The Effects of Out-of-Order Execution on the Memory System

Ph.D. Research Proposal

Aamer Jaleel
Dept. of Electrical and Computer Engineering
University of Maryland, College Park
ajaleel@eng.umd.edu

ABSTRACT

The use of large instruction windows coupled with aggressive out-of-order and prefetching capabilities has provided significant improvements in processor performance. We quantify the effects of increased out-of-order aggressiveness on a processor's memory ordering/consistency model as well as an application's cache behavior. A preliminary study reveals that increasing reorder buffer sizes cause less than one third of total memory instructions issued to be executed in actual program order. Additionally, increasing the reorder buffer size from 80 to 512 entries results in an increase in the frequency of memory traps by a factor of 100-150 and an increase in total overhead of 20-60%. Even more, the reordering of memory instructions increases the cache misses by 25-80% in the L1-cache and 50-200% in the L2-cache.

These findings reveal that increased out-of-order capability can waste energy in two ways. First, re-fetching and re-executing instructions flushed due to replay traps requires the fetch, map, and execution units to dissipate energy on work that has already been done before. Second, an increase in the number of cache accesses and cache misses needlessly dissipates energy. Both these side effects can be related to the reordering of memory instructions. Thus, to avoid wasting both energy and performance, we propose a mechanism called "windowing" to throttle the degree by which memory instructions are issued out-of-order. By investigating various window sizes in traditional load/store queues, we propose to explore dynamic and static mechanisms to reduce the negative effects of out-of-order execution of memory instructions.

1 INTRODUCTION

The instruction window is categorized as one of the most important design parameters in the design of modern high performance processors. Many previous studies have shown that increasing the size of reorder buffers, issue queues and load/store queues can lead to increased performance. Consequently, much research has looked at the feasibility of increasing the size of these hardware data structures without negatively impacting clock cycle time. However, when one considers “real” effects due to the reordering of memory instructions, the potential performance gains largely disappear.

By varying the aggressiveness of an out-of-order core in terms of reorder buffer sizes, issue queues, load/store queues, and renaming registers, we observe two potential pitfalls of aggressive out-of-order mechanisms present in real systems that many previous simulation based studies have not addressed. First, aggressive out-of-order execution conflicts with a processor’s memory consistency and ordering model by requiring the processor to take frequent expensive *replay traps*, i.e. flushing the pipeline and re-executing a window of instructions. Second, highly aggressive out-of-order mechanisms can destroy cache locality, thereby causing an application to suffer from a higher number of cache misses than a lesser aggressive out-of-order mechanism. Our preliminary study shows that even though aggressive out-of-order mechanisms enhance performance, the reordering of memory instructions can cause significant overhead in the memory system. Thus, we propose to investigate mechanisms required to control the degree by which future aggressive out-of-order processors issue memory instructions out of program order.

1.1 Motivation

The designs of modern high performance microprocessor architectures rely on very aggressive hardware mechanisms to maximize processor performance. Techniques such as branch prediction, data speculation, load speculation, hardware and software prefetching, cache line prediction and pipeline scheduling speculation are a few of the numerous techniques utilized by modern high performance microprocessors to tolerate the growing gap between the processor and DRAM system. The different techniques mentioned strive towards one common goal—boost processor performance by continuing to do *possibly* useful work rather than stay idle.

To avoid staying idle, most ILP processors improve processor performance by executing instructions in an order different from sequential program order. This is called instruction reordering and is also more commonly known as out-of-order execution. The motivation for out-of-order execution is to overlap useful work with work that takes a while to do. To be capable of executing instructions in an order different from actual program order, instructions are fetched into an instruction window. Each cycle the processor's out-of-order hardware consults the instruction window for instructions that are ready to execute. If an instruction has all of its dependencies resolved and is ready to execute, the out-of-order hardware issues it to the appropriate functional unit. Thus, by overlapping useful work with work that takes a while to do, modern microprocessors achieve a much higher performance with out-of-order execution than with in-order execution.

Processor performance, in general, is determined by the amount of time it takes to execute a given program. Mathematically, processor performance is expressed by *IPC x clock speed*, where IPC is Instructions Per Cycle, i.e. the average number of program instructions completed in a processor clock cycle. From this equation, it is easy to realize that processor performance can be improved by either increasing IPC, clock speed or both. These methods for increasing microprocessor performance belong to two well known trends in the architecture community: *brainiacs* and *speed demons*. *Brainiacs* improve processor performance by concentrating only on increasing IPC. They attempt to build smarter processors that are capable of dynamically exploiting maximum application ILP. Using large instruction window and queue sizes and complex issue logic schemes to execute several instructions at a time, the *brainiac* approach

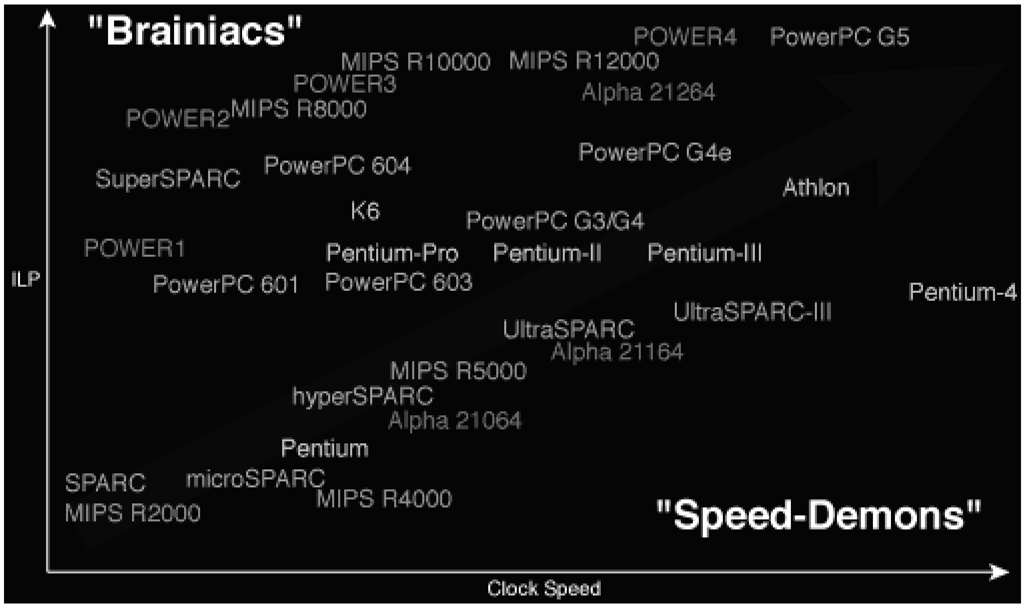


Figure 1: Brainiacs Vs. Speed-Demons.

increases IPC while maintaining low clock speeds (e.g. IBM's POWER2, MIPS R10000 and others as shown in Figure 1). *Speed demons*, on the other hand, only concentrate on increasing clock speeds to improve processor performance. Their design philosophy is to accommodate any amount of design complexity as long as it does not compromise the primary goal of maintaining high clock speeds. With continued decrease in feature size and improved microarchitectural techniques in microprocessor design, *speed demons* have been able to continue to increase clock speeds and achieve high processor performance by relying on smart compilers to expose an application's inherent ILP (e.g. Pentium 4, UltraSPARC-III).

Exactly which path of design decision (*brainiac* or *speed demon*) is the "right" path for improving microprocessor performance is a hot debate. From Figure 1, we observe that some microprocessor vendors such as the DEC/Compaq and MIPS started off as *speed demons* (Alpha 21064, Alpha 21164, MIPS R2000, MIPS R4000) and then changed their design philosophy to *brainiac* (Alpha 21264, MIPS R5000, MIPS R8000, MIPS R1000, MIPS R12000). Sun on the other hand started off as a *brainiac* (SPARC, microSPARC, superSPARC, hyperSPARC), however of late has changed to the *speed demon* design philosophy (UltraSPARC-III).

In general, it is desirable if the design philosophy of a microprocessor were both *brainiac* and *speed demon*, however, the two design philosophies are often at odds against one another. This is because the complex logic required to extract ILP in the *brainiac* approach cannot handle the high clock speeds desired by the *speed demons*. One such example is the complex out-of-order issue logic, the core of an ILP processor. It is a well known fact that a processor's out-of-order efficiency (i.e. ILP extraction capability) depends on the total number of instructions it views at any given time, i.e. the instruction window size. The more instructions an out-of-order core views, the more opportunity a processor has to exploit an applications inherent ILP [4, 17, 19, 22]. With the growing gap between the processor and DRAM system, the need for larger instruction windows to exploit ILP has become extremely important to avoid processor idle time. However, with increasing instruction window sizes, the instruction selection and issue logic, synchronization logic, and required data paths become critical paths with latencies that cannot meet high clock frequencies. Consequently, on going research has proposed several novel techniques to provide the functionality of large instruction window sizes yet maintain high clock speeds [13, 17, 18].

1.2 The Problem

Though large instruction windows and aggressive instruction schedulers provide the processor with a large number of instructions deep into an application's instruction stream, selecting and issuing to execute such distant independent instructions inherently causes an application's instructions to be reordered. The reordering of ALU instructions poses minimal effects on program execution; however, the reordering of memory instructions can affect program execution in two distinct ways:

- **Increased Replay Traps:** The reordering of memory instructions can create a variety of hazards that can affect the correct execution of an application. For example, when using load speculation [20, 18], if it is later determined that the speculated load utilizes the same effective address as an older but unresolved store, then the load causes a fault, and the processor must replay the faulting load instruction. This is known as a “replay trap”. A replay trap can either be handled by flushing the pipeline and restarting execution at the faulting instruction or re-executing only the faulting instruction and all of it’s direct and indirect dependent instructions. Even though the re-execute method is better than the pipeline flush method, the complexity in logic required to determine and re-execute an entire dependence chain of the replay trap causing instruction is relatively expensive and can become even more so with increased instruction window sizes [20]. Thus, with the pipeline flush method of handling traps, if the frequency of traps increases, significant performance and energy can be wasted in re-fetching and re-executing instructions flushed.
- **Increased Cache Misses:** Executing memory instructions speculatively or in an order different from actual program order can negatively impact an application’s cache locality. For example, a load instruction issued out-of-order can evict data required by both older and future memory instructions that are waiting to be issued. When the older or future memory instruction later executes and misses in the data cache, energy is needlessly wasted in re-fetching and re-filling the recently evicted data cache line. Even more, if the out-of-order issued load instruction is speculative, energy is unnecessarily dissipated by accessing the data cache and evicting a data cache line in the event of a cache miss. Thus, with increase in out-of-order capability, an increase in the frequency of conflict misses due to speculative or non-speculative memory instructions can result in unnecessary thrashing of the data cache resulting in wastage of energy.

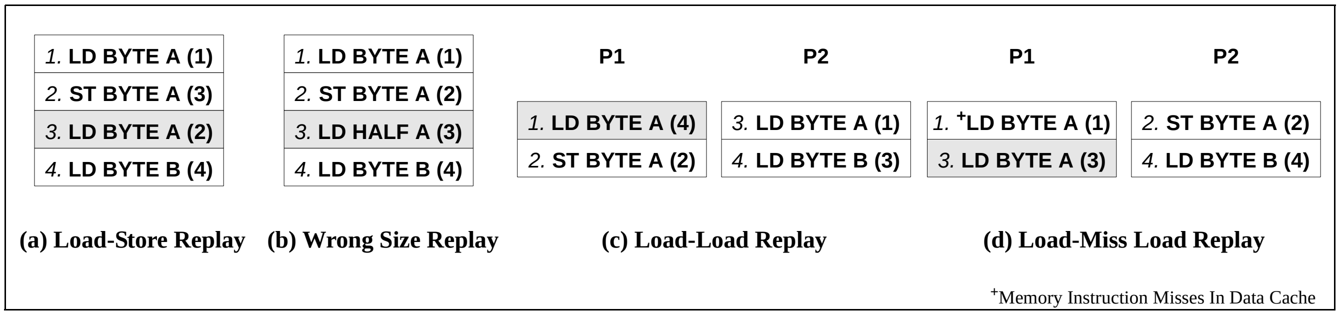


Figure 2: Classification of Replay Traps. The figure illustrates the different types of replay traps that can occur in both uniprocessor and multiprocessor scenarios. In the examples, due to the replay trap, re-execution starts from the shaded instruction.

2 BACKGROUND

2.1 Reordering of Memory Instructions

By allowing a processor to exploit instruction level parallelism (ILP) via large instruction windows, both ALU and memory instructions are executed out of program order. Since register renaming maintains the dependencies of ALU instructions, out-of-order issue of ALU instructions poses no threat to functional correctness. On the other hand, since memory dependencies are resolved only after issuing to execute, out-of-order issue of memory instructions can pose threats to functional correctness especially if two out-of-order issued memory instructions access the same memory location. In such a scenario, the processor may be required to initiate a replay trap. A replay trap occurs when the processor must roll back the state to force accesses to a particular memory location in order, or to handle different-sized accesses to the same memory location. Figure 2 illustrates the different types of replay traps. Numbers in parenthesis signify the order which instructions are issued to execute and numbers in italics signify actual program order.

