ABSTRACT

Title of dissertation: THE EFFECTS OF AGGRESSIVE OUT-OF-ORDER MECHANISMS ON THE MEMORY SUB-SYSTEM

Aamer Jaleel, Doctor of Philosophy, 2005

Dissertation directed by: Associate Professor Bruce L. Jacob
Department of Electrical and Computer Engineering,
University of Maryland, College Park, MD

Contrary to existing work that demonstrate significant improvements in performance with larger reorder buffers, the work presented in this dissertation shows that larger instruction windows do not necessarily provide the significant improvements in performance. By using detailed models of the DRAM system and the memory subsystem, we show that increasing out-of-order aggressiveness by increasing reorder buffer sizes beyond 128 entries no longer buys any improvement in processor performance. In fact we observe that it can actually degrade processor performance. Additionally, this dissertation demonstrates a non-intuitive problem associated with the out-of-order execution of memory instructions: the reordering of memory instructions can cause a degradation in the performance of the memory subsystem. Specifically, we show that increasing out-of-order aggressiveness in terms of reorder buffer sizes increases the frequency of replay traps and data cache misses. The presentation of this problem in itself is of utmost significance: *the very mechanisms commonly used to improve performance are sources of performance degradation in the memory subsystem*. We observe that while the negative effects of out-of-order execution existed for only a small fraction of the time with small reorder buffers, eliminating other sources of stalls by increasing out-of-order capability introduces these

unexpected side effects in the memory subsystem to represent significant overhead. This reveals that one can not overlook rarely occurring events in the memory subsystem. To gain insight on the source of the problem, we attempt to measure the degree to which memory system performance relies on out-of-order execution. Using the network communication concept of *windowing*, we decided to change the load/store scheduling window independently of the ALU scheduling window. Our study revealed that memory instructions issued out-of-order are the primary reason for the increase in the frequency of replay traps. On the other hand, the out-of-order issue of memory instructions is responsible for the constructive and destructive references to the data cache. Incorporating detailed memory subsystem models and a realistic DRAM model into existing simulators and filtering out the destructive references from the total cache references can allow for aggressive out-of-order cores to reap the true benefits of out-of-order execution.

THE EFFECTS OF AGGRESSIVE OUT-OF-ORDER MECHANISMS ON THE MEMORY SUBSYSTEM

by

Aamer Jaleel

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2005

Advisory Committee:

Associate Professor Bruce L. Jacob, Chair
Assistant Professor Rajeev Barua
Associate Professor Manoj Franklin
Associate Professor Chau-Wen Tseng
Associate Professor Donald Yeung

To my wonderful parents

Dr. Yawer Jaleel
and
Dr. Taheniyet Farzana

ACKNOWLEDGEMENTS

First, and foremost, I would like to thank God for giving me the inspiration and faith both of which are responsible for my achievements so far.

I would like to thank my parents, Dr. Yawer Jaleel and Dr. Taheniyet Farzana, for believing in me, instilling in me the importance of education, and encouraging me to pursue higher goals. Many thanks to my sister Asfia Talat, my brothers Yaser Jaleel and Naser Jaleel for their continuous support during graduate life. Special thanks to my two-year old niece, Sofia Khaja, for keeping me in touch with the brighter sides of life by knocking on my door (almost daily) during the course of completing this dissertation.

I am indebted to my advisor, Dr. Bruce Jacob, for his unconditional support, advice and perspective in the classroom, countless hours in the office, and many, many informal meetings. I am also grateful to Dr. Donald Yeung for his continuous feedback and insights in all aspects of my educational and research career at this institution. Additionally, I would also like to thank the members of my committee: Dr. Rajeev Barua, Dr. Manoj Franklin, and Dr. Chau-Wen Tseng for their feedback.

Finally, I would like to thank the members of my research group and lab for tolerating me and being a part of all aspects of my graduate work and social life. Many many thanks to Brinda Ganesh and Dr. David Wang for their continuous feedback and insightful discussions. Besides them, many thanks to Abdel-Hameed, Ankush, Bharath, Choi, Ohm, Sadagopan, Sam, Steve, and Sumesh. Their presence and contributions during life at Maryland have helped me become what I am. Thank You!

Table of Contents

List of Figures

List of Tables

CHAPTER 1 *Introduction*

As the gap between the processor and DRAM system continues to grow, a processor can stop instruction processing due to the latencies associated with misses in the last on-chip cache. This is because instructions directly or indirectly dependent on the instruction missing in the last-level cache cannot be issued until the data is delivered to the processor by the DRAM system. To tolerate the long latencies associated with DRAM, out-of-order execution has been one of the fundamental techniques used to tolerate the long latencies associated with misses in the largest on-chip cache. The primary goal behind out-of-order execution is to allow the processor to continue doing possible useful work rather than stay idle. To do so, the processor maintains several different out-of-order hardware structures to schedule and issue instructions from. One such hardware structure is the instruction window or reorder buffer.

It is a widely held belief that the efficiency of an out-of-order core is directly dependent on the number of instructions available to the instruction scheduler. The larger the number of instructions, the more a processor is able to exploit an applications inherent instruction level-parallelism. One of the most popular mechanism to maximize out-of-order efficiency is to provide the instruction scheduler with a gigantic window of instructions to support the scheduling and issuing of instructions. Large instruction windows and aggressive instruction schedulers provide the processor with a large number of instructions deep into an applications instruction stream. The larger the number of instructions an

instruction scheduler is able to view, the better the instruction scheduler can extract multiple independent instructions.

In efforts to exploit maximum ILP, recent trends have categorized the instruction window as one of the most important design parameters in the development of high performance superscalar processors. Many previous studies have illustrated that increasing the size of out-of-order hardware structures like the instruction windows or reorder buffer, issue queues and load/store queues (even to enormous sizes) can lead to increased performance [5, 7, 31, 63, 74]. Consequently, much research has looked at the feasibility of increasing the size of these hardware data structures without negatively impacting clock cycle time [30, 51, 38, 55]. In presenting the huge performance improvements, however, most of the existing studies have discounted real effects that occur in the memory subsystem (due to which potential performance gains largely disappear).

1.1 Contributions of Dissertation

1.1.1 Problems With Aggressive Out-of-Order Mechanisms

The work presented in this dissertation demonstrates that continuing to increase the aggressiveness of an out-of-order core to improve processor performance can come at the cost of a degradation in performance in the memory subsystem. By varying the aggressiveness of an out-of-order core in terms of reorder buffer sizes, issue queues, load/store queues, and renaming registers, this dissertation brings to light problems present in real systems that many previous simulation-based studies have not addressed.

- With a detailed model of the memory subsystem and DRAM system, we show that application performance saturates beyond a 128-entry reorder buffer. In fact, we observe applications can observe a 5-10% degradation in performance beyond the use of a 128-entry reorder buffer.
- Increasing out-of-order capability conflicts with a processor's memory ordering model and requires the processor to take frequent expensive *replay traps*. An increase in the frequency of replay traps causes a processor to re-fetch and re-execute instructions beyond the trap causing instruction. This can require the fetch, map, and execution units to unnecessarily dissipate energy on work that has already been done before
- Increasing out-of-order capability destroys cache locality, thereby causing an application to suffer from a higher number of cache misses than a lesser aggressive out-of-order mechanism. The increase in the number of cache misses is associated with the increase in the amount of speculation as a result of large instruction windows or reorder buffer sizes. The increase in the number of cache accesses and cache misses needlessly dissipates energy.

1.1.2 *Disorder* — A New Metric To Measure Reordering

Having presented the problem with mechanisms to increase out-of-order capability, we show that the side effects in the memory subsystem are primarily due to the reordering of memory instructions and increased speculation. Increasing out-of-order aggressiveness allows for both ALU and memory instructions of a program to be reordered. We show that speculation and the reordering of memory instructions in aggressive out-of-order processors

can cause significant overhead in the system, i.e. the very mechanisms commonly used to improve performance can cause significant performance degradation in the system. To measure the reordering of memory instructions we introduce a new metric called *disorder* to quantify the degree by which memory instructions are issued out-of-order.

Disorder can be of two types: *global disorder* and *local disorder*. Global disorder is the degree by which memory instructions are issued out-of-order when compared to program order. The global disorder metric is used to measure how memory instructions issue out-of-order when compared to program fetch order. Local disorder on the other hand is the degree by which memory instructions issue out-of-order when compared to those memory instructions issued in the same cycle or a prior cycle. Our disorder study of the workloads illustrates that increasing out-of-order aggressiveness causes large global disorder and small local disorder in the system. The disorder results indicate that on a program level memory instructions are heavily re-ordered but when compared to other memory instructions they issue in close proximity to each other.

After illustrating the existence of global and local disorder, we investigate any correlation between increased global disorder with the degradation in the memory subsystem. We show that the global disorder metric correlates well with the degradation in the memory subsystem: the larger the global disorder the more the degradation. Based on this finding we conclude that mechanisms to reduce global disorder are required to reduce the sources of performance loss in the memory subsystem.

1.1.3 Observing the Reordering of Memory Instructions

To determine the degree to which memory system performance relies upon the out-of-order execution of memory instructions. Rather than reduce the size of the reorder buffer, which restricts the reordering of both ALU and memory instructions, we decided to change the load/store scheduling window independently of the ALU scheduling window. To do this, we use the network communication concept of *windowing*. Windowing essentially introduces a virtual load/store queue (VLSQ) within the existing physical load/store queue. The VLSQ reduces the reordering of memory instructions by limiting the number of memory instructions visible to the select and issue logic. Thus, the instruction scheduler is restricted to issue only those memory instructions that reside within the virtual load/store queue. The virtual window "slides" onto younger memory instructions only when the instruction at the virtual head is issued. Thus, younger memory instructions that are ready to be issued can only be issued when the *virtual window* slides onto them. By restricting the number of memory instructions visible to the instruction scheduler, windowing reduces the reordering of memory instructions. The smaller the size of the virtual window, the smaller the degree to which memory instructions are reordered. The larger the size of the virtual window, the larger the degree of memory instruction reordering.

Our study using windowing provides important insights on the effects of reordering memory instructions in aggressive out-of-order systems. We observe that memory instructions issued out-of-order are responsible for the frequent replay traps. Furthermore, the out-of-order issue of memory instructions in the presence of speculation is also responsible for constructive and destructive cache references. We observe that by reducing the reordering of memory instructions windowing reduces the frequency of replay traps.

On the other hand, even though reducing the reordering of memory instructions reduces the total number of cache misses, it can degrade overall processor performance. This is because the use of smaller virtual windows eliminates early cache miss detection, thus delaying the request to get data from DRAM. Based on our study we conclude that filtering the constructive memory references from the destructive references can allow for aggressive out-of-order cores to avoid frequent replay traps and the unneeded cache misses and reap the benefits of increased out-of-order aggressiveness.

1.1.4 Importance of Dissertation

The main contributions of this dissertation are as follows:

- The work presented in this dissertation shows that continuing to increase out-of-order aggressiveness to improve processor performance will come at the cost of a degradation in performance. By using a realistic model of the memory subsystem, contrary to existing work, we show that increasing out-of-order aggressiveness by increasing instruction window sizes beyond 128 entries does not buy any improvement in processor performance. In fact, we show that it can actually degrade processor performance.
- The degradation in processor performance comes as a result of an increase in the frequency of replay traps This can lead to a degradation in both performance and energy in redoing work already done before.
- The degradation in performance also comes from an increase in the total number of cache misses. The increase in the amount of speculative instructions in flight can cause destructive interference in the caches resulting in an increase in the total

number of cache misses when compared to smaller instruction windows.

- The side effects of increased out-of-order aggressiveness in the memory subsystem, in the presence of speculative execution, is primarily due to the reordering of memory instructions. We introduce a metric called *disorder* to measure the reordering of memory instructions and correlate increase in disorder with increased out-of-order capability. Furthermore, we also correlate the performance degradation in the memory subsystem with increased disorder.
- To determine the degree to which out-of-order execution of memory instructions affects processor performance, we investigated the degree to which processor performance is dependent on the out-of-order issue of memory instructions. We use the network communication concept of *windowing* to control the reordering of memory instructions while allowing ALU instructions to execute in any order. We show a direct correlation between the size of the instruction window and the frequency of replay traps and cache misses. Our investigations by statically varying the size of the window revealed that reducing the reordering of memory instructions via the use of smaller windows causes fewer occurrences of replay traps and cache misses.
- The dissertation places significance in the presentation of a problem: *continuing to increase aggressiveness of an out-of-order core to tolerate the long latencies associated with DRAM will cause a degradation in the performance of the memory subsystem*.

1.2 Organization of Dissertation

The work presented in this dissertation is organized as follows. First in Chapter 2 we provide a detailed description of out-of-order execution, the different hardware structures used, and different mechanisms used to improve out-of-order efficiency and tolerate DRAM latency. Next, Chapter 3 describes the trends in both industry and academia to improve the performance of high performance microprocessors. Next, Chapter 4 discusses in further detail memory instruction speculation and the issues associated with speculatively executing memory instructions. Next, after a description of the simulation methodology, Chapter 5 presents the pitfalls associated with increasing out-of-order aggressiveness. For aggressive systems that use blind load speculation and sequential data prefetching, we present in this chapter the increase in the frequency of replay traps and the number of cache misses. In this chapter we arrive at the conclusion that the problems are primarily associated with the reordering of memory instructions. Next, Chapter 6 presents a metric called *disorder* to measure the degree by which memory instructions are reordered. In this chapter we show that increasing out-of-order aggressiveness causes significant memory disorder and conclude that reducing the reordering of memory instructions can reduce the overheads associated with the negative effects in the memory subsystem. Next, Chapter 7 correlates the degradation in performance with increased global disorder. Next, Chapter 8, for systems with controlled load speculation and stride data prefetching, we present the use of *windowing* to gain insight on the effects of out-of-order execution of memory instructions on processor and memory subsystem performance. This chapter also discusses the effects of reducing the reordering of memory instructions on the data caches, replay traps, processor performance and power consumption. Finally, Chapter 9 concludes the dissertation and

provides future work to reduce the unexpected negative effects in the memory subsystem due to increased out-of-order execution.

CHAPTER 2

High Performance Computing Techniques

2.1 Out-of-Order Execution

Out-of-order execution is a widely used technique to tolerate the long latencies associated with cross-chip delays and last-level cache misses. Unlike in-order execution, out-of-order execution has the capability to schedule ready instructions independently of long latency instructions. The hardware needed to support this activity involves hardware structures such as instruction windows or reorder buffers, issue queues, and load-store queues. In general, increasing out-of-order aggressiveness implies an increase in the size of each of these hardware data structures.

We now provide a brief description of the different out-of-order hardware data structures mentioned and their functionality in out-of-order processors.

2.1.1 Instruction Window or Reorder Buffer (ROB)

The instruction window or reorder buffer is a hardware queue that keeps track of all instructions fetched into the pipeline. The instruction window is maintained via two pointers: *head* and *tail*. New instructions that are fetched into the pipelined are enqueued at the tail of the instruction window after being decoded. At the back end of the pipeline, old instructions are retired from the head. The main purpose of an instruction window is to queue up completed instructions so that they may be retired in program order, thus providing

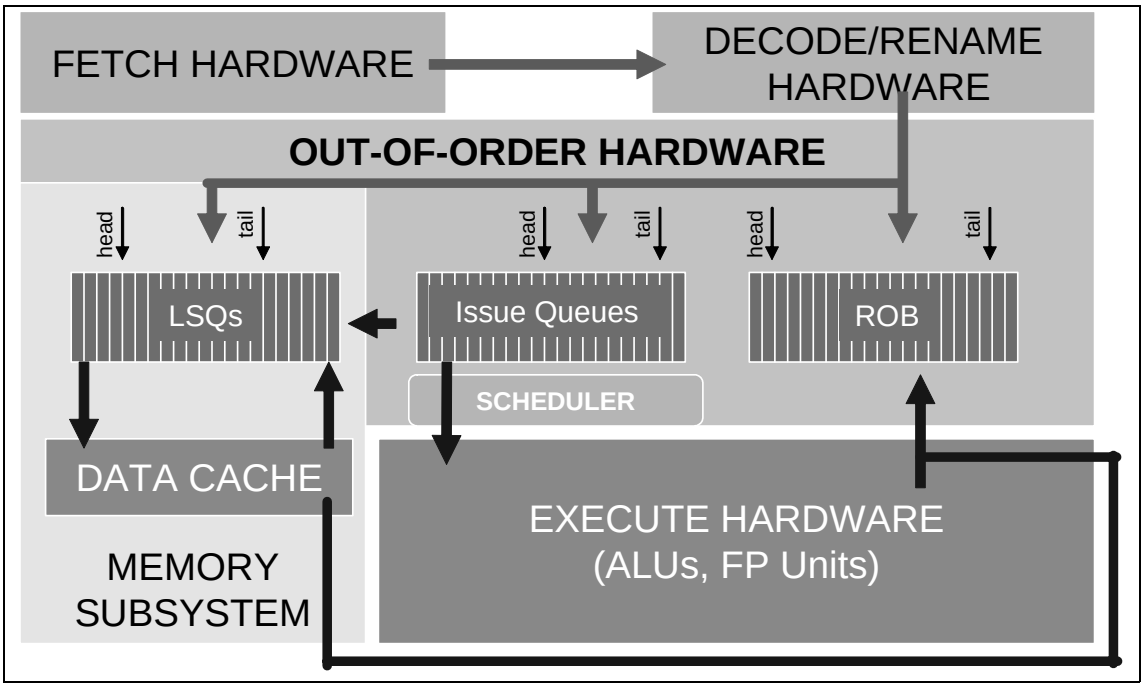


Figure 2.1: Out-of-Order Hardware Data Structures.

the illusion that all instructions are executed in sequential order—this simplifies the process of handling interrupts precisely [65, 68].

2.1.2 Issue Queue or Scheduling Window

The issue queues hold a subset of instructions that reside within the instruction window or reorder buffer. Issue queues are responsible for tracking the dependencies of instructions, determining when instructions are ready to be issued, and arbitrating among all instructions that are ready to issue. Each cycle, all members of the issue queue are informed about instructions that are about to be completed so that dependents of completed instructions may mark themselves as ready to be issued [42]. To reduce the overhead of searching for ready instructions, issue queues are commonly partitioned into integer and floating point queues.

Integer queues hold all integer operations as well as the address computation operations of memory instructions and floating point queues hold all floating point operations.

