A Day in the Life of a Data Cache Miss

Tejas Karkhanis and J. E. Smith
Dept. of Electrical and Computer Engineering
University of Wisconsin-Madison

Abstract

The activity within a processor following a cache miss is studied via a series of simulation experiments. This is a preliminary step toward developing ways of mitigating data cache miss penalties, especially for long misses. With a modest-sized reorder buffer (ROB) of 64 entries, structural blockages due to a full ROB are the major cause of the cache miss penalty. For the SpecINT2000 benchmarks, about 90% of long cache misses result in a blocked ROB. After structural constraints are removed, data dependences are found not to be a major cause of performance loss. In fact, instruction issuing can proceed at full speed for at least one thousand cycles (the length we sampled) beyond a data cache miss. In some cases we found control dependences do pose a problem, however. That is, there are many mispredicted branches that depend on data from a cache miss. With an 8K gshare predictor, 30% or more of data cache misses feed into a mispredicted branch in 7 of 12 benchmarks. Finally, we discuss implications for future processor designs. This includes lengthening ROBs (and enlarging physical rename locations), coupled with a more modest enlarging of instruction issue windows.

1 Motivation

The overall performance impact of data cache misses is a function of the number of misses and the penalty per miss. Most research is aimed at reducing the number of misses, for example by prefetching, using victim buffers, etc. To reduce the miss penalty, common solutions are focused on the memory system, e.g. by adding levels to the memory hierarchy. In contrast, we are interested in reducing the data cache miss penalty by applying methods *within the processor*. As a first step, we have been studying what happens within a processor when a data cache miss occurs.

Before proceeding, we define two terms. First, *miss delay* is the latency for fetching miss data from the lower level in the memory hierarchy where it resides. Second, *miss penalty* is the performance loss due to a cache miss. Miss penalty is related to miss delay, but in some cases all or part of the miss delay may be hidden. In a processor with hierarchical caches, miss delays vary over a wide range, from 10 cycles or less (for an L1 to L2 miss) to perhaps hundreds of cycles (for a miss all the way to main memory). To keep the discussion simple, we will consider two cases: *short misses* and *long misses*. And, most of our emphasis is on long misses because of their increasing importance due to the widening processor/memory performance gap.

From the processor perspective, there is a penalty following a long miss because issuing of useful instructions

slows, and eventually stops, when the miss occurs. There are three reasons this can happen:

1) *There is a structural blockage*. For example, the load that misses moves to the head of the reorder buffer and blocks; waiting to retire. Then, the reorder buffer fills, and/or the processor runs out of physical registers, dispatch stops, eventually issue stops, and nothing more happens until the miss data returns from memory (or some lower level in the cache hierarchy).
2) *There is a data blockage*. That is, instructions that depend on the load data, either directly or indirectly, begin to pile up in the issue window, eventually the window is full of data dependent instructions, and issue stops. In a sense, this is a structural blockage, but it is a direct result of data dependences. This is different from case 1) because the blockage in 1) occurs purely for structural reasons; no dependences are necessarily involved.
3) *A control dependence stops issuing of useful instructions*. In particular, if a mispredicted branch is dependent on the load miss data, all instructions after the branch will eventually be flushed. In effect, issuing of *useful* instructions is blocked by the control dependence on the miss data.

In the research described here, we study the above three causes of the data cache miss penalty and discuss the relative importance of each. We then make observations regarding future processor microarchitectures that are tolerant of data cache miss delays.

2 Methodology

In order to investigate the three causes of performance loss, we performed a number of simulation-based experiments. Because the types of blockages, especially the structural blockage, depend on a processor's microarchitecture, we first describe the processor model. Then we describe the simulation model, benchmarks, and the general method used.

2.1 Processor Model

The relevant components of the processors we study are listed below. They have been chosen to be consistent with the characteristics of modern superscalar processors. Key components are:

1) Modest issue width – 4-way. It appears that the trend toward wider issue widths may have topped out, with emphasis shifting to deeper pipelining as a way of increasing parallelism.
2) Renaming in physical register file. The physical register file is larger than the logical file, and all renamed values are held in the physical file. This has become the standard method for implementing dynamic superscalar processors.
3) The issue window and reorder buffer are separate and may be sized independently. All in-flight instructions have a ROB entry, but only instruction waiting for data and/or functional units wait in the window. After they issue, instructions are removed immediately from the window, making room for others.