- **Load-Store Replay:** A load-store replay trap occurs when a newer load is issued before all prior store addresses are resolved. In the event that the processor detects a newer load executing out-of-order with respect to an older store that it depends upon, a load-store trap is initiated. This is required so that the newer load acquires data from the store rather than stale data from the cache. For example, in Figure

2(a), if memory instruction number three (a load) executes before the store instruction (both of which access the same memory location A), then the value loaded from the data cache will be incorrect. Microprocessors that use load speculation must handle this replay trap to ensure functional correctness, e.g. Alpha, POWER4. [1, 2, 24]

- **Wrong Size Replay:** A wrong-size replay trap occurs when the data for a newer load is partially in the store queue and partially in the data cache. Figure 2(b) illustrates this via an example. The second load instruction in the program requires reading a half word (two bytes) starting at memory location A, however a prior store writes a byte to the same memory location. When the processor detects this, the load must be re-executed after the older store instruction drains it's data into the cache. Note that this replay trap can occur even if memory instructions are issued in program order [1, 2, 24]. As a result, all high performance microprocessors must be able to detect and overcome this hazard.
- **Load-Load Replay:** A load-load replay trap is initiated when two loads to the same memory address are issued out-of-order. In a uniprocessor environment this poses no problems, however in the case of a multiprocessor environment, out-of-order issue of loads can cause subtle memory consistency problems. For example, if two loads to the same address are issued out-of-order, and a different processor changes the value between the execution of these two loads, then, the newer load instruction may obtain the older value and the older load may obtain a newer value. Figure 2(c) illustrates this via an example. This load-load ordering problem can either be handled in hardware or explicitly by software programmer. In the software approach, if a relaxed memory consistency model is supported, processors provide a *memory barrier* instruction that allows the programmer to enforce ordering among memory instructions wherever needed. However, extensive use of memory barriers can negatively hurt performance [19]. Thus, hardware support, via replay traps, is provided by some processors to guarantee load-load ordering to the same address. (e.g., Alpha[1, 2], POWER4[24], and MIPS R10000[3])

- **Load-miss Load Replay:** A load-miss load replay trap is initiated when two loads to the same memory address are issued and the first load misses in the data cache and already has a miss information/status holding register (MSHR) allocated to it (Figure 2(d)). An MSHR keeps track of an outstanding memory request to a single cache line [14]. It is used to "merge" multiple requests to the same cache line by keeping track of all destination registers waiting for data from memory. When the data arrives from memory, the MSHR fills all the outstanding destination registers that were waiting for the same cache line. Like the load-load replay trap, a subtle case of memory inconsistency can occur if there is an intervening store from a different processor between two loads to the same memory address. In such a scenario, the MSHR provides the second load instruction with stale data. Thus, to avoid this source of memory inconsistency, the processor replay traps until the data for the first load is loaded into the destination register. Note that this replay trap occurs even though memory instructions are issued in program order. (e.g., Alpha[1, 2]).
- **Cache Line Conflicts:** With out-of-order execution there can be a number of memory instructions in progress at the same time. A cache line conflict occurs when two outstanding misses to different physical addresses request the same cache line/set. When the processor detects a cache miss for an older memory instruction and detects that an MSHR is already allocated for the same cacheline, a replay trap is generated for the newly issued memory instruction [1, 2].

Irrespective of the type of replay trap, the mechanisms currently used to handle a replay trap are identical to those involved in handling branch mispredicts. When an instruction executes and causes a replay trap, the fact is noted in the reorder buffer entry. While committing instructions, if the processor detects a replay trap, the pipeline is flushed and execution is restarted at the faulting instruction. We show that while these replay traps occur only a fraction of the time, increasing out-of-order capability exposes them to be an overwhelming hazard to performance.

2.2 Related Work

It is widely believed that a processor's out-of-order efficiency depends on the number of instructions it views at a given time, i.e. the reorder buffer/instruction window size. The more instructions an out-of-order core views and the wider the issue widths, the more an out-of-order core can exploit an application's instruction level parallelism (ILP). Furthermore, more aggressive techniques such as load speculation and data-value prediction allow the instruction scheduler to be less stricter, thereby exploiting more ILP.

It is also known that larger instruction windows conflict with increasing clock speeds. A good deal of recent effort is aimed at designing efficient and fast issue/selection logic that allows for larger instruction window sizes while still maintaining high clock speeds. Henry et al. proposed an alternate binary tree circuit implementation for the wakeup logic [13], Onder et al. proposed explicit wake-up lists associated with executing instructions [17], Lebeck et al. tackle the instruction window size by proposing an alternate *waiting instruction buffer* (WIB) [15], and Akkary et al. propose a checkpoint and recovery mechanism to recover from branch mispredicts with larger instruction window sizes [4].

Since larger instruction windows expose aggressive out-of-order processors to more load/store communications, Park et al propose techniques to scale the load/store queue size using segmentation [19]. Furthermore, to allow for load speculation, Calder et al. tackle the false memory aliasing problem and propose four different mechanisms for load speculation. Loads predicted to not alias to older stores are issued speculatively. If the load is mispredicted, instructions are squashed and re-executed [20].

Burger et. al. point out that when using aggressive latency tolerance techniques, memory bandwidth, particularly pin bandwidth, and not raw access latencies prevents processors from gaining higher performance. To quantify this they decomposed execution time into processing cycles, raw memory latency stall cycles, and limited bandwidth stall cycles. Using this mechanism they were able to show that applications running on future aggressive processors will stall primarily due to memory-bandwidth limitations [6].

3 EXPERIMENTAL METHODOLOGY

For the purpose of our study, we use a validated execution-driven Alpha 21264 simulator [9, 10]. The simulator has a detailed memory system with two-way set associative L1 instruction and data caches, 4-way set associative unified L2 cache, 8 MSHRs per cache, and 128-entry fully associate TLBs. The simulator also models a detailed SDRAM memory and bus model[8]. The simulator also models two prefetching schemes a) sequential prefetching without stream buffers and b) stride prefetching with a 256 entry 2-way associative stride table and eight 8-entry stream buffers. With sequential prefetching, the processor requests the next four cache-lines on a cache miss. The data gathered in this proposal considers only sequential prefetching. Like the Alpha 21264 processor, the simulator allows for aggressive out-of-order techniques such as load speculation; i.e., the processor issues load instructions even though prior store instructions aren't resolved. Additionally, like the Alpha 21264 processor, the simulator detects memory ordering problems like those mentioned in Section 2.2 and handles them in the same way exceptions are handled—the pipeline is flushed and instructions are re-fetched starting from the faulting memory instruction. Note that these exceptions do not require handler support—they merely require re-execution of instructions starting from the older memory instruction. Additionally, like the Alpha 21264 processor, the simulator maintains a 1024-entry store-wait data structure to avoid recurring store-replay traps. If a load instruction causes a store-replay trap, the fact is noted in the store-wait table. When issuing a load instruction and there are prior unresolved stores, the processor issues a load only if the PC associated with it is not listed in the store-wait table. The store-wait table is cleared unconditionally every 16,384 cycles [1, 2].

Table 1: Processor Parameters

Configuration Name	ROB Size	Issue Width INT/FP	IssueQ Size INT/FP	# Functional Units**	LQ/SQ Size	Renaming Registers INT/FP
Alpha 21264 x 1	80	2/1 — 8/4 Way	20/15	4/4/1/1	32/32	41/41
Alpha 21264 x 2	128	4/2 — 8/4 Way	40/30	8/8/2/2	64/64	82/82
Alpha 21264 x 4	256	4/2 — 8/4 Way	80/60	16/16/4/4	128/128	164/164
Alpha 21264 x 8	512	4/2 — 8/4 Way	160/120	32/32/8/8	256/256	328/328

Table 2: Cache Configurations

L1 Size	L1 Latency	L1 Line Size	L2 Size	L2 Latency	L2 Line Size
16 KB	3	32 Bytes	512 KB	8	64 Bytes
64 KB	3	64 Bytes	2 MB	15	64 Bytes

Table 3: Benchmarks

SPEC SUITE	art	gcc	mcf	parser	perlbmk
	swim	twolf	vortex	vpr	
Olden	em3d	health	mst	treeadd	

Table 4: Processor Instruction Issue Configurations

Configuration Name	Configuration Description
ALU-in / MEM-in	Instructions are issued only when they reach the head of the reorder buffer (ROB). Speculation is enabled. By definition of a ROB this enforces inorder issue.
ALU-out / MEM-in	ALU and memory operations are issued out-of-order with speculation enabled.
ALU-out / MEM-out	ALU operations are issued out-of-order. Memory operations are issued from the issue queues in program order.

****INT ALU/INT MULT/FP ALU/FP MULT**

For this baseline study, we vary the aggressiveness of the out-of-order core by changing the ROB size, issue widths, issue queue and load-store queue size, number of functional units, and the number of renaming registers as shown in Table 1. Additionally, we vary the data cache parameters as shown in Table 2 and assume a perfect instruction cache.

To measure the impact of out-of-order execution, we define three different issue logics: ALU-in/MEM-in, ALU-out/MEM-in, and ALU-out/MEM-out as shown in Table 4. The cores for these three different configurations differ in the issue logic otherwise are identical. In the ALU-in/MEM-in configuration, the core issues instructions only when the instruction reaches the head of the reorder buffer (ROB). By definition of a ROB this enforces in-order execution. The configuration does allow for speculative execution but by definition mandates that both ALU and memory operations be issued in strict program order. The ALU-out/MEM-in configuration allows the issue of ALU operations out-of-order but mandates issuing of memory instructions from the load and store queues in strict program order. Finally, the ALU-out/MEM-out configuration allows the issue of both ALU and memory operations out-of-order (and represents the working of actual hardware).

We use a mixture of SPECINT & SPECFP 2000 [12], and Olden [7] benchmarks as shown in Table 3. Each benchmark was allowed to warm up and perform its initialization routines before statistics and data were gathered. For the Olden benchmarks, data was gathered via entry and exit points embedded into the benchmarks and implemented in the simulator model. The benchmarks were compiled with the *-O2* optimization flag. The SPEC benchmarks were acquired from the SimpleScalar developers [25] and were warmed up by fast-forwarding the recommended number of instructions [21]. Data was gathered over the next half billion instructions. The SPEC benchmarks operate on their reference input sets. The execute commands for the Olden benchmarks are the following: bisort 250000; em3d 10000 5 100 1 5; health 5 500 5; mst 1024; perimeter 11; treeadd 20.

4 IMPACT OF INCREASED OUT-OF-ORDER AGGRESSIVENESS

4.1 Replay Traps

Figure 3 shows the number of instructions executed between traps (trap rate), and trap overhead, i.e. the total amount of execution time wasted due to traps, for the 16KB and 64KB cache configurations averaged across all 15 of our benchmarks. The per benchmark data is provided for reference at the end of this proposal. The graphs first of all show that even though the ALU-in/MEM-in configuration issues memory instructions in program order, the processor can still suffer from replay traps. This is because some replay traps such as the wrong-size and load-miss load replay trap can occur even though memory instructions are issued in program order. The figure also illustrates that as the CPU gets more aggressive (increasing issue widths and reorder buffer sizes), it exposes traps as an important source of overhead. This is primarily due to

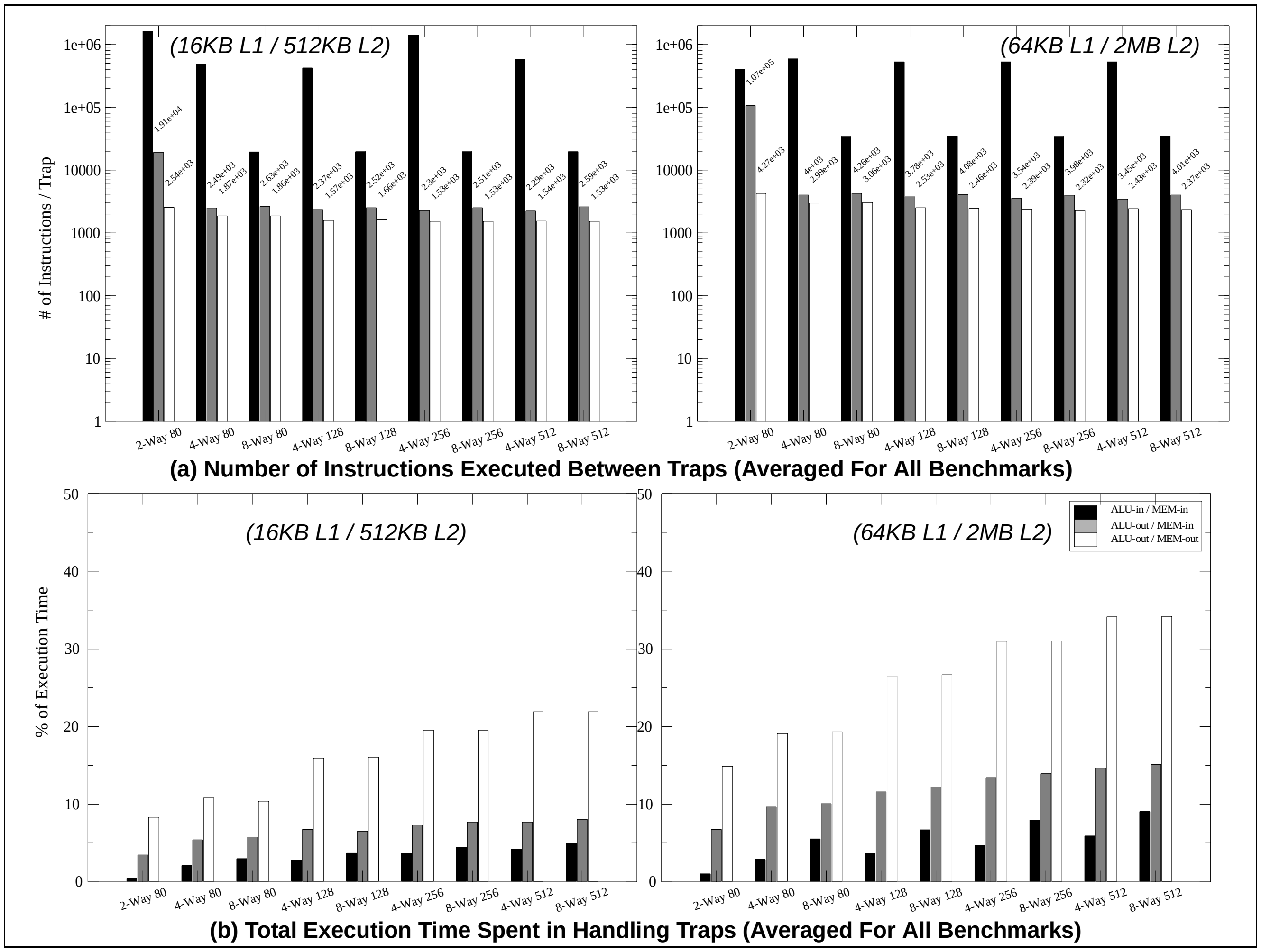


Figure 3: Reorder Traps. (a) Trap Rate— Average Number of Instructions Executed Between Traps (b) Trap Overhead—Total Amount of Execution Lost Due to Traps Trends show that increase in out-of-order aggressiveness by increasing issue widths and reorder buffer sizes increases the trap rate and trap overhead. For an ALU-out/MEM-out core, the figure illustrates that trap overhead and trap rate can be reduced by more than 50% if the core is forced to issue memory instructions in order.

the current mechanisms of handling traps— flushing the pipeline and restarting from the faulting memory instruction. Larger issue widths and reorder buffer sizes allow for a processor to exploit ILP by executing instructions deep into an application's instruction stream; the overheads of flushing and re-fetching an entire window of instructions can become expensive due to the amount of work that needs to be redone. For example, if a trap occurs on a system with a 512-entry reorder buffer, and if at the time of the trap the reorder buffer is full, then it takes a minimum of 64 cycles on an 8-way and 128 cycles on a 4-way processor to restore the state of the reorder buffer to what it was before the trap. Furthermore, this latency can be even higher due to the functional unit and cache access latency. This implies that it is imperative that the frequency of traps be low on systems with larger instruction windows.