2.1.3 Load and Store Queues

Load and store queues are content-addressable-memory (CAM) structures that hold all memory instructions fetched into the pipeline. At the instruction decode and renaming pipeline stage, if an instruction is determined to be a load or store instruction, it is queued into the load or store queue. Load and store queues have the capability of supporting multiple searches to support memory dependencies or different memory consistency models [55]. For example, when a load instruction is issued, the address of the load instruction is CAMed in the store queue to determine if the data should be read from the data cache or from an existing older store residing in the store queue. In modern microprocessors, the load/store queues are one of the main hardware structures used to enforce consistency via the use of replay traps as described in Chapter 4.

2.2 Speculation

To maximize performance improvements, there are a variety of speculation techniques that are also used in conjunction with out-of-order execution. Speculation allows processors to predict the outcome of a given operation and act according to the predicted outcome. Speculation is necessary if the latency to determine the result of an operation is more than one cycle and the prediction can be made with some reasonable accuracy. As with any prediction technique, mechanisms to recover from mispredictions are also required to ensure

program correctness. We now discuss a few common speculation techniques used in high performance processors.

2.2.1 Branch Prediction

Branch instructions are control instructions that determine the flow of execution. The direction and target of branch instructions can either be determined at decode time or execute time if the branches are unconditional or conditional respectively. In general, 20% of all program instructions have been identified as control transfer instructions. Since instructions are fetched during the first stage of the pipeline, and the decode and execution stages are several stages down the pipeline, the fetch stage somehow needs to be provided intuition of branch instructions being fetched and their appropriate targets.

One method of handling branch instructions is to allow the instruction fetch unit to continue fetching and be oblivious to branches. This is also known as always fetching down the untaken path. However, such a mechanism can cause frequent pipeline flushes if a program has frequent taken branches. Thus, hardware data structures are introduced into the pipeline that track branch outcomes and targets based on the program counter (PC) of the branch instruction. At instruction fetch time, the fetch hardware consults the branch prediction hardware based on the fetchPC. If the PC hits in the branch prediction hardware, the fetch hardware redirects fetching of instructions from the target address provided in the prediction hardware.

There have been several branch predictors implemented e.g. two-level predictors, McFarling predictions, global history predictors, etc. that have provided the capability of predicting the outcomes of branches with predictions as good as 90-95%. Such accurate

branch predictors allow for the out-of-order execution hardware a larger view of real program instructions thus providing them the capability of dynamically extracting application ILP.

2.2.2 Data Prefetching

The ability to predict what data a processor will be using in the near future and be able to bring that data into the cache from lower levels of memory (e.g. DRAM) *before* the processor actually requests the data is known as data-prefetching. The ability to accurately predict what data to prefetch and the mechanisms to allow for data prefetching is an important area of research to tolerate long DRAM latencies. Modern high performance microprocessors use prefetching mechanisms that are implemented using software, hardware, or helper threads. We now provide a brief description of each of these mechanisms.

Software prefetching [6, 40, 46] is a technique by which explicit prefetch instructions can be introduced into the application stream to bring data into the cache ahead of time. Normally this is done by the compiler by inserting a load instruction to the same memory address several instructions before the value of the load is actually used. Since the compiler or program writer is directly involved in explicitly prefetching the data, the task of software prefetching is non-trivial. Furthermore, the fact that explicit instructions are inserted into the existing instruction stream can also increase the instruction bandwidth requirements.

In attempts to reduce the overhead in terms of instruction bandwidth and the non-trivial nature of predicting the address stream of the application statically, hardware prefetching techniques [6, 18, 34, 22] can be used to dynamically detect the access pattern of

applications and prefetch data based on the access pattern. The most common hardware prefetch techniques are the sequential prefetching [20, 21, 64] and stride prefetching [14, 21]. Sequential prefetching attempts to exploit spatial locality by prefetching consecutive blocks of data (on a hit or miss) in hopes that future accesses will not miss in the data cache. On the other hand, stride prefetching detects and prefetches data based on access stride patterns that the hardware prefetcher dynamically detects. The benefit of a stride prefetcher is that it does not require spatial locality to be present. The main problem with hardware prefetching is the hardware cost and complexity to build a prefetcher that can accurately determine the different types of access patterns. Additionally, if the accuracy of the hardware prefetcher is low, cache pollution and wasted memory bandwidth can degrade processor performance.

With the introduction of multi-threaded architectures [13], prefetching can also be exploited by assigning helper threads to prefetch data [17, 39, 75]. By using idle threads, a processor can schedule tasks threads to help the primary thread [13]. The helper threads execute code that prefetch data required by the primary thread. However, the main disadvantage of thread-based prefetching techniques is that they require idle threads and the availability of spare resources to handle the demands of the helper thread.

2.2.3 Load-Hit Speculation

In general, the latency from the time when an instruction is issued to the time when it is actually executed is usually greater than one. This is because of the delays associated with register file accesses or the transfer of data across bypass paths. To accommodate this extra latency, the instruction issue logic allows for the early issue of instructions dependent on

older instructions from the queues. The early issue of instructions handles well for non-load dependent instructions, however, the early issue of instructions dependent on load instructions poses unwanted behavior as load instructions have a non-deterministic latency due to their unknown hit/miss status.

There are a couple of ways of handling the non-deterministic latency. The issue logic can wait until the result of the load is determined before it issues instructions dependent on the load. However, this can be a source of performance loss as the first level caches usually have a high hit rate. Thus, a more aggressive approach is to assume that all loads issued hit in the L1 data cache. This is known as load-hit speculation, and allows for instructions dependent on the load to be issued early. However, this approach requires the need to handle the re-issue of instructions dependent on the load incase the load misses in the data cache [42].

2.2.4 Load Speculation

Since the effective addresses associated with memory instructions are determined after they are issued to execute, prediction techniques can be used to determine the dependencies of instructions in memory. For example, a load instruction to memory address M must only be issued after a store instruction to memory address M (if it exists). However, the question arises as to what to do if a load instruction is ready to be issued but an older store instruction has not been issued yet. The conservative approach is to wait till the older store instruction is issued before the newer load instruction can be issued. Such a mechanism can be a source of performance loss if the load instruction was not dependent on the store instruction. Thus,

prediction mechanisms can be used to predict whether a younger load is dependant on older stores.

Rather than requiring all loads to wait on prior unresolved stores, load speculation is commonly used. To allow for load speculation, Calder et al. [10] perform a survey of techniques that tackle the false memory aliasing problem. They analyze four different mechanisms that allow for load speculation: dependence prediction, address prediction, value prediction, and memory renaming. Loads predicted to not alias to older stores are issued speculatively. If the load is mispredicted and it actually depends on an older store, instructions are squashed and execution restarts at the mispredicted load instruction.

Load speculation is a mechanism used to send loads that are ready to execute to the memory subsystem as soon as possible even before older store addresses are not resolved. However, processors need mechanisms to ensure that the memory independence prediction is valid. To do so, after the store instruction computes its effective address, the store CAMs the load queue to determine if any younger loads that were dependent on the store were issued before the store did. If so, the younger load instruction must be replayed (as it acquired stale data from the cache) and is flagged with a replay trap. Replay traps are further discussed in Chapter 4.

CHAPTER 3

Trends in Improving Processor Performance

3.1 Industry Trends

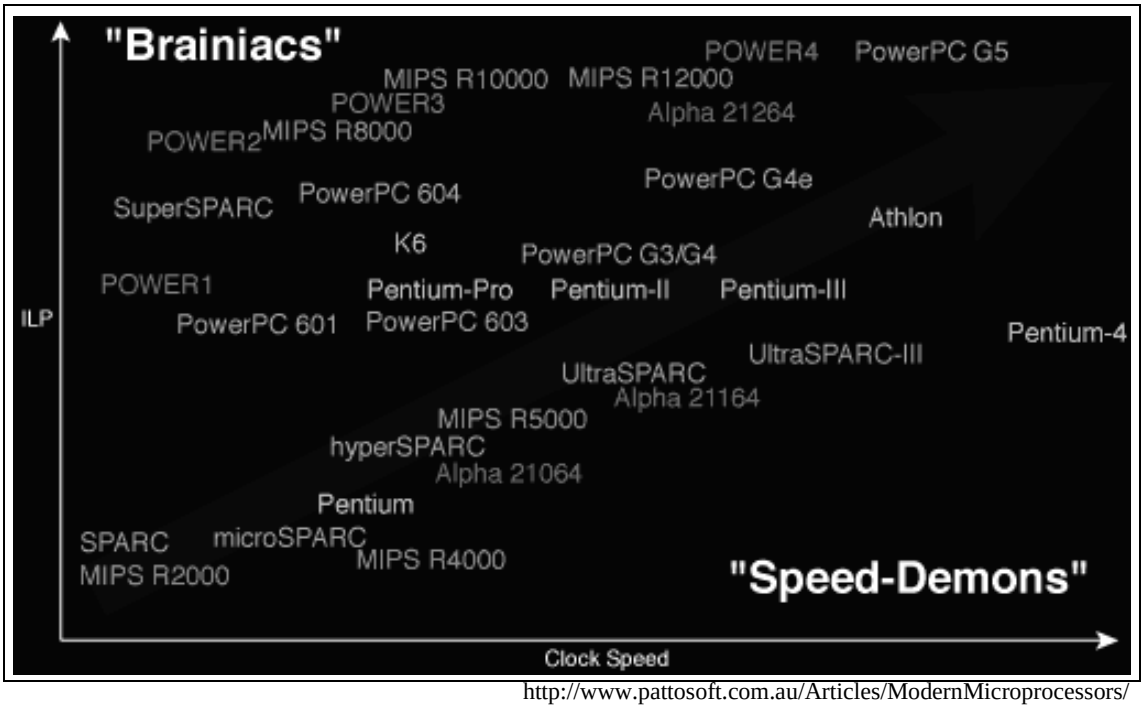
The designs of modern high performance microprocessor architectures rely on very aggressive hardware mechanisms to maximize processor performance. Techniques such as branch prediction, data speculation, load speculation, hardware and software prefetching, cache line prediction and pipeline scheduling 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 belonged 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 increases IPC while maintaining low clock speeds (e.g. IBM's POWER2, MIPS R10000 and others as shown in Figure 3.1). *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 decreases in feature size and improved micro-architectural 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 was a hot debate. From Figure 3.1, we observe that



http://www.pattosoft.com.au/Articles/ModernMicroprocessors/

Figure 3.1: Brainiacs Vs. Speed-Demons. Trends in industry microprocessor designs.

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 R10000, 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).

More recently, industry is now moving towards multiple cores per die. With the growing amount of transistors available with each new shrink in the process technology, the trend is now to increase computation power by adding more cores rather than cache space. With the growing amount of parallel programs in the fields of multimedia, transaction processing and many other emerging fields, the goal of multi-cores now is to provide the capability of more compute power per square millimeter of chip area.

3.2 Related Work

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 [5, 51, 55, 63]. 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 have proposed a variety of techniques to improve processor performance.

Since memory latency is one of the major hurdles that a microprocessor has to overcome, most research investigations have proposed and investigated mechanisms to tolerate this latency. Prediction techniques such as load speculation allow processors to send load requests as early as possible to the memory subsystem so as to reduce latencies in the event of a cache miss. Other research methods have proposed novel techniques to increase ILP by providing mechanisms to achieve the performance of large instruction windows

without sacrificing clock speeds. We now discuss some of the relevant work that others have proposed to tolerate memory latency.

Even though there exists a longing desire to improve processor performance by increasing the sizes of instruction windows and reorder buffers, in general clock speeds and the size of instruction windows are at odds against one another. Increasing the size of the instruction windows requires a longer amount of time to consult all entries within the instruction window to determine potential instructions for scheduling. Thus, researchers have arrived at an understanding that larger hardware structures (e.g. instruction windows) conflict with increasing clock speeds and alternative design methodologies must be investigated.

Based on this, a good deal of recent effort has aimed at solving the large instruction window problem by investigating alternative mechanisms. These mechanisms can be categorized in two different ways:

- **Better Algorithms and Circuit Implementations:** The research contributions in this category aim at designing better algorithms and/or efficient circuit techniques that are not on the critical path when scaling the size of the instruction window.
- **Emulate the Behavior of Large Instruction Windows with Smaller Instruction Windows:** The research contributions in this category introduce additional hardware data structures, when used, emulate the behavior of large instruction windows without impacting clock cycle time. Such mechanisms may also be required to maintain checkpoints at regular intervals in the event a roll back is needed.

We now describe some of the important contributions to both these categories.

Better Algorithms and Circuit Implementations

Henry et al. [30] proposed new circuit implementations for the logic components that limit the critical path of a processor with large instruction windows. Specifically, they proposed new circuit implementations for the rename, schedule, wake-up, and commit logics. They propose the use of log-depth cyclic segmented prefix (CSP) circuits to re-implement the schedule and wake-up logic. They show that their modified circuits can be used with existing technology to build a 500 MHz processor with 8-way issue width and a 128-entry instruction window. Thus, using an alternative circuit design for the timing critical portions of the instruction window they were able to build larger instruction window sizes while still maintaining high clock speeds.

Onder et al [51] illustrated that existing mechanisms to wake-up instructions that are dependent on instructions that have finished execution do not scale well with increase instruction window sizes. They propose a new wake-up algorithm to dynamically associate explicit wake-up lists associated with each executing instruction. The wake-up list is essentially a list of all direct dependents of any instruction. The insight here is that rather than repeatedly examining an instruction to determine if it can be woken up, only a subset of waiting instructions can be woken up based on the explicit wake-up list associated with the instruction. Such a mechanism allows for the reduction in the fan-out of the wake-up logic and hence allows for the implementation of larger instruction windows.

Emulate Behavior of Large Instruction Windows

Lebeck et al. [38] illustrate the need for large instruction windows to be able to continue doing useful work while an older instruction misses in the data cache. The insight here is

that those instructions that are dependent on the long latency operation cannot execute until the source operation completes. Thus, all instructions dependent on the long latency operation can be moved from the instruction window into an alternate *waiting instruction buffer* (WIB). This frees up room for new instructions to be fetched and executed, essentially emulating a large instruction window. When the older long latency operation has finished execution, the instructions are moved from the WIB back into the instruction window. Thus, this mechanism emulates the behavior of a large instruction window rather without having to physically increase the size of the instruction window.

Scaling up the size of the instruction window makes sense only if all associated out-of-order hardware structures are scaled accordingly as well. One such hardware structure is the load/store queue which maintains a list of memory instructions in flight in-order. Since larger instruction windows bring on chip a larger number of memory instructions, the increase in size can lead to a extensive store to load communications. Since the load/store queue is a CAM structure, frequent store to load communications may not be able to be handled in the same cycle. Thus, Park et al. [55] propose the use of segmentation to scale the load/store queue size.

Akkary et al. [5] show that large instruction windows usually have four critical components that need to be dealt with: a) scheduling instructions b) recovering from branch mispredicts c) buffering stores and forwarding data to loads and d) reclaiming physical registers. They show that scheduling window size (i.e. choice (a)) is not as sensitive to the other three issues when dealing with large instruction windows hence provide novel solutions for each one of them. They propose a novel checkpoint and recovery mechanism to recover from branch mispredicts, a hierarchical store queue organization for quick data

forwarding to loads, and a novel algorithm to reclaim physical registers as quickly as possible.

Since the primary reasoning behind large instruction windows is to tolerate the long latencies to memory, Onur et al. [48] propose the use of runahead execution. With runahead execution, the processor checkpoints the current architectural state and unblocks the instruction window (which is blocked by a long latency operation). The long latency operation is provided with a a bogus value. The processor then enters “runahead mode” and continues executing as if normal, but does not commit state to the architectural register file. Eventually when the blocked operation finishes, the pipeline is flushed and the processor enter “normal mode” and execution restarts at the blocked operation. Such a mechanism effectively creates very accurate data and instruction cache prefetches. Thus, the use of runahead execution provides the emulation of a large instruction window to prefetch data.

Even though much work has discussed mechanisms to tolerate long latencies, it is important to also mention that Burger et. al. [8] point out that when attempting to use aggressive latency tolerance techniques, memory bandwidth, particularly pin bandwidth, and not raw memory access latencies will prevent future 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.

Our work differs from prior work in that it explicitly illustrates that even though increasing out-of-order capability by increasing instruction window sizes does improve performance, it however, comes at a cost of a degradation in performance in the memory

subsystem. Existing work [47, 57] has discussed the effects of speculative execution on the performance of caches, and have proposed mechanisms to filter the effects [47]. Our work shows that continuing to increase aggressiveness of the out-of-order core can result in an increase in the frequency of replay traps and the number of cache misses in the memory subsystem. We correlate these sources of performance degradation to the reordering of memory instructions and propose the concept of *windowing*. Windowing essentially reduces the reordering of memory instructions, the performance degradation in the memory subsystem, and the power and performance overheads of speculatively issuing memory instructions.

CHAPTER 4 *Memory Subsystem Issues*

In attempts to improve performance, the out-of-order issue logic exploits an application's inherent instruction level parallelism (ILP) by finding and issuing to execute instructions that are independent of long-latency instructions. As a result of instruction scheduling both ALU and memory instructions are executed in an order different from program order. Since register renaming maintains the dependencies of ALU instructions, the out-of-order issue of ALU instructions poses no threat to functional correctness. This is because ALU instruction dependencies are defined by their source registers; the source and destination registers are defined in the bytes of the instruction itself. Thus, at decode time, the instruction decoder *knows* which instructions ALU instructions depend on and the register renaming logic sets the appropriate dependency chains to determine when to dispatch instructions to execute. Thus, the out-of-order issue logic issues instructions only when the appropriate producer instructions have finished execution.

Though the dependencies of ALU instructions are easy to determine, the dependency chains of memory instructions are not as trivial. The execution of memory instructions usually involves the computation of the effective address and then the loading/store of data to caches/memory. The computation of the effective address of memory instructions cannot be determined statically, instead the memory instruction must actually be issued to functional units. Thus, any dependencies associated with these instructions in memory can be determined only after they have been issued to execute. If it so happens that after having

issued to execute, if a direct or indirect memory dependence is determined e.g. if two memory instructions access the same memory location, then measures must be taken to ensure functional correctness. In general, issuing memory instructions that access the same memory location (regardless of the order in which they are issued) can cause inconsistency issues depending on the consistency model defined by the processor. To handle the issues of inconsistency or functional correctness, mechanisms to ensure correct execution are required. Modern processors use the mechanism of *replay traps* to ensure consistent state and functional correctness.