4) The issue window "collapses"; that is, the holes due to issued instructions may be re-filled. This maximizes the issue opportunities following a cache miss and reduces the likelihood that a full window will eventually cause a structural blockage.

Table 1: Processor Configuration

ROB size	64 entries
Issue Window Size	32 entries
LSQ size	64 entries
IF, ID, IS, IC Width	4 instructions/cycle
Branch Predictor	gshare: 8K entries, 13 bit GHR
Return Address Stack	64 entries
Branch Target Buffer	4K sets, 4 way
Functional Units	4 Int. ALUs, 1 Int. MULT/DIV
	4 FP ALUs, 1 FP MULT/DIV
L1 I and D Caches	512 sets, 4-way SA, 16 byte block size, LRU
L2 Unified Cache	N/A; all L1 misses are 1000 cycles
Pipeline Depth	5 stages before the issue stage

2.2 Simulation Model

To evaluate performance, we used a modified version of the SimpleScalar simulator [1]. The main modification is that the register update unit (RUU) is replaced with a separate issue window and re-order buffer. Up to four independent instructions per clock cycle may issue out-of-order from the issue window. Table 1 summarizes the processor parameters used in all simulations.

2.3 Workload

We simulated SPEC 2000 INT benchmarks compiled with base optimization level (-arch ev6 –non_shared –fast). The reference inputs were used and all benchmarks were fast forwarded 400 million instructions before being simulated for 800 million committed instructions.

2.4 Simulation Approach

We perform simulation experiments intended to highlight the above-listed causes of data cache miss penalties. To do this, we remove other events that disrupt smooth instruction issue. In particular, we assume an ideal instruction cache, and, for some experiments, we assume the branch predictor is ideal as well. A typical simulation experiment does the following:

1) Simulate until the data cache is warmed up. After warmup, wait for the first data cache miss.
2) When the miss occurs, disable all additional data cache misses so that the effects of the single miss can be monitored.
3) Wait a very long time (1000 clock cycles) and monitor the instruction window, ROB, and instruction issue, observing the effects of the miss.
4) During (and immediately after) the 1000 clock cycles, collect relevant data, then let the processor settle back to normal issuing

Table 2: Structural Blockage Statistics

Benchmark	Avg. # insns issued after the miss	Avg. #Insns in window after 1000 cycles	Fraction of samples where ROB fills
bzip2	44.1	13.1	1.0
crafty	44.6	9.6	0.8
eon	55.2	6.0	1.0
gap	56.8	10.7	1.0
gcc	51.7	8.2	0.9
gzip	42.0	8.7	0.9
mcf	55.8	5.5	0.9
parser	44.2	7.4	1.0
perl	49.6	6.6	0.8
twolf	49.6	12.9	0.8
vortex	49.7	3.5	1.0
vpr	27.0	16.9	0.6

and re-enable data cache misses to monitor the next miss.

The above is repeated, data is accumulated, and then is summarized. In effect, we collect data from a number of isolated samples. Intentionally, this method avoids the complications of multiple overlapping data cache misses. The advantage is that it provides clearer insight into the first-order phenomena that take place. Toward the end of the paper we briefly discuss multiple cache misses.

3 Experiments

3.1 Structural Blockage

The first set of experiments is focused on determining the nature of the structural blockage caused by a finite ROB and physical register set. We also study the transient behavior as the miss occurs and instruction issue eventually blocks.

For these initial experiments, we set the window to 32 entries, the ROB to 64 entries and use 64 physical registers. These numbers are typical of what one might find in a current superscalar processor. Also, the Load/Store queues are sized large enough (64 entries) that they will not be the initial cause of a structural blockage. We assume ideal branch prediction.

As described above, after warmup, the simulator takes the first data cache miss and does not return the data for 1000 cycles. In the meantime, other loads and stores are forced to hit in the cache.

Table 2 contains statistics for each of the benchmarks (listed in column 1). The second column is the average number of instructions that issue *after* the miss is detected. These instructions are all independent of the load data. The third column is the number of instructions left in the window after issuing eventually stops. In general, these instructions are dependent, either directly or indirectly, on the cache miss data. The fourth column is the fraction of the samples where instruction issue blocks because the ROB fills (as opposed to data dependences filling the window). Because the ROB and issue window are relatively small compared with the long miss delay (1000 cycles), in-

struction issue always blocks because either the ROB fills or the window fills.