Contrary to the desire for less frequent traps, by moving from an in-order, ALU-in/MEM-in, core to a more aggressive out-of-order core, ALU-out/MEM-out (white bars), we note a factor of 8-9 increase in trap rate, causing an application to waste on average 15-30% of it's execution time redoing work already done before. We observe that by restricting the out-of-order core to issue memory instructions in-order and executing ALU instructions out-of-order (ALU-out/MEM-in), the overhead of redoing work already done before can be reduced by more than 50%. However, this comes at the penalty of not exploiting ILP among memory instructions. This suggests the need for modern out-of-order processors to statically or dynamically throttle the degree by which they issue memory instructions out-of-order rather than issuing memory instructions all out-of-order or all in-order. If during a certain window of execution the processor notes frequent reorder traps, it should have a mechanism to ease back and restrict the reordering of memory instructions completely or partially.

4.2 Cache Performance

To measure cache performance of an application we compute the non-speculative cache miss rate, i.e. the number of non-speculative cache misses divided by the total number of memory instructions committed. The number of non-speculative cache misses is tracked by adding two bits in the ROB entry that denote whether the memory instruction hits or misses in the L1 and L2 caches respectively. When a memory instruction accesses the data cache(s) these two bits are updated. At commit time a counter is incremented to keep track of non-speculative L1 and L2 cache misses. The reason why we choose non-speculative misses rather than both speculative and non-speculative misses is because it better characterizes the real stalls an application faces for data required from memory.

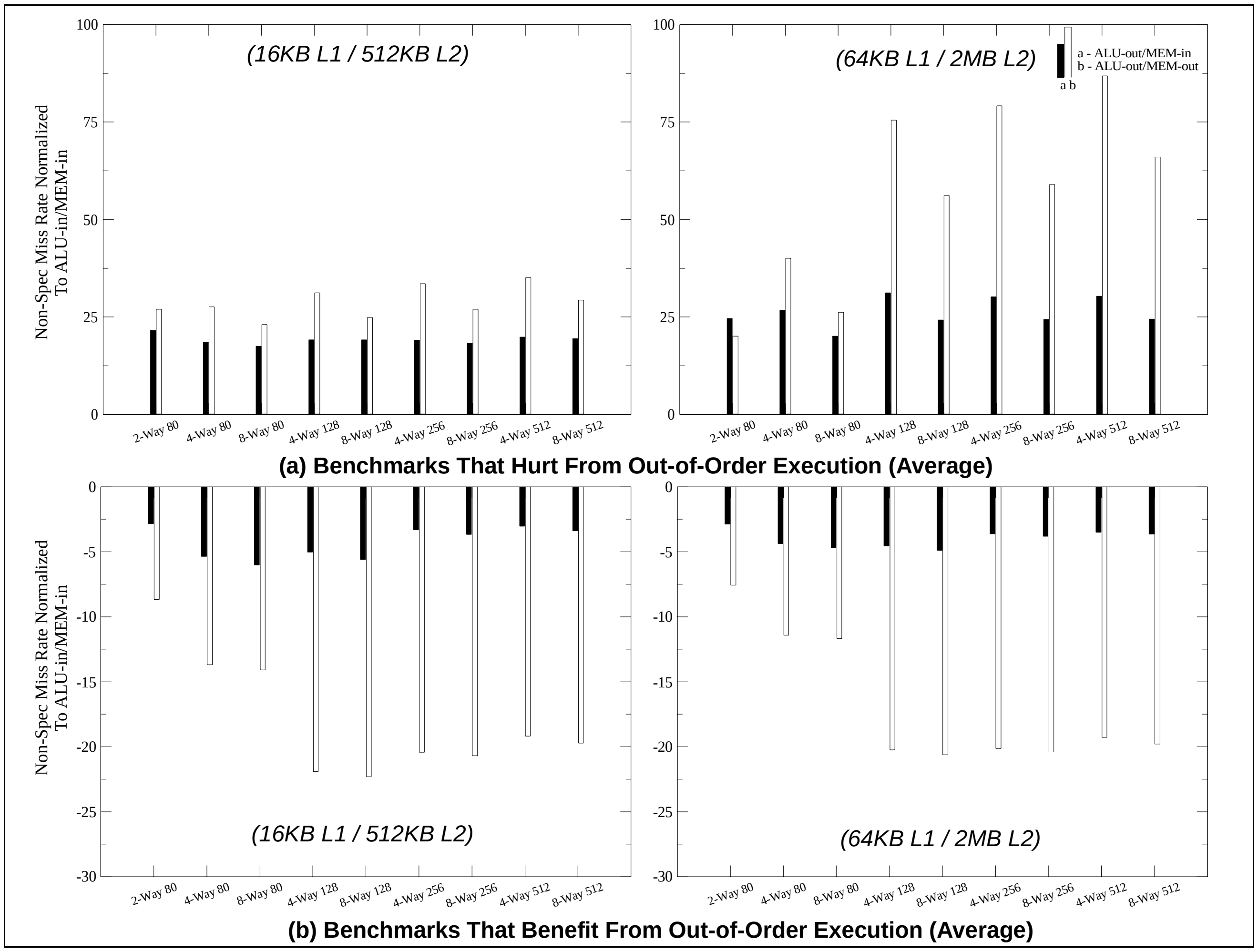


Figure 4: Non-Speculative L1 Cache Miss Rates. The figures shows the non-speculative L1 cache miss rates for the ALU-out/MEM-in and ALU-out/MEM-out configurations normalized to the ALU-in/MEM-in configuration. The graphs show that out-of-order execution of both memory and ALU operations can benefit an application's cache performance by 7-22%. On the other hand, it can hurt cache performance by up to 80%. We observe that restricting the memory operations to be issued inorder eliminates more than 50% of the non-speculative misses.

Figure 4 illustrates the 16KB and 64KB L1-cache non-speculative miss rate (the number of non-speculative memory reference misses divided by the total number of memory references committed) normalized to the ALU-in/MEM-in configuration. The normalized graphs are separated into those applications whose cache miss rate benefitted from out-of-order execution and those whose cache miss rate worsened from out-of-order execution. The graphs display the different out-of-order parameters on the x-axis and the percent change in miss-rate with respect to the ALU-in/MEM-in configuration on the y-axis. The percent change for the ALU-out/MEM-in configuration is represented by black bars and the ALU-out/

MEM-out by white bars. These graphs report the average increase/decrease in non-speculative miss rate across all benchmarks differing by more than +/- 5%. The per-benchmark behavior is discussed in the analysis and is provided as reference at the end of this proposal.

Clearly, depending on an application's memory access pattern, some applications benefit from out-of-order execution and some don't. We find that applications such as bisort, em3d, twolf, and vpr have a memory access pattern that benefits from out-of-order execution; we also observe that these benchmarks tend to perform better with a less strict instruction scheduler, i.e. one that issues both ALU and memory operations out-of-order.

On the other hand, we find applications such as art, gcc, mcf, mst, swim, and treeadd hurting from the out-of-order execution of memory instructions by on average up to 80% increase in miss rate, and some individual benchmarks showing absolute miss rate increases up to 200%. Since larger data caches in general have a higher hit-rate, they allow a processor to speculate much better than a smaller data cache. Executing memory instructions down a speculative path has a higher probability of evicting data that is required or would later be reused by non-speculative memory instructions if the cache is small. We also observe from Figure 4 that for those benchmarks hurting from out-of-order execution, keeping the issue widths constant and increasing the reorder buffer sizes tends to have little effect (2-3% reduction in performance) in the 16KB L1 cache, but by as much as 30% in the 64KB L1 cache for 4-way and 8-way issue-widths respectively. For such benchmarks we also observe that restricting the memory operations to be issued in-order reduces the non-speculative miss rate by 50-90%. This again emphasizes the negative impact of increased out-of-order aggressiveness on cache performance and requires for a processor to be able to dynamically scale the degree by which it issues memory instructions out-of-order.

Similarly, in Figure 5 we plot the normalized non-speculative 512KB and 2MB L2 cache miss rates for benchmarks differing from the ALU-in/MEM-in configuration by more than +/-5%. Simulations revealed

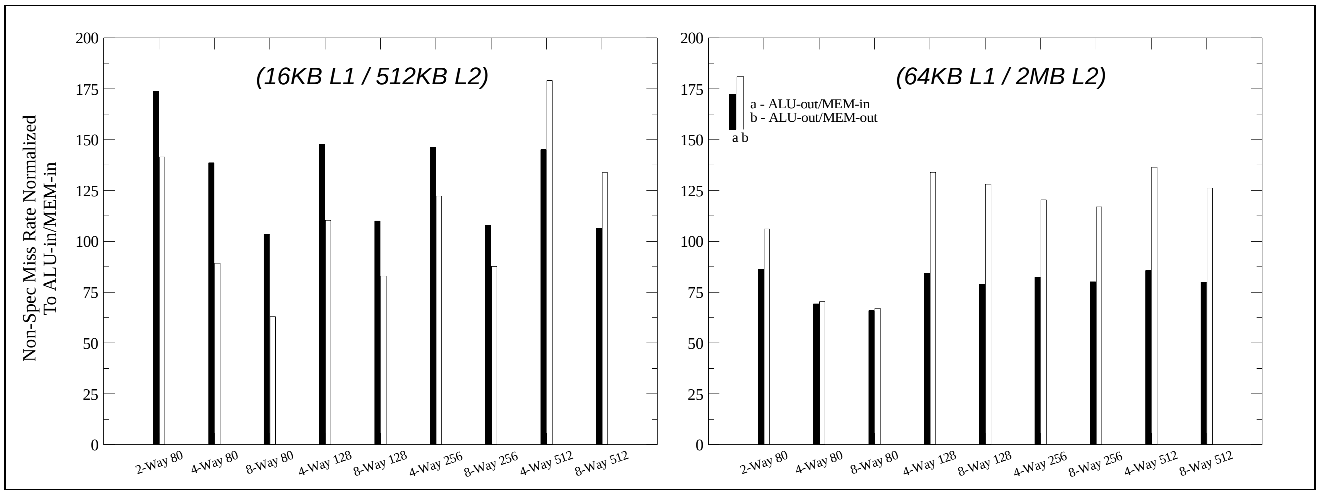


Figure 5: Non-Speculative L2 Cache Miss Rates. The figures shows the non-speculative L2 cache miss rates for the ALU-out/MEM-in and ALU-out/MEM-out configurations normalized to the ALU-in/MEM-in configuration. The graphs show that out-of-order execution of both memory and ALU operations can hurt cache performance by up to 175%.

only one benchmark, em3d, that benefited from out-of-order execution by up to 30%. On the other hand, the average miss rates across 8 benchmarks reveal an increase in non-speculative miss rate by 50-175% in the 512KB cache and 60-120% in the 2MB L2 caches. On a per-benchmark basis we observe degradation in non-speculative miss-rate of 5-20% for parser, vortex and perimeter, by as much as 90% for health and mcf, and finally by 250, 200, and 600% for the benchmarks swim, gcc, and art respectively. In the case of the 2MB and 512KB L2 caches, we observe that restricting the processor to issue memory instructions in-order reduces the miss rate by 50% or more. We also observe in the 512 KB L2 cache, except for the 512-entry ROB sizes, the in-order issue of memory operations increases the miss rate by up to 20%. Since the difference between ALU-in/MEM-in and ALU-out/MEM-in is a higher degree of speculation, this behavior implies that speculative memory accesses tend to evict data required by non-speculative memory accesses.

Our different choice of benchmarks show a varying behavior as a result of out-of-order execution of memory instructions. Depending on the memory access pattern of an application, out-of-order execution can either benefit, hurt, or bring no change to cache performance. Of the 15 benchmarks we observe the increase in non-speculative cache misses (on average as much as 190%) in both the L1 and L2 cache is a factor 9 larger than the benefits (up to 20%) received by some benchmarks. The data reveals two important findings. First, the out-of-order issue of memory instructions tends to increase non-speculative cache miss rates. Second, even if the memory instructions are issued in order, speculative memory accesses tend to increase non-speculative cache miss rates as well. We also observe that non-speculative cache miss rates can be decreased by more than 50% if the core is restricted to issue only the memory operations in-order. These findings again reinforce the need for a mechanism in out-of-order processors to dynamically reduce the degree of speculative issue of memory operations when faced with frequent reorder traps as well as non-speculative cache misses.

4.3 Performance (CPI)

Our simulations revealed that increasing the aggressiveness of the out-of-order core in general increased the non-speculative cache misses and the number of replay traps. We know that such trends in normal cases significantly hurt performance. The question however is, does the increase in out-of-order aggressiveness overcome these hurdles to provide net performance improvements? Figure 6 illustrates the performance graphs for the ALU-in/MEM-in, ALU-out/MEM-in, and ALU-out/MEM-out configurations for the two different cache configurations. The graphs show benchmarks on the x-axis and cycles per instruction (CPI) on the y-axis. The CPI is distributed into three parts: cycles where memory instructions couldn’t retire due to memory latency (black bars), cycles where ALU and memory instructions couldn’t retire because they hadn’t started or finished execution (grey bars), and overhead cycles. (white bars).