4.1 Replay Traps

A replay trap occurs when the processor must roll back its state to force accesses to a particular memory location in order, or to handle different-sized accesses to the same memory location. The Alpha Compiler Writer's Guide describes replay traps as:

> "Replay traps occur when there are multiple concurrent loads and/or stores in progress to the same address or the same cache index. The best way to avoid replay traps is to keep values in registers so that multiple references to the same address are not in progress at the same time" [2].

Replay traps preserve a programs producer-consumer semantics between load and store instructions and is an internal processor mechanism that should not be confused with software managed traps/interrupts.

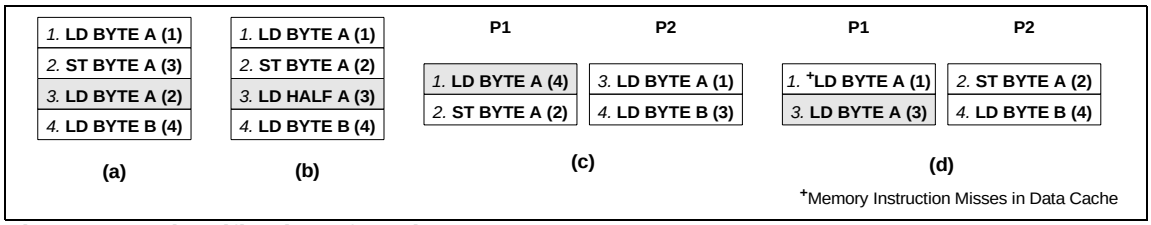


Figure 4.1: Classification of Replay Traps. The figure illustrates the different types of replay traps that can occur in both uniprocessor and multiprocessor environments. (a) Load-Store Replay (b) Wrong Size Replay (c) Load-Load Replay (d) Load-Miss Load Replay. In the examples, due to a replay trap, re-execution starts from the shaded instruction. Numbers in parenthesis show program execution order and numbers in italics show actual program order.

We now describe the different types of replay traps that are possible when issuing memory instructions are executed. Figure 4.1 lists the different types of replay traps, and we now provide a detailed description of the replay trap and the reasoning behind their use.

4.1.1 Replay Traps For Functional Correctness

When executing memory instruction in a modern superscalar, certain conditions must be met to ensure the execution of any application is functionally sound. In some cases, even with in-order processors certain corner cases may arise where a processor needs to handle the special case in order to avoid complexity during the design process. We now discuss two replay traps that are required to ensure functional correctness.

Load-Store Replay Trap

Load-store replay traps, in the presence of load speculation, are required to handle the fact that applications can communicate with each other via memory rather than explicitly through registers. This usually happens when the compiler is required to spill data to memory due to register pressure. For example, for a given program, a store instruction may write to a particular location 0xABB0 in memory (e.g. STQ r10, 0xABB0) and at a later point in the program a load instruction reads from location 0xABB0 (e.g. LDQ r4, 0xABB0)

the value stored by the older store instruction. To ensure correctness, the order of the store and the load must be maintained as there is a dependence associated between the load instruction and the store instruction. The dependence is based on the fact that they both share the same effective address, and this is commonly termed as a *memory dependenc*e.

In the presence of load speculation [60], load instructions can be issued before all prior store instructions are executed, i.e. before a stores effective addresses are resolved. Without load speculation, a load instruction would issue after all older stores are resolved and either gets its data from the data cache or the store queue (in case the store has not retired). However, with load speculation, if a load instruction did issue out-of-order when compared to an older store instruction, then the load instruction acquires stale data from the cache (since the store has not executed or stored data into the cache). To avoid this source of inconsistency, in the presence of load-speculation, store instructions are required to CAM the load queue at execute time to detect if any younger load instructions had the same effective address. If so, then the younger load instruction and all of its dependents must be re-executed to propagate the correct value of the store. To identify that the younger load instruction must be re-executed, the processor flags the load instruction to be *replayed*, and this fact is noted within the load instruction reorder buffer entry.

P1.I1: STB R4, 0xABB2(R1) (*R1 = 0*)

P1.I2: LDB R4, 0xABB0(R2) (*R2 = 2*)

P2.I3: STB R8, 0xABB2(R0) (*R0 = 0*)

P1.I4: ADD R1, R4, R6

P1.I5: LDB R5, 0xABB4(R0) (*R0 = -2*)

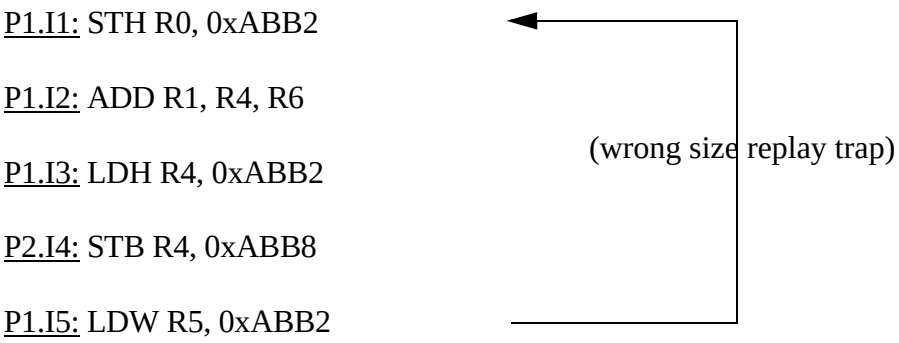
To illustrate the need for a a load-store replay trap, we now provide an example below. If the processor executes the program in the order I1, I2, I5, I4, and I3, then memory instruction ID 5 will receive the data stored by memory instruction ID 1, rather than memory instruction ID 3. This is incorrect execution of the program and must be rectified by re-executing memory instruction ID 5 and all of its dependent instructions.

Of course, load-store replay traps are only required if a microprocessor employs load speculation, i.e. it issues load instructions before older store instructions have been resolved. Studies have shown that being conservative and not using load speculation can cause unnecessary performance degradation especially if younger load instructions did not depend on any older unresolved stores. The performance degradation become significant especially if the younger loads miss in the data cache [60]. This is because sending the load to the memory subsystem as early as possible can allow for early cache miss detection and can overlap useful processor work with cache miss latency. Thus, some microprocessors employ the use load speculation to enhance performance, e.g. Alpha, POWER4 [1, 2, 24, 70].

Wrong Size Replay Trap

A wrong-size replay trap is required when the data required by a load instruction is partially present in the data cache and partially present in the store queue. This usually occurs when the address of the load instruction overlaps partially or completely with the address of an older store instruction, i.e. there is a partial memory dependence between a younger load and an older store instruction.

For example, if a store instruction writes two bytes (a half word) worth of information to location 0xABB2 (e.g. sth r0, 0xABB2), then on a byte addressed machine, data is written to



location 0xABB2 and 0xABB3. If a younger load instruction wishes to read two bytes of information from location 0xABB2 (e.g. ldh r2, 0xABB2) or one byte of information from 0xABB3 (.e.g. ldb r1, 0xABB3), the data can be forwarded from the store queue to the appropriate destination registers. However, if the load instruction attempts to read four bytes (a word) worth of information from address location 0xABB0 (e.g. ldw r4, 0xABB0), then for such a request, the data associated with addresses 0xABB2 and 0xABB3 is in the store queue and data associated with addresses 0xABB0 and 0xABB1 is in the data cache. To avoid additional circuitry to MUX and merge portions of the needed data from the data cache and the store queue, the younger load instruction is flagged with a replay trap.

Since store instructions write data to the data cache only after commit time, the younger load instruction must wait till the store instruction commits and drains its data into the data cache. Thus, the overhead of a wrong-size replay traps is the total time for the store instruction to commit and drain the data in the data cache. It is important to point out here that a wrong size replay trap can occur even if memory instructions are issued in program order [1, 2, 70]. In general, a compiler is responsible for ensuring that partial memory dependencies do not exist between loads and stores. However, there are times during the compilation process that this is unavoidable. Hence, all high performance microprocessors must be able to detect and overcome this hazard [24].

4.1.2 Replay Traps To Enforce Memory Consistency

To be able to write correct and efficient shared memory programs, programmers need to understand exactly how memory behaves when it comes to read and write operations that originate from multiple processors in a shared-memory multiprocessor environment. To shed some light on this behavior a memory consistency model is defined for shared-memory multiprocessor systems. Memory consistency models [4, 26, 44, 59, 76] provide a formal specification of exactly how the memory system will appear when processors read or write to locations in memory. In general, the consistency model places restriction on the data value that is returned upon reads. Intuitively, a read should always return the last data value written. In a uniprocessor system, this is easy to do: the *last* value written is based on program order. However, in a multi-processor environment this is harder to do as reads and writes can be on different processors of a multi-processor system. To solve this problem, a number of consistency models have been proposed to allow for hardware and compiler optimizations. A detailed discussion of memory consistency is out of the scope of this work, however we refer the reader to the following excellent tutorial [4].

Load-Load Replay Trap

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 lead to 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 can obtain the older value and the older load can obtain a newer value.

We now illustrate an example of load-load replay traps with two processors P1 and P2. Assume that processor P1 and P2 execute the following instructions that are listed in program order:

P1		**P2**	
P1.I1: LDB R4, 0xABB0(R2) (*R2 = 2*)	**(2)**	P2.I1: MOV R1, R8	**(1)**
P1.I2: ADD R1, R4, R6	**(3)**	P2.I2: STB R8, 0xABB2	**(4)**
P1.I3: LDB R5, 0xABB4(R0) (*R0 = -2*)	**(6)**	P2.I3: SUB R1, R2, R3.	**(5)**

The global in-order execution requirements as required by the programmer is shown in parenthesis next to each instruction. That is, the multi-processor state should be comparable to the state if the program were to execute on a uniprocessor with instructions committing in the order described by the numbers in parenthesis. In an out-of-order core processor, instructions on both of these processors can be reordered. The ordering of instructions on P1 for example could be P1.I3, P1.I1, P1.I2. Thus, memory instructions can be reordered because the instructions producing R0, and R2 could produce the values at different times based on their producers or the individual instruction latencies. From a uniprocessor perspective such a reordering of the memory instructions is absolutely ok. However with a multiprocessor perspective this will lead to an inconsistent result because now P1.I3 will acquire the old value of the data stored at memory location 0xABB2 by P2.I2 and P1.I1 will now have the new value stored by P2.I2. This is a problem and needs to be handled to allow for correct execution of programs.

The above load-load ordering problem can either be handled in hardware or explicitly by the software programmer. In the software approach, if a relaxed memory consistency model is supported, a programmer can use *memory barrier* instructions provided by the processor. Memory barrier instructions allows the programmer to enforce ordering among memory instructions where ever needed. With *memory barrier* instructions, all processors wait at a particular location in the program until everybody reaches that barrier. Thus, instructions beyond the barrier aren't executed until everybody reaches the barrier. For example, in the code above, a programmer can insert barrier instructions before P1.I2 and after P2.I2. In such a case, the processor can handle instructions beyond the barrier in one of two ways: (1) do not execute instructions beyond the barrier, or (2) speculatively issue instructions but restart execution of instructions beyond the barrier after all processors reach the barrier. By inserting the barriers programmers can ensure that instructions execute and produce results as they expect them to do.

However, it has been addressed that extensive use of memory barriers can negatively hurt performance [55]. This is because application level ILP is lost whenever barriers are reached as no further useful progress can be made until all processor reach the barrier. Thus, hardware support, via replay traps, is also provided by processors to guarantee load-load ordering to the same address. (e.g., Alpha [1, 2, 24], POWER4 [70], and MIPS R10000 [3], Sparc)

Load-miss Load Replay Trap

A load-miss load replay trap occurs 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. An MSHR keeps track of an outstanding memory request to a single cache line [37]. It is used to coalesce multiple requests to the same cache line by keeping track of all requests that are waiting on the same cache line of data to arrive from memory. In doing so, MSHRs prevent multiple requests for the same data to be sent to memory. Eventually, when the data arrives from memory, the MSHR provides all outstanding destination requestors with the data they were waiting on.

Just like the load-load replay trap, memory inconsistency issues can occur if two loads miss in the data cache and are waiting on an MSHR and an intervening store from a different processor exists. In such a case the data written by the remote store would be lost as the MSHR provides the waiting loads with data from DRAM. We point out here that this problem does not require for the out-of-order issue of load memory instructions. Instead, two loads to the same address issued in program order can cause memory inconsistency in a multi-processor environment. We now illustrate an example of load-miss-load replay traps with two processors P1 and P2. Again, if we assume the fetch sequence and illustrated by the order of instructions listed for each processor and the needed execution sequence as described by the numbers in parenthesis.

P1		**P2**	
P1.I1: LDB R4, 0xABB0(R2) (*R2 = 2*)	**(2)**	P2.I1: MOV R1, R8	**(1)**
P1.I2: ADD R1, R4, R6	**(3)**	P2.I2: STB R8, 0xABB2	**(4)**
P1.I3: LDB R5, 0xABB4(R0) (*R0 = -2*)	**(6)**	P2.I3: SUB R1, R2, R3.	**(5)**

In our example we assume that cache coherence protocols are present to update or invalidate remotely cached copies of data. Again, we assume that there are no memory

barrier instructions used by the programmer. If we assume that instructions on processor P1 are issued in order and instruction P1.I1 misses in the data cache and has a miss status holding register assigned to it. Meanwhile P1.I3 also executes and since it is to the same address it will also miss in the data cache, however with an outstanding MSHR to the same cache line, P1.I3 will merge with the existing outstanding MSHR. Eventually the data arrives from memory and the data is provided to both P1.I1 and P1.I3. However, now P1.I3 receives stale data and not the most up-to-date copy of the data that should have been provided by P2.I2. Thus, to avoid this source of memory inconsistency, when the processor detects an outstanding load to the same memory address, it flags a replay trap on the newer memory instruction and waits until the data for the first load is loaded into the destination register. (e.g., Alpha [1, 224]).

4.2 Handling Replay Traps

Having illustrated the need and detection of replay traps, we now require mechanisms to fix the problems associated with replay traps. Since replay traps are associated with incorrect data being consumed by load instructions, it is required that the load instruction be re-executed with the correct data value. However, it is not enough to just re-execute the load instruction as any dependents on the load need to be propagated with the correct value of the load, and in turn their dependents, and so on. Thus, handling a replay trap requires the re-execution of the entire direct and indirect dependency chain of the affected load instruction. There are two mechanism of handling this re-execution: the squash and re-execute method and the re-execute method. We now briefly discuss both these mechanisms.

4.2.1 Squash and Re-Execute

With the squash and re-execute mechanism, replay traps can be handled in the same way that branch mispredicts are handled: flush the entire pipeline and re-execute from the replay trap causing instruction. With such a mechanism, when a load instruction is detected to be replayed, the fact is noted in the instructions reorder buffer entry. While committing instructions, if the processor detects that the instruction caused a replay trap, the pipeline is flushed and execution is restarted at the replay trap causing instruction. With such a mechanism, all instructions younger than the load instruction are re-fetched, and re-executed. This can be a tremendous source of performance and energy loss as instructions that are independent of the trap causing instruction must be re-fetched and re-executed. However, such a scheme can be easily built into existing hardware as it can use existing mechanisms already built into the processor e.g. interrupts, branch mispredicts. Thus, if the frequency of replay traps are relatively low then the overheads of re-fetching and re-executing instructions can be negligible or overlooked.

4.2.2 Re-Execute Architecture

An alternative to flush and re-execute is to only execute those instructions that are dependent on the exception causing instruction. Such a mechanism would require the hardware to do one of two things:

- Retain all instructions dependent on load instructions in the reservation stations or issue queues until it is determined with good certainty that the load instruction does not cause a replay trap
- Remember the dependency chain of all instructions dependent on all load

instructions in case it is determined later that it causes a replay trap

Both these solutions are complex and require additional hardware structures and complexity. The first approach is overkill as the latency to determine whether a load causes a replay trap or not can be extremely long, in fact, it cannot be determined until the commit time of the load. Such a mechanism can cause undue performance degradation in ensuring that no replay traps occur. With the second approach, attempting to maintain or remember the direct and indirect dependency chains of a load instruction can be relatively expensive and complex. This can become an even larger problem as the sizes of instruction windows are scaled up. Hence this mechanism will not scale well with growing demands to increase instruction window sizes. However, assuming that the complexity and latency to determine all the dependencies of the load instruction are negligible, the hardware now requires a mechanism to re-inject the instructions back into the issue queue so that they can be re-executed.

4.3 Summary

This chapter provided a detailed overview of the different types of replay traps that can occur due to the execution of memory instructions. We showed the existence of four replay traps, two which are required for the correct execution of programs on both single and multi-processor systems and two that are required to maintain memory consistency in a multi-processor environment. We showed that these traps need not occur due to the reordering of memory instructions, in fact they can occur even if memory instructions are issued in program order. Furthermore, we showed that the mechanisms of handling replay traps can

be rather expensive either in the way that they are handled or in the hardware required to implement them. With a flush and execute method needless time and energy is spent in re-executing a window of instructions. On the other hand, the re-execute only method can be expensive to build into existing hardware. Thus, based on the fact that the mechanisms that are required to handle replay traps themselves are expensive, it is of extreme importance that the frequency of replay traps be minimal.

CHAPTER 5 *Pitfalls of Increased Out-of-Order Capability*

5.1 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 in modern superscalar processors can affect program execution in two distinct ways: increased replay traps and cache misses. We observe that while these events occur only a fraction of the time with lesser aggressive out-of-order mechanisms, increasing out-of-order capability exposes them as an overwhelming hazard to overall performance.

5.1.1 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 [52, 60], 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 be handled either by flushing the pipeline and restarting execution at the faulting instruction or by re-executing only the faulting instruction and all of its 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 [60]. However, with either method of handling traps, as the frequency of replay traps increases, significant performance and energy is wasted in re-fetching and re-executing instructions.

To measure the increase in the number of replay traps with increased out-of-order aggressiveness, we define two metrics: *replay trap frequency* and *replay trap overhead*. We define the replay trap frequency as the total number of times a processor is required to handle a replay trap per 1000 instructions committed. Replay trap overhead on the other hand is a measure of the total amount of work wasted (in cycles) normalized due to the occurrence of replay traps.

5.1.2 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 the wastage of energy.

5.2 Simulation Methodology

5.2.1 Performance Simulator

For the purpose of our study, we use a validated execution-driven Alpha 21264 simulator [22, 23]. 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 associative TLBs. The simulator models a detailed SDRAM memory and bus model[19]. The simulator also models two prefetching schemes for the L2 data cache: 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. 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 Chapter 4.1 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. We remind the reader that these hardware events do not require handler support—they merely require re-execution of instructions starting from the older memory instruction.