Table 3: Data Dependence Blockage Average Statistics

Benchmark	Avg. # insns issued after the miss	Avg. #insns dependent on load miss data
bzip2	3950.1	17.8
crafty	3746.9	20.1
eon	3922.9	22.4
gap	3292.9	31.6
gcc	3678.4	17.2
gzip	3501.6	96.2
mcf	3862.7	11.5
parser	3648.8	32.6
perl	3518.8	30.3
twolf	3672.8	44.7
vortex	3606.5	7.8
vpr	2371.8	24.0

As shown in column four, about 90% of the time, the ROB usually fills first and causes dispatch (and issue) to stop. This indicates that the purely structural (ROB) blockage eventually leads to the miss penalty. Or, conversely data dependences are usually *not* the problem; there are enough instructions independent of the load so that issue could have kept going if it were not for the ROB. In columns two and three we see this reflected in the large number of issuing instructions after the cache miss occurs (typically over 40), and the small number of dependent instructions left in the window (about 9 on average) waiting for the load to return. This highlights the fact that data dependences on the missing load are not the bottleneck. The bottleneck occurs for purely structural reasons. The only exception is benchmark *vpr*, where a significant number of dependent instructions do pile up and block issue because of a full window about 40% of the time.

3.2 Data Dependences

Because the previous section shows that structural blockage caused by the ROB (and physical register file) is the major performance impediment, the next set of experiments remove these limitations by making the ROB, physical register file, and load/store queues very large (4K entries each). The latency to main memory is set at 1000 cycles, and statistics are given in Table 3. The main difference is that now a large number of instructions can issue after the miss occurs (column 2). Because the maximum issue width is four and the miss delay is 1000 cycles, at most 4000 instructions can issue under ideal conditions. For most benchmarks, over 3500 actually do issue. This indicates that the vast majority of instructions following the miss are independent of the miss data. Meanwhile, after the 1000 cycle interval, we measure only a few tens of dependent instructions left in the window; the average is about 30, with the largest number being 96 for *bzip*.

To get additional insight, we studied the average number of dependent instructions each clock period after the data cache miss occurs. These graphs indicate that in most cases, the dependent instructions follow the load fairly closely, after which all remaining instructions are independent. Representative graphs for SpecINT2000 benchmarks are in Fig 1. In some benchmarks, like *mcf*, dependent instructions come immediately after the missed load. Other benchmarks such as *gcc* show a growing number of dependent instructions through the entire 1000 clock period interval.

Table 4: Load-to-Mispredicted Branch Statistics

Benchmark	Fraction of load misses which drive a mispredicted branch	Avg. distance to mispredicted branch
Bzip2	0.01	33.5
Crafty	0.30	20.3
Eon	0.18	30.6
Gap	0.33	27.0
Gcc	0.35	32.4
Gzip	0.01	27.7
Mcf	0.44	32.4
Parser	0.08	35.9
Perl	0.40	30.2
Twolf	0.37	65.6
Vortex	0.16	41.2
Vpr	0.47	31.3

3.3 Control Dependences

If a mispredicted branch is dependent on the load miss, either directly or indirectly, all speculative instructions following the branch perform no useful work. In effect, useful instruction issue stops immediately after the mispredicted branch. To determine the importance of these control dependences on the data cache miss penalty, we performed experiments as in the previous section, but branch prediction is no longer ideal. We collect data to determine how often a dependent misprediction follows the load miss, and how far from the load it occurs. Note that mispredictions *independent* of the load are still properly handled because they can be resolved and fetch/issue can continue "under" the miss.

Results are in Table 4. In column 2 we see that for many of the benchmarks, a significant fraction of load misses feed branches that are mispredicted. Overall, about 25% of the miss loads lead to a mispredicted branch, but there is a lot of variance across benchmarks. For 7 of the 12 benchmarks 30% or more of the miss loads provide data that eventually resolves a mispredicted branch. This makes some intuitive sense, because tests of data loaded from memory are more likely to be mispredicted than branches based on an internally generated value like a loop count, for example. Furthermore, load miss data is data that has not been accessed for a

long period of time (and consequently has not been used to decide a branch for a long period of time).