One would expect that with increased aggressiveness comes decreased CPI, however excluding the 2-way issue system, the graphs show performance to be relatively flat. We observe that moving from an ALU-in/MEM-in core to an ALU-out/MEM-in core improves performance by 33% or more. This performance improvement is attained by overlapping useful work while waiting for memory operations to complete. This is evident by the ALU wait portion of CPI reducing by 60% or more. Comparing the ALU-out/MEM-in

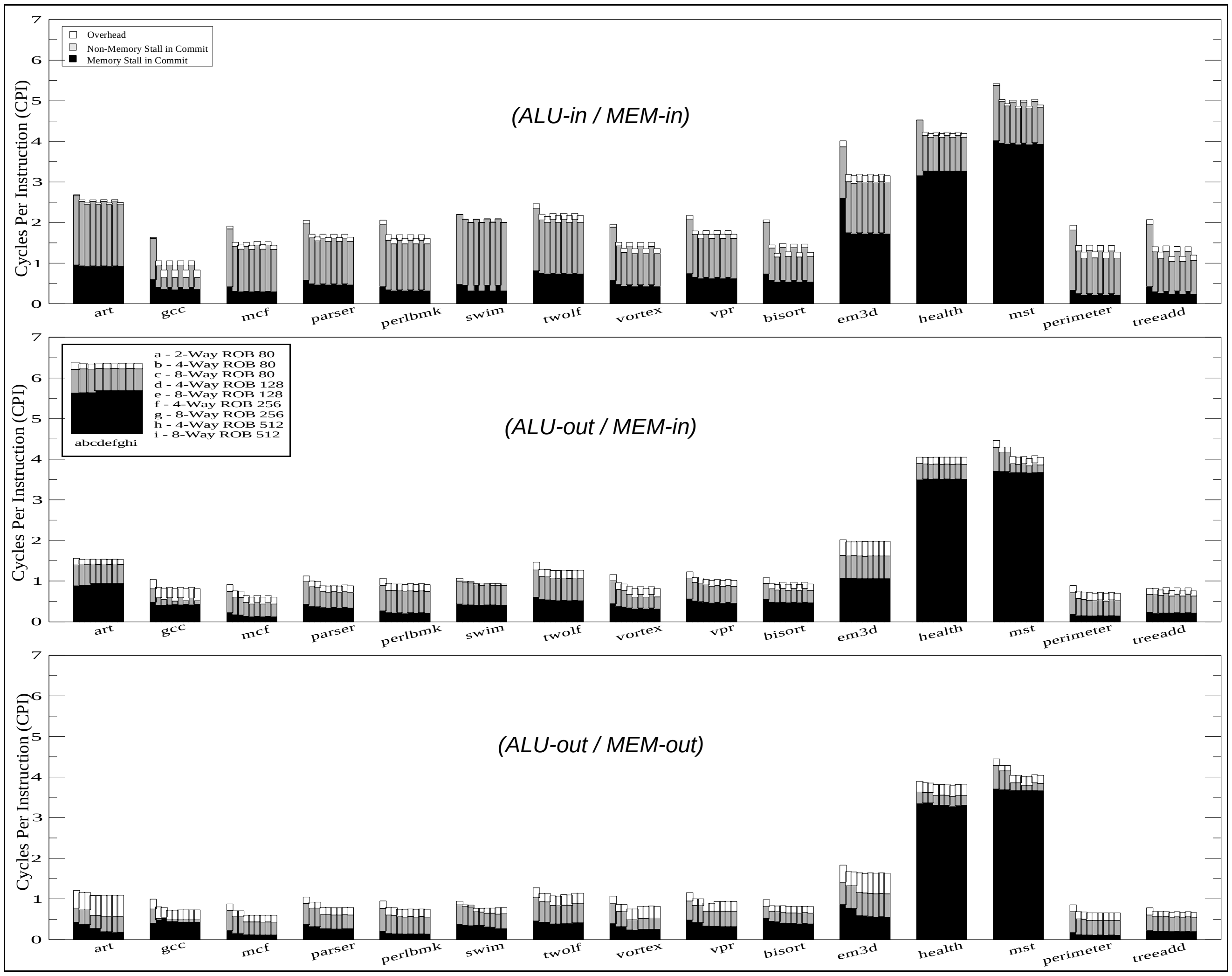


Figure 6: 64KB/2MB L1/L2 Performance - CPI. The figure illustrates the CPI for the ALU-in/MEM-in, ALU-out/MEM-in, and ALU-out/MEM-out configuration. The figure shows that increasing the aggressiveness of the out-of-order core provides minimal performance gain due to increase in trap overhead. The figure shows that much of the wasted work in the ALU-out/MEM-out configuration can be eliminated by restricting the memory instructions to be issued in-order (ALU-out/MEM-in).

with the ALU-out/MEM-out core, a cursory glance of the performance graphs show no remarkable speedups. The performance of the these two systems are within 2-10% (with the exception of art for the 64KB L1 cache). For the ALU-out/MEM-out configuration, we observe that issuing memory instructions out-of-order reduced the portion of time waiting for memory instructions to retire (black bars), but at the same time increased the overhead due to traps (white bars). We observe that, in general, whenever the memory latency portion of CPI decreases, it results in the overhead portion of CPI increasing. This reveals that the out-of-order execution of memory instructions and the memory ordering model conflict with each other. Continued increase in aggressive out-of-order techniques expose non-speculative cache misses and reordering traps as a bottleneck to increased performance. We conclude that a mechanism to dynamically throttle the reordering of memory instructions is required to reap the potential benefits of out-of-order execution.

5 MEASURING OUT-OF-ORDER AGGRESSIVENESS

The results in the previous section showed that moving from an ALU-out/MEM-out configuration to ALU-out/MEM-in configuration reduces the trap frequency and cache misses. Since the change of processor configuration involved merely controlling the order in which memory instructions are issued, we now introduce a metric to measure the degree by which an instruction is issued out of program order.

5.1 Disorder—Measuring the Reordering of Memory Instructions

When executing instructions on an in-order processor, every instruction executes in strict program order. With out-of-order execution however, instructions are executed in an order different from fetch/program order. The degree by which an instruction is issued out-of-order we call *disorder*. Disorder can be classified into two types: *absolute* or *relative*. For the purpose of our study, we will only be measuring the disorder of memory instructions. To measure disorder, at the time of instruction decode, we assign each memory instruction a sequential ID, i.e. first memory reference gets sequential ID one, the next gets sequential ID two, and so on. (In the case of pipeline flushes, the sequential ID is restored to the last successfully retired memory instruction ID + 1.) Disorder is computed ONLY after a memory instruction has all of its dependences resolved and is about to be issued to the cache memory system.

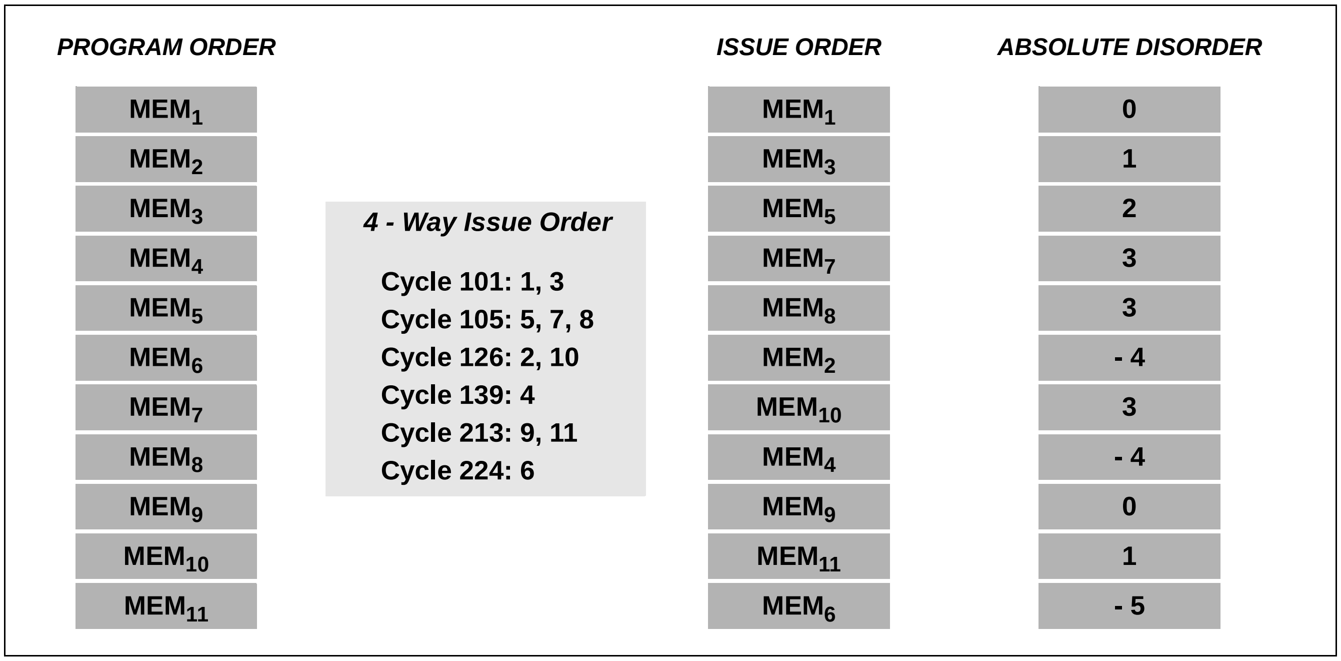


Figure 7: Absolute Disorder. The degree to which a memory instruction is issued out-of-order with respect to actual program order. The disorder is computed by computing the difference between a memory instruction issued and the memory instruction that should have been issued.

5.1.1 Absolute Disorder

Absolute disorder is the degree by which a memory instruction is issued out-of-order with respect to actual program order. Absolute disorder is computed by calculating the difference between the current memory instruction and the memory instruction that should have been issued had the processor executed the program in sequential order. Figure 7 illustrates an example on computing absolute disorder. The figure shows in cycle 101 memory instructions 1 and 3 issued to the cache system. If the system were in-order, then memory instructions 1 and 2 would have been issued instead. Thus, the absolute disorder of memory instruction 1 is 0 (1-1) and the absolute disorder of memory instruction 3 is 1 (3-2). An absolute disorder value of zero indicates that the memory instruction was issued on time, a value less than zero indicates that the memory instruction was delayed, and a disorder value greater than zero indicates that the memory instruction was issued earlier than it should have.

It is important to point out here that out-of-order execution is not the only source of absolute disorder. Modern microprocessors issue load and store instructions to the cache system out of program order. This is because loads access the cache when they reach the memory stage of the pipeline while store instructions access the data cache at commit time. Since load instructions merely read the contents of the data cache, they can access the cache as soon as their effective address is available. A store on the other hand must wait until commit time before writing to the data cache. This is to avoid speculative writes to the data cache. Thus, for a particular program, if a store is immediately followed by a load, the newer load instruction will access the data cache earlier than the store, hence causing disorder.

5.1.2 Relative Disorder

Relative disorder on the other hand is the degree by which a memory instruction is issued out-of-order with respect to other memory instructions issued in the previous and current cycle. Absolute disorder computed disorder from a “program order” perspective, however relative disorder compares how memory instructions issue with respect to each other. Figure 8 provides an example on how to compute relative disorder for memory instruction number 10. Memory instruction 10 is issued by the processor in cycle 126 along with memory instruction number 2. The last time the processor issued memory instructions was in cycle 105, and the memory instructions issued then were 5, 7, and 8. To compute relative disorder, we first compute the disorder between memory instruction 10 and memory instructions 2, 5, 7, and 8 respectively (done by subtracting from 10 each of the other memory instruction IDs). This yields the disorder of memory instruction 10 with every other memory instruction issued in the same and previous cycle. We define relative disorder to be the minimum of all the computed disorders. Thus, the disorder of memory instruction 10 with 2, 5, 7, and 8 is 8, 5, 3, and 2 respectively. Therefore, the relative disorder for memory instruction 10 is the minimum of all computed disorders, i.e. 2.

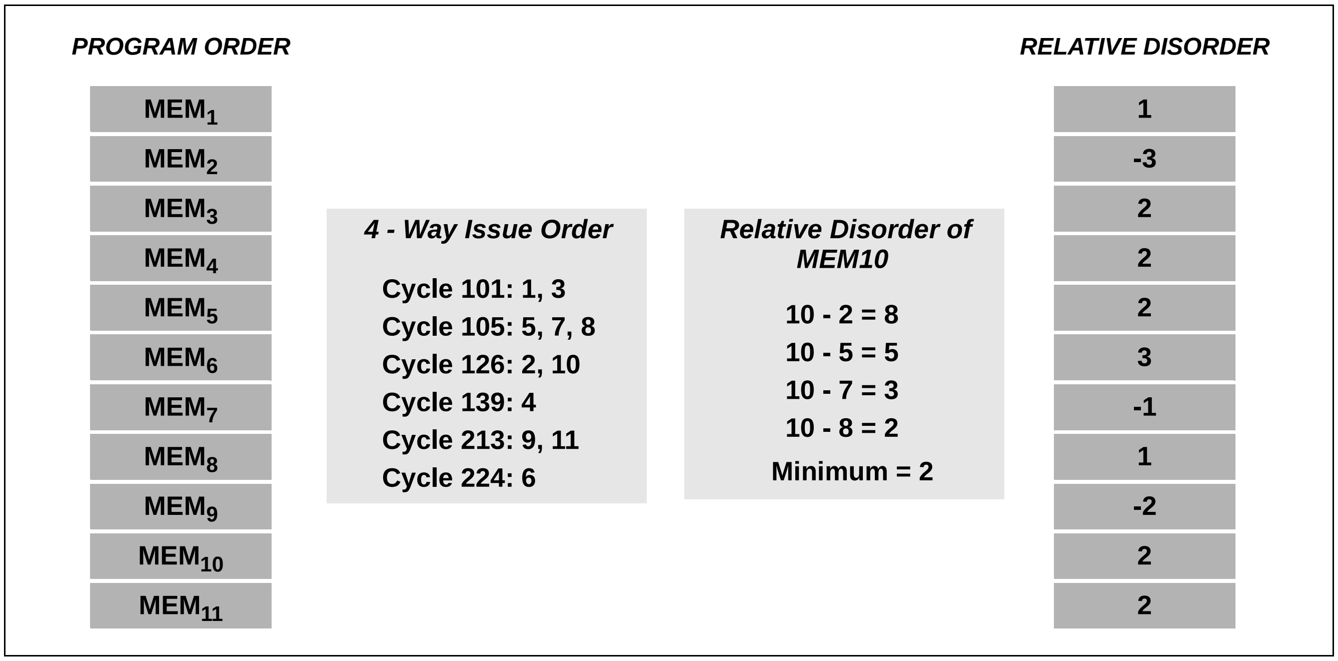


Figure 8: Relative Disorder. The degree by which an instruction is issued out-of-order as compared to other instructions issued in the same and previous cycle. The disorder is computed by extracting the minimum difference between a memory instruction and other memory instructions issued in the same and previous cycle.

Relative disorder measures how a processor issues memory instructions compared to one another other, i.e. if a processor issued memory instruction N in a given cycle, how far apart in the instruction stream are other memory instructions that are issued in the current and following cycle. We chose the minimum value of all computed disorders and not anything else (e.g. standard deviation, or average of all computed disorders) because the value will capture any in-order issue trend. A relative disorder of plus or minus one indicates that there exists one memory instruction in the same or previous cycle that immediately follows or precedes the relevant memory instruction. Relative disorders other than plus or minus one indicate the degree by which memory instructions are separated in the sequential instruction stream.

