Real-Time Penalties in RISC Processing

Steve Dropsho
dropsho@cs.umass.edu

Department of Computer Science
University of Masschusetts-Amherst

December 12, 1995

Abstract

The RISC processor features that provide high performance are *probabilistic* (e.g., cache, TLB, writebuffers, branch prediction, etc.), so worst-case analysis in real-time systems must regularly assume the pathological conditions that make these features perform poorly (e.g., every cache access conflicts). This report presents analytical results of performance penalties due to worst-case execution time (WCET) estimates for RISC processors in real-time systems. The results clearly indicate where efforts should be made to reduce variability in processor designs.

1 Introduction

In real-time computing the correctness of an answer depends not only on its logical value but also *when* it is produced. The need to *guarantee* timing behavior of applications often requires worst-case assumptions be made about runtime behavior such as assuming every cache access is a miss or all branches are incorrectly predicted. While such features as caching and branch prediction significantly enhance performance, many real-time systems turn these features off in preference of execution time *predictability*.

This report looks at the potential penalties that must be assumed when calculating worst-case execution time (WCET) estimates of real-time code. The importance of general purpose processing in real-time computing requires predictable RISC processors [20]. The results here demonstrate where effort should be spent on redesign to significantly improve WCET estimates. We list the hardware features of interest and perform a first order analysis of their potential effects on code execution times. The result is a quantified ranking of the features in decreasing order of their influence on performance.

The complete list of features in a processing system that can add variability to the execution time are best found by analyzing each function in the processor. In the Von Neumann model, a processor has the following functions: instruction fetch, decode, dispatch, execution, and write. Instruction fetch touches the instruction buffer, instruction cache, and address translation hardware (TLB). Decode is self contained in a logic block. Dispatch must conform to instruction issue constraints, i.e., inter-functional unit dependencies. Execution must conform to data dependencies and execute various instruction types. The write stage touches internal registers, the data cache, and the writebuffer.

The various instructions executed are ALU, load/store, and control transfer (i.e., branch). ALU instructions can have data dependent execution times and touch the register file. Load/store instructions touch the register file, data cache, TLB, and potentially arithmetic unit for address calculation. Through cache accesses, load/store instructions have contact with main memory and its DRAM refresh cycles. Control transfer instructions touch the branch target buffer, branch prediction hardware, and possibly the arithmetic unit for target address calculation.

Thus, the features of interest are: variable execution time ALU instructions, instruction and data cache loads and stores, writebuffer effects, pipeline effects (inter-functional unit dependencies, data dependencies, and register file access coordination), TLB accesses, control transfer hardware, exceptions, and DRAM refresh cycles. The following analyses are first order approximations to the effects of each of these features.

Assumptions to note, since most microprocessor architectures have moved to a Harvard architecture with separate instruction and data caches (PowerPC, DEC Alpha, HP-PA RISC, Intel), we will assume systems have the Harvard architecture. In addition, we assume systems prohibit self modifying code. These assumptions help simplify some of the analyses.

To highlight the maximum potential effects of each individual factor, the features are addressed assuming best-case conditions for all but the feature under consideration. For example, looking at instruction cache effects we assume all instructions are single-cycle ALU operations (no data references) without interdependencies, thus eliminating the data cache, writebuffer, pipeline, and branch hardware from having influence.

During the discussion, worst-case results are described in terms of relative performance to the best-case conditions. In the conclusion we also show average-to-best-case results and average-to-worst-case results.

2 ALU Operations

The basic types of operations in a RISC processor ALU are limited. The operations can be categorized as arithmetic (integer and floating point), logical, shift, and swap. Of these the shift and arithmetic instructions may still have variability in some processors due to data dependent operation. Logical operations (e.g., AND, OR, NOT) and swap operations (the interchange of values between two registers) have always been of fixed duration due to their simplicity.

2.1 Shift Instructions

Primarily due to increases in transistor density current processors have barrel shifters to allow single-cycle multi-bit shifts, thus removing variability in shifting a large number of bit positions. Low variability barrel shifters have been available in older processors such as the SPARC2 and have been maintained in recent designs such as the PowerPC family. The R4000, however, allowed up to 64 bit shifts to be specified but only allowed up to 32 bit shifts in a single cycle with larger shifts causing a slip in the pipeline of one cycle [11].

2.2 Arithmetic Instructions

The multiply and divide instructions still suffer from data dependent execution times in some processors. While there are many algorithms for implementing these functions [8] each involves a tradeoff between time and hardware resources. Today's microprocessor's use aggressive

hardware designs to minimize the cycle time of integer multiply, but opt for longer and less hardware intensive integer division implementations. For example, the PowerPC 603 requires between 2 to 6 cycles [3], a 1:3 ratio, for multiply depending on the operands while division is a fixed 37 cycles. In contrast, the Alpha 21064 uses a software division algorithm with a best-case of 16 cycles and a worst-case of 144 cycles [4]. This is a 1:9 ratio and the largest ALU instruction best-case to worst-case ratio we know of in current processors.

For floating point operations the PowerPC 603 has a single-cycle throughput with a fixed three cycle latency for all operations except for division. Division is not pipelined and requires a fixed 18 cycles for single precision and 33 for double. Single cycle latency with multiple cycle throughput on floating point multiplication is standard in the popular processors as is a longer, but fixed delay for division (DEC Alpha 21064 [4], Intel Pentium [1], MIPS R4400 [18]).

The conclusions we can draw on variability due to instruction data dependencies is that most instructions add no variability. Shift, multiplication, and division instructions can contribute variance in some processors, however, if one or more operands are constants then compilers can predict the execution time. The largest best-case to worst-case ratios we have seen for shift, multiplication, and division are 1:2 (MIPS R4000), 1:3 (PowerPC 603), and 1:9 (MIPS R4400), respectively.

