make local safe decisions. To see this, assume that the small section contains l different paths. When simulating the execution of the different paths in the pipeline we may end up with l different pipeline states, leading to the same problem as for the variable-latency instructions. It is not possible to know

which pipeline state (path) that gives us the longest overall execution time.

An example of when local decisions are used to reduce the path complexity is the prune operation used in [4, 5]. It is used to discard some combinations of basic blocks that will execute in shorter time than another combination of blocks found. To make this pruning decision, one must know how the execution of some basic blocks will influence the execution of other parts in the program. Due to, e.g., the domino anomaly (Figure 4), this can be difficult or even impossible. The same problem exists in [2] where the longest path is chosen in each iteration of a loop.

To conclude, when doing timing analysis in the presence of timing anomalies, it is not possible to make safe local decisions, i.e., safe choices between the different pipeline states that an unknown event may give rise to.

Fortunately, we will in the next section show two approaches that can make it possible for previously published timing analysis methods to handle dynamically scheduled processors.

4. Methods for Elimination of Anomalies

In this section, we will present two new approaches to estimating the WCET of a program running on a dynamically scheduled processor where we might experience timing anomalies. Both approaches can be used together with previously published timing analysis methods. We first present the serial-execution method, a pessimistic but safe method to handle architectures with timing anomalies. After this, we present a method based on program modifications—by modifying the program we make it possible for timing analysis methods to rely on safe local decisions. At the end of this section, we present a case study of how the program-modification method can be used together with our previously published method based on cycle-level symbolic execution [6]. We only focus on instruction and data cache analysis and the out-of-order resource use of the functional units.

4.1. The pessimistic serial-execution method

A straight-forward way to make safe estimations for architectures containing anomalies is to use the pessimistic serial-execution estimate. This means that we model all instructions as being executed in-order in the functional units. That is, we sum all instruction latencies in the functional units. In addition to this, we add the miss penalties for all instruction and data cache misses. We now formulate a claim that needs to be proven although intuitive in nature.

Claim: The WCET corresponding to a serial execution of the instructions, assuming their worst-case latencies, is always higher than the WCET corresponding to any pipelined execution of the same instruction sequence.

Proof: Instructions can not execute slower than in-order since this would mean that some functional units are idle sometime. This can not be true since instructions are always available for execution. The only possibility for an instruction to stall is cache misses which we add separately.

The serial-execution estimate will be safe but maybe too pessimistic. A big advantage, however, is that unknown events in the system are handled in a safe way. They can not lead to a greater execution time than the one estimated for serial execution.

4.2. The program modification method

The serial-execution method is very pessimistic. If we want a tighter estimated WCET we must model the pipelined execution accurately and deal with the problem of timing anomalies. One way of accomplishing this is to modify the program so that we can rely on safe local decisions. In short, we want to make sure that the following conditions are true:

1. All variable-latency instructions that have an unknown latency must, when simulated, still result in a predictable pipeline state. Also, we must make sure that the worst-case latency is used for the instruction. In addition, other unknown events such as unknown instruction cache accesses must also result in a predictable pipeline state.

2. If the number of paths in a small section of the program is being reduced by selecting the longest one or discarding the shortest ones, then the state of the pipeline and the caches at the beginning and the end of the paths must not differ when comparing them.

One way of fulfilling the first condition is to force an in-order resource use when executing the variable-latency instruction. Then, the pipeline state must be predictable before allowing out-of-order resource use again. The way to accomplish this is highly architecture dependent. Unfortunately, no support for in-order resource scheduling is present in processors today, but other instructions may be used for this purpose. For example, in the POWERPC architecture, there is a memory synchronization instruction called `sync`, which inhibits further dispatching until the `sync` instruction completes. This instruction can be used as a way to force serialization together with a variable-latency instruction.

If one `sync` is placed after the variable-latency instruction then the pipeline state will be known afterwards. If one `sync` is placed before the variable-latency instruction we will know for sure that the instruction will execute in-order

and the maximum latency will be the worst-case latency. Also, for other unknown events, like an unknown instruction cache access, we can also use the same method to make the pipeline state predictable. In the rest of this paper we will assume that an instruction such as `sync` exists.

To fulfill the second condition above we can again use the `sync` instruction to handle the pipeline state. For example, by placing such an instruction at the end of two paths, the pipeline states in the two paths are made equal to each other. The state of caches is more tricky to handle. It is necessary to set the state of the caches corresponding to the two paths being compared equal to each other. How this can be done is also highly architecture dependent. There are several options available:

1. One can invalidate all blocks in the caches. This should be possible in almost all processors.
2. One can invalidate only the blocks that differ in the two caches. This requires support for invalidation on the block level.
3. One can replace the blocks that differ with blocks that will be needed in the future by preloading blocks into the caches. This requires support for explicitly loading blocks into a cache.