5.2.2 Baseline Study Processor Configuration

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 5.1. Additionally, we vary the data cache parameters as shown in Table 5.2, and assume a perfect instruction cache. In this study, we assume aggressive processor configurations in both prefetching and load-speculation. The purpose of choosing aggressive mechanisms as part of our initial study is to measure the effects of increased out-of-order aggressive mechanisms on the memory subsystem. Secondly, our choices of aggressive mechanisms (prefetching and load-speculation) are easy to build into existing systems without any additional hardware support. As prefetch mechanisms are becoming standard in modern high performance processors, the prefetch mechanism we chose to model is the fetch next sequential line mechanism that automatically prefetches the next four lines on a cache miss. In terms of load-speculation, i.e. the issuing of load instructions even if older store instructions are not resolved, the simulator models “blind speculation.” This means that the out-of-order issue logic assumes that a load instruction is independent of all older store instructions. In the event that a younger load instruction depends on an older store instruction, the processor replay traps the younger load instruction. Later in Chapter 8 we investigate the effects of increased out-of-order aggressiveness with stride prefetching and load speculation using predictive mechanisms (store sets) to track the dependencies between younger load instructions and older store instructions.

To measure the impact of out-of-order execution, we define three issue logic configurations: ALU-in/MEM-in, ALU-out/MEM-in, and ALU-out/MEM-out as described

Table 5.1: 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	4/2	20/15	4/4/1/1	32/32	41/41
Alpha 21264 x 2	128	4/2	40/30	8/8/2/2	64/64	82/82
Alpha 21264 x 4	256	4/2	80/60	16/16/4/4	128/128	164/164
Alpha 21264 x 8	512	4/2	160/120	32/32/8/8	256/256	328/328

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

Table 5.2: Cache Configurations

L1 Size	L1 Latency	L1 Line Size	L2 Size	L2 Latency	L2 Line Size
16 KB	2	32 Bytes	512 KB	8	64 Bytes
32 KB	2	32 Bytes	1 MB	12	64 Bytes
64 KB	3	64 Bytes	2 MB	15	64 Bytes

Table 5.3: Issue Logic 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 in-order issue.
ALU-out / MEM-in	ALU operations are issued out-of-order. Memory operations are issued from the issue queues in program order.
ALU-out / MEM-out	ALU and memory operations are issued out-of-order with speculation enabled.

Table 5.4: Benchmarks

SPEC SUITE	art	swim	mcf	parser	perlbmk
	twolf	vortex	vpr		

in Table 5.3. The cores for these three different configurations are identical and only differ in their respective issue logics. 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 out-of-order issue of ALU operations but mandates issuing of memory instructions from the load and store queues in strict program order. The ALU-out/MEM-out configuration allows the issue of both ALU and memory operations out-of-order, which is representative of current hardware.

5.2.3 Benchmarks

For our preliminary baseline study, we use a subset of SPEC2000 integer and floating point [29] benchmarks as shown in Table 5.4. Each benchmark was allowed to warm up and perform its initialization routines before statistics and data were gathered. The SPEC benchmarks were acquired from the SimpleScalar developers [71] and were warmed up by fast-forwarding the first 250 million instructions and then data was gathered over the next 500 million instructions. The benchmarks all operate on their reference input sets.

5.3 Effects of Increased Out-of-Order Aggressiveness

5.3.1 Replay Traps

To illustrate the effects of increased out-of-order aggressiveness on replay traps, Figure 5.1 shows the replay trap frequency and the replay trap overhead averaged for the different

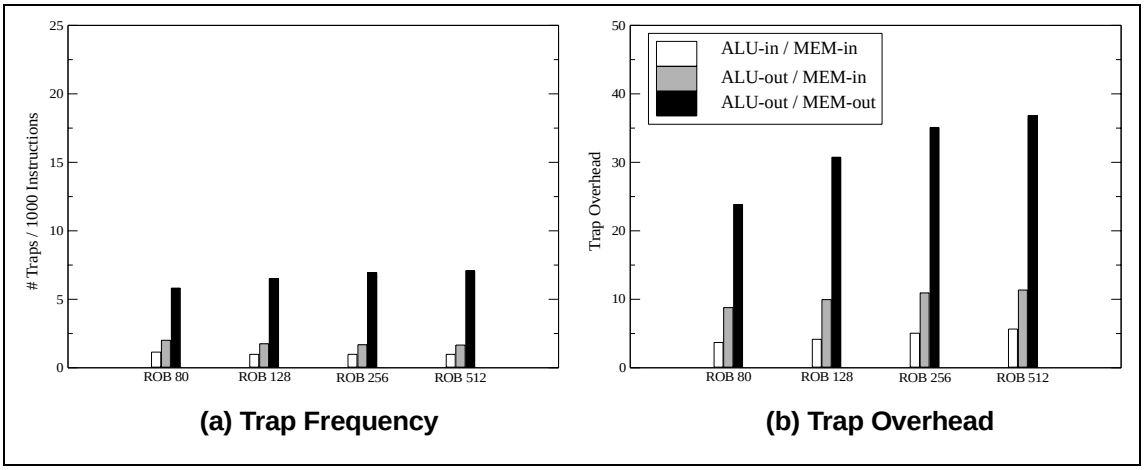


Figure 5.1: Reorder Traps. (a) Trap Rate— Average Number of Instructions Executed Between Traps (b) Trap Overhead—Total Amount of Execution Lost Due to Traps Trends show that increase in out-of-order aggressiveness by increasing reorder buffer sizes increases the trap rate and trap overhead. For an ALU-out/MEM-out 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.

benchmarks. In both graphs, the x-axis represents the four different reorder buffer sizes and the y-axis represents the trap frequency and trap overhead. For each reorder buffer size we present three bar graphs representing the three different issue-logic configurations. The first bar graph represents the ALU-in/MEM-in configuration, the next bar represents the ALU-out/MEM-in configuration, and finally the last bar represents the ALU-out/MEM-out configuration.

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 reorder buffer sizes), it exposes traps as an important source of overhead. This is due to the mechanisms of handling traps—i.e. flushing the pipeline and restarting from the faulting memory instruction. Larger 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 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 is usually higher due to functional unit and cache access/miss latency. The bottom line: it is imperative that the frequency of traps be low on systems with larger instruction windows.

Contrary to the desire for less frequent replay traps, by moving from an in-order, ALU-in/MEM-in, core to an aggressive out-of-order core, ALU-out/MEM-out (black bars), we note a factor of 8-9 increase in trap rate, causing an application to waste on average 15-30% of its total 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.

The mere difference between the ALU-out/MEM-in and ALU-out/MEM-out configuration shows that the reordering of memory instructions, due to speculative execution, causes significant overhead in the system. This suggests the need for modern out-of-order processors to 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.

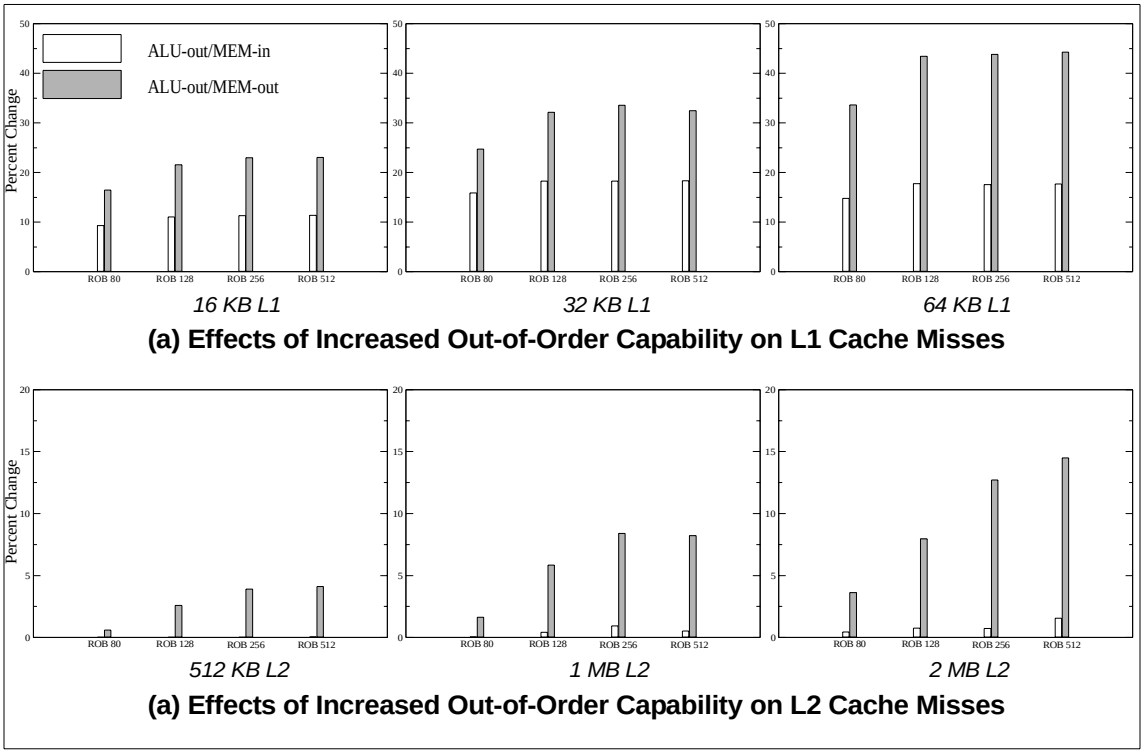


Figure 5.2: Effects of OoO on Cache Performance. The figures shows the increase in the L1 and L2 cache misses for the ALU-out/MEM-in and ALU-out/MEM-out configurations normalized to the ALU-in/MEM-in configuration. The graphs show that out-of-order execution of both memory and ALU operations can hurt cache performance by up to 40% in the L1 cache and 20% in the L2 cache. Additionally, we observe that restricting the memory operations to be issued in-order eliminates more than 50% of the cache misses.

5.3.2 Cache Performance

To measure the cache performance of the memory subsystem, we measure the total number of cache misses in the level one and level two data cache. Note that a cache miss is defined as miss in the cache as well as a miss in the MSHRs. Figure 5.2 illustrates the 16KB, 32KB, and 64KB L1 and L2 cache misses normalized to the ALU-in/MEM-in configuration. The data is averaged across all benchmarks used in this study. The graphs display the different reorder buffer sizes on the x-axis and the percent change in cache misses with respect to the ALU-in/MEM-in configuration on the y-axis. For each reorder

buffer size, the percent change for the ALU-out/MEM-in configuration is represented by the first bar and the ALU-out/MEM-out by the second bar.

From the figure we observe that applications can observe a degradation in cache performance due to the out-of-order issue of instructions. On average we observe a 40% increase in the total number of cache misses with an ALU-out/MEM-out configuration, with some individual benchmarks showing miss rate increases by as much as 20% when compared to a total in-order system. For these applications we also observe that the worst of the performance degradation is due to the reordering of memory instructions. This observation is based on the fact that moving from an ALU-out/MEM-in system to an ALU-out/MEM-out configuration causes an increase in the total number of cache misses when compared to ALU-in/MEM-in by a factor of 2 to 3. Thus, for these applications we observe that an increase in the reordering of memory instructions and speculation causes significant performance degradation in the memory subsystem.

Similarly, Figure 5.2(b) illustrates the 512KB, 1MB, and 2MB L2 cache misses normalized to the ALU-in/MEM-in configuration. From the figure we observe an increase in the total number of L2 cache misses by 2% in the 512KB L2 cache, 7% in the 1MB L2 cache, and 15% in the 2MB L2 caches. On a per-benchmark basis we observe increase in cache misses by as much as 20% in art and 35% in twolf. In the case of the L2 caches, we observe that restricting the processor to issue memory instructions in-order reduces the total number of L2 cache misses by a factor of 10 or more. Since the difference between the ALU-out/MEM-in and ALU-out/MEM-out configurations is a higher degree of speculation and the out-of-order issue of memory instructions, the increase in the number of cache misses can be associated with the reordering of memory instructions that are speculatively

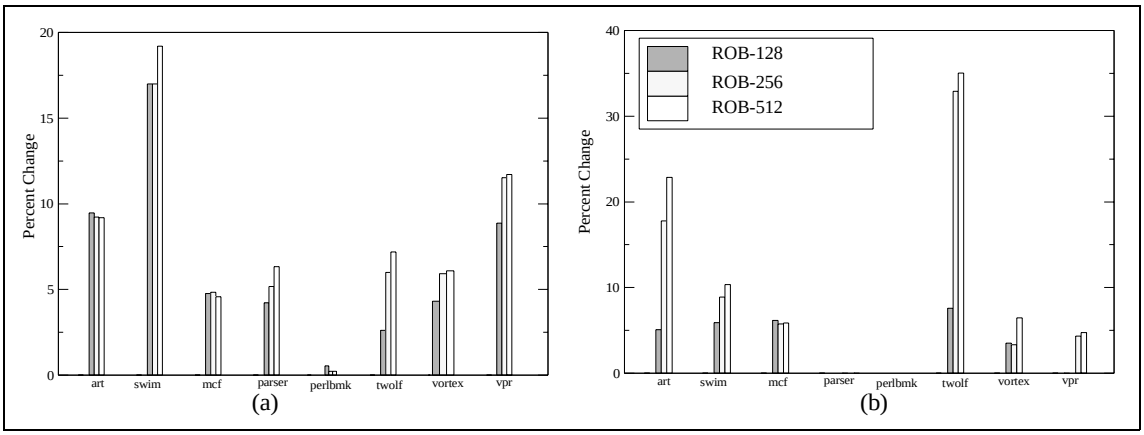


Figure 5.3: Effects of OoO on Cache Misses. Increasing the out-of-order capability of a processor can cause (a) increase in L1 cache misses (b) increase in L2 cache misses.

executed. Even though speculative execution of memory instructions is classified as yet another type of data-prefetching [47, 49], speculative memory accesses can evict data required by younger or older non-speculative memory accesses.

From the figure, we also observe that the performance degradation in the caches increase with larger cache sizes. For example, when compared to an in-order system, we observe on average a 20%, 30% and 40% cache performance degradation in the 16KB, 32KB, and 64KB cache. Similarly, we observe on average a 2%, 7%, and 15% cache performance degradation in the 512KB, 1MB, and 2MB L2 cache respectively. This can be explained by the fact that larger data caches allow for better speculation than smaller data caches due to the high hit-rates in large data caches. With larger data caches, an out-of-order system has more opportunity to progress deep into an application's instruction stream while with smaller data cache the processor can spend most of the time stalling for data from lower levels of memory to arrive.

To understand the effects of increased out-of-order aggressiveness on cache performance, we now plot in Figure 5.3 the increase in the number of cache misses in the 64KB L1 and 2MB L2 caches on a per-benchmark basis. For each benchmark we present

the increase in the number of cache misses normalized to a processor with an 80-entry reorder buffer. The first bar represents the increase with respect to a 128-entry ROB, the second bar represents the increase with respect to a 256-entry ROB, and the last bar represents the increase with respect to a 512-entry ROB.

Based on the figure, we see that increasing the reorder buffer size from 80 to 512 negatively impacts application cache locality by increasing the total number of cache misses by 5–20% in the L1 cache and by 5-35% in the L2 cache. Based on this data we conclude that eliminating sources of stalls by increasing the out-of-order capability causes additional overhead in the memory subsystem. We observe that the very mechanisms commonly used to tolerate long latencies associated with memory can themselves cause degradation in the memory subsystem. Such degradation not only leads to a performance degradation but also dissipates energy in doing unnecessary work. Thus, based on this data, we conclude that it is imperative that the negative effects in the memory subsystem be reduced with increasing out-of-order aggressiveness.

5.4 Summary

This chapter reveals two important findings. First, increasing out-of-order capability conflicts with the memory ordering requirements of a processor causing the processor to incur frequent memory replay traps. Second, the speculative, out-of-order issue of memory instructions tends to increase the number of cache misses in the L1 and L2 caches. By using different issue-logic configurations, in the presence of branch prediction, we identify that both these negative effects are due to the reordering of memory instructions. By issuing

memory instructions in-order we observe that the negative effects in the memory subsystem are drastically reduced. Since recent research and industry trends are focusing on increasing out-of-order capability, it is imperative that the frequency of traps and the number of cache misses be reduced so that future high performance processors can realize the full potential of more complex out-of-order designs. Thus, these findings motivate the understanding of how memory instructions are reordered in aggressive out-of-order core systems. The next chapter introduces a metric to measure the reordering of memory instructions.

CHAPTER 6 *Disorder of Memory Instructions*

The transition from an ALU-out/MEM-in configuration to an ALU-out/MEM-out configuration increased the trap frequency by more than a factor of 2 and the total number of cache misses by more than 25%. Since the difference between the two processor configuration merely involves controlling the order in which memory instructions are issued, it is highly likely that the degradation in performance is primarily due to the reordering of memory instructions. Thus, we propose a methodology to measure and understand the degree by which memory instructions are issued out-of-order. In doing so, we can quantitatively understand the reasons for the degradation in the memory subsystem. We define a new metric called *disorder* to measure the degree by which dynamic instruction schedulers issue memory instructions.

6.1 Defining Disorder

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. We define *disorder* as the degree by which an instruction is issued out-of-order. We classify disorder into two types: *global disorder* and *local disorder*. Disorder can be measured for any type of instruction. Since the performance

degradation in the memory subsystem is primarily due to the reordering of memory instructions, we only measure the disorder of memory instructions.

To measure disorder, at the time of instruction decode, each memory instruction is assigned a sequential ID. The first memory reference is assigned sequential ID one, the next memory instruction is assigned sequential ID two, and so on. In the event 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. Thus, disorder for a load instruction is measured when it is issued to the caches to read data, and the disorder for a store is computed when it is issued to the cache system at commit time.