2.3 Observations

The fraction of shifts, multiplications, and divisions in the SPEC89 benchmarks are 0.012, 0.030, and 0.005, respectively [10]. Assuming no performance penalties from other factors (cache misses, exceptions, branch mispredictions, etc.) we can show the impact of ALU instruction variability on an application with similar instruction mix. We shall use a best-case cost of one cycle for shifts, 5 for multiplication, and 16 for division. For worst-case times, two cycles for shifts, 15 for multiplication, and 144 for division. While no machine actually has all these values, the numbers preserve the worst-case ratios noted above and will enhance the effects of the variability. In other words, the scenario describes a fictitious processor with worse real-time ALU operations than any actual processor.

Equation 1 gives the relative performance between best-case to worst-case. Inserting the appropriate time values gives a relative performance between worst-case and best-case of 1.8. Or, equivalently, the effects of variable instruction execution times can cause the worst-case time to be 1.8 times the best-case time.

$$RP = \frac{0.953 \times NOP + 0.012 \times SHIFT_{wc} + 0.030 \times MULT_{wc} + 0.005 \times DIV_{wc}}{0.953 \times NOP + 0.012 \times SHIFT_{bc} + 0.030 \times MULT_{bc} + 0.005 \times DIV_{bc}} \quad (1)$$

3 Memory Accesses

Memory accesses include instruction accesses by the fetch unit and explicit data accesses by software loads and stores.

3.1 Instruction Cache Misses

The effects of instruction cache miss predictions on execution time is determined by summing the cost of each instruction and any memory access penalty then dividing by the cost of each

instruction. Dividing by the number of instructions gives an average relative performance value for each instruction. This is shown in equation 2.

Missing in equation 2 is explicit consideration of superscalar pipelines. The effect of multi-issue architectures is to decrease the *effective* cost of an instruction by a factor of the issue rate. To avoid the complexity of instruction scheduling and non-blocking pipelines and their interaction on a cache miss we will assume only single issue pipelines in the following discussion on memory accesses. It should be noted that the results only get worse in a superscalar environment by a factor equal to the maximum issue rate.

$$RP = \frac{1}{n}\sum_{k=1}^{n}\frac{InstCost_k + MissPenalty_k}{InstCost_k} \tag{2}$$

Equation 2 can be simplified if we assume a fixed worst-case instruction cost and average the miss penalty over each instruction. Equation 3 shows the linear relationship between the instruction miss rate and relative performance. For current RISC processors an ALU instruction cost of 1 cycle is common. The metric is relative to the same sequence of operations with no memory penalties. The miss penalty to memory can vary significantly depending on the system design (personal computer vs. single cpu workstation vs. multiprocessor). For example, the Sun SPARC2 workstation has a penalty of at least 24 cycles while the penalty on an SGI Onyx multiprocessor system is about 100 processor cycles. Figure 1 plots Equation 3 for miss penalties of 20, 50, and 100 cycles.

$$RP = \frac{InstCost + MissPenalty \times MissRate}{InstCost} \tag{3}$$

This analysis shows the primary performance penalties suffered by instruction cache misses. A subtle effect is not captured in the simple equation. High performance processors usually provide the requested word first to the pipeline to minimize the read miss stall then complete loading the cache in subsequent cycles. Provided that another miss does not occur in the next few cycles, the additional cycles for loading the cache are hidden behind processor computation effectively shortening the read penalty by a few cycles. Equation 3 does not account for this effect which would be seen when the miss rate is very high. The result would be to tail the curves up a small number of cycles (approx. 1 to 3 cycles) as the miss rate approached 100%.

3.2 Data Accesses

Data accesses are more complex than instruction accesses due to the variety of data access policies: non-cacheable data vs. cacheable data, write-through policy vs. writeback policy, and allocate cache line on miss vs. not allocate. These memory design options are summarized in table 1 along with each option's effective use of the memory system. The two actions on the memory system are either a memory read (Mem RD) or a memory write to an assumed writebuffer (WB Effects) that may delay the actual write to main memory.

The actions on the memory system are either a read, a write, or a combination of the two. Actions of just reads or just writes can be analyzed fairly simply and provide interesting information. However, when the operations occur in conjunction the analysis becomes complex due to the many design tricks used to hide writes partially or entirely behind outstanding reads. We can bound the performance penalties by assuming the individual operations are independent

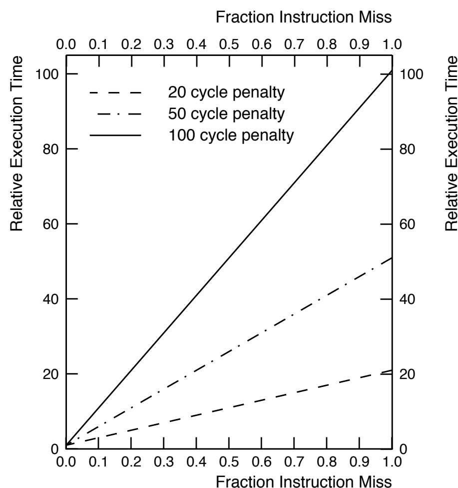

Figure 1: Instruction Cache Effects

and their respective penalties additive. In effect, serializing the operations to ensure worst-case analysis. This is discussed in more detail below.

3.3 Data Cache Loads

From table 1 a data load cache miss is simply a memory read and is identical to an instruction cache miss except in the case of a writeback cache where a modified (dirty) cache line must be displaced. In this scenario, both a read of the new line and a write of the dirty line must be performed. As mentioned previously, we will ignore writes for now so we assume no modified data cache lines are displaced.

By ignoring writes, the equation for the performance penalty of data cache loads is identical to that of instruction cache accesses in equation 3. And since the access penalty for data is usually identical that of instructions the associated graph in figure 1 is accurate.

3.4 Data Cache Stores and Writebuffer Effects

The scenarios described in table 1 for data stores are more varied than for data loads. For scenarios that result in only memory reads the previous analysis for data loads can be used. A more interesting case occurs when just writes are issued to the memory system. In this scenario the writebuffer's effects must be considered.

A writebuffer provides small, temporary storage for the pipeline or cache controller to store data that must go to main memory. Once data is added to the writebuffer, the writebuffer assumes responsibility for moving the data to main memory and maintaining the data integrity in the interim (e.g., an old data item will not be read from memory if a more current value exists in the writebuffer). The delay to the writebuffer in modern processors is usually a single cycle, much less time than if the data had to go directly to main memory. This immediately frees the writing unit (pipeline or cache controller) to continue running since the write has been *effectively* completed.