The first option of invalidating the entire contents of the caches is obviously not an attractive solution since the performance will most probably become poor. This is true also for the second option since each invalidate operation will in many cases cause an additional cache miss later on. The third option is the most promising one but requires special instructions to preload the cache. Examples of such instructions are the instruction and data cache block touch instructions (`icbt` and `dcbt`) found in the POWERPC architecture.

When preloading blocks, it is best to preload a block that will be needed somewhere along the worst-case path. Then, no unnecessary pessimism is added due to additional cache misses. In addition, it is often best to place a preload instruction outside loops if possible to reduce the overhead. The best way to preload is a complex issue, which we do not investigate further in this paper. In the experimental evaluation, we derived this information manually (see Section 5).

When safe local decisions can be made, one can use previously published timing analysis methods when estimating the WCET for programs running on a dynamically scheduled processor. However, to really use one of these methods one must also specify at which points in the program a particular method relies on safe local decisions. Furthermore, the timing model used by the method must be extended to model the dynamically scheduled pipeline. If this is possible and how it is done for each individual method is beyond the scope of this paper. Yet, in the next section, we will describe how it is done for our previously published method based on symbolic execution.

4.3. Case study: symbolic execution method

We will now take a closer look at how the program modification method can be used together with our previously published WCET estimation method [6], based on cycle-level symbolic execution. We start with a brief description of our timing analysis method.

Our WCET estimation method is based on a cycle-level architectural simulator, which can be seen as an instruction-level simulator together with a detailed timing model of the architecture. By using such a simulator, it is possible to get tight estimations of the WCET for single paths through the program. However, in order to estimate the WCET for the whole program, the simulator has been extended to handle unknown data values to enable symbolic execution of programs. In addition to exploring all feasible paths in the program, many infeasible (non-executable) paths are also eliminated. The number of paths to explore can easily become prohibitive. Therefore, a path merge strategy is used to reduce the number of simulated paths. Typically, if a loop contains two feasible paths, these will be merged into one path before starting a new iteration, thereby reducing the number of paths to simulate to at most two in this case.

In order to estimate the WCET for a dynamically scheduled processor we must first attach the simulator to a timing model which accurately models the execution of instructions in the pipeline including the instruction and data caches. Then, we must modify the program to be able to make safe local decisions. This is done by first estimating the WCET of the unmodified program. In this process, we identify all places in the program where the analysis needs to make local decisions. In our case, this is when variable-latency instructions with unknown latency are found and whenever a merge operation is done during the analysis. At all identified places in the program, modifications are applied in order to make all the local decisions safe, i.e., `sync` instructions are inserted to handle pipeline states that differ, and all blocks that differ in the instruction and data cache are replaced by preloading other blocks that will be needed in the future. Finally, a safe estimation of the WCET of the modified program can be made.

The integration of the program modification and our WCET estimation method described here is the one used in the next section where we evaluate the program modification method and also compare it with the pessimistic serial-execution method.

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)

Table 1. The benchmark programs used.

5. Experimental Evaluation

We have evaluated the amount of pessimism introduced when estimating the WCET of seven benchmark programs, using the two methods presented in Section 4: the pessimistic serial-execution method, and the program modification method. The modeled architecture is the one presented in Section 2.2, consisting of a dual-issue pipeline with instruction and data caches.

The key question to answer is how much pessimism is introduced by the two methods. If the pessimism is too severe, it will prompt for advancements in timing analysis methods for dynamically scheduled processors. If it is reasonable, previous methods can be used in combination with the method presented in this paper to enable tight estimations of WCET for programs on dynamically scheduled processors.

5.1. Methodology

An overview of the seven benchmark programs can be seen in Table 1. There are four small programs: *matmult*, *bsort*, *isort*, and *fib*, and three larger programs: *DES*, *jfdctint*, and *compress*. The GNU compiler (gcc 2.7.2.2) and linker has been used to compile and link the benchmarks. No optimization was enabled.

To estimate the WCET of the benchmark programs, the WCET simulator and method described in Section 4.3 has been used. The implementation is built upon the instruction-level simulator, PSIM [1], which simulates the POWERPC instruction set. The original simulator has been extended with a WCET algorithm that uses the simulator to estimate the WCET by exploring and merging paths in the program.