6.1.1 Global Disorder

Global disorder is the degree by which a memory instruction is issued out-of-order with respect to actual program order. Global 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 6.1 illustrates an example on computing global disorder. The figure shows in cycle 101 memory instructions 1 and 3 issued to the cache memory system. If the system were in-order, then memory instructions 1 and 2 would have been issued instead. Thus, the global disorder of memory instruction 1 is 0 (1-1) and the global disorder of memory instruction 3 is 1 (3-2). A global disorder value of zero indicates that the memory instruction was issued on time, a disorder value less than zero indicates that the memory instruction was delayed, and a disorder value

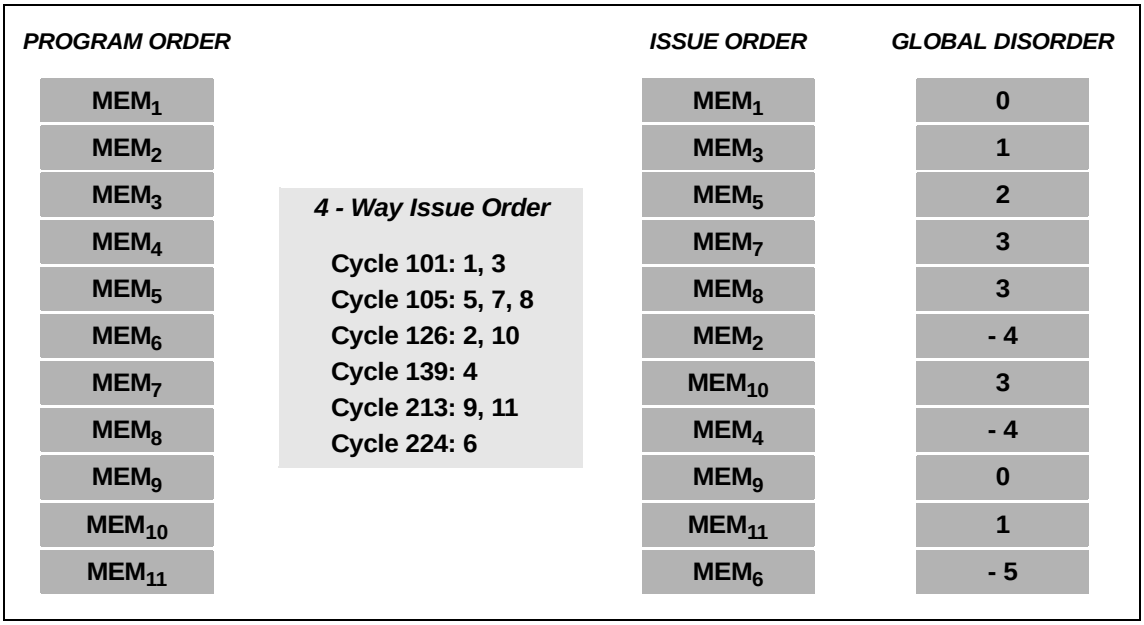


Figure 6.1: Global Disorder. he degree to which a memory instruction is issued out-of-order with respect to actual program order. The disorder is computed by computing the difference between a memory instruction issued and the memory instruction that should have been issued.

greater than zero indicates that the memory instruction was issued earlier than it would have were it an in-order processor.

We point out here that out-of-order execution is not the only source of global disorder. Modern microprocessors issue load and store instructions to the cache system out of program order with respect to each other. 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 ensure that only non-speculative writes are sent 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 before the store instruction, hence inherently causing disorder in the system.

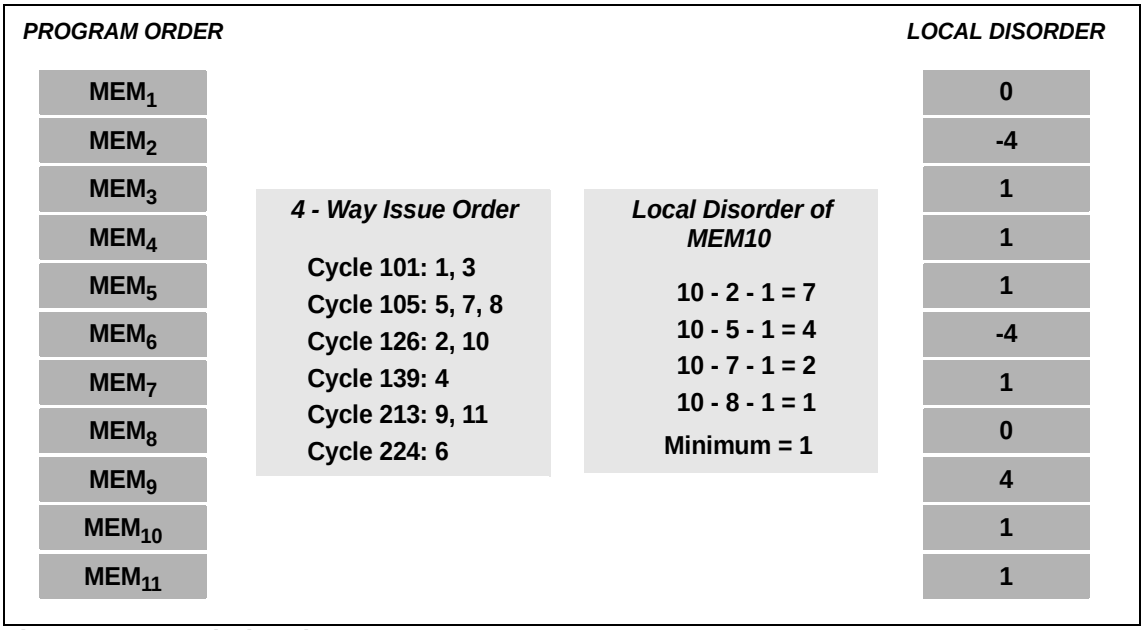


Figure 6.2: Local Disorder. The degree by which an instruction is issued out-of-order as compared to other instructions issued in the same and previous cycle. The disorder is computed by extracting the minimum difference between a memory instruction and other memory instructions issued in the same and previous cycle.

6.1.2 Local Disorder

Local disorder is the degree by which a memory instruction is issued out-of-order with respect to recently issued memory instructions. The intuition behind global disorder was to measure disorder from a "program order" perspective. The intuition behind local disorder is to measure how a memory instruction issues with respect to other memory instructions issued in the previous cycle and younger memory instructions issued in the same cycle. Figure 6.2 provides an example on computing local disorder of memory instruction number 10. Memory instruction 10 is issued by the processor in cycle 126 along with memory instruction number 2. In the recent past the processor issued memory instructions in cycle 105, and the memory instructions issued were 5, 7, and 8. To compute local disorder, we first compute the distance in instruction stream (in terms of memory instructions) 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). Furthermore, to be consistent with the definition of global disorder, we subtract 1 for uniformity between the two disorder metrics. This was done because global disorder defined in-order issue as a value of zero, thus to ensure that back to back issue of 7, 8 in-order provides local disorder of zero, we needed to subtract 1. Thus, the local disorder of 8 would be 8 - 7 - 1 = 0.

After computing individual disorders of memory instruction 10 with other memory instruction issued in the previous cycle and younger memory instructions in the same cycle, local disorder is defined to be the minimum of the magnitudes of all the computed disorders. Since the disorder values of memory instruction 10 with memory instructions 2, 5, 7, and 8 are 7, 4, 2, and 1 respectively, the minimum of all computed disorders is 1, thus making the local disorder of memory instruction 10 to be 1. Similarly, for memory instruction 9, the instructions issued in the same cycle is 11 and in the previous cycle is 4. Since memory instruction 11 is older than memory instruction 9, it is omitted. Thus, the local disorder of memory instruction 9 is only computed with 4, hence the local disorder of memory instruction 9 is 9-4-1 which equals 4.

Local disorder is a measure of how a processor issues memory instructions compared to other memory instructions, i.e. if a processor issues memory instruction M in a given cycle, how far apart in the instruction stream are other memory instructions that are issued in the current and following cycle. It is used to determine whether memory instructions closer to or further from M issue in the same or next 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 we wanted to be able to capture how well a processor issues

memory instructions in-order. By keeping track of the minimum of the computed disorder values, the minimum value provides intuition of how far apart in the instruction stream other memory instructions reside. By doing so, we are able to quantitatively capture in-order streams of issue embedded in the out-of-order stream. A local disorder of 0 indicates that there exists one memory instruction in the same or previous cycle that immediately precedes the relevant memory instruction. Local disorders other than 0 indicate the degree by which memory instructions are separated from others in the sequential instruction stream.

6.1.3 Why Measure Disorder?

Having introduced global and local disorder, we now briefly discuss what we expect to learn by experimentally measuring the degree of disorder. From the previous chapter, we concluded that the negative effects in the memory subsystem are primarily due to the reordering of memory instructions. By defining a methodology to measure both global and local disorder, our first attempt at understanding the source of the problem is to measure the total amount of disorder prevalent in the system. Indeed, the out-of-order issue logic will reorder the issue of memory instructions; we are looking to quantitatively measure by how much memory instructions are reordered. This initial study will provide us with an intuition on the amount of disorder that can prove to be harmful in the memory system. Once we have identified the upper limit on the amount of disorder that proves to be harmful, our next approach would be to fine tune the amount of disorder by mechanisms that restrict the reordering of memory instructions. In doing so, we can then determine the threshold below which disorder is useful and beyond which disorder is harmful.

Before we propose mechanisms to reduce disorder, based on the above motivation, we now describe our experimental approach on measuring the amount of disorder present in aggressive out-of-order systems

6.2 Experimental Measurements of Disorder

6.2.1 Disorder Study Simulator Parameters

To measure disorder we vary the aggressiveness of the out-of-order core by scaling both the issue widths as well as the reorder buffer sizes. Using the same simulator parameters as described in the previous chapter, we vary out-of-order aggressiveness by scaling issue widths and reorder buffer sizes. We scale the issue widths from a 2-way issue system to an aggressive 32-way issue system, and the reorder buffer sizes from 80 entries to 512 entries as shown in Table 6.1. We varied both issue widths and reorder buffer sizes so as to quantitatively measure the effects of both issue-widths and re-order buffer sizes on the re-ordering of memory instructions. In doing so, we hope to determine the parameter that contributes to the maximum disorder.

To measure the impact of out-of-order execution on disorder we re-use the three issue-logic configurations ALU-in/MEM-in, ALU-out/MEM-in, and ALU-out/MEM-out as described in Table 6.2. We remind the reader that the cores for these three different configurations are identical and only differ in their respective issue logic. The ALU-in/MEM-in configuration mandates that both ALU and memory operations be issued in

Table 6.1: Simulator Configurations

	Configuration Name	ROB Size	Issue Width INT/FP	IssueQ Size INT/FP	# Functional Units**	LQ/SQ Size	# Renaming Registers INT/FP
a)	Alpha 21264 - 2way	80	2/1 Way	20/15	4/4/1/1	32/32	41/41
b)	Alpha 21264 x 1	80	4/2 Way	20/15	4/4/1/1	32/32	41/41
c)		80	8/4 Way	20/15	4/4/1/1	32/32	41/41
d)		80	16/8 Way	20/15	4/4/1/1	32/32	41/41
e)		80	32/16 Way	20/15	4/4/1/1	32/32	41/41
f)	Alpha 21264 x 2	128	8/4 Way	40/30	8/8/2/2	64/64	82/82
g)		128	16/8 Way	40/30	8/8/2/2	64/64	82/82
h)		128	32/16 Way	40/30	8/8/2/2	64/64	82/82
i)	Alpha 21264 x 4	256	16/8 Way	80/60	16/16/4/4	128/128	164/164
j)		256	32/16 Way	80/60	16/16/4/4	128/128	164/164
k)	Alpha 21264 x 8	512	32/16 Way	160/120	32/32/8/8	256/256	328/328

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

Table 6.2: Issue Logic 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 in-order issue.
ALU-out / MEM-in	ALU operations are issued out-of-order. Memory operations are issued from the issue queues in program order.
ALU-out / MEM-out	ALU and memory operations are issued out-of-order with speculation enabled.
ALU-out/MEM-out/PerfL2	ALU and memory operations are issued out-of-order, L2 cache is perfect.
ALU-out/MEM-out/PerfL1	ALU and memory operations are issued out-of-order, L1 cache is perfect.

strict program order. The ALU-out/MEM-in configuration allows the issue of ALU operations out-of-order but mandates issuing of memory instructions in strict program order. The ALU-out/MEM-out configuration allows the issue of both ALU and memory operations out-of-order. Additionally, to determine the primary source of disorder, in addition to these issue logic configurations we also model the ALU-out/MEM-out/PerfL2 and ALU-out/MEM-out/PerfL1 processor configurations. The ALU-out/MEM-out/perfL2 and ALU-out/MEM-out/perfL1 configurations are identical to the ALU-out/MEM-out

configuration and only differ in the configuration of the caches. The ALU-out/MEM-out/PerfL2 models a perfect L2 data cache while the ALU-out/MEM-out/PerfL1 models a perfect L1 cache. To clarify, a perfect cache is identified as a cache configuration where all accesses to the cache provide a 100% hit-rate.

Based on the above experimental methodology, we now present data on the existence and measurement of global and local disorder in a system.

6.2.2 Global Disorder Results

Disorder, as mentioned earlier, is the degree by which a memory instruction is issued out of program order. To verify the existence of global disorder, Figure 6.3 illustrates the global disorder measurements for the application SWIM on ten different configurations of the processor. Additionally, Figure 6.4 illustrates pictorially the trends in global disorder with different reorder buffer sizes. The x-axis represents the disorder of a memory instruction, and the y-axis represents the percent of overall memory instructions exhibiting the disorder.

Figure 6.3 and 6.4 show that out-of-order execution can create significant disorder with respect to actual program order. We observe that roughly half of the memory 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 could not be issued). The wide variation in disorder is most likely due to memory references missing in the data caches or functional unit latency. The low disorders primarily due to misses in the L1 cache (L2 hit latency 15 cycles) or functional

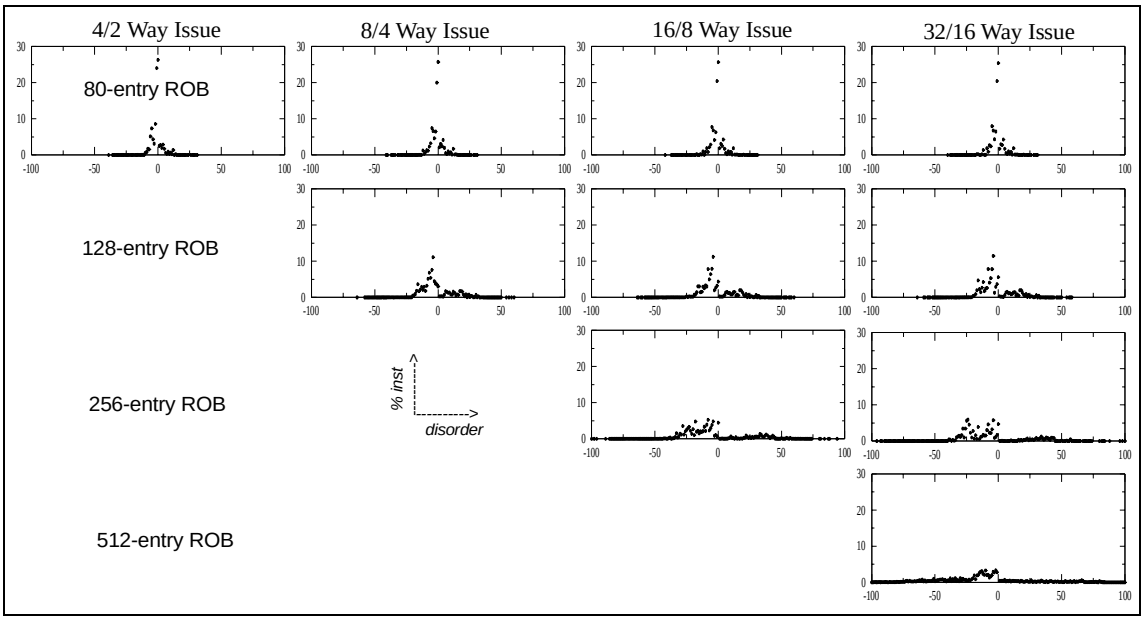


Figure 6.3: Illustration of Global Disorder. The figure shows the global disorder for the application SWIM for increasing issue widths (left to right horizontally) and increasing ROB sizes (top to down vertically). The x-axis represents the disorder, and the y-axis represents the percent of instructions 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.

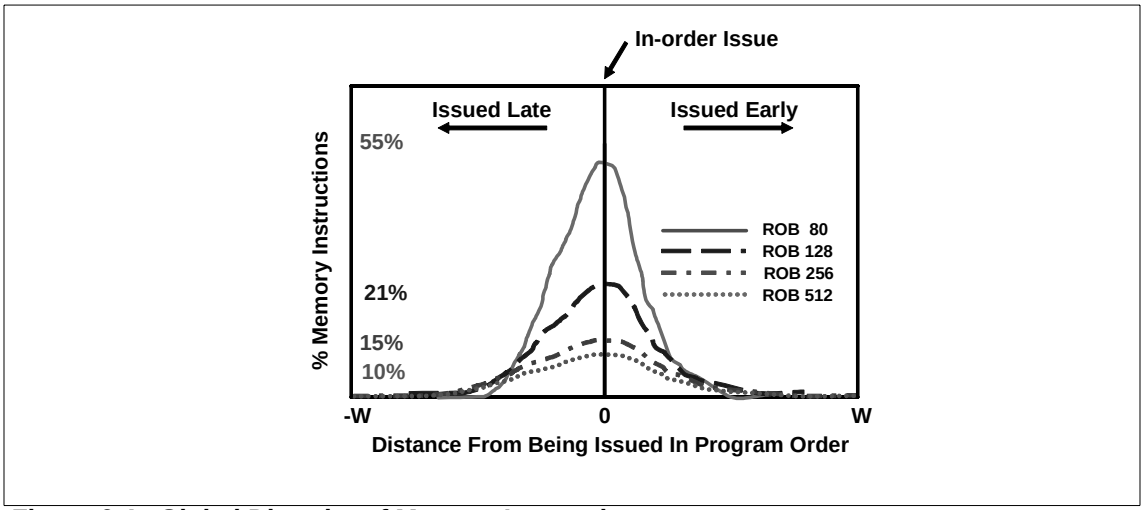


Figure 6.4: Global Disorder of Memory Instructions. The figure illustrates how memory instructions are reordered in an out-of-order issue machine: they are either issued late, on time, or early.

unit latency while the extreme disorders are due to misses in the L2 cache, i.e. due to DRAM latency. We observe 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 observe that the number of memory instructions issued on time (global

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 10% (3-4% for swim). The graphs also show that increasing window sizes correlate with increasing global disorder (10-25%), where as increasing issue widths change global disorder by only a few percent (2-4%). This can be explained by the fact that with increasing issue widths and constant reorder buffer sizes, the window of instructions available to the processor stays constant. If the processor is unable to issue from the window, then, irrespective of the issue width, instructions cannot be issued until older instructions retire, eventually causing the reorder buffer to become full and the processor to stall. On the other hand, increasing the window size provides the processor a much larger choice of instructions to issue from, hence causing an increase in total global disorder.