Architectural Options	Data Load	Data Store
I. Non-Cacheable Data	Mem RD	WB Effects
II. Cacheable Data	...	...
1. Write Through Cache	...	...
(A) Allocate on miss	Mem RD	Mem RD + WB Effects
(B) Not allocate on miss	Mem RD	WB Effects
2. Writeback Cache	...	...
(A) Allocate on miss	...	...
i. Replace clean line	Mem RD	Mem RD
ii. Replace dirtly line	Mem RD + WB Effects	Mem RD + WB Effects
(B) Not allocate on miss	Mem RD	WB Effects

Table 1: Data Access Scenarios

The difficulty arises in the limited storage of the writebuffer. If writes occur more quickly than they can be retired to main memory the writebuffer will fill and stall subsequent writes. Clearly the spacing between writes is a factor in determining the effects of the writebuffer and this leads to two analyses. In the worst-case all writes are issued back to back overloading the capacity of the writebuffer for essentially the entire series of writes. The other extreme, the best-case, has the writes evenly spaced throughout the instruction sequence providing maximum time for retiring data to main memory before the next write.

Equation 4 gives the relative performance of a sequence of instructions in the presence of contention during writes. Because of its generality equation 4 is not particularly revealing. Instead, let us maximize the effects of the write penalties by assuming all instructions take a single cycle in the ALU. This results in a more useful representation for the worst-case in equation 5.

$$RP = \frac{1}{n}\sum_{k=1}^{n} \frac{InstCost_k + WritePenalty_k}{InstCost_k} \tag{4}$$

$$RP_{wc} = 1 + FracWr \times WritePenalty \tag{5}$$

Since in the worst-case all writes are consecutive, once the writebuffer fills each write must stall for the duration of a write to memory ($WritePenalty$) until a slot empties in the writebuffer. We are assuming that the number of writes to fill the writebuffer is small relative to the overall number of writes (*writebuffer depth* $\ll n \times FracWr$) and can be ignored. Note that we are assigning only a single cycle to store to the writebuffer for a zero penalty (i.e., requiring zero stall cycles) in the processor pipeline.

Equation 6 uses the same assumptions above to quantify the performance in the best-case with evenly distributed writes. The relative performance is a cycle for each instruction plus the stall penalty due to contention in the writebuffer. With the stores evenly distributed the spacing between writes is approximately equal to the inverse of the fraction of writes. Note, the actual

number of non-storing cycles is one less. For example, with 10% of the instructions as stores, a write occurs once every 10 instructions providing 9 cycles with no stores.

$$RP_{bc} = 1 + FracWr \times max(0, WritePenalty - (\frac{1}{FracWr} - 1)) \tag{6}$$

Equation 6 is an approximation since non-integer intervals are realized in practice by averaging intervals of multiple writes. The $max()$ function sets the floor of the penalty at zero when the interval of writes exceeds the time to empty the writebuffer to main memory. In other words, the writebuffer is able to completely hide the store to memory.

Figure 2 plots equations 5 and 6 for $WritePenalty$ times of 10, 25, and 50 cycles, or half the times of the read penalties used in figure 1 since writes to memory generally take less than half the time of reads.

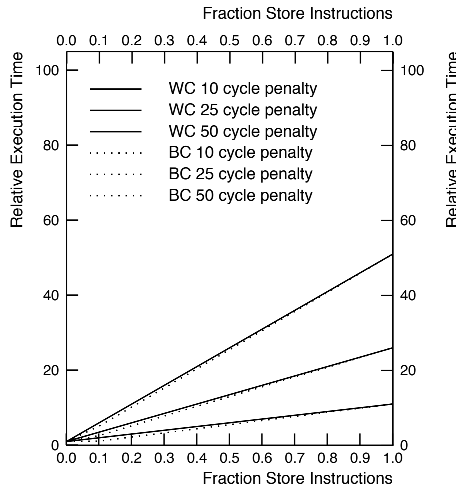

Figure 2: Writebuffer Effects

Note that the best-case curves (dotted) do not deviate much from the worst-case (solid). Figure 3 is the same plot but enlarged around the point of 10% store instructions. It becomes clear that the relative performance of the best-case and worst-case differ by only a small amount.

3.5 Data Loads Displacing Dirty Cache Lines

We can estimate an upper bound on the performance impact of a writeback cache that allocates a cache line on a store miss if we assume every miss causes a read with a writeback of a displaced dirty cache line. In our worst-case assumption we serialize the operations to maximize the penalty in order to bound the performance we would see in a real machine. Thus, we assume the writeback of the dirty cache line proceeds first followed by the read. The effect is to increase the memory access penalty as shown in equation 7 and is graphed in figure 4. Here we are assuming 100% of the cache lines are dirty ($FracDirty = 1$) and, as above, the value of $WrbackPenalty$ is set to one half the miss penalty for a read.

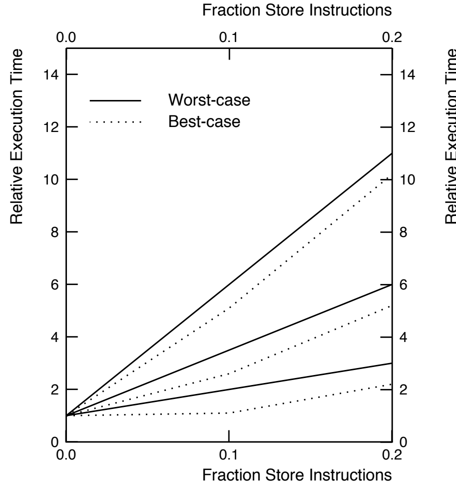

Figure 3: Writebuffer Effects (close-up)

$$RP_{RDwWB} = \frac{InstCost + (MissPenalty_{RD} + FracDirty \times WbackPenalty) \times MissRate}{InstCost} \quad (7)$$

3.6 Observations

The performance penalty for cache misses is high. For worst-case timing analysis a miss penalty must be assumed if a cache hit cannot be *guaranteed*. Current work in instruction cache performance prediction [12] has accurately determined hit rates of 70% during compile time for instruction cache accesses. We must assume the remaining 30% as instruction cache misses. Referring to the graph in figure 1 a 30% miss rate shows that estimates of best-case to worst-case differ by a *factor of 7.0* assuming a 20 cycle miss penalty.