Column 3 shows the number of instructions that separate the load from the mispredicted branch (i.e. averaged over those cases where there is a dependent mispredicted branch). This number is typically about 30, with a range from 20

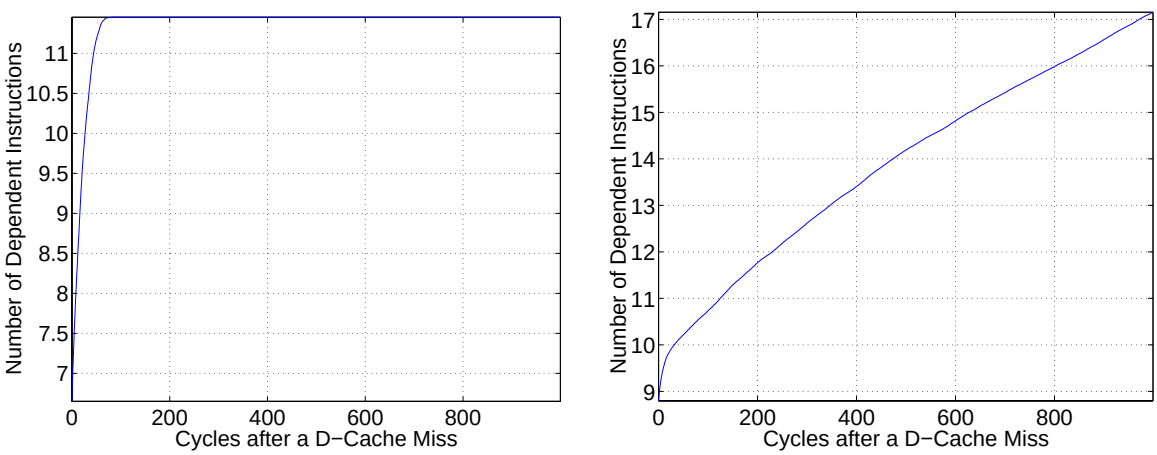

Figure 1: Accumulation of dependencies on a missed load

to 65. This indicates relatively little useful work is done in the control dependent case.

Our conclusion is that these control dependences may be a major cause of performance loss when there is a long data cache miss. Furthermore, this loss is a property of the program and the quality of the branch predictor. A larger window or ROB will not help in this case.

4 Implications for Processor Design

Assuming extremely long miss delays was an experimental device for collecting basic data. As stated earlier, in reality, misses will vary in length from about ten cycles to hundreds of cycles (for a miss all the way to main memory).

For short misses, we can conclude that a 10-cycle miss delay is likely hidden with windows and ROBs about the same size as are currently being used. From Table 2, we see that about 40 instructions can issue after the miss is detected, and at the rate of four per cycle, this is ten cycles worth of instructions. For multiple outstanding misses to the L2 cache, a slightly larger ROB may be called for, but in most cases a window of 32-48 instructions is probably enough, because the miss load has a relatively small number of nearby dependent instructions that take up window space.

For long misses, implications are more interesting, and in many respects this is a more important case for future designs. Relatively speaking, main memory latencies continue to grow; they now consume over 100 cycles and may reach many hundreds of cycles in the near future. Although they are less common than the short misses, the penalty is much higher. The remainder of this section focuses on the long miss case.

The first conclusion is that data dependences do *not* inhibit performance in a major way. If the structural resources are available, instructions can continue issuing and can overlap even very long miss latencies.

A second conclusion is that control dependences are significant in some cases. In these cases, even if structural blockages are removed, mispredicted branches will inhibit performance. Consequently, we have one more reason to use highly accurate branch predictors. In this study, we used a relatively modest predictor by today's standards (8K gshare).

A third conclusion is that if the window and ROB are somehow enlarged to cover long misses, they should be scaled up in different ways. As future research we plan to investigate this further, but we can make some rough estimates of how they may scale. First, the ROB must be large enough to sustain a high rate of instruction issue during the miss delay. For example, if the issue width is n and the miss delay is D cycles, then a ROB of size nD should be adequate. In other words, to cover a miss delay of 200 cycles with a four-issue machine would require a ROB of about 800 entries. Of course, the number of rename locations, store queue entries, etc. for uncommitted values must be commensurately sized.

On the other hand, the window must be enlarged to hold the dependent instructions that back up behind each outstanding long miss. If the original window size is W, if there are m outstanding long misses, and if each has d dependent instructions, then the window should be enlarged to about $W'=W+dm$. For example, if m is 6 and d is 30, then the window should be enlarged by 180 slots, for a total of slightly more than 200. That is, it is significantly smaller than the ROB.