The timing model used in the WCET simulator is based on the model of the POWERPC architecture discussed in Section 2.2 with the timing parameters according to Figure 1. However, instead of a detailed simulation model of the pipeline, we use an analytical approach. During simulation, the functional unit latencies of the simulated instructions are added together with instruction and data cache miss penalties. This we call the serial time, T_{serial}. We then assume that the time T to execute the program on the dual-issue architecture is:

$$T = \frac{T_{serial}}{2}$$

The relation between T and T_{serial} is obviously not this simple in reality. The above formula would represent the ideal situation of dispatching two instructions each cycle. This is often not possible in reality due to cache misses and pipeline stalls and is highly program dependent. Nevertheless, this formula makes it easy to compare the different estimation methods. When estimating the WCET our model automatically produces the pessimistic serial-execution estimate. The other estimates are derived by using the formula above.

When modifying the programs we used `sync` instructions to handle the pipeline state and preload instructions to handle the instruction and data cache states as described in Section 4.3. We assumed that a single `sync` placed at a merge point in the program incurs a penalty of 5 cycles in the dual-issue architecture. When one `sync` instruction is placed before and one after a variable-latency instruction, we assumed a penalty of 8 cycles, i.e., the second `sync` incurs less penalty than the first one since the pipeline is already flushed by the first `sync`. When adding preload instructions, the program becomes bigger. The effect of this on the latency and possible additional instruction cache misses has been estimated manually and accounted for in the results. Three integer multiply instructions were assumed to be variable-latency: `mulhw`, `mulhwu`, and `mullw`. The multiply immediate instruction, `mulli`, and all other instructions were assumed to have fixed latencies.

5.2. Evaluation results

The results from our evaluation of the seven benchmark programs can be seen in Table 2. The actual WCET has been determined by simulating the program using the worst-case input data, or using random input data if the worst-case input was to complex to determine. The table also shows the estimated WCET when using the serial method and when using the modified program method. Also included for comparison purposes is the unsafe program estimate, i.e., the dual-issue timing model has been assumed but no program modifications have been made. This is unsafe since timing anomalies can lead to an underestimation of the WCET. The ratio columns in the table is the estimated WCET values divided by the actual WCET. The modified slowdown is the modified program estimate di-

Program	Measured Actual WCET	Estimated WCET						
		Unsafe program		Serial method		Modified program		Modified
		WCET	Ratio	WCET	Ratio	WCET	Ratio	slowdown
matmult	5283287	5283287	1	10566574	2	6323287	1.20	1.20
bsort	230490	230490	1	460981	2	256854	1.11	1.11
isort	2085	2085	1	4170	2	2325	1.12	1.12
fib	797	797	1	1594	2	797	1	1
DES	186166	186358	1.001	372716	2.002	186358	1.001	1
jfdctint	9409	9409	1	18819	2	9921	1.05	1.05
compress	16486	54583	3.31	109167	6.62	69291	4.20	1.27

Table 2. The estimated WCET using the serial method and when using modified programs.

vided by the unsafe program estimate and shows the amount of pessimism introduced when modifying the programs.

The serial method overestimates the WCET by at least a factor of 2. This is expected and is a result of our assumed timing model. However, for *DES* and *compress*, additional sources contribute. In *DES*, the small additional overestimation is due to data accesses with an unknown reference address. These unpredictable accesses must not be cached in order to keep the cache state predictable. This is accomplished by mapping the accessed data structures into a non-cacheable part of the memory as suggested in [6]. Then, unpredictable accesses will not interfere with the cache and will always cause a cache miss. In *compress*, a small part of the overestimation is also due to unpredictable data accesses. In addition to this, the path analysis fails to eliminate all infeasible paths due to a pessimistic upper bound on a loop (a more thorough description of this loop can be found in [6]).

The estimated WCET of the modified programs is shorter than the serial estimate for all examined programs. In *fib* and *DES*, the program modification method gave no slowdown at all since no modifications were needed. These two programs contain no variable-latency instruction and during the analysis, no merging was done.

In *matmult* and *jfdctint*, the slowdown is caused entirely by variable-latency instructions. No merging was done during the analysis. In *jfdctint*, variable-latency multiplications are only used in the beginning of the program and the inserted `sync` instructions have therefore quite small impact on the estimated value. In *matmult*, however, the multiplications are common and the inserted `sync` instructions give a slowdown of 20 %.