Data loads can also be a source of large variability. Work in predicting the hit rates in the data cache [15, 2] have not been nearly as successful as similar work for instruction references. Experimental results have shown miss rate predictions ranging from 30% up to 100% for applications with very low actual miss rates. Because of the large variance we shall assume a 100% miss rate in the WCET. From the SPEC89 suite [10], the frequency of data loads is about 35% and the frequency of data stores about 10%. The *effective miss rate per instruction* for data loads is the miss rate of loads multiplied by the fraction of data loads occurring in code segments. Here, we are assuming a 100% miss rate for the 35% of instructions that are data loads producing an effective miss rate of 35%. Since data load cache misses are similar to instruction cache misses figure 1 can again be used to show best-case estimates to worst-case estimates at the 35% point differ by a *factor of 8.0* for a 20 cycle miss penalty.

Data stores occur much less frequently than data loads, so effects of the writebuffer add much less variability in worst-case time estimates. Under a worst-case distribution of the stores for a store penalty of 10 cycles and an *effective per instruction miss rate* of 10% (100% miss

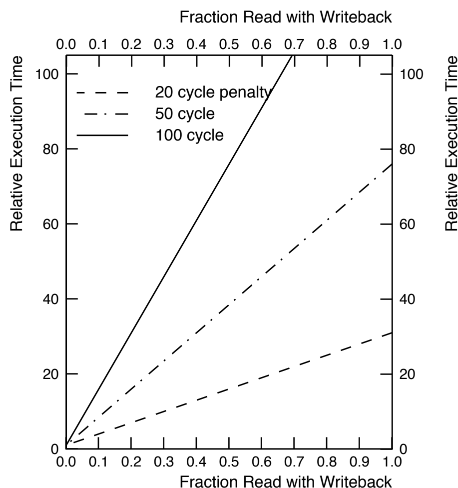

Figure 4: Writeback cache policy with allocate on miss and dirty line displacement

rate on data stores times 10% frequency of data stores in applications) figure 3 shows best-case to worst-case differing by a *factor of 2.0* due to worst-case distribution of writes and the writebuffer.

If the cache policy is a writeback cache with allocate on a write miss and we assume the most costly scenario that a dirty cache line is always displaced then the worst-case time estimates must be increased significantly. We can refer to figure 4 at the point for 45% of the instructions missing in cache data access (group the frequency of data loads and stores together since it is the same penalty for both). The graph shows best-case to worst-case differing by a *factor of 14.5*. This quick analysis highlights the effect that cache management policy has on estimating performance.

4 Pipeline Effects

The structure of high performance processors has evolved in two ways: breaking the functionality into a series of stages to allow faster clock rates (i.e., pipelined) and putting functional units in parallel so multiple instructions can be issued simultaneously (i.e., superscalar). Designs that do both are called pipelined superscalar processors.

The best-case performance occurs when the maximum number of instructions enter and leave the pipeline each cycle. Obviously, this is limited by the number of instructions that can be issued per cycle. For example, the R4000 can issue one instruction per cycle while the PowerPC 604 can issue up to four (e.g., two integer, one floating point, and a branch).

$$RP_{PPL} = \frac{1}{n}\sum_{k=1}^{n} \frac{(InstCost_k + MaxPplPenalty - 1) \times \frac{1}{MaxIssueRate_{wc}}}{\frac{InstCost_k}{MaxIssueRate_{bc}}} \tag{8}$$

The worst-case performance relative to the best-case is described in equation 8. The denominator reflects the best-case where multiple instructions are issued per cycle for an effective

execution time less than one cycle. The numerator reflects the worst-case in that without additional knowledge we must assume that there are pipeline conflicts between every instruction. These pipeline conflicts can be from too many instructions trying to be issued on too few execution units, data dependencies between instructions, or inter-unit dependencies (e.g., floating point unit also requires the integer unit or a branch blocks issues to other units to simplify rollback logic). The effect of these conflicts in the worst-case is to serialize the instruction issue reducing the worst-case maximum issue rate ($MaxIssueRate_{wc}$) to 1.0 and potentially adding additional time due to a stalled pipeline ($MaxPplPenalty$, this penalty does not include cycles counted in the previous instruction, $InstCost_{k-1}$). In the worst-case with no knowledge we must assume the worst stall penalty occurs on every instruction.

The relative performance factor is maximized- and hence shows worst-case variability- if all instructions execute in the minimum amount of time. From equation 8 we can see that as the instruction time $InstCost_k$ *decreases* from infinity to the minimum of one cycle the relative performance factor increases from $MaxIssueRate_{bc}$ to $MaxIssueRate_{bc} \times (1 + MaxPplPenalty)$. Thus, we shall assume an instruction time of one cycle ($InstCost_k = 1$), the minimum value, to determine the maximum relative performance factor.

4.1 Observations

The maximum pipeline penalties in RISC processors are data dependencies between instructions which use high latency functional units. For example, in the PowerPC 604 conflicts between the branch unit and the dispatch unit result in a single cycle pipeline stall, data dependencies in the integer unit, complex integer unit, floating point unit, and store unit can cause pipeline stalls of zero (no delay), one, two, and two cycles, respectively [19] (note, the units have *latencies* of $1, 2, 3$, and 3 cycles and can stall the pipeline up to one cycle less than the latency). Therefore, a single cycle instruction in the PowerPC 604 can stall the pipeline for a maximum of two cycles. Thus, the relative performance factor between worst-case and best-case in the PowerPC due to pipeline conflicts is $(1 + 2) \times 4 = 12$.

An interesting contrast is the MIPS R4000 [11] which can issue only a single instruction per cycle and has a maximum stall penalty of two cycles (e.g., a load whose result is required immediately) for a relative performance factor of 3. This is considerably smaller than the superscalar PowerPC 604 and indicates that as designs incorporate more parallelism into the pipelines the gap between best- and worst-case will grow.