5.2 Disorder Results

5.2.1 Absolute Disorder

Absolute disorder, as mentioned earlier, is the degree by which a memory instruction is issued out of program order. Figure 9 shows the absolute disorder for the application SWIM on different configurations

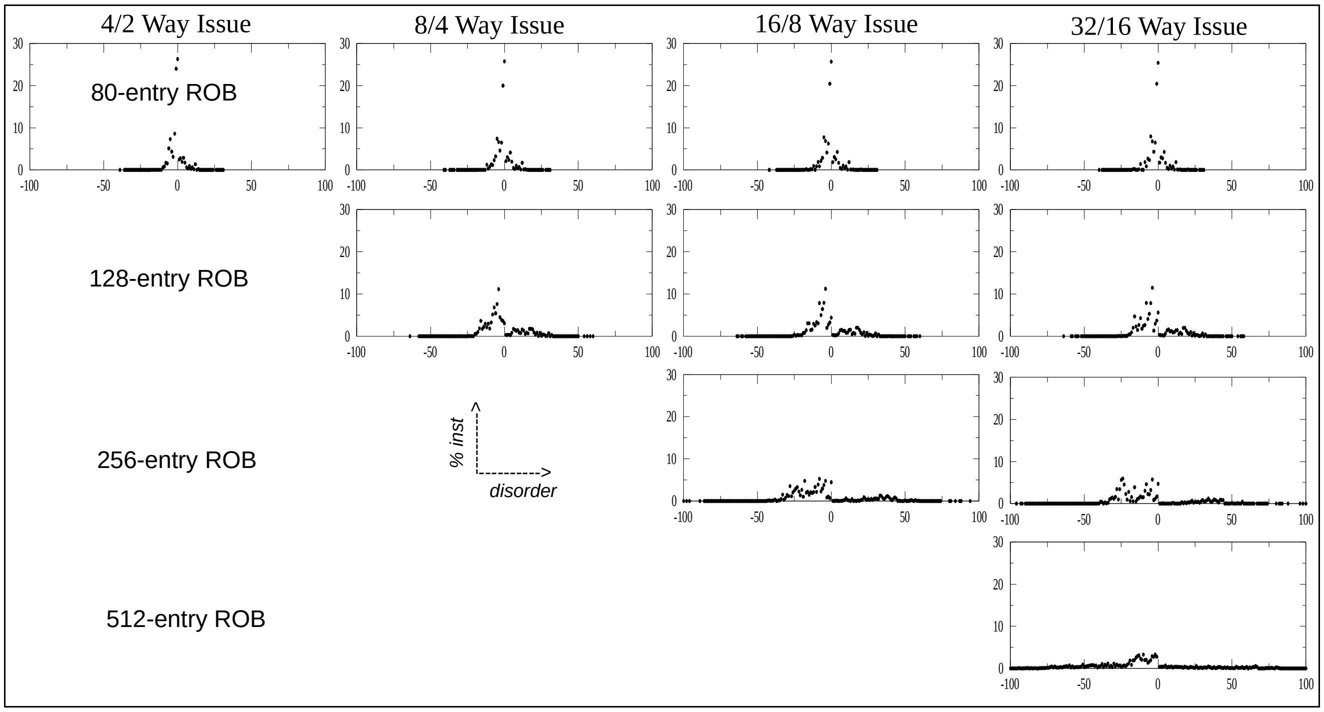


Figure 9: Absolute Disorder (SWIM). The figure shows the absolute disorder for the application SWIM for increasing issue widths (left to right horizontally) and increasing ROB sizes (top to down vertically). The x-axis represents the disorder, and the y-axis represents the percent of instructions with the disorder. One obvious feature from the graph is that memory instructions are significantly reordered, and as out-of-order aggressiveness increases fewer and fewer memory instructions are issued in program order. This re-ordering may have side affects in terms of cache performance as well as memory re-ordering issues.

of the Alpha processor. The graphs are representative of the several other SPEC2000 applications. The x-axis represents the disorder of a memory instruction, and the y-axis represents the percent of instructions that exhibited that disorder.

Figure 9 shows that out-of-order execution creates significant disorder with respect to actual program order. We see that approximately 30% of the instructions are issued in actual program order on an Alpha 21264 with 4/2-way issue and an 80-entry reorder buffer. The remaining instructions either have a negative disorder (issued late due to dependencies or missing in the data cache) or a positive disorder (issued early because older memory instructions couldn’t be issued). It is likely that the wide variation in disorder is mostly due to memory references missing in the data caches, the low disorders primarily due to misses in the L1 cache (L2 hit latency 15 cycles), and the extreme disorders due to misses in the L2 cache, i.e. due to memory latency. We see that increasing aggressiveness of the out-of-order core (increasing issue widths going across and increasing ROB sizes going down) allows for increased speculation; thus we see the number of memory instructions issued on time (absolute disorder of zero) decreasing. For a 32/16 way processor with a 512-entry ROB, the number of memory instructions issued on time is about 3-4%. The graph also shows that increasing window sizes correlate with increasing absolute disorder (10-25%), where as increasing issue widths change absolute disorder by only a few percent (2-4%). This is because with increasing issue widths and constant reorder buffer sizes, the window of instructions available to the processor stays the same. If the processor isn't able to issue from the window, then irrespective of the issue width, instructions won't be issued thus the reorder buffer eventually fills up. Increasing the window size provides the processor for a wider choice of instructions to issue from, thus increasing the absolute disorder.

5.2.2 Relative Disorder

We now analyze an application's relative disorder, i.e. the disorder with respect to other memory operations issued in the same and previous cycle. In the earlier section, we observed high absolute disorder— less than one third of all memory instructions issued are actually issued on time. Figure 10 shows the relative disorder for SWIM and is representative of all other benchmarks. We observe that memory instructions issued are usually in close proximity to each other, i.e. memory instructions are often scheduled from the same basic block or section of code; thus yielding an average relative disorder that is low, i.e. less than or equal to |5|. This implies that when executing instructions out-of-order, instructions that are close to each other are more likely to be scheduled in close proximity to each other i.e. there seems to exist a "spatial locality" with respect to issuing memory instructions.

From the earlier section, absolute disorder was used to measure the degree by which a memory instruction is issued out of program order. However, it can also be used to measure the degree of speculation. The higher the magnitude of the absolute disorder, the more the processor is speculatively

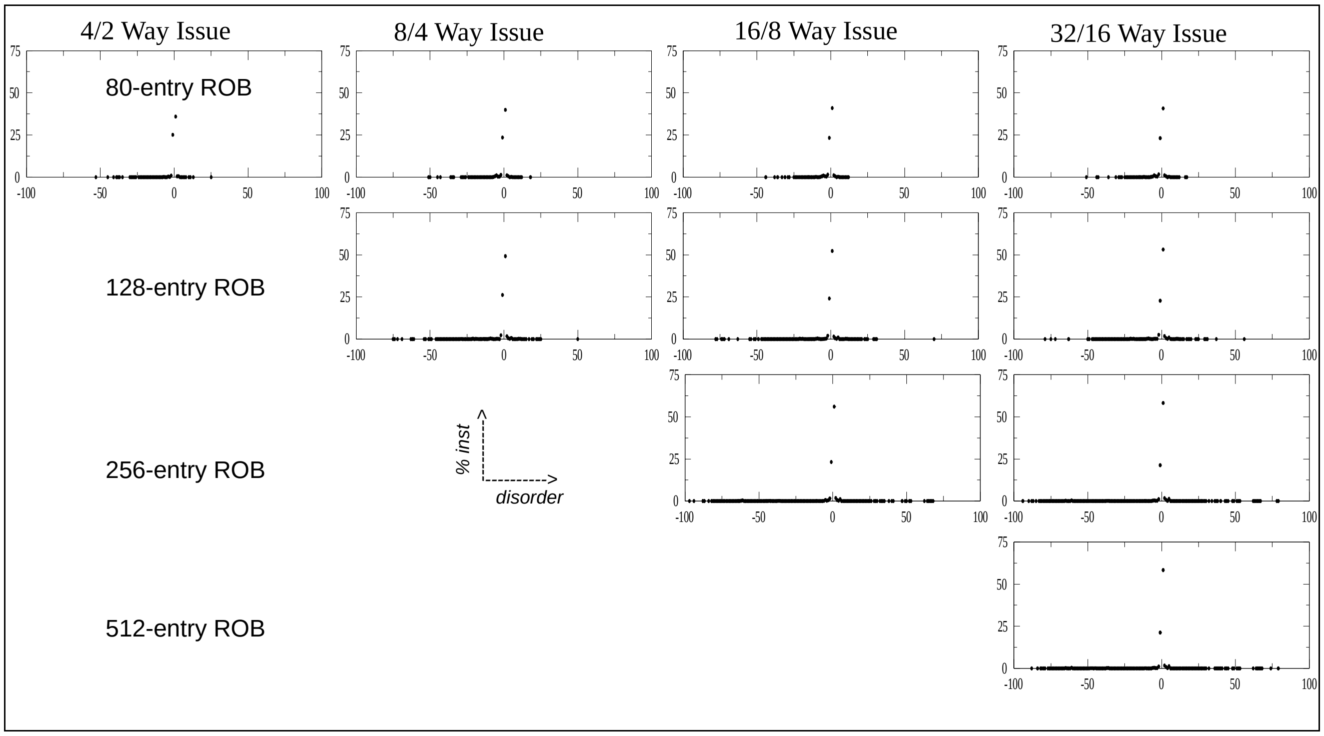


Figure 10: Relative Disorder (SWIM). The figure shows the relative disorder for the application SWIM for increasing issue widths (left to right horizontally) and increasing ROB sizes (top to down vertically). We see that the bulk of the memory instructions issued to the memory system have a relative disorder of zero signifying that instructions issued are usually in close proximity to each other, i.e. they are from within the same basic block or section of the code.

executing memory instructions. With increasing out-of-order aggressiveness, a processor speculatively executes memory operations distant from older memory operations that missed in the L1 data cache. While executing instructions in the distant, it is more likely that the processor would be able to execute other memory operations in the same locality, thus contributing to lower relative disorder.

6 PROPOSED WORK

We now look at possible avenues to address the problems associated with increasing out-of-order aggressiveness. Results from section 4.1 and 4.2 revealed a correlation between the reordering of memory instructions, replay traps and cache performance. With the absolute disorder metric in mind, we propose to throttle the degree by which memory instructions are issued out-of-order and reduce absolute disorder via a *windowing* mechanism. With regards to cache performance alone, we propose to investigate different caching strategies and determine any correlation between cache configuration, increased out-of-order aggressiveness, and cache performance.

6.1 Windowing Memory Instructions

We observe that simply restricting memory instructions to be issued in program order reduces both the negative effects of out-of-order execution. However, we also observe that issuing memory instructions in program order hurts ILP among memory instructions. Thus, rather than issuing all memory instructions in order, processors could utilize a mechanism to throttle the degree by which they issue memory instructions out-of-order. We propose to restrict the reordering of memory instructions based on a window of instructions by using the network communication concept of windowing [23]. By using a *sliding window protocol*, we can restrict the scheduler to issue only those memory instructions that lie within the current window of memory instructions. The size of the *sliding window* can either be determined statically or dynamically. Such a mechanism can reduce the disorder of memory instructions, hence possibly reduce the negative effects of out-of-order execution of memory instructions.

Windowing is a commonly used technique for implementing flow control while transferring data over networks. With typical network communication, a sender normally sends out data packets and the receiver acknowledges (*acks*) them. The window size determines the maximum number of data packets that can be sent without waiting for an ack. Once an ack is received and it is for the oldest packet in the senders queue, the window is extended by sliding the window down to accommodate sending newer packets.

We attempt to reduce the reordering of memory instructions by utilizing the aforementioned property of windowing. The length of the window determines the amount of memory instructions available to the select and issue logic. The window essentially acts as a virtual load/store queue (VLSQ). The virtual load/store queue is maintained using two pointers in the existing load/store queue: *virtual head* and *virtual tail*; *virtual head* always points to the oldest non-issued memory instruction and *virtual tail* is a pointer to the end of the