To better understand disorder, in Figure 6.5 we compare the average global disorders for our five different processor configurations of the Alpha 21264: ALU-in/MEM-in, ALU-out/MEM-in, ALU-out/MEM-out, ALU-out/MEM-out/perfL2, and finally ALU-out/MEM-out/perfL1. For all the benchmarks, the figure illustrates increased out-of-order capability on the x-axis and the average global disorder on the y-axis.

Global disorder is a measure of the degree by which memory instructions are reordered when compared to in-order execution. In other words, the re-ordered listing of memory instructions executed is merely a permutation of the ordered sequential listing of the same execution stream. Mathematically, the average global disorder must be zero because the sum of the differences of the re-ordered and in-order list is zero. Graphs in Figure 6.5, however, do not reflect this. We wish to emphasize that this is not a mathematical or simulation inaccuracy. There are two reasons behind this. First, the fact that existing systems have a

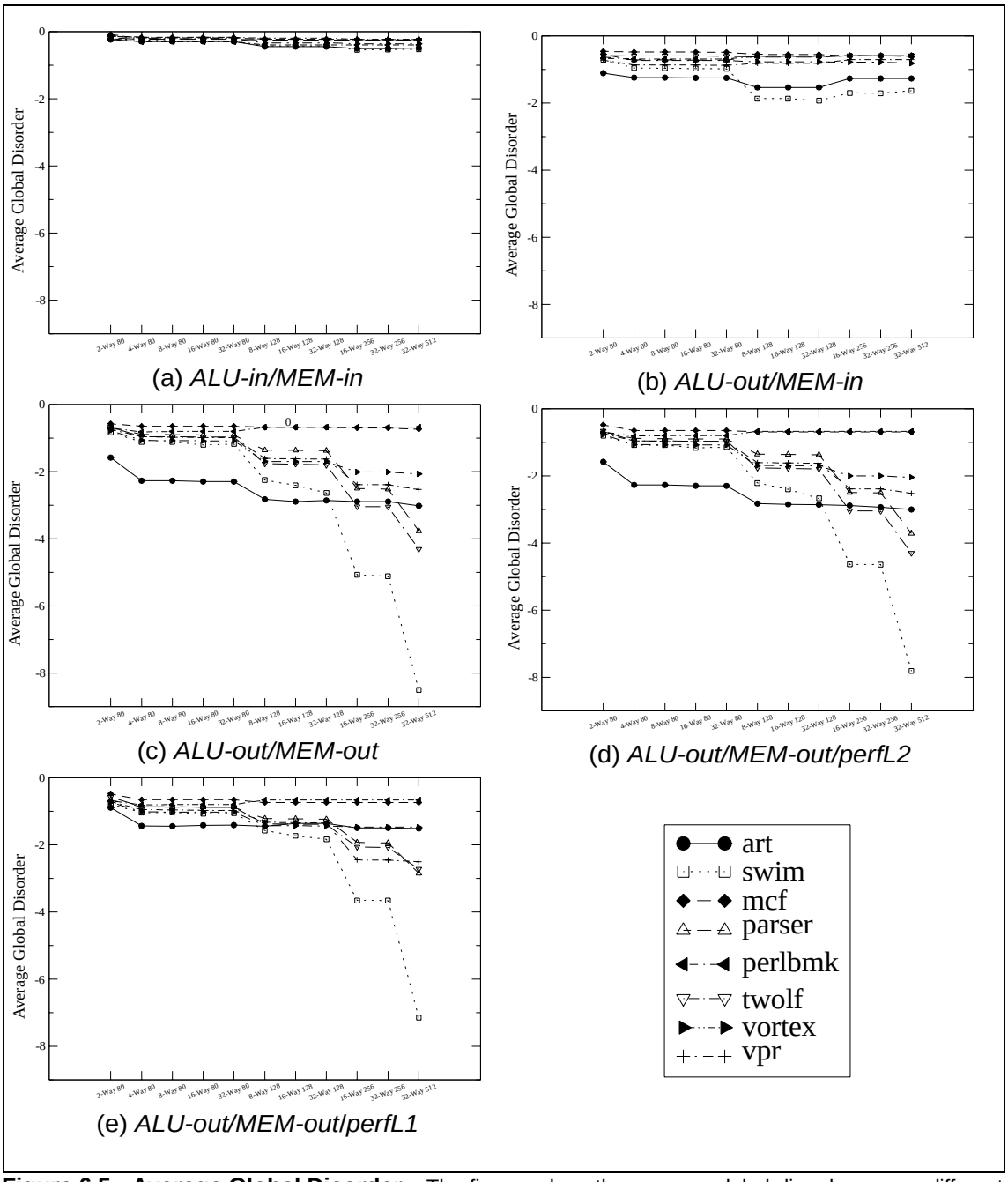


Figure 6.5: Average Global Disorder. The figures show the average global disorder across different issue-logic and memory system configurations. A general trend shows an increase in disorder with increase in out-of-order aggressiveness.

built-in mechanism that causes loads to be issued earlier than stores. As mentioned earlier, this is because loads and stores access the memory subsystem in different pipeline stages after being issued. Thus, this can cause the average global disorder to deviate from the intuitive average value of zero. Second, the non-zero global disorder is also because the

calculation of average global disorder includes the global disorders of ALL memory operations executed; those that were committed as well as those that were executed speculatively. In doing so, we are able to capture the overall behavior of memory instructions executed in the system.

For all benchmarks, the graphs in Figure 6.5 show an average global disorder that is negative. Recalling our definition of global disorder, a disorder value less than zero implies that memory instructions are being issued later than they should have, i.e. newer memory instructions are issued before older memory instructions are issued to the memory system. This implies that on average, the processor usually delays the issue of a memory instruction to the memory system most likely due to dependencies on older long latency operations such as functional unit latency or cache miss latency.

Additionally, from the graphs in Figure 6.5, we note disorder values other than zero for processor configurations that issue memory operations in strict program order (ALU-in/MEM-in and ALU-out/MEM-in). Based on our definition of MEM-in, this should not be the case because every memory operation is issued in strict program order. The disorder values for these systems is not an error, but are due to the design of modern high performance microprocessors. Such processors wait till the commit stage of the pipeline for store instructions to access the data cache. This is because store instructions can only write data to the cache if they are non-speculative. Thus, even though store instructions are issued in-order, the store waits in the reorder buffer until commit time. Meanwhile, other load instructions are issued and access the memory system before the store does, thus causing global disorder. With in-order execution, if a bulk of the instructions awaiting commit in the reorder buffer are store instructions, the disorder can become more negative. This is because

newer load instructions access the data cache before the older store instructions. For exactly this scenario, in the configurations where memory instructions are issued in-order, we observe an increase in disorder (a decrease in the value of global disorder) with increasing issue widths and reorder buffer sizes. However, we observe that such disorder is negligible (on average it is less than 0.3), (see Figure 6.5(a)), implying that stores do not wait too long in the ROB to be committed.

Increasing the issue widths and reorder buffer sizes introduced some disorder into an already in-order system. By allowing just the ALU instructions to be issued out-of-order and still maintain the in-order issue of memory instructions, i.e. the ALU-out/MEM-in configuration we observe a factor of 2-4 increase in total global disorder when increasing issue widths and reorder buffer sizes. This can be explained by the fact that the ALU-out/MEM-in configuration exploits ILP amongst ALU instructions by speculatively executing independent ALU instructions. Since memory instructions are scheduled in order from the issue queue, the issue logic configuration can issue load and store instructions simultaneously or in clusters. Since load and store instructions access the data cache in different stages of the pipeline, the issuing of multiple loads and stores in clusters can increase the global disorder. For a system that issues memory instructions in order, we observe that average global disorder is essentially a measure of the average number of loads issued to the memory subsystem prior to a store instruction is committed.

We now analyze the effects of allowing both ALU and memory instructions to issue out-of-order. Figure 6.5(c) shows the impact on global disorder for the ALU-out/MEM-out processor configuration. Comparing Figure 6.5(b) and Figure 6.5(c), we observe a factor of 2-16 increase in global disorder by allowing memory instructions to be issued out-of-order.

On average, we observe that the issue of memory instructions to the memory subsystem can be delayed by as much as 8 or more memory instructions. Furthermore, Figure 6.5(c) shows that increasing out-of-order aggressiveness by means of increasing issue widths and reorder buffer sizes (moving from left to right on the x-axis) can increase the global disorder by a factor of 2 or more. Systems with a 32-way 512-entry reorder buffer have on average a global disorder value that is 6-8 orders of magnitude higher than a system with a 2-way 80-entry reorder buffer. Thus, from this data we quantitatively observe that the more the out-of-order aggressiveness, the more the global disorder in the system.

In general, we observe that the trends of increasing out-of-order aggressiveness causes the average global disorder to become more negative. The fact that the average global disorder value becomes more negative implies that memory instructions are being reordered and that they are being issued late to the memory subsystem. Since the average disorder is negative and not positive, what this also means is that the processor is speculatively executing memory instructions. Thus, we point out that average global disorder is not only a measure of the degree by which memory instructions are issued out-of-order, but it is also a measure of speculative execution of memory instructions. In general, if the processor is doing a lot of speculation—which is the case with large window sizes and large issue widths—the more negative the global disorder.

We know that an application's global disorder is primarily due to misses in the caches or functional unit latency. To determine the source of global disorder, we also present the average global disorder for the ALU-out/MEM-out/perfL1 and ALU-out/MEM-out/perfL2 configurations. Graphs in Figure 6.5(d) and Figure 6.5(e) show that a perfect L2 cache has little or no effect on the value of average global disorder. However, we observe that with a

perfect L1 cache system, global disorder values are significantly less than the ALU-out/MEM-out configuration and are more comparable to the ALU-out/MEM-in configuration. Based on these findings, we observe that global disorder is primarily due to latencies associated with older memory instructions that miss in the L1 cache. This implies that global disorder is due to memory instructions that are indirectly waiting on the result of a memory instruction that misses in the primary data cache. In the meantime, younger memory instructions independent of the missing memory instruction issue to the memory system speculatively, hence creating global disorder.

Furthermore, we also note that even with a perfect L1 cache configuration, the average global disorder is still a factor of 1-8 worse than that of a core that issues all operations in order (ALU-in/MEM-in) or one that issues just memory operations in order (ALU-out/MEM-in). From this behavior, we gather further insight in that global disorder is not only due to cache misses but also due to latencies associated with functional units and producer consumer relationships.

From the data presented in this section, we have quantitatively illustrated the existence of global disorder in a system with out-of-order capability. We showed that increasing out-of-order capability by increasing issue-widths and reorder buffer sizes causes an increase in global disorder in the system. In general, we observe that global disorder is negative, illustrating that memory instructions are usually issued late to the memory subsystem. We also quantified that maximal re-ordering of memory instructions is primarily due to misses in the L1 data cache. We will illustrate later in this dissertation that it is due to the reordering of memory instructions that we observe the unexpected side effects in the memory subsystem.

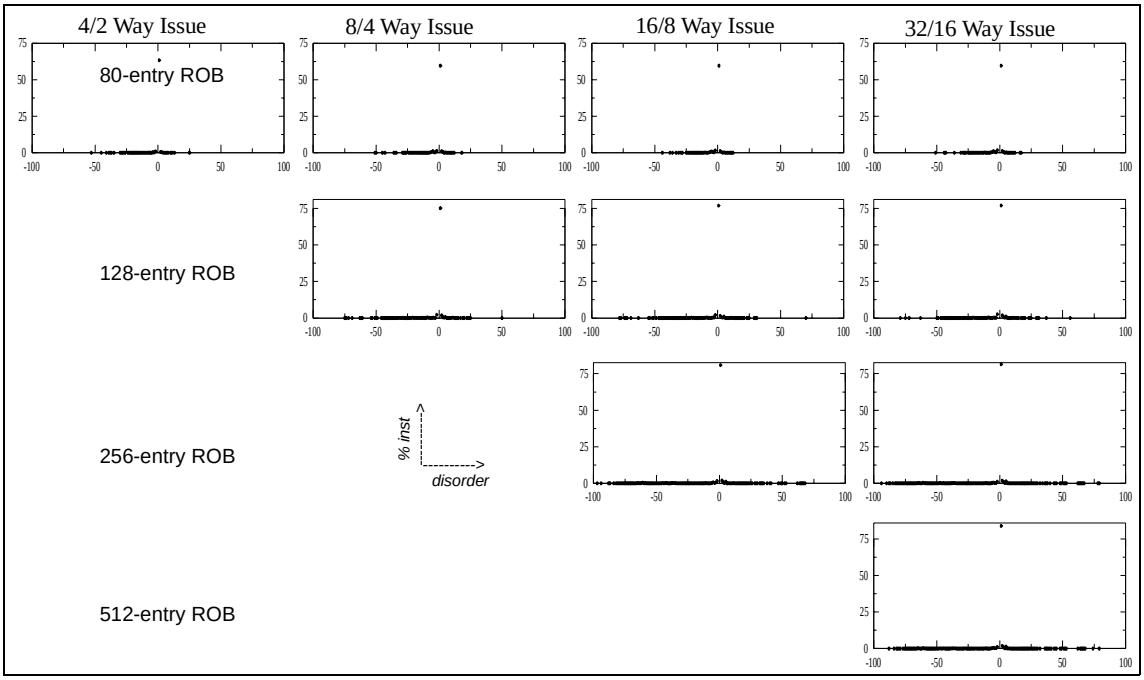


Figure 6.6: Illustration of Local Disorder. The figure shows the local disorder for the application SWIM for increasing issue widths (left to right horizontally) and increasing ROB sizes (top to down vertically). We see that the bulk of the memory instructions issued to the memory system have a local 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.

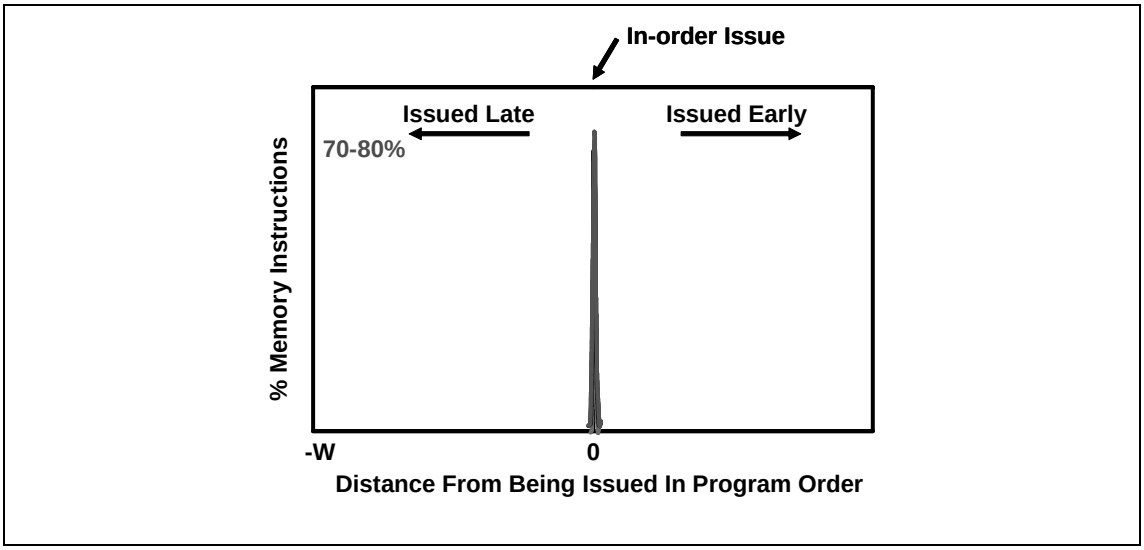


Figure 6.7: Local Disorder of Memory Instructions. The figure illustrates the behavior of local disorder as the reorder buffer size is increased. In general, memory instructions have small local disorder with increased reorder buffer sizes.

6.2.3 Local Disorder Results

Global disorder illustrated the degree of out-of-order execution of memory instructions

from a program-order perspective. We now analyze an applications local disorder, i.e. the disorder with respect to other memory operations issued in the same and previous cycle. Based on data from the earlier section, we observed significant global disorder—about 30-50% of memory instructions are issued on time. Figure 6.6 shows the local disorder for the memory application SWIM and is representative of other benchmarks. Based on the data, we observe that applications have extremely low local disorder—on average 70-80% of memory instructions have local disorder 0. This implies that memory instructions issued to the memory subsystem are usually in close proximity to each other i.e. memory instructions issued are usually from the same basic block or region of code. Since in most cases local disorder is 0, this implies that when memory instructions are issued to the memory subsystem they are usually issued back to back in-order. For example, for two memory instructions X, and X+1, if X is issued in a particular cycle, with high probability X+1 will also be issued either in the same cycle or the following cycle. Based on our data, we find that in general 70-80% of memory instructions follow this trend of issue. This implies that when executing instructions out-of-order, memory instructions that are close to each other are more likely to be scheduled together i.e. there seems to exist a “spatial locality” with respect to issuing memory instructions. From this behavior, we can gather that when a processor speculates in a particular region of code, it spends time issuing memory instructions in the same region of code rather than moving back and forth between older and younger memory instructions. Thus, we can conclude that when a processor speculates, it continues to speculate and delays the issue of older memory instructions to the memory subsystem.

A closer look at the data shows that in general increasing ROB sizes causes the local disorder to decrease, i.e. a system with an 80-entry ROB has 68% of its memory instructions

issued with local disorder 0, while in a 512-entry ROB we observe as much as 85% of its memory instructions issue with local disorder 0. This can be explained by two reasons. First, increasing ROB sizes causes the processor to speculate deep into an applications instruction stream causing memory instructions from the same program region to be issued back to back. Secondly, as we will show later, the in-order stream of issue in the out-of-order stream is also due to one of the negative effects of increasing out-of-order capability. We observe that increasing ROB sizes cause a significant increase in replay traps causing memory instructions to be re-issued and re-executed in the correct order. The fact that memory instructions are re-issued and re-executed causes them to be issued closer to each other, hence causing the local disorder to be low.

To better understand the trends of local disorder, Figure 6.8 presents the average local disorder for the five different processor configurations. As expected, we observe that the average local disorder for memory instructions issued in-order for the ALU-in/MEM-in and ALU-out/MEM-in configurations is relatively close to zero. Furthermore, of all the benchmarks, we observe that the average local disorder of art, parser, and twolf increases with increase in out-of-order aggressiveness. This implies that the memory instructions of these benchmarks have small latencies either due to functional units or misses in the L1 data cache. Thus, during these small latencies the out-of-order issue logic speculates for a little while and then goes back to executing older memory instructions. Thus, the increase in local disorder with increase in out-of-order aggressiveness can be explained by the out-of-order core going back and forth in the issue of memory instructions between different basic blocks or regions of code.