Pipeline timing analysis has been used by Harmon et al. [6] and their results show *exact* matches between the observed timing and predictions when only variations due to pipeline effects are considered. The two results that did not match in their experiments are easily explained as a deficiency in the tools to account for the proper worst-case path (resulting in a 2% error) and determine a data dependent loop bound (over estimated by a factor of 2). The conclusion from this work is that given a path pipeline effects can be determined precisely by the state of the art in timing tools *without modification to the basic pipeline structure*.

5 TLB Accesses

Modern systems use virtual addressing to simplify system use and improve resource utilization. The advantages of virtual addressing include relocatable code, efficient memory use,

and process protection. A disadvantage is that virtual addresses must be *translated* to actual physical addresses before memory is accessed. There are many different translation methods used across the different microprocessors and each makes tradeoffs between potential memory fragmentation, simplicity of translation, size of translation table information, and protection. The second edition of Hennessy and Patterson's architecture book [7] gives a good survey of the issues and possible solutions.

All methods of translation follow pointers to one or more tables that contain information for translating between the virtual address and the corresponding physical address. Thus, the cost of translation is primarily in memory accesses to follow the links between the various tables. The TLB is a cache for the most recent translations to save this table walking. Unfortunately, under worst-case conditions we may have to assume misses in the TLB for many of the instruction and data accesses.

TLB miss times depend heavily on the number of pointers that have to be followed in determining a physical address. This might be as few as one or potentially a large number, however, practical considerations usually limit the number to two (POWER2 architecture [17]) or three accesses (Alpha AXP architecture [7]) per miss. Actual measured times by Saavedra and Smith [16] on a variety of systems support this relationship. Some RISC chips such as the Alpha 21064 trap on a TLB miss to run specialized code for loading the TLB. This adds additional time that is not considered here. Instead, we present the effects of TLB misses for systems that require 2, 3, 4, and 5 memory references to resolve the physical address. We will assume only a 20 cycle read penalty for corresponding TLB miss penalties of 40, 60, 80, and 100 cycles.

$$RP_{TLB} = \frac{InstCost + MissPenalty_{TLB} \times MissRate_{TLB} \times MemAccessesPerInst}{InstCost} \quad (9)$$

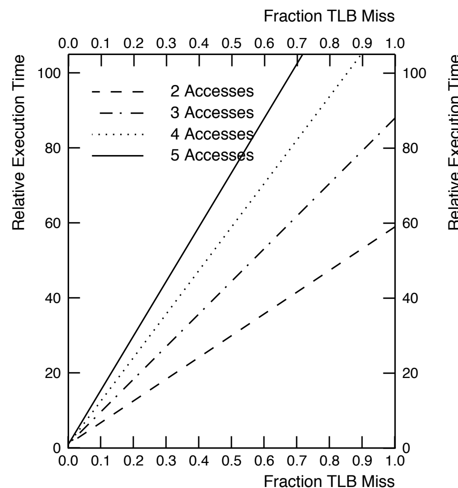

Figure 5: TLB miss effects for translations with 2, 3, 4, and 5 memory accesses

Equation 9 and figure 5 show how performance is affected by misses in the TLB. Again, we assume single cycle instructions and note that we can have more TLB misses than instruction

references since a data load or store includes a reference for an instruction and then one for the data. For typical applications there are 1.45 memory references per instruction.

5.1 Observations

Unfortunately, we are not familiar with any work that parallels the work done in instruction cache worst-case performance prediction [14] and consequently we do not have numbers showing what reasonable analysis can predict for TLB behavior. It is also unfortunate that the TLB miss rate can not be bounded by the instruction or data cache miss rates. From figure 5 all that can be said is that in the worst-case the relative performance factor is no greater than 88, which is not particularly comforting. However, separate instruction TLBs and larger page sizes should allow good worst-case miss rate prediction. *Average-case* data of TLB miss rates in actual machines show miss rates for the SPEC Benchmarks ranging from 0.0% up to 10.0% [16].

6 Control Transfer Instructions

Control transfer instructions change the control flow of programs. Generally, there are two problems in tracking control flow. One is the *instruction flow problem*. When control transfers from one sequence to another the instruction fetch unit must adjust for this change in flow. The second is the *decision problem*, determining whether control transfer is to take place in conditional branches.

In high performance RISC processors both problems increase variance of the WCET. To address the flow problem high performance RISCs have *branch target buffers (BTB)* that cache target addresses of control transfer instructions. When a control transfer instruction is encountered the target address from the BTB is used to start fetching instructions before the target address has been calculated. Some designs may even cache the instruction at the target address. The target instruction can then be immediately used in place of the control transfer instruction. In such a design the branch essentially executes in zero cycles. Variability is added if the BTB does not contain the target of a control transfer instruction.

The decision problem results from a dependency between an operation to write condition bits and the branch that uses the condition bits. In a worst-case scenario the branch immediately follows the instruction that sets the condition code. In order to use the actual value of the condition code in the branch decision the branch may have to stall one or more cycles. High performance RISCs eliminate many of these stalls by predicting the direction of the branch and verifying the prediction later. In the case of a correct prediction no cycles are wasted in the pipeline. In the case of an incorrect prediction the instructions following the branch must be flushed and instructions from the correct target must be fetched.

The ratio between best-case and worst-case time estimates depends on the number of cycles for correcting a misprediction or calculating a target address due to a BTB miss, the number of instructions issued per cycle in the pipeline, and the fraction of branch instructions in the code. The ratio can be approximated as in equation 10. The denominator reflects that in the best-case some architectures can completely hide branches. This equation is adequate as an approximation for any reasonable code sequence with branch occurrences as a minority of the total instructions. When the fraction of branches becomes large it is increasingly difficult to completely hide control transfer with branch folding and other techniques because of storage

limits for the needed information.

$$RP = \frac{1 + FracBr \times max(AddrCalcPenalty, MispredictPenalty) \times InstIssueNum}{1 - FracBr} \qquad (10)$$

In the SPEC89 codes branches are almost 15% of the instructions [10]. For an example we shall use 15% for $FracBr$. The IBM's Power2 does not use a BTB (it always calculates the target address) but has a one cycle delay on mispredictions and can issue up to four instructions simultaneously, giving a penalty factor of 1.9. The MIPS R4000 also does not use a BTB but has a three cycle penalty and can issue only a single instruction per cycle [11] for a similar performance penalty factor of 1.7.