Others have noted that when there is a long data cache miss, resources may sit idle for long periods of time. Consequently, it has been proposed that the processor should use the resources by context switching to a different thread (in a fine-grained multi-threaded paradigm) [2], or execute speculative threads to perform pre-fetching or pre-execution [3,4]. Our results show that, in fact, many blockages are not inherent, but are structural. By providing additional window and ROB resources (which may be needed by the aforementioned methods, also) the otherwise idle resources can be used for *real* execution of the *main thread*.

Of course, enlarging the window, ROB (and load/store queues and physical registers) is not a simple matter, although proposals have been put forward for each. Ways of building more scalable windows have been proposed in [5,6], and these may provide adequate window sizes. Recently, a proposal for extremely large windows [7] has been put forward, with the same objective as we are studying – covering long cache miss latencies. For large physical register files, using a hierarchy as in [8,9,10] may be a good approach. The IBM Power4 [12] has ROB entries composed of blocks of instructions rather than single instructions. Consequently, the Power4 can maintain 200 in-flight instructions. This coarser granularity approach could be used to expand the ROB even larger. Increasing the ROB also entails increasing the number of physical registers. Schemes such as speculative retirement and large history buffers [13,14] may be useful as alternatives to increasing the number of physical registers.

5 Conclusions and Future Research

The main conclusion is that in current superscalar processors, data and control dependences are typically not the

main performance limiter when there is a long cache miss – rather structural features are often the limiters (e.g. ROB size and window size). If the ROB and window size are separated (i.e. not an RUU), then it appears that the ROB needs to be enlarged significantly more than the window size in order to cover long misses.

Future research should be directed at ways of covering cache misses by implementing structural resources that allow the covering of longer latency misses. The work in [7, 13] are interesting steps in that direction, but we feel that the design space is large and a variety of other solutions should be studied

6 Acknowledgements

We would like to thank Timothy Heil and Eric Rotenberg for several valuable discussions. This work is being supported by SRC grant 2000-HJ-782, NSF grants EIA-0071924 and CCR-9900610, Intel and IBM.

References

[1] D. Burger et al., "Evaluating Future Microprocessors: The SimpleScalar Tool Set," *Technical Report TR-1308*, University of Wisconsin-Madison Computer Sciences Dept., June 1996.

[2] Anant Agarwal, et al., "April: A Processor Architecture for Multiprocessing," *Proceedings of the 17th Annual International Symposium on Computer Architecture,* June 1990, pages 104-114.

[3] R. Balasubramonian et al., "DynamicallyAllocating Processor Resources Between Nearby and Distant ILP," *28th Int. Symp. on Computer Architecture*, pp. 26-37, July 2001.

[4] C. B. Zilles and G. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," *27th Int. Symp. on Computer Architecture*, pp. 172-181, June 2000.

[5] R. Canal and A. Gonzalez, "A Low-Complexity Issue Logic", 2000 *Int. Conf. on Supercomputing*, pp. 327-335, May 2000.

[6] P. Michaud and A. Seznec, "Data-flow Prescheduling for Large Instruction Windows in Out-of-Order Processors", *7th Int. Symp. on High Performance Computer Architecture*, pp. 27-306, Jan. 2001.

[7] A. Lebeck, et al., "A Large, Fast Instruction Window for Tolerating Cache Misses," *29th Int. Symp. on Computer Architecture*, May 2002.

[8] J. L. Cruz et al., "Multiple-Banked Register File Architectures", *27th Int. Symp. on Computer Architecture*, pp. 316-325, June 2000.

[9] J. Zalamea, et al., "Two-Level Hierarchical Register File Organization for VLIW Processors", *33rd Int. Symp. on Microarchitecture*, pp. 137-146, Dec. 2000.

[10] R. Balasubramonian et al., "Reducing the Complexity of the Register File in Dynamic Superscalar Processors," *34th Int. Symp. on Microarchitecture*, get pages, Dec. 2001.

[11] D. Sorin, et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," *29th Int. Symp. on Computer Architecture*, May 2002.

[12] J. M. Tendler, et al., "IBM Power4 System Microarchitecture," IBM Journal

of Research and Development, pp. 5-26, Jan. 2002.

[13] P. Ranganathan, et al., "Using Speculative Retirement and Larger Instructions Windows to Narrow the Performance Gap between Memory Consistency Models," *9th ACM Symposium on Parallel Algorithms and Architectures*, June 1997.

[14] C. Gniady, et al., "Is SC + ILP = RC?," *26th Int. Symp. on Computer Architecture*, May 2002.