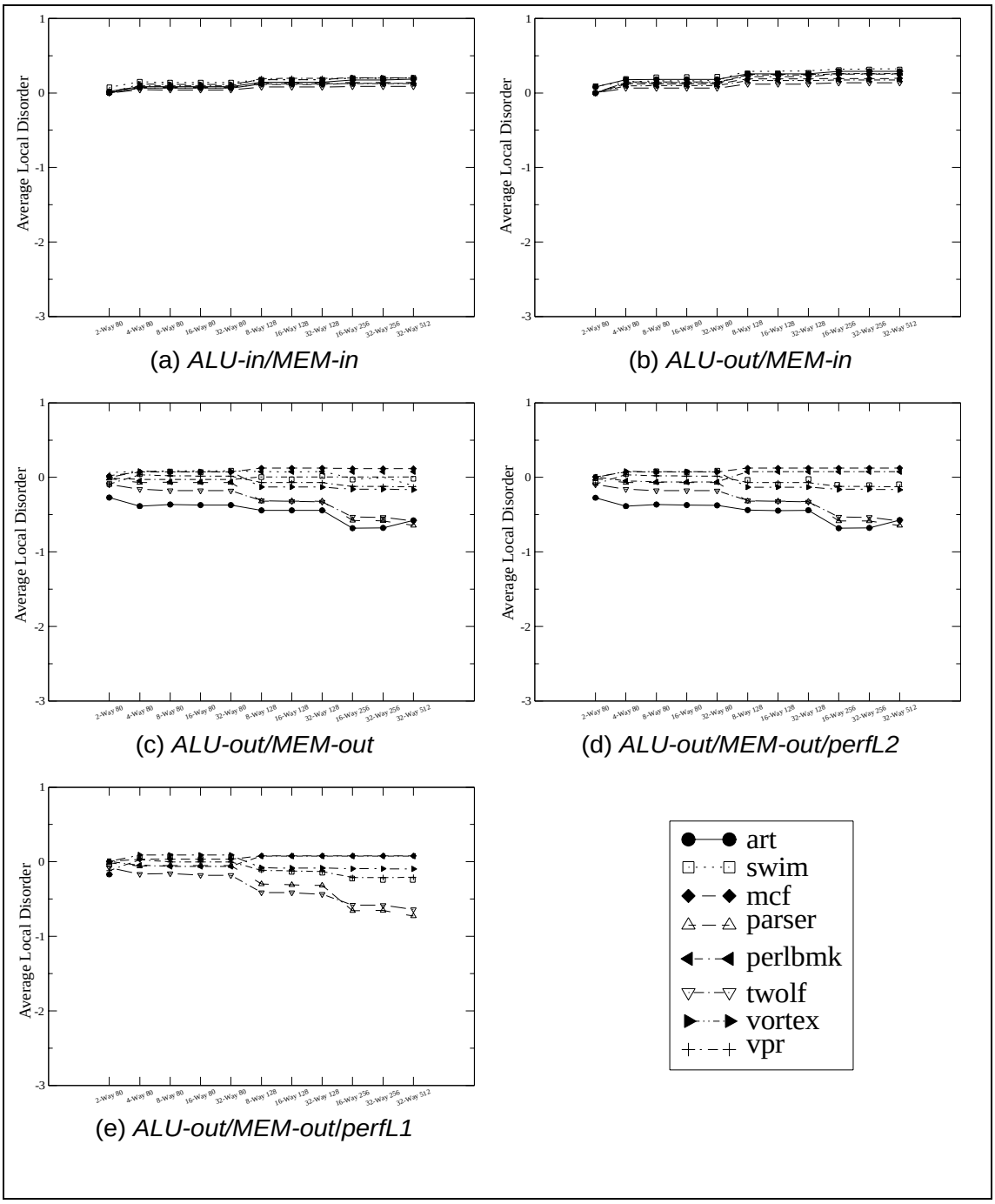


Figure 6.8: Average Local Disorder. The figures show the average local disorder across different issue-logic and memory system configurations. The general trend shows an increase in disorder with increase in out-of-order aggressiveness.

Based on our introduction of the local disorder metric, we observe that in general memory instructions are issued in close proximity to each other. We conclude that when a processor issues instructions in a particular region of code, it usually finds memory

instructions that are ready to be issued in the same region, hence causing low local disorder. This local disorder results illustrate that when a processor speculates, it continues to speculate in the same region of code.

6.3 Summary

This chapter introduced the *disorder* metric to measure the degree by which memory instructions are issued out-of-order. We defined two types of disorders: *global disorder* and *local disorder*. Global disorder is the degree by which memory instructions are issued out-of-order when compared to program order. Local disorder on the other hand measures the degree by which memory instructions issue out-of-order when compared to other memory instructions. From our study on measuring disorder, we observe that applications exhibit large global disorder and small local disorder. We showed that global disorder is primarily due to increasing reorder buffer sizes with only 10% of memory instructions issued to the memory subsystem on time. We showed that large global disorder and small disorder are a direct result of an out-of-order core processor speculatively issuing instructions.

CHAPTER 7 *Relating Disorder To Negative Effects*

Having defined disorder and illustrated its existence, we now present the correlation between disorder and the negative effects in the memory subsystem.

7.1 Replay Traps

Figure 7.1 compares the trap rate for the different issue logic and cache configurations. Trap rate is defined as the total number of replay traps that are handled per 1000 instructions committed. The figure illustrates on the x-axis the aggressive out-of-order configurations with respect to issue widths and reorder buffer sizes, and on the y-axis the number of traps that are handled per 1000 instructions committed. In general, for the different configurations of the issue logic and cache, we observe that increasing out-of-order capability (going right on the x-axis) causes the trap rate to increase. We observe that the more aggressive the use of out-of-order mechanisms, the more frequently the occurrence of traps. This implies that increasing out-of-order aggressiveness comes at the cost of an increase in the frequency of replay traps. In general, we observe that while replay traps occurred infrequently with lesser aggressive out-of-order mechanisms, increasing out-of-order efficiency by increasing issue-widths and reorder buffer sizes causes a degradation in performance in the memory subsystem.

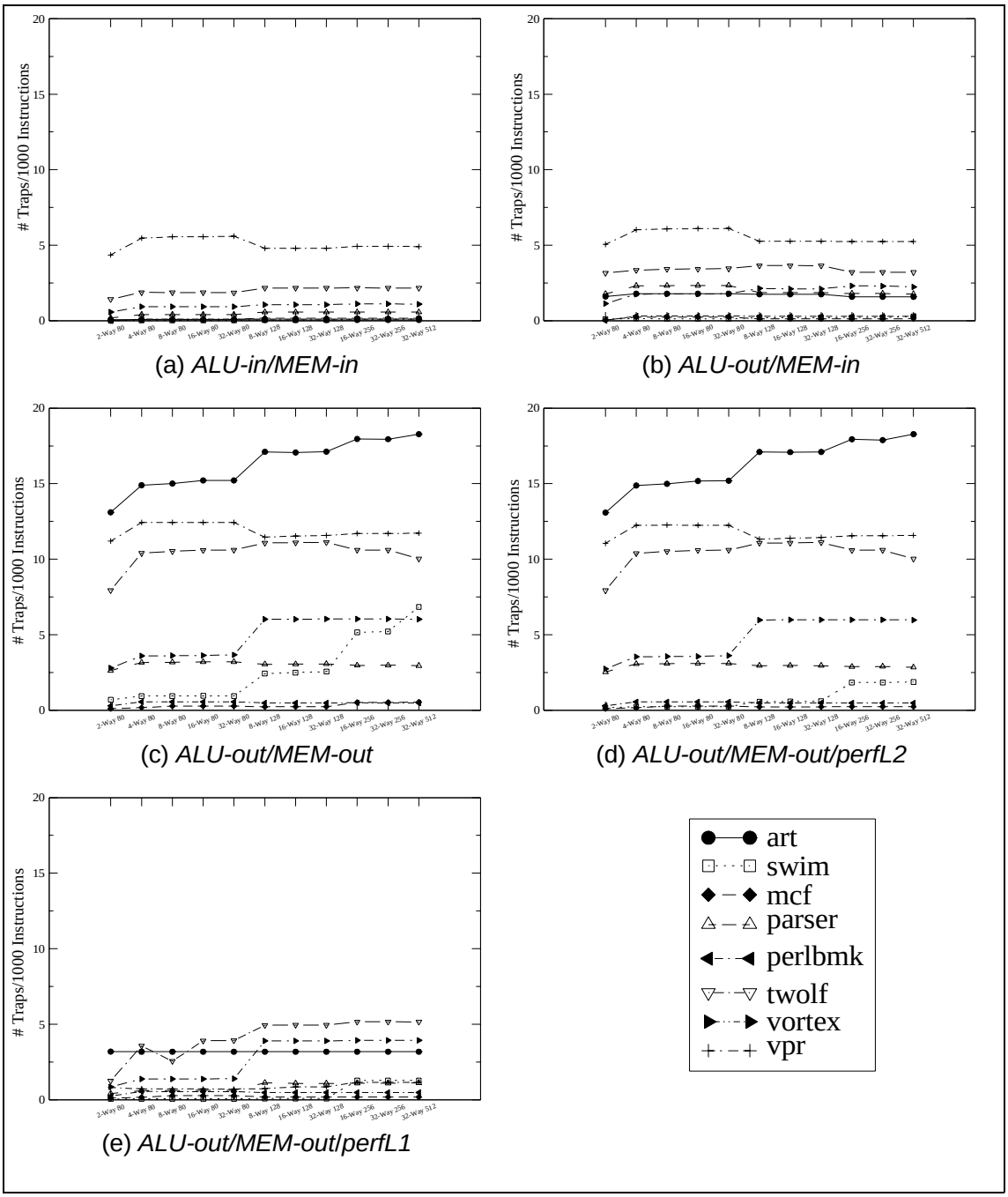


Figure 7.1: Memory Reorder Trap Rate. The figure compares the number of memory re-ordering traps that occur with increased out-of-order aggressiveness for three different system cache configurations. The graphs show that increasing aggressiveness increases the trap rates by more than 50%. We also note that a good deal of reorder traps occur due to misses in the primary L1 data cache.

From the graphs, we observe that while the occurrence of replay traps was rare when issuing memory instructions in-order, the out-of-order issue of memory instructions causes as much as a factor of 7 increase in the frequency of replay traps. Of the different

benchmarks, we observe that *art*, *twolf*, and *vpr* suffer heavily from replay traps. Further investigation revealed that these benchmarks suffered extensively from store-replay traps, load-load replay traps, and load-miss load replay traps. We observed that these benchmarks have extensive load-store communication, thus with the use of blind load speculation these benchmarks hurt from frequent store replay traps. Another observation about most of the benchmarks is that they tend to reuse the same memory addresses frequently. This usually happens when a compiler is required to spill data to memory due to limited architectural registers and then at a later point in time re-load the data from memory. With larger re-order buffers, the spilling and re-loading of data to and from the same memory address is exposed more frequently than smaller reorder buffers as the processor is able to observe a larger view of the application instruction stream. With hardware mechanisms to maintain memory consistency by ensuring reads to the same memory address occur in program order, such replay traps increase with the increase in re-order buffer sizes. Thus, based on these reasons, we observe that increasing out-of-order capability by increasing reorder buffer sizes increases the frequency of the replay traps.

From the figure, we observe that the trends for all five configurations closely match the trends for the average global disorder as depicted in Figure 6.5. Like global disorder, the frequency of traps increases with increased out-of-order capability. Like global disorder, we observe that reducing L1 data cache misses reduces the number of replay traps by a factor of 2 or more. We observe that as the memory system becomes more “perfect”, the trap rate decreases, implying that the bulk of the replay traps occur primarily due to misses in the L1 data cache. This behavior is consistent with the behavior of global disorder. To correlate the two metrics, we now present a scatter plot in Figure 7.2 with the average global disorder on

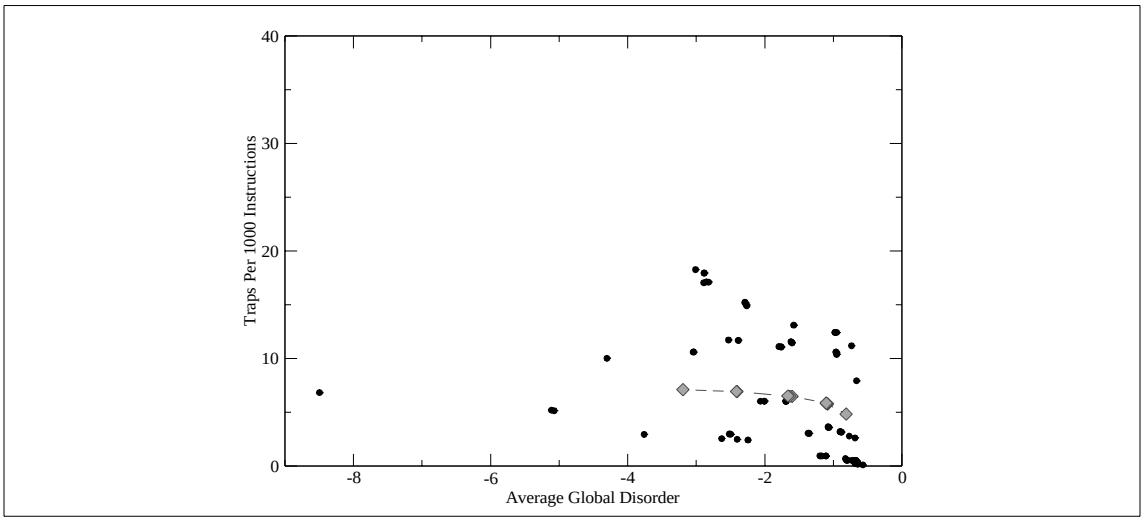


Figure 7.2: Trap Rate Vs. Global Disorder. This figure correlates the trap rate with average global disorder. We observe a direct correlation between the increase in global disorder and the increase in the total number of replay traps.

the x-axis and the trap rate on the y-axis. In the graph, the circles represent the behavior of all the applications studied in this chapter, and the diamonds connected via a dashed line is the overall average of global disorder and trap rate. From the figure, it can be observed that different benchmarks have varying trap frequencies with similar global disorders. However, in general, there is a direct correlation between the two: the more the global disorder, i.e. the later memory instructions are issued, the larger the trap rate. From the graph, if we only consider the average values (i.e. diamonds connected via dashed lines), we observe that the increase in trap rate correlates directly with increased global disorder. From the right going left, the five points are the 2-way 80, 80-entry ROB, 128-entry ROB, 256-entry ROB, and finally a 512-entry ROB. As before, we observe that issue-widths have no impact on global disorder with increased out-of-order aggressiveness. From the figure we observe that the smaller the global disorder the lesser the frequency of traps, thus we can conclude that it is imperative that global disorder be reduced with increasing out-of-order aggressiveness. We notice that there is not too much of difference between different global disorders and trap

frequency, i.e. the average graph stays relatively flat. However, we expect this to change when we move from blind speculation methodology of issuing loads to controlled speculation. This will be evident in the next chapter of this dissertation.

7.2 L1 Cache Performance

We now investigate the effects of out-of-order execution on the performance of the L1 data cache. To measure performance of the data cache we measure the total number of cache misses an application encounters while changing the processor configuration from the ALU-in/MEM-in configuration to the ALU-out/MEM-in and ALU-out/MEM-out configurations. We remind the reader that a cache miss is one that not only misses in the data cache but also in the outstanding MSHRs.

Figure 7.3 plots the total number of cache misses encountered in the ALU-out/MEM-in and ALU-out/MEM-out configurations when normalized to the ALU-in/MEM-in configuration for three different L1 data cache sizes: 16K, 32K, and 64K. The x-axis represents the different benchmarks; for each benchmark we present two line graphs for the 11 different configurations of increased out-of-order aggressiveness. The y-axis presents the percent increase in the number of cache misses when compared to the ALU-in/MEM-in configuration. The squares (dark) connected via lines represent the percent increase in cache misses of the ALU-out/MEM-out configuration as compared to ALU-in/MEM-in configuration, and the circles (light) connected via lines represents the percent increase in cache misses of the ALU-out/MEM-in configuration when compared to the ALU-in/MEM-

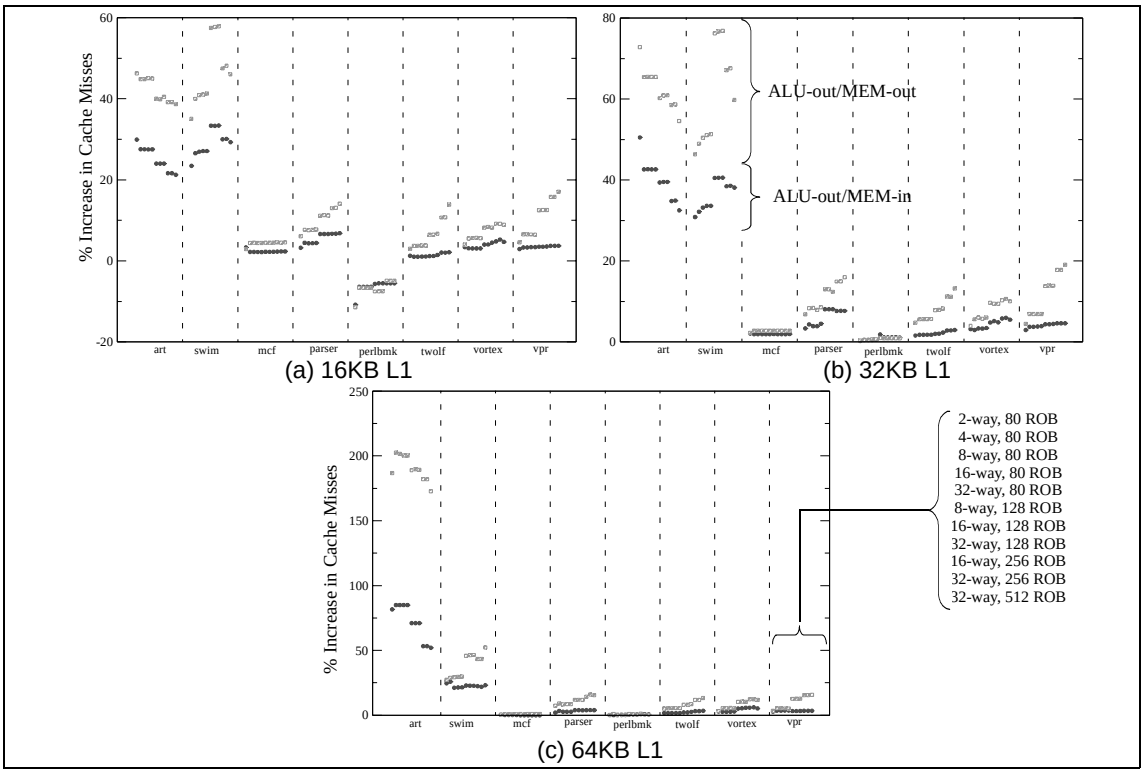


Figure 7.3: L1 Cache Misses: In-Order Vs. OoO. The figure shows the L1 cache miss rates as a percent for the different benchmarks for a system with purely in-order execution, a system that executes memory instructions in order and ALU out-of-order, and a system that allows execution out-of-order.