6.1 Observations

The effect of control transfer instructions on the worst-case to best-case ratio is less than 2.0 for current architectures. This ratio reflects the low frequency of branches and the relatively small penalty for a branch misprediction. As pipelines become more complex we can expect the effects of branches to grow somewhat. However, since architects stress minimizing the branch penalty, we expect the ratio to grow slowly in response to the increasing degree of multiple issue in superscalar pipelines.

7 Exceptions

The cost of an exception is: 1) the delay to correct the state (T_{State}) for a precise exception plus 2) the delay to start the handler routine (T_{Begin_H}) plus 3) the delay in returning from the handler (T_{End_H}) plus 4) the delay in restoring the post-exception instructions to their same or better state of progress in the pipeline as before the exception ($T_{Ppl_Restore}$). The best-case and worst-case times for each of these steps in taking an exception define the potential range of variability. Equation 11 shows the relative performance of worst- to best-case timings for exceptions. We will ignore potential cache conflicts between the exception handler and original code stream in order to isolate the exception overhead.

$$RP_{excpn_ovrhd} = \frac{T_{State_{WC}} + T_{Begin_H_{WC}} + T_{End_H_{WC}} + T_{Ppl_Restore_{WC}}}{T_{State_{BC}} + T_{Begin_H_{BC}} + T_{End_H_{BC}} + T_{Ppl_Restore_{BC}}} \qquad (11)$$

7.1 Observations

Precise Exceptions. For the precise exception model, the state of the processor when the exception is taken must be the same as if the processor were not pipelined. In processors that do not support out-of-order execution maintaining precise exceptions is straightforward. However, when out-of-order execution is allowed then instructions must be cancelled that have "completed" ahead of the instruction where the exception occurred even though they logically follow it. Also, instructions still in progress must be completed if they are logically before the exception and cancelled if they occur logically after. In general, the pipeline must be emptied so the protection mode can be changed from user to supervisor, to allow exception handlers to access the kernel.

In superscalar processors it is possible to have an earlier instruction still in progress when the exception occurs. Interrupt processing must be delayed sufficiently so any updates to machine state will be complete before the interrupt handler saves state. In the best-case zero delay cycles will be needed, however, in the worst-case a long latency instruction such as a double precision floating point divide might force a significant delay. Many processors support two modes, one that forces serialization of long instructions (at a performance penalty) if precise interrupts are required and a second mode for high speed processing that does not guarantee precise exceptions. We will assume the pipeline is very efficient at correcting machine state and causes no penalty ($T_{State} = 0$).

Handler Start. Once an exception is raised the proper handler code must be run. The general method for determining the proper code is to put an ID in an *exception register* that is used as an offset from a special base address pointer into an exception table. The exception table is essentially an indirect jump table with a list of pointers to various handler routines. The appropriate address is retrieved and loaded into the program counter. So, one memory read is required before the start of the handler routine can be fetched. In the best-case this memory read hits in the cache. In the worst-case the read misses in the cache and suffers a penalty of $MissPenalty$ cycles. A value of 20 cycles has been used in previous sections.

Handler End. When an exception is taken machine state is pushed onto a stack. Upon returning from the exception handling routine this state is popped off and restored. If this exception stack is implemented on chip then restoring is done in parallel with the return instruction and no delays are incurred. Use of a hardware stack is common so we will use $T_{End_H} = 0$ for both best- and worst-case.

Restoring the Pipeline. Upon returning from an interrupt additional time penalty cycles are assessed until the original instruction stream progresses to at least the same point in the pipeline as before the exception. In the best-case this can be as few as two cycles in the PowerPC 604 if the exception occurred when subsequent single-cycle instructions had to be serialized. In this case only the cost of fetching and decoding instructions for dispatch have to paid again. In the worst-case we have to pay not only the penalty of fetching and dispatch but also the maximum potential delay due to a pipeline conflict between instructions in the handler routine and the original code stream and the cost of executing the instruction where the exception occurred minus one cycle. After paying the maximum penalty ($MaxPplPenalty$) for potential pipeline conflicts then we can assume, even in the worst-case, that the instruction scheduling will be as good or better than before the exception.

Bringing It All Together. Analyzing exceptions is uses many resources of the processor and not a single feature like the cache or TLB. Thus, the best- and worst-case performance of many features must be considered together in assessing the overhead of taking an exception. This creates an issue that must be quickly discussed here even though it turns out to have no effect.

There is a difference between the number of *cycles* lost due to delays and the amount of *work* lost. In a single-issue machine the two are equivalent, but in a superscalar machine they are not. In superscalar designs the amount of work equals the number of cycles times the number of instructions issued per cycle. For the relative performance ratio we are interested in the *work*

lost due to delays between the worst-case and best-case. Since we are looking at worst-case relative to best-case exception overhead *on the same* processor with *the same code sequence* in both cases then both the numerator and denominator have the same issue rate factor and, thus, it has no effect on the relative performance factor.

$$RP = \frac{(1 + MissRate_{WC} \times MissPenalty) + (2 + InstCost - 1 + MaxPplPenalty)}{(1 + MissRate_{BC} \times MissPenalty) + (2 + InstCost - 1)} \tag{12}$$

Equation 12 gives the relative performance of the overhead for taking and returning from an exception on the PowerPC 604. Using 20 cycles for $MissPenalty$, 1.0 and 0.0 for $MissRate$ in the worst- and best-cases respectively, and 2 cycles for $MaxPplPenalty$, this ratio is maximized when instruction cost is minimized. Thus, we will use 1 cycle for $InstCost$ which gives a relative performance factor of 8.0 between the best-case overhead and the worst-case overhead, with over 80% due to the potential cache miss. However, the total impact of interrupt overhead on system performance depends on the frequency of interrupts and the size of the handler.

$$RP_{excpn_frac} = \frac{1 + FracExcpn \times ExcpnOvrhd_{WC}}{1 + FracExcpn \times ExcpnOvrhd_{BC}} \tag{13}$$

Figure 6 graphs equation 13 showing the effects of exception overhead given the fraction of instructions experiencing an exception and assuming a handler that requires zero time. We will use the values above for $ExcpnOvrhd_{WC} = 24 cycles$ and $ExcpnOvrhd_{BC} = 3 cycles$.