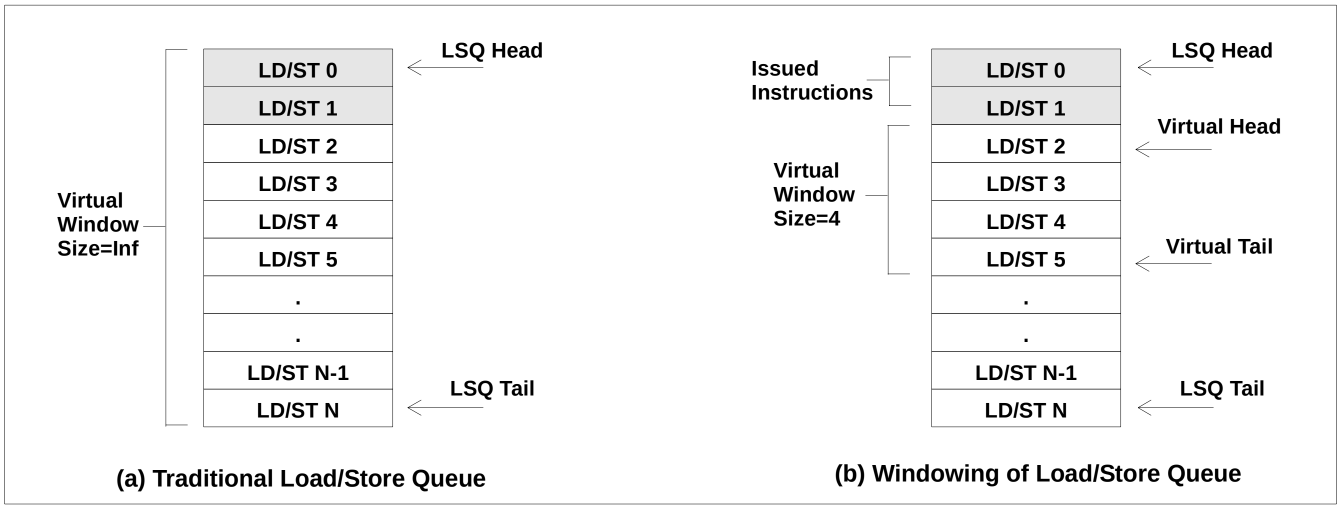
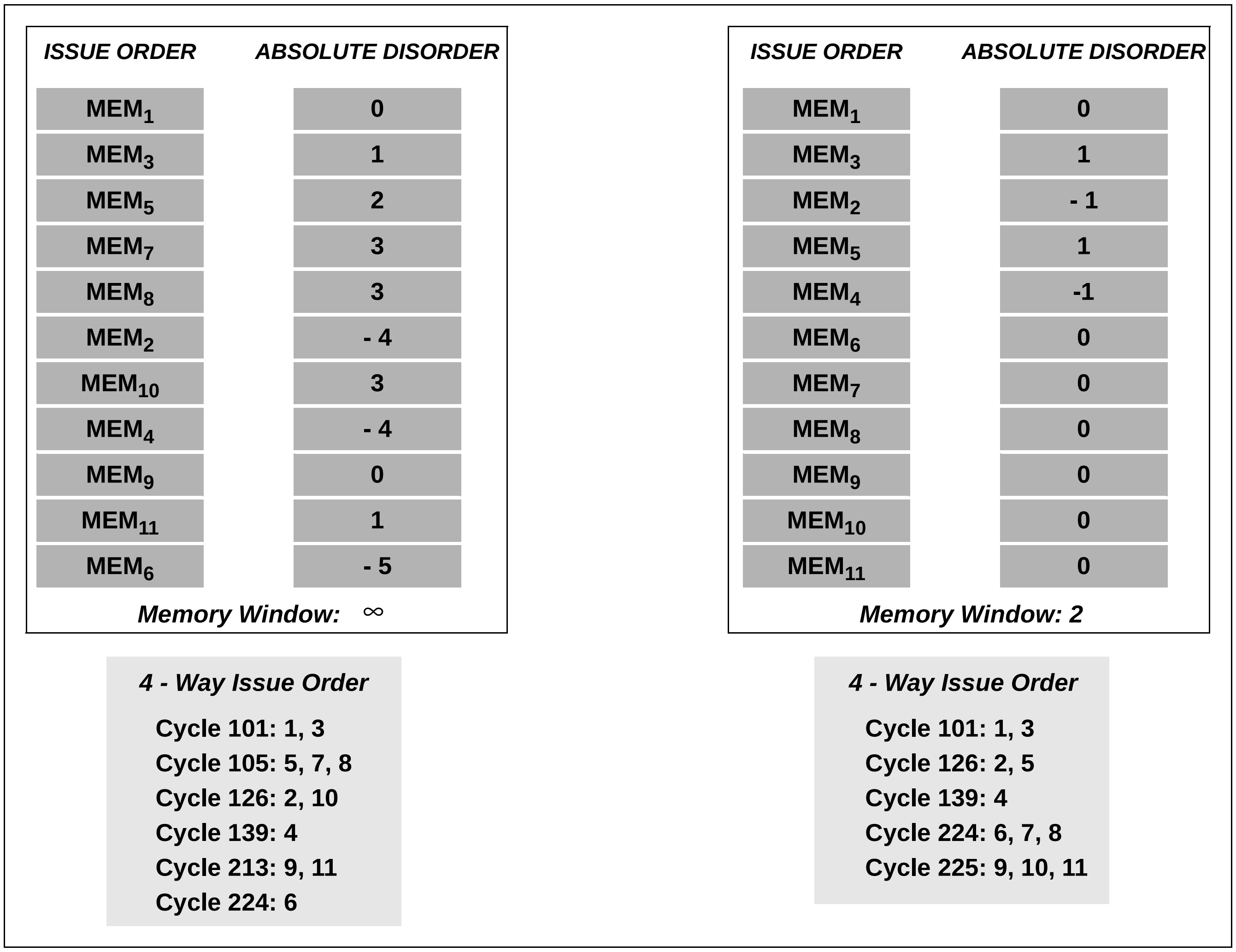


Figure 11: Windowing Memory Instructions: A mechanism to reduce the reordering of memory instructions. (a) The figure illustrates the traditional implementation of a load-store queue. (b) Using windowing, only memory instructions that lie within the virtual head and virtual tail pointers are issued to execute. Other memory instructions must wait till these lie within the virtual window before they can be issued to the memory system.

virtual load/store queue. The difference between *virtual head* and *virtual tail* is W_{size}, the size of the virtual load/store queue. During instruction scheduling, the select and issue logic must ensure that only memory instructions residing within the virtual load/store queue are selected to be issued. The *virtual head* and *virtual tail* pointer is changed when the memory instruction at the *virtual head* is issued.

Figure 11(a) illustrates a traditional load/store queue with head pointer at index 0 and the tail pointer at index N. The shaded load/store queue entries indicate instructions that have already been issued. With a traditional load/store queue, the issue logic can schedule any memory instruction (between 2 and N) whose operands are ready. Figure 11(b) illustrates an example of windowing with $W_{size} = 4$. The *virtual head* pointer points to the first non-issued memory instruction, i.e. memory instruction 2. Using windowing, the issue logic can schedule only memory instructions 2, 3, 4, or 5. If none of the instructions in the virtual load/store queue have their operands ready, the issue logic will stall the issue of memory operations. Only when memory instruction two is issued does the window slide down until it reaches a non-issued memory instruction.



ISSUE ORDER	ABSOLUTE DISORDER
MEM_1	0
MEM_3	1
MEM_5	2
MEM_7	3
MEM_8	3
MEM_2	- 4
MEM_{10}	3
MEM_4	- 4
MEM_9	0
MEM_{11}	1
MEM_6	- 5

Memory Window: ∞

ISSUE ORDER	ABSOLUTE DISORDER
MEM_1	0
MEM_3	1
MEM_2	- 1
MEM_5	1
MEM_4	-1
MEM_6	0
MEM_7	0
MEM_8	0
MEM_9	0
MEM_{10}	0
MEM_{11}	0

Memory Window: 2

4 - Way Issue Order

Cycle 101: 1, 3
Cycle 105: 5, 7, 8
Cycle 126: 2, 10
Cycle 139: 4
Cycle 213: 9, 11
Cycle 224: 6

4 - Way Issue Order

Cycle 101: 1, 3
Cycle 126: 2, 5
Cycle 139: 4
Cycle 224: 6, 7, 8
Cycle 225: 9, 10, 11

Figure 12: Windowing of Memory Instructions. The figure illustrates the effect of windowing on absolute disorder.

The benefits of windowing would be two fold. First, windowing will reduce the reordering of memory instructions by maintaining a small load/store queue to schedule instructions from without affecting instruction fetch bandwidth and the execution of ALU instructions. By reducing the reordering of memory instructions, i.e. disorder, windowing can reduce the number of replay traps and cache misses. We show in Figure 12 a comparison of the absolute disorder of a processor with an infinite memory window and a memory window size of 2. From the figure, we observe that windowing aids in the reduction of absolute disorder. The second benefit of windowing is that it will reduce the total number of speculative memory instructions issued to execute. The benefits of reducing speculative memory instructions are reduced number of load/store queue searches for memory disambiguation and reduced number of cache accesses. A reduction in the number of speculative memory instructions issued and a reduction in replay traps caused due to the reordering of memory instructions can lead to significant amount of power and energy savings in the data caches and fetch, map, and execution hardware.

However, as mentioned earlier, a downside associated with windowing is a reduction in memory ILP. Applications that are heavily memory dependent can suffer from a degradation in performance due to the late issue of memory instructions to the memory system. Such memory intensive applications may require a larger virtual load/store queue than those applications that are compute intensive. Characterizing application behavior with different window sizes statically can help determine an optimal virtual load/store queue size.

7 CONCLUSIONS

Large instruction windows coupled with out-of-order execution has been the widely proposed technique to tolerate the long latencies associated with data cache misses and cross chip communication. From our preliminary study we've observed two pitfalls of aggressive out-of-order mechanisms. First, increased out-of-order capability conflicts with the memory ordering requirements of a processor resulting in frequent replay traps to maintain correct state. Second, increasing out-of-order capability can destroy an application's cache locality by causing it to suffer from a higher number of cache misses than a lesser aggressive out-of-order mechanism. We observe that both these side effects of out-of-order mechanism can cause significant performance and energy loss and can be attributed to the reordering of memory instructions.

Based on these observation, we believe that future aggressive microprocessors require mechanisms to overcome both pitfalls. With respect to replay trap and cache performance, we propose a windowing mechanism in the load/store queue to reduce the reordering of memory instructions. The window essentially acts as a virtual load/store queue within the traditional load/store queue. Developing the windowing mechanism requires us to investigate static or dynamic mechanisms to determine VLSQ sizes that can aid in the reduction of replay traps and cache misses. With respect to cache performance alone, we propose to investigate different caching strategies to reduce the number of cache misses with increased out-of-order capability. Favorable solutions can aid in reducing the wastage of performance and energy and allow future aggressive out-of-order processors to reap the true benefits of out-of-order execution.

8 BIBLIOGRAPHY

1. Compaq Computer Corporation. “Alpha 21264 Microprocessor Hardware Reference Manual.” June 1999.
2. Compaq Computer Corporation. “Compiler Writer’s Guide for the Alpha 21264” June 1999.
3. Silicon Graphics, Inc. *MIPS R10000 Microprocessor User’s Manual version 2.0*, October 1996.
4. H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpointing Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th International Symposium on Microarchitecture*, December 2003.
5. M. D. Brown, J. Stark, and Y. N. Patt. Select-Free Instruction Scheduling Logic. In *Proc. 34th International Symposium on Microarchitecture*, December 2001.
6. D. Burger, J. Goodman, and A. Kagi. “Memory Bandwidth Limitations of Future Microprocessors.” In *Proc. 23rd International Symposium on Computer Architecture (ISCA'96)*. Philadelphia, PA, May 1996.
7. M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. “Early Experiences with Olden.” In *Proceedings of the 6th International Owrkshop on Languages and Compilers for Parallel Computing*, pages 1-20, August 1993.
8. V. Cuppu and B. Jacob. “A Performance Comparison of contemporary DRAM architectures.” In *Proc. 26th International Symposium on Computer Architecture (ISCA'99)*. Atlanta GA, May 1999.
9. R. Desikan, D. Burger, and S. Keckler. “Sim-alpha: a Validated, Execution-Driven Alpha 21264 Simulator.” Tech Report TR-01-23, University of Texas at Austin.
10. R. Desikan, D. Burger, and S. Keckler. “Measuring experimental error in microprocessor simulation.” In *Proc. 28th International Symposium on Computer Architecture (ISCA'01)*. Goteborg, Sweden, June 2001.
11. M. K. Gowan, L. L. Biro, D. B. Jackson. “Power Considerations in the Design of the Alpha 21264 Microprocessor.” In *Design Automation Conference (DAC’98)*. San Francisco, CA, June 1998.
12. J. L. Henning. “SPEC CPU2000: Measuring CPU Performance in the New Millenium”. IEEE Computer, 33(7):28-35, July 2000.

13. D. Henry, B. Kuszmaul, G. Loh, and R. Sami. “Circuits for wide-window superscalar processors.” In *Proc. 27th Annual International Symposium on Computer Architecture (ISCA’00)*, Vancouver BC, June 2000.

14. D. Kroft. “Lockup-Free Instruction Fetch/Prefetch Cache Organization.” In *Proc. 8th International Symposium on Computer Architecture (ISCA'81)*. Minneapolis MN, May 1981.

15. A. Lebeck, J. Kppanalil, T. Li, J. Patwardhan, and E. Rotenberg. “A Large, Fast Instruction Window for Tolerating Cache Misses.” In in *Proc. 29th Annual International Symposium on Computer Architecture (ISCA’02),* Anchorage, Alaska, May 2002.

16. R. Natarajan, H. Hanson, S.W. Keckler, C.R. Moore, and D. Burger. “Microprocessor Pipeline Energy Analysis.” *IEEE International Symposium on Low Power Electronics and Design (ISLPED),* pp. 282-287, Seoul, Korea, August, 2003.

17. S. Onder and R. Gupta. “Instruction Wake-Up in Wide Issue Superscalars.” In *Proc. ACM/IEEE Conference on Parallel Architectures and Compilation*

18. S. Onder and R. Gupta. “Dynamic Memory Disambiguation in the Presence of Out-of-Order Store Issuing.” In *International Symposium on Microarchitecture*, 1999.

19. I. Park, C. L. Ooi and T. N. Vijaykumar. “Reducing Design Complexity of the Load/Store Queue.” In *Proc. 36th International Symposium on Microarchitecture*, San Diego, CA, December 2003.

20. G. Reinman and B. Calder. “A Comparative Survey of Load Speculation Architecture”. In *Journal of Instruction Level Parallelism 1 (JILP)*. pp. 1-39, 2000.

21. S. Sair and M. Chaney. “Memory Behavior of the SPEC2000 Benchmark Suite.” IBM Research Report RC 21852 (98345), IBM T.J. Watson, October 2000.

22. K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Tradeoffs and Simulation Techniques. *IEEE Transactions on Computers*, 48(11):1260-1281, November 1999.

23. A. S. Tanenbaum. Computer Networks. Third Edition.

24. J.M. Tendler, J.S. Dodson, J.S. Fields, H.Le Jr., and B. Sinharoy. Power4 system microarchitecture. IBM *Journal of Research and Development*, 45(1), October 2002.