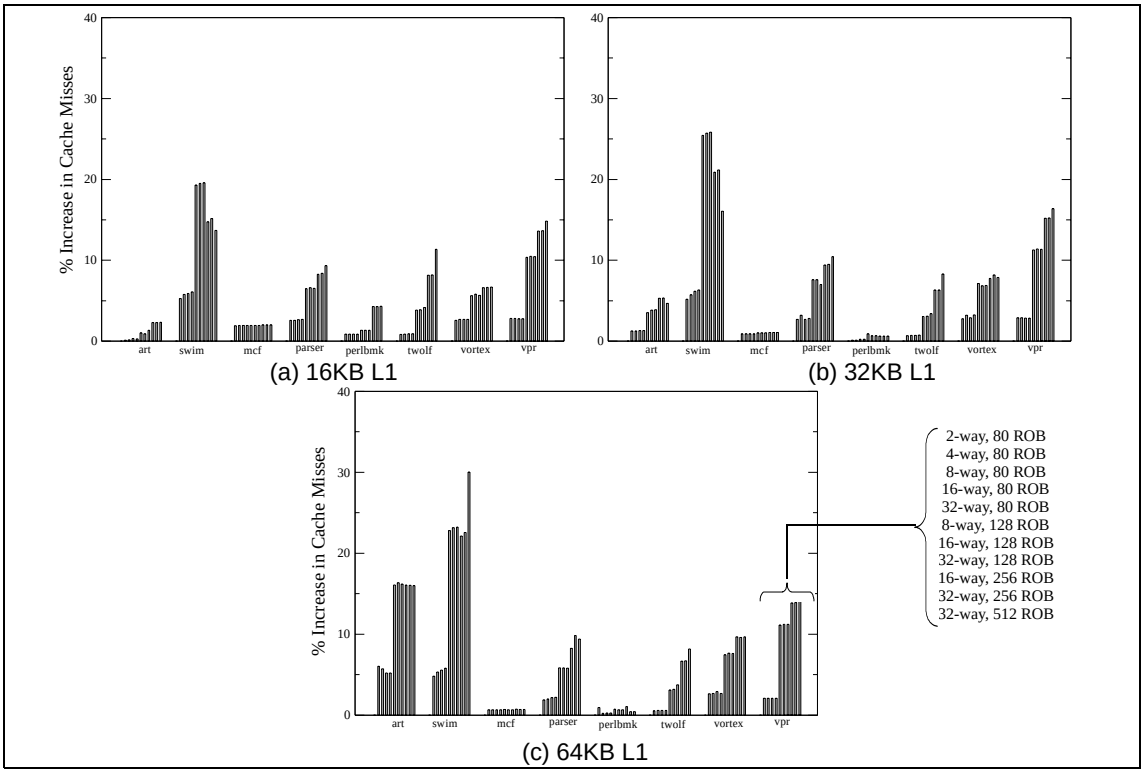


Figure 7.4: Effects of OoO on L1 Cache Misses. (a) 16K L1 (b) 32K L1, (c) 64K L1

in configuration. The benchmarks on the x-axis are separated via dashed lines to distinguish between the different data sets for each benchmark. Note that the graphs for the different cache sizes are not on the same scale.

From the graphs we observe that increasing out-of-order aggressiveness can increase the total number of application cache misses. For the three different cache sizes, we observe that for each benchmark, the general trend with increasing out-of-order aggressiveness (going right) in general increases the total number of cache misses when compared to a lesser aggressive out-of-order mechanism. When comparing the ALU-out/MEM-in issue-logic configuration to an in-order system, we observe an average of 5% increase in the total number of cache misses, with some benchmarks being affected by as much as 75% with a 64K L1 data cache. Similarly, with the ALU-out/MEM-out configuration, we observe a much larger degradation in cache performance. In general, we observe a factor of 2-3 degradation in cache performance when compared to the ALU-out/MEM-in configuration. When comparing the ALU-out/MEM-in configuration, an all out-of-order issue machine causes a 25-250% increase in cache misses when compared to the ALU-in/MEM-in configuration. These results thus show that increasing out-of-order aggressiveness comes at the cost of a degradation in the memory subsystem.

We observe two interesting behaviors from the graphs. First, increasing out-of-order aggressiveness in some cases can be beneficial. For example, for the benchmark *perlbmk*, when compared to the ALU-in/MEM-in configuration, the number of cache misses decrease. This can be explained by data prefetching due to the out-of-order issued memory instructions or due to wrong-path memory instructions prefetching data into the data caches. The second interesting behavior that we observe with increased out-of-order aggressiveness

is that increasing the cache size from 16K to 64K worsens the degradation in performance. At first this is non-intuitive as one would expect that a larger data cache should reduce the degradation in the memory subsystem. However, this can be explained by the fact that larger data caches provide a better hit rate allowing a processor to execute further into an application's instruction stream. On the other hand, with smaller data caches frequent cache misses can cause an application to eventually stall. The fact that larger data caches allow for an application to do more speculation implies that the processor will execute more memory instructions out-of-order speculatively. This can cause an increase in the number of cache misses either due to constructive or destructive interference. Thus, even though the number of cache misses with a larger data cache is smaller than that of a smaller data cache, the overall degradation in the memory subsystem is larger due to the out-of-order issue of memory instructions.

To further understand the impact of increased out-of-order capability, Figures 7.4 (a,b,c) illustrates the ALU-out/MEM-out behavior of the cache misses for the three different cache sizes. For each benchmark, the percent increase in the total number of cache misses is normalized to a processor configuration of the Alpha 21264 with a 2-way issue logic (see Table 6.1). From the figure, we observe that across all benchmarks and cache sizes, increasing issue-widths and reorder buffer sizes can increase the number of cache misses by 25-30%. For most of the benchmarks, we observe that maximum degradation in cache performance occurs when reorder buffer sizes are increased, hence the step shaped pattern of the bar graphs. The data presented here clearly shows that continuing to increase out-of-order aggressiveness leads to a degradation in performance in the first level data cache.

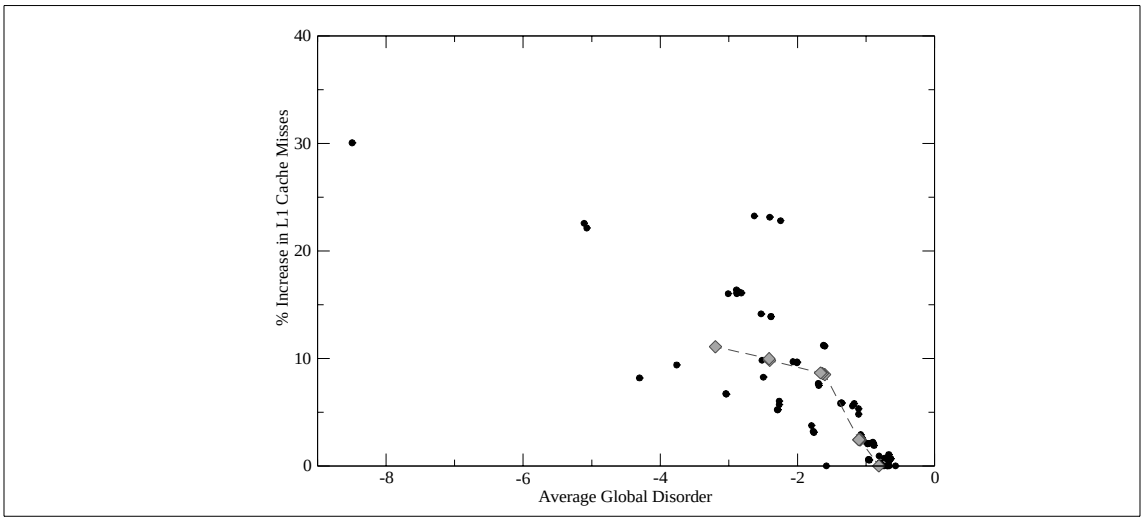


Figure 7.5: L1 Cache Misses Vs. Global Disorder. This figure correlates the increase in the number of L1 cache misses with average global disorder. We observe a direct correlation between the increase in global disorder and the increase in the number of L1 cache misses.

Furthermore, we also observe that the trends in the degradation of cache performance closely match the trends in global disorder: increasing out-of-order capability also increased global disorder. To illustrate this explicitly Figure 7.5 presents a scatter plot of average global disorder on the x-axis and the increase in L1 cache misses when compared to a 2-way ROB-80 configuration on the y-axis. As before, we present the behavior of the different benchmarks (circles) as well as the average across all the benchmarks (diamonds connected via dashed line). Again, we observe that the degradation in performance varies across different benchmarks for the same average global disorder. However, we observe there is a direct correlation between average global disorder and the degradation in L1 cache performance. Based on the scatter plot, workloads with the largest global disorder have the worst degradation in cache performance, for example, a greater than 20% degradation in cache performance is experienced when memory instructions are delayed by more than 4 or 5 memory instructions. Thus, we again conclude that increased global disorder correlates with the degradation in the performance of the L1 cache and we believe that mechanisms to

reduce the reordering of memory instructions must be employed so as to reduce this source of performance loss.

7.3 L2 Cache Performance

Like the L1 data cache, Figure 7.6 plots the total number of cache misses encountered in the ALU-out/MEM-in and ALU-out/MEM-out configurations when normalized to the ALU-in/MEM-in configuration for three different L2 data cache sizes: 512KB, 1MB, and 2MB. The x-axis represents the different benchmarks with the 11 different configurations of increased out-of-order aggressiveness and the ALU-out/MEM-in and ALU-out/MEM-out configurations. As before, the y-axis presents the percent increase in the number of cache misses when compared to the ALU-in/MEM-in configuration. Again the connected squares represent the ALU-out/MEM-out configuration and the connected circles represent the ALU-out/MEM-in configuration. Note that the graphs for the different cache sizes are not on the same scale.

The benchmarks show a varying behavior as a result of out-of-order execution of 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. We observe workloads can benefit by 2-10% with the out-of-order issue of instructions while others can experience up to a 30% degradation in cache performance. The decrease in the number of cache misses can be explained by useful prefetching performed by either the speculative issue of memory instructions or the sequential prefetch engine. On the other hand, the

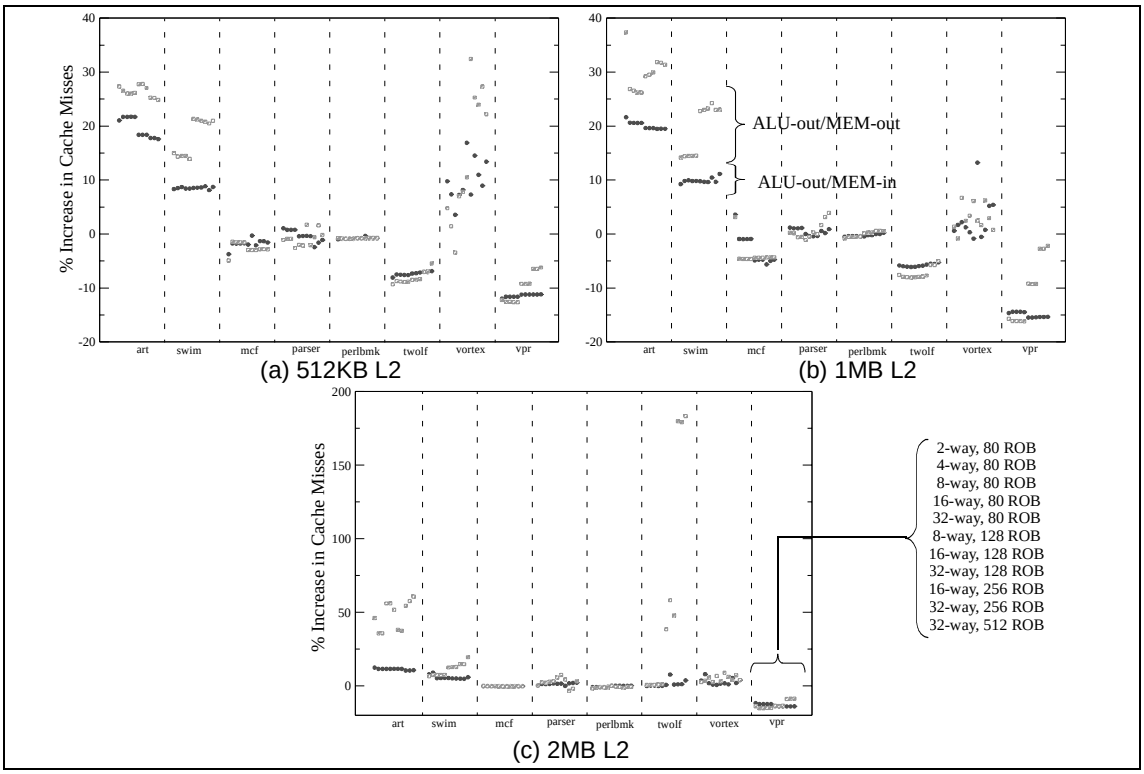


Figure 7.6: L2 Cache Misses: In-Order Vs. OoO. The figure shows the L2 cache miss rates as a percent for the different benchmarks for a system with purely in-order execution, a system that executes memory instructions in order and ALU out-of-order, and a system that allows execution out-of-order.

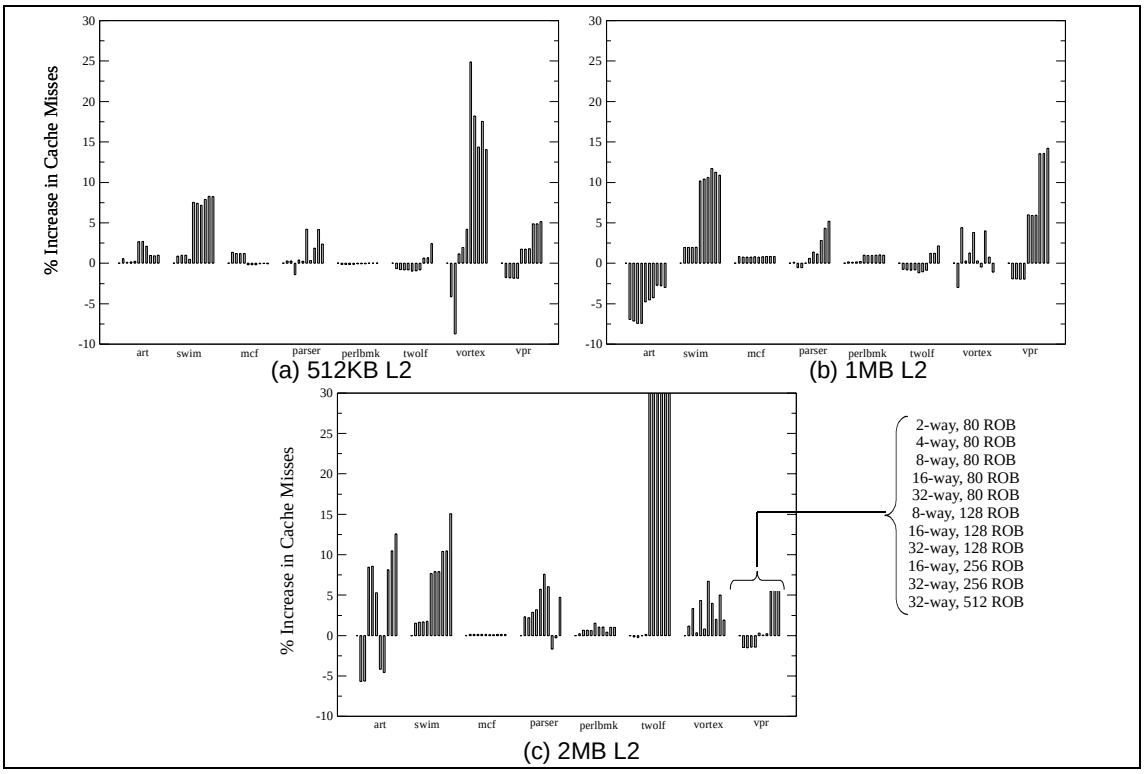


Figure 7.7: Effects of OoO on L2 Cache Misses. (a) 512K L2 (b) 1MB L2 (c) 2MB L2

increase in the number of cache misses can be explained by an increase in the number of conflict misses due to constructive or destructive interference between older and younger memory instructions.

Unlike the L1 data cache, with increasing L2 data cache sizes, only three benchmarks *art*, *swim*, and *twolf* suffer 50%, 15%, and 180% degradation when compared to an in-order configuration. Except for these benchmarks, such behavior implies that the L2 cache is resistant to the out-of-order issue of instructions while keeping out-of-order aggressiveness constant. However, from Figure 7.7, we observe that increasing out-of-order aggressiveness for the ALU-out/MEM-out configuration in most cases results in a degradation in L2 cache performance by 25% (twolf showing 180%) with a 2MB L2 cache. Of the different benchmarks, *art*, *parser*, and *vortex* display erratic behavior with increasing out-of-order aggressiveness. Attempts to investigate the reasoning behind such erratic behavior led us to arrive at the fact that such behavior is due to the randomness in which instructions become ready due to the different latencies on producers and the order in which the out-of-order issue logic selects the ready instructions. The different issue-widths and reorder buffer sizes add to the randomness thus causing erratic behavior in some of the benchmarks. However, such erratic behavior provides more reason to illustrate that the order in which memory instructions are issued to the memory subsystem can change the performance of the memory subsystem.

To observe if there is a correlation between global disorder and the increase in L2 cache misses, we present in Figure 7.8 a scatter plot of the average global disorder on the x-axis and the increase in L2 cache misses on the y-axis. As before