To put exception overhead into perspective, a PowerPC 604 100 MHZ processor that misses the cache on every instruction and data reference and experiences the worst pipeline conflicts would still issue instructions at a rate greater than one per 100 cycles. So, using this instruction completion rate as a bound, to achieve interrupts on only 1% of the instructions requires a rate of 10,000 exceptions per second, or one exception per 0.1 milliseconds. This is a very high rate for current systems whose tasks have execution times in excess of 1.0 millisecond. Thus, to estimate the worst-case impact of exception overhead in a typical system we feel comfortable using the relative performance factor at 1% which equation 13 shows is 1.2.

8 DRAM Refresh

Main memory is usually constructed with low cost and low power **dynamic random access memory (DRAM)** chips. However, DRAM technology slowly loses electrical charge that must be restored with **refresh cycles**. The problem for real-time systems is that actual memory reads are delayed additional cycles if a refresh cycle is in progress. Since refresh cycles are asynchronous to application code, execution time variability is added by these unanticipated memory access delays.

In most systems, logic to control the DRAM refresh operations is situated with the logic that manipulates the control signals to the DRAM chips (address and write enable strobes). In expensive systems this logic is placed close to the DRAM chips with only a small number of cycles between initiation of each refresh and its completion. In simpler, less expensive systems such as might be found with embedded real-time applications this DRAM control logic is further from the DRAM chips and closer to the processor. Control of a refresh at the DRAM

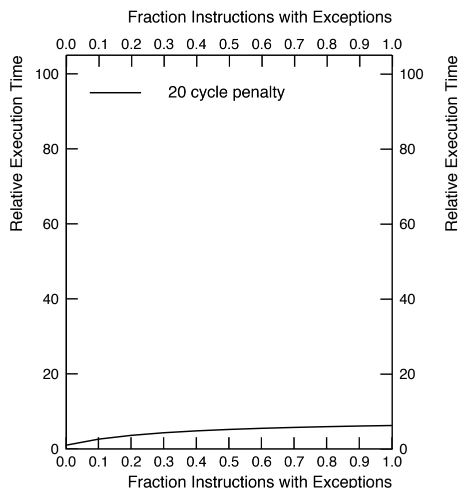

Figure 6: Exception Overhead

chip interface is similar to a memory read, however, the timing at the system level is more characteristic of a write from the processor. In the extreme, a single memory refresh operation will require less time than a memory write, thus, for worst-case analysis we will assume a refresh takes the same amount of time as a write.

In the best-case scenario no memory operations collide with refresh cycles and no penalties are paid. In the worst-case scenario every memory access that could possibly collide with the start of a refresh operation does so and must wait the additional time of the refresh which we is bounded by the time of a write. The total effect of DRAM refresh cannot be more than the total time the system is actually doing refresh.

8.1 Observations

A bound on DRAM refresh effects can be determined if some reasonable assumptions about the size and configuration of main memory are be made. Equation 14 gives relative performance due to DRAM refresh effects provided some system parameters are known. An industry constant in DRAM memory modules (DRAM SIMMs) is that a single SIMM must receive a refresh operation *at least* once every 15.625 microseconds *on average* [13]. Thus, each SIMM must receive a refresh operation every 15.625 microseconds and multiple banks may be refreshed in parallel (if permitted by power constraints). Assuming a single bank of 1M×32 bit SIMMs (*NumBanks = 1*, *SIMMSz = 4 MB*), a system with 64 MB (*MemSz = 64 MB*) has 16 SIMMS and requires a refresh operation every 976 ns. In a 100 MHz system (*CPUClk = 10 ns/cyc*) that is 97.6 cycles between refresh operations. If a refresh requires 10 cycles (*RefTime = 10 cyc*) then 10.3% of the time is spent in refresh. Thus, the maximum increase is 10.3% in the time of memory accesses for a relative performance of 1.10. Doubling memory size to 128 MB will give a relative performance of 1.20. Clearly, refresh has a small effect compared to other features.

Architectural Feature	WC:BC	Ave:BC	WC:Ave
TLB Accesses	≤ 88	3.6	≤ 24.4
Data Cache Load	[14.5, 8.0]	[1.5, 1.3]	[9.7, 6.2]
Pipeline Effects (PowerPC 604)	12.0	Not avail.	Not avail.
Instruction Cache Load	7.0	1.8	3.9
Writebuffer Effects on Data Stores	2.0	1.5	1.3
Branch Instructions	1.9	1.1	1.7
ALU instructions	1.8	1.4	1.3
Exceptions	1.2	1.0	1.2
DRAM Refresh	1.1	1.0	1.1

Table 2: Hardware Performance Effects: Ratios (Approx.)

$$RP_{DRAM} = 1 + \frac{MemSz(MB)}{SIMMSz(MB) \times NumBanks} \times \frac{RefTime(cyc) \times CPUClk(ns/cyc)}{15625ns} \quad (14)$$

9 Conclusions

9.1 Worst-Case to Best-Case Results

Table 2 in column 2 consolidates the results in order of decreasing effects on performance. The range indicated for the data cache reflects differences due to possible caching policies listed in table 1. The high value is for cache policies where data loads experience both the memory read penalty and writebuffer effects while the low value reflects cache policies where the writebuffer effects can be ignored. The apparent difference of writebuffer influence between data cache loads and data stores is due to the difference in the number of instructions using the writebuffer (45% in the first case and only 15% in the second).

9.2 Average-Case to Best-Case Results

While the range between worst- and best-case times is instructive, it is also necessary to consider the average-case execution times to see what features force undue pessimism in worst-case execution time estimates over the performance typically seen. To demonstrate this we first derive average-case to best-case ratios as we did for the worst-case. From the average-to-best ratios and the worst-to-best ratios a simple division generates worst-to-average ratios.