25. C. T. Weaver. Pre-compiled SPEC2000 Alpha Binaries. Available: http://www.simplescalar.org

26. A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. “Speculation Techniques for Improving Load Related Instruction Scheduling”. In *Proc. 26th International Symposium on Computer Architecture (ISCA'99)*. Atlanta GA, May 1999

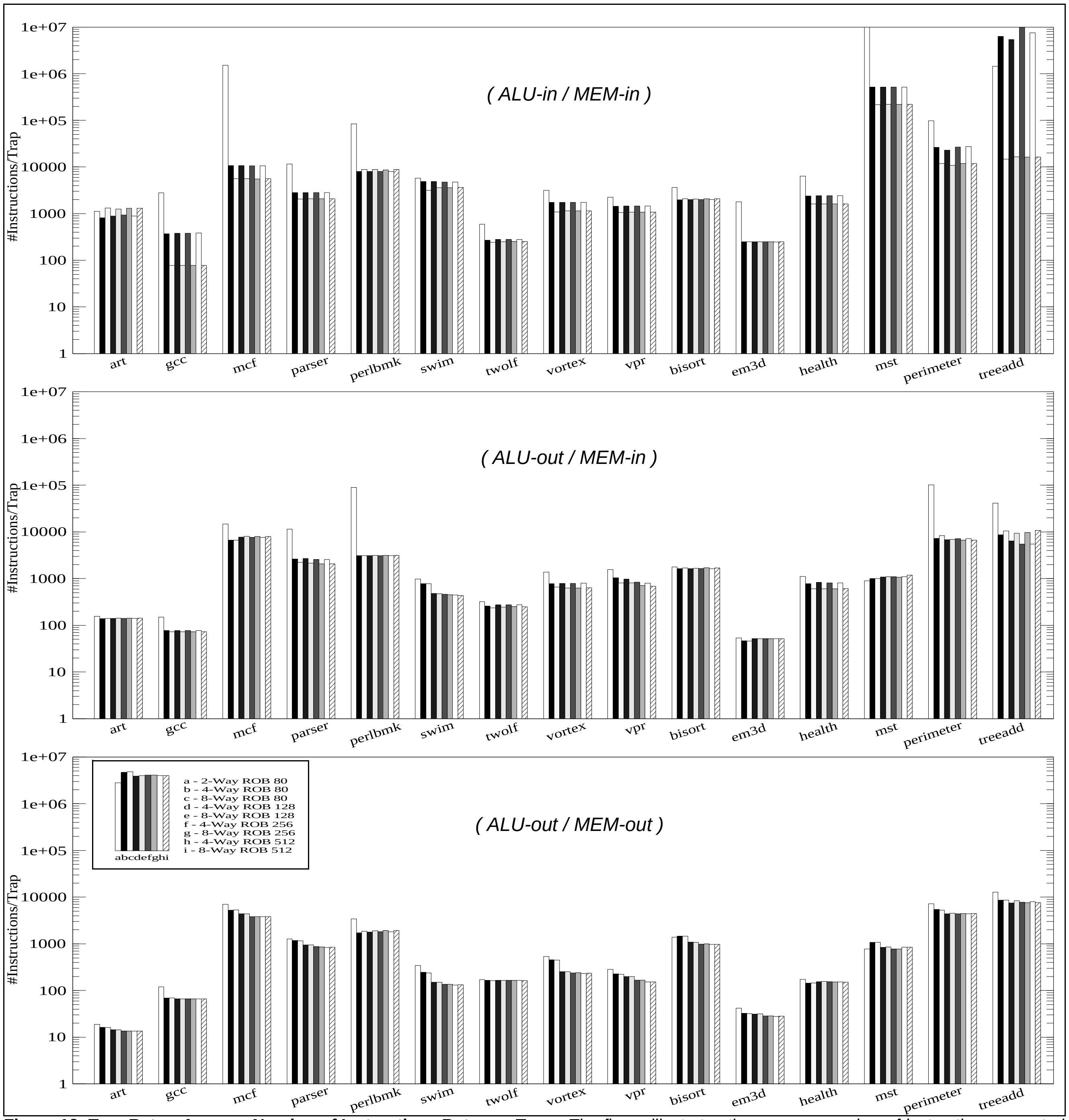


Figure 13: Trap Rate—Average Number of Instructions Between Traps. The figure illustrates the average number of instructions executed between traps. The figures show that trap rate increases by a factor of 8-9 when moving from an inorder core (ALU-in/MEM-in) to out-of-order core (ALU-out/MEM-out). Since trap rate is caused due to the reordering of memory instructions, we see that forcing the inorder issue of memory instructions (ALU-out/MEM-in) increases the average number of instructions executed between traps.

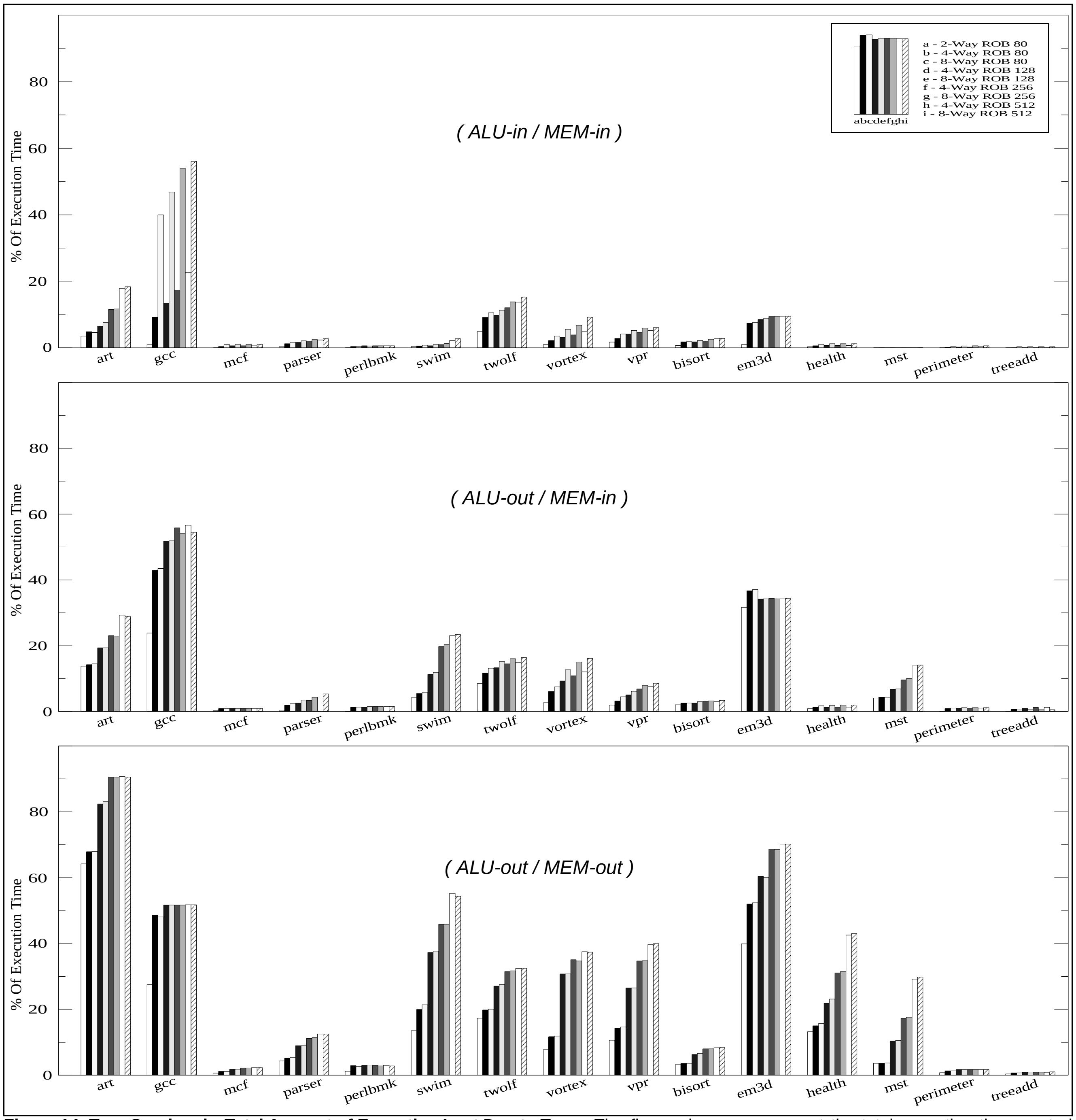


Figure 14: Trap Overhead—Total Amount of Execution Lost Due to Traps. The figure show as a percent the total execution time wasted due to trap handling. Trends show that increase in out-of-order aggressiveness by increasing issue widths and reorder buffer sizes increases the trap overhead. When compared to a purely out-of-order core, the figure illustrates that trap overhead can be reduced by more than 50% if the core is forced to issue memory instructions in order.

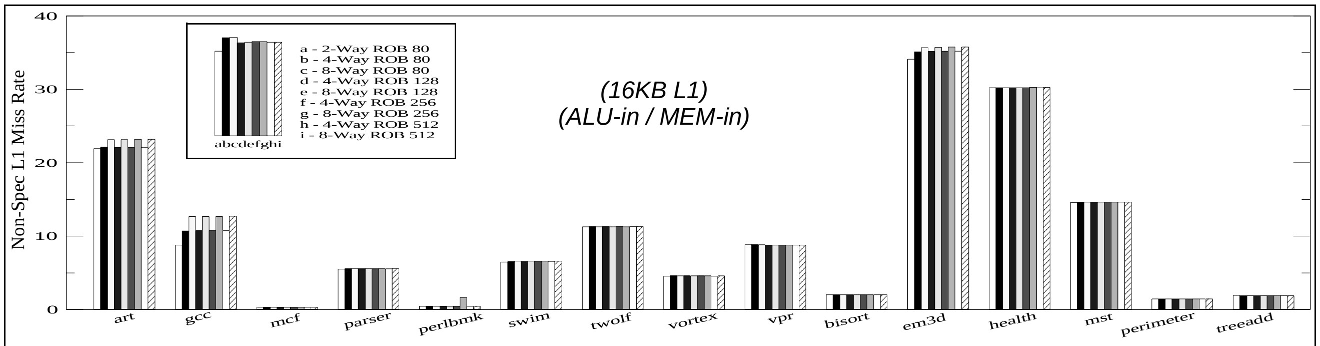
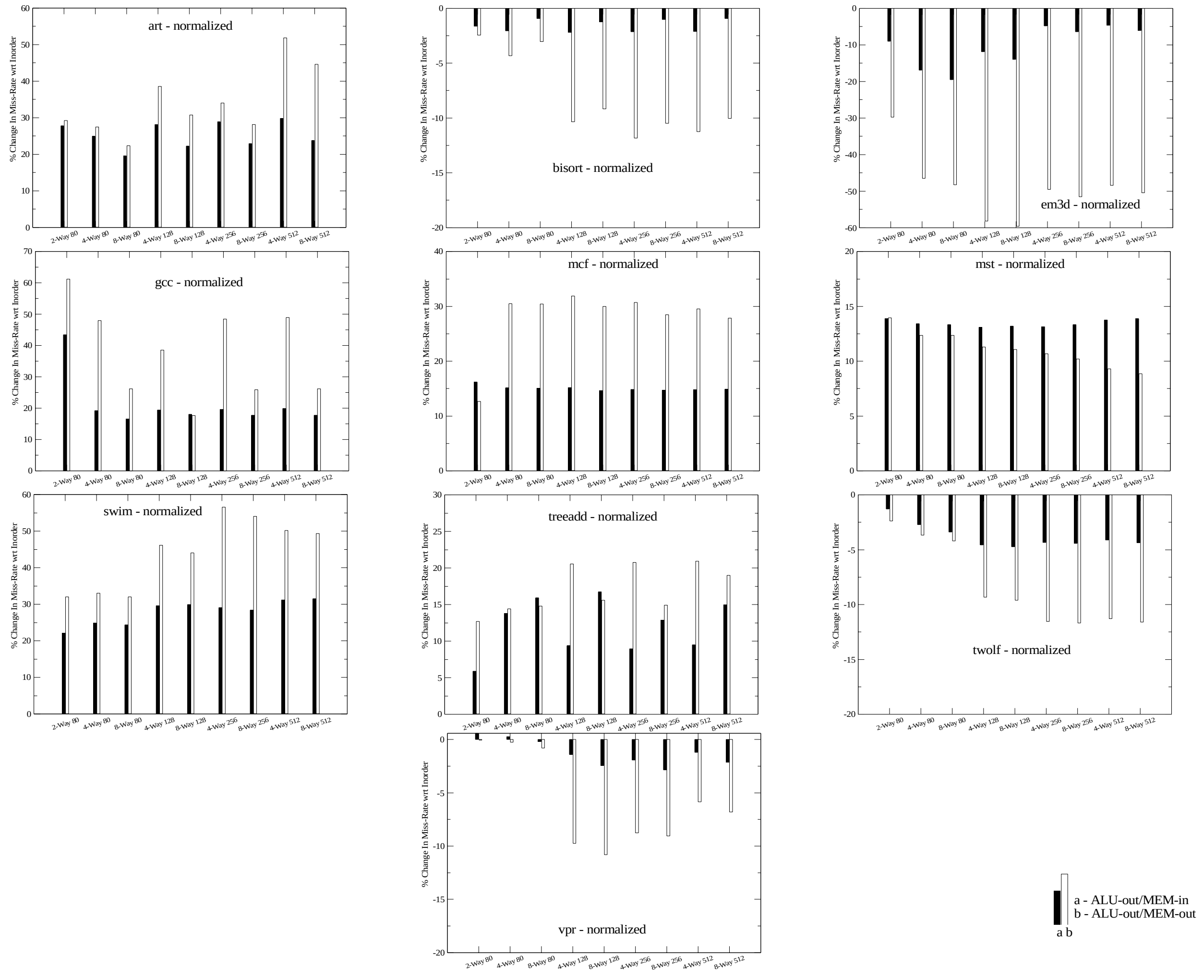


Figure 15: 16KB L1 Non-Speculative Cache Miss Rate. (above) The figure shows the non-speculative L1 cache miss rates as a percent for the ALU-in/MEM-in configuration. (below) Using the inorder core as a base machine, the graphs show the ALU-out/MEM-in and ALU-out/MEM-out non-speculative cache miss rates normalized to the ALU-in/MEM-in configuration. The graphs show that four applications benefit from out-of-order execution by 10-50%, six of them hurt by 20-60%, and the remaining five have little or no difference. In scenarios where out-of-order execution hurts cache performance, the ALU-out/MEM-in configuration in most cases helps reduce the non-speculative cache misses by 50% or more.