For the remaining programs, *bsort*, *isort*, and *compress*, it is the merging that contribute most to the slowdown. In *bsort* and *compress*, there are a small number of variable-latency multiplications but the effect of those instructions is negligible. In *bsort* and *isort*, the merging occurred at one place in the program. At this place, a `sync` instruction was added, which resulted in a slowdown of 11 % and 12 % for *bsort* and *isort*, respectively. The highest slowdown experienced, 27 %, was for *compress*. This is explained by the fact that merging occurred at four different places in *compress*, each requiring a `sync` instruction.

At the merge place in *bsort* and *isort*, and at two of the four merge places in *compress*, preload instructions for the instruction cache were needed. At these merge places, the instruction cache states differed in the paths being merged. The number of blocks to preload varied between 6 and 10 among the three programs. By preloading blocks that were needed along the worst-case path no extra cache misses occurred and the effect of these preload instructions is very small compared to the merging. The data cache states never differed when merging paths in the programs.

In summary, our program modification method can perform well in conjunction with our symbolic execution method for all our benchmark programs. It works especially well for programs that have few variable-latency instructions and only one feasible path so that merging is avoided when analyzing the program. On the other hand, if a program contains many variable-latency instructions or many feasible paths then the serial method could perform nearly as well or maybe better. For example, if optimization is enabled when compiling *matmult*, the variable-latency multiplications becomes relatively more frequent. This would change the slowdown from a factor of 1.2 to approximately 1.5, thus approaching the slowdown of the serial method.

6. Discussion and Future Work

The results show that our program-modification method can be used to obtain safe and fairly tight estimations of the WCET for our benchmark programs. This suggests that, for a certain class of programs, running on dynamically scheduled processors, it is possible to make safe and tight estimations of the WCET. However, to use the method, there must be some support in the architecture to be able to explicitly control the state of caches and the resource allocation in the pipeline. Ideally, one would need explicit program control

of all internal state in a processor that may influence the future timing of instructions. If no support exists for explicit control of the state of caches or the pipeline, then one is forced to use the serial estimation method which often leads to more pessimism in the estimated WCET.

When using the program modification method the resources in the processor can be used out-of-order except at the modification points in the program where we force an in-order execution. An important consequence of this is that we must statically account for all unknown events and modify the program at the proper places. This forbids the use of preemptive scheduling where a program can be interrupted at any time. However, limited preemption would be possible by treating preemption points in the program as being similar to merge points. The cache and pipeline state must be predictable at all points, regardless of the program being preempted or not. The serial-execution method does not rely on making unknown events safe and can be used together with preemptive scheduling.

It is quite possible that a better analysis method can be invented that results in tighter estimations of the WCET. However, when the processor allows out-of-order resource allocation, timing anomalies can occur. A better analysis method could avoid the program modifications, but each unknown event must still be statically known and statically analyzed. An alternative interesting option would be to include the possibility to control the resource allocation in a processor. Then, the processor could be forced to allocate resources in-order resulting in a stable scheduling of instructions but probably also lower performance.

In this paper, we have only dealt with the handling of caches and the basic pipeline. To make the methods presented here useful, other features in an architecture must also be analyzed. For example, further research is needed to analyze the effect of speculative branches and branch history buffers and how to explicitly control the state of these features. Moreover, we only consider dynamic scheduling done for the functional units. To assure a safe estimate, other features need to taken into account, such as out-of-order load/store-accesses and contention between accesses from the instruction and data cache going to the main memory.

7. Conclusions

Most high-performance processors today use several features that allow out-of-order execution. We have shown that previous methods fail in estimating WCET because they assume that one can rely on worst-case assumptions for local entities such as instructions and basic blocks to estimate the effect on the overall WCET.

In order make available methods useful, we propose to make program modifications to make unknown instruction latencies predictable. This allows existing methods to estimate the WCET safely. We applied these program modifications to seven benchmark programs and estimated the WCET of these programs using a model of a dual-issue pipelined processor with instruction and data caches. We found that the pessimism imposed by the program modifications is less than 27 % for the programs in our benchmark. This suggests that for a certain class of programs, useful estimates of the WCET can be obtained for dynamically scheduled processors.

8. Acknowledgment

We are grateful to Dr. Jan Jonsson for his constructive comments. This research is supported by a grant from Swedish Research Council on Engineering Science (TFR) under contract number 221-96-214.

References

[1] A. Cagney. PSIM, a POWERPC simulator. http://sourceware.cygnus.com/psim/.

[2] 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.

[3] 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.

[4] 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.

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

[6] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2/3):183–207, November 1999.

[7] 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*, pages 47–55, June 1997.

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