Column 3 in table 2 shows the average- to best-case results. For the average-case we assume a 4.0% miss rate for both the instruction and data caches. This is a high miss ratio for today's microprocessors which have demonstrated significantly lower values in the SPEC92 suite [9] and other commercial benchmarks [5] that include transaction processing, file servers, and multiuser development environments. For loads that might displace a dirty cache line we assume that only 50% of the cache lines are dirty. Again, note that these factors are for single issue processors and it is likely that superscalar processors will have larger factors which depend on their sustained instruction issue rate in the absence of cache misses.

We also assume 85% accuracy from the branch prediction mechanisms for a misprediction rate of only 15%. Assuming 15% of the instructions are branches then only $0.15 \times 0.15 = 0.023$ mispredictions are made per 100 instructions.

For writebuffer effects, we assume that writes are randomly distributed and we average the worst-case distribution results (*RP = 2.0*) and the best-case distribution results (*RP = 1.0*) for a value of *RP = 1.5*. Similarly for ALU instruction times, we use times that are the average between the best and worst execution times for the values in the numerator of equation 1: 1.5 cycles for shifts, 10 cycles for multiplication, and 80 cycles for division.

For average-case TLB effects we will select the TLB miss rate of 3.0%. This is approximately the median miss rate for the HP 9000/720 on the SPEC Benchmarks [16]. The HP 9000/720 was chosen because it is most representative of current microprocessor systems in the study. With a TLB miss penalty of three memory accesses or 60 cycles, equation 9 shows a performance ratio of 3.6.

We are not aware of any data on the average-case performance of a pipeline without stalls from cache misses. However, as discussed previously, current techniques appear to have solved the pipeline performance prediction problem and are not considered further.

For exception overhead, its influence is so minor in comparison to other features that we will assume the average-case matches the best-case to accentuate the ratio between the worst- and average-cases.

For average-case DRAM refresh effects, we assume that memory operations are evenly distributed and randomly conflict with refresh operations. Assuming the worst average-case memory factor of 1.5 (see table 2), 33% of the time is spent doing memory accesses. The time that a memory access will overlap a refresh operation is approximately $0.33 \times 0.10 = 0.033$, or 3 percent delay due to refresh.

9.3 Worst-Case to Average-Case Results

Column 4 in table 2 lists the worst-to-average ratios. While the relative order of the effects is almost identical to the worst-to-best results in column 1 there appears to be greater room for decreasing penalties due to branching compared to penalties from the writebuffer and ALU instructions. Overall, however, the TLB and caching are still the dominant factors in determining the worst-case execution time.

To simplify the discussion, little has been said about the interactions between the hardware features. Indeed, these interactions can have significant impact on the degree to which features affect the run-times. For example, Healy et al. [6] show how memory latency can be overlapped with pipeline stalls and Saavedra and Smith [16] show that the loose coupling of functional units in superscalar designs can result in an overlap between computation and data fetching resulting in limited prefetching. Thus, the penalties are not strictly additive.

10 Conclusion

This work develops first order approximations to the performance penalties that real-time systems must assume when using high performance RISC processors. The results prioritize the areas of processor designs that should be considered for significantly improving WCET

estimates. While the needs of real-time systems require high performance, design changes are needed to effectively exploit the potential of RISC processors.

References

[1] Donald Alpert and Dror Avnon. Architecture of the Pentium Microprocessor. *IEEE Micro*, pages 11–21, June 1993.

[2] S. Basumallick and K. Nilsen. Cache Issues in Real-Time Systems. *ACM PLDI Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[3] Brad Burgess, Nasr Ullah, Peter Van Overen, and Deene Ogden. The PowerPC 603 Microprocessor. *Communications of the ACM*, pages 34–42, June 1994.

[4] Digital Equipment Corporation, editor. *DECchip 21064-AA Microprocessor Hardware Reference Manual*. Digital Equipment Corporation, 1992.

[5] M.T. Franklin, W.P. Alexander, R. Jauhari, A.M.G. Maynard, and B.R. Olszewski. Commercial workload performance in the IBM POWER2 RISC System/6000 processor. *IBM Journal of Research and Development*, pages 555–562, September 1994.

[6] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. *Proc. of the IEEE Real-Time Systems Symposium*, December 1995.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., second edition, 1996.

[8] Israel Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[9] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, pages 15–26, October 1994.

[10] Larry McMahan and Ruby Lee. Pathlengths of SPEC Benchmarks for PA-RISC, MIPS, and SPARC. *IEEE COMPCON*, pages 481–490, 1993.

[11] Sunil Mirapuri, Michael Woodacre, and Nader Vasseghi. The Mips R4000 Processor. *IEEE Micro*, pages 10–22, April 1992.

[12] Frank Mueller, David B. Whalley, and Marion Marmon. Predicting Instruction Cache Behavior. *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[13] Inc. NEC Electronics, editor. *Memory Products Data Book, DRAMs, DRAM Modules, Video RAMs*, volume 2. NEC Electronics, Inc., 1993.

[14] D. B. Whalley R. Arnold, F. Mueller and M. Harmon. Bounding Worst-Case Instruction Cache Performance. *IEEE Symposium on Real-Time Systems*, pages 172–181, Dec. 1994.

[15] Jai Rawat. Static Analysis of Cache Performance for Real-Time Programming. Technical Report TR93-19, Iowa State University of Science and Technology, November 1993.

[16] R.H. Saavedra and A.J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, pages 1223–1235, October 1995.

[17] D.J. Shippy and T.W. Griffith. POWER2 fixed point, data cache, and storage control units. *IBM Journal of Research and Development*, pages 503–524, September 1994.

[18] S. Simha. R4400 Microprocessor Product Information. Technical report, MIPS Technologies Inc., September 27 1993.

[19] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC Microprocessor. *IEEE Micro*, pages 8–17, October 1994.

[20] Chip Weems and Steve Dropsho. Real-Time RISC Processing. Technical Report TR-95-41, University of Massachusetts- Amherst, 1995.