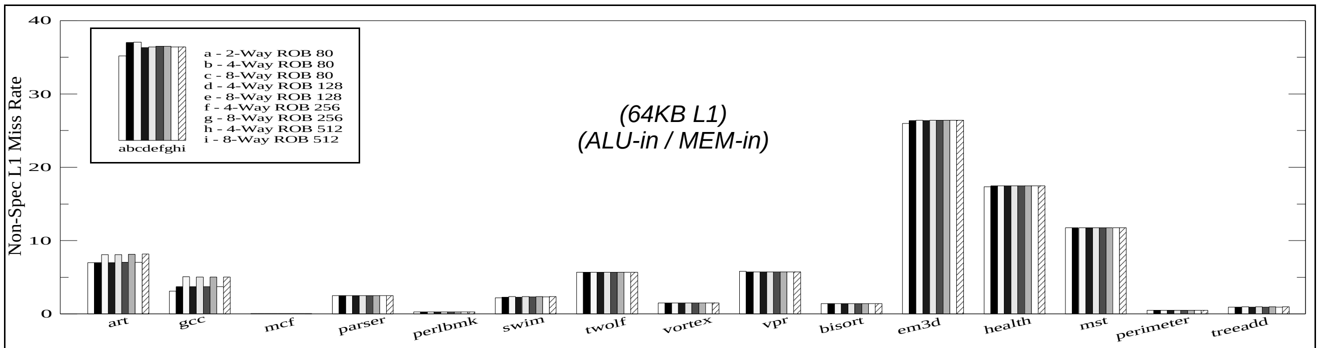
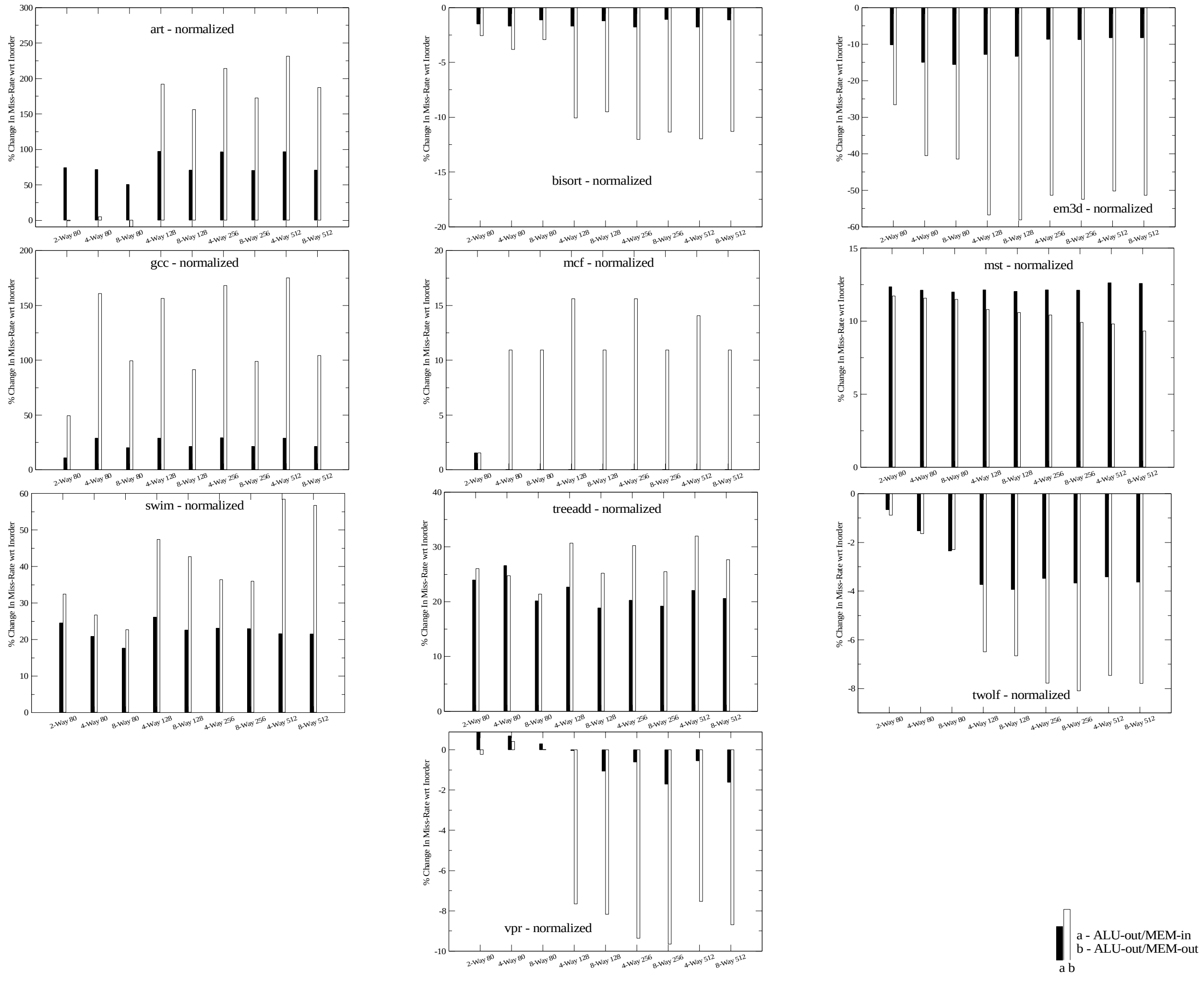


Figure 16: 64KB L1 Non-Speculative Cache Miss Rate. (above) The figure shows the non-speculative L1 cache miss rates as a percent for the ALU-in/MEM-in configuration. (below) Using the inorder core as a base machine, the graphs show the ALU-out/MEM-in and ALU-out/MEM-out non-speculative cache miss rates normalized to the ALU-in/MEM-in configuration. The graphs show that four applications benefit from out-of-order execution by 10-50%, six of them hurt by 20-200%, and the remaining five have little or no difference. In scenarios where out-of-order execution hurts cache performance, the ALU-out/MEM-in configuration in most cases helps reduce the non-speculative cache misses by 50% or more.



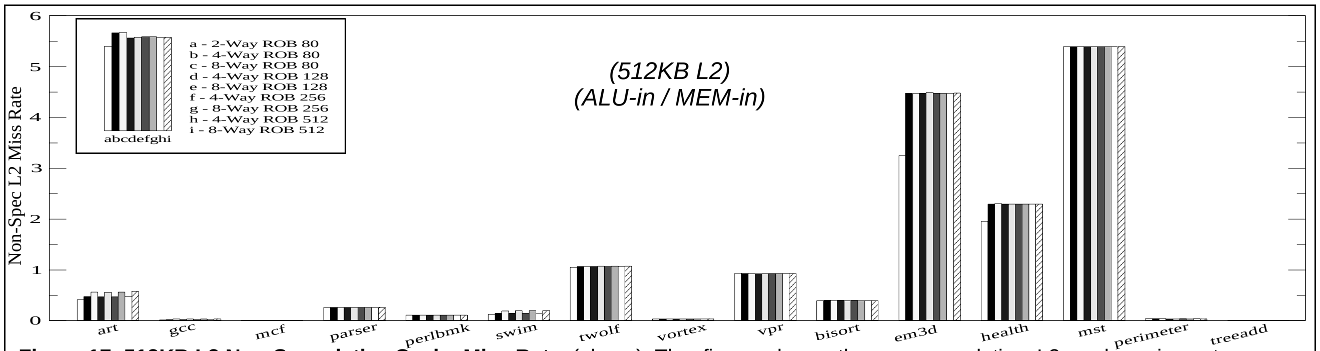
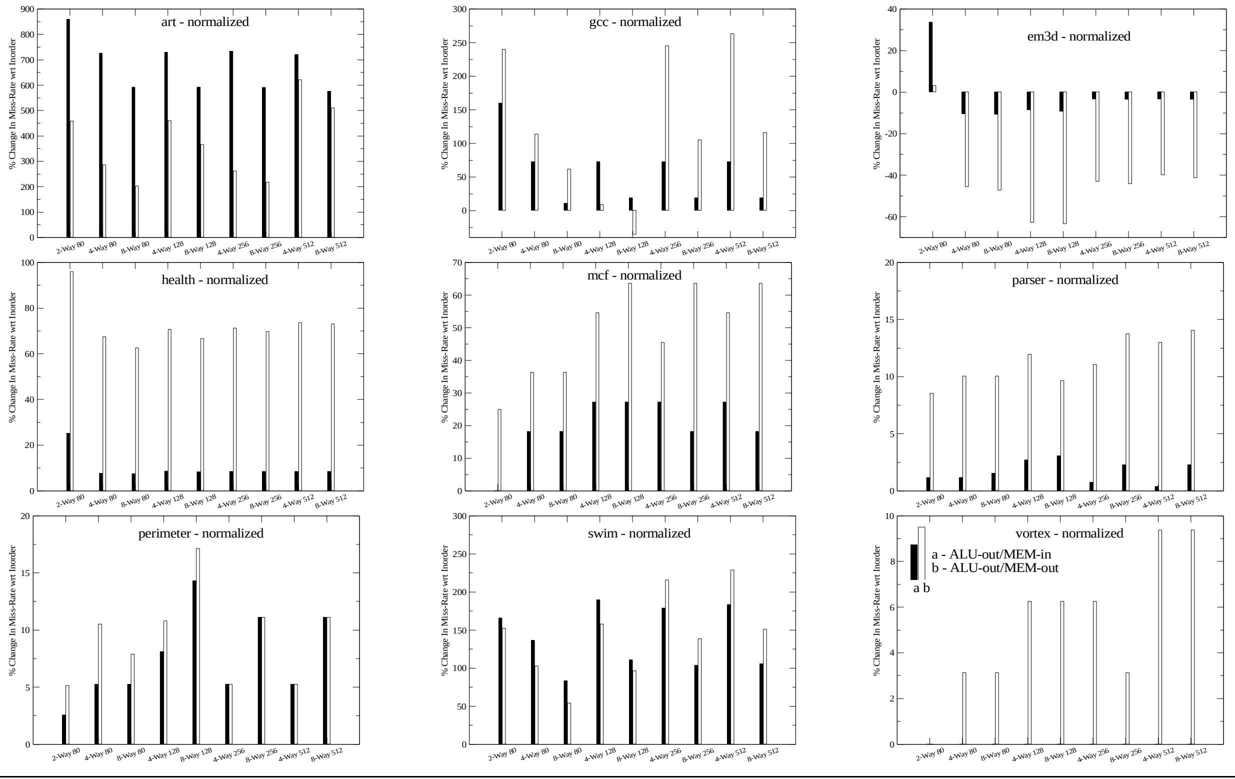


Figure 17: 512KB L2 Non-Speculative Cache Miss Rate. (above) The figure shows the non-speculative L2 cache miss rates as a percent for the ALU-in/MEM-in configuration. (below) Using the inorder core as a base machine, the graphs show the ALU-out/MEM-in and ALU-out/MEM-out non-speculative cache miss rates normalized to the ALU-in/MEM-in configuration. The graphs show that only em3d benefits from out-of-order execution by 40-60%, eight of them hurt by 5-400%, and the remaining six have little or no difference. In scenarios where out-of-order execution hurts cache performance, the ALU-out/MEM-in configuration in most cases helps reduce the non-speculative



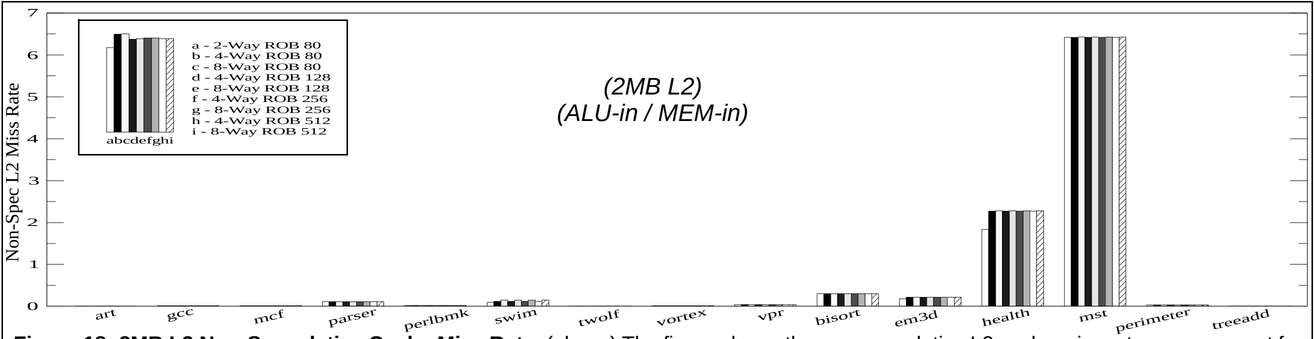
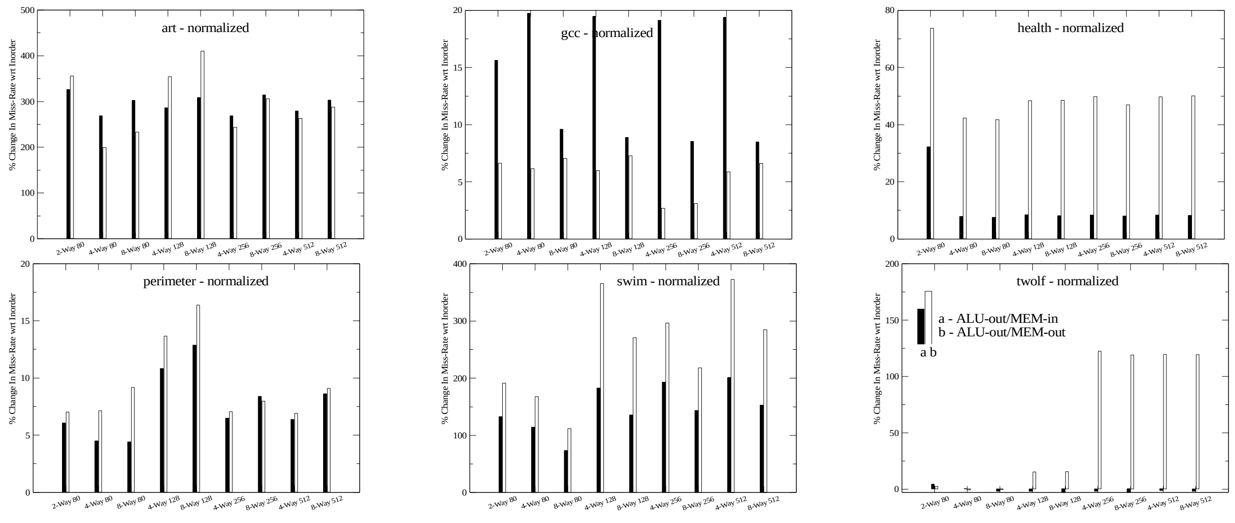


Figure 18: 2MB L2 Non-Speculative Cache Miss Rate. (above) The figure shows the non-speculative L2 cache miss rates as a percent for the ALU-in/MEM-in configuration. (below) Using the inorder core as a base machine, the graphs show the ALU-out/MEM-in and ALU-out/MEM-out non-speculative cache miss rates normalized to the ALU-in/MEM-in configuration. The graphs show that only out-of-order execution hurts performance by 5-350% for six of the benchmarks, and the remaining nine have little or no difference. In scenarios where out-of-order execution hurts cache performance, the ALU-out/MEM-in configuration in most cases helps reduce the non-speculative cache misses by 50% or more.


art - normalized
gcc - normalized
health - normalized
perimeter - normalized
swim - normalized
twolf - normalized
% Change In Miss-Rate wrt Inorder
a - ALU-out/MEM-in
b - ALU-out/MEM-out
a b
2-Way 80
4-Way 80
8-Way 80
4-Way 128
8-Way 128
4-Way 256
8-Way 256
4-Way 512
8-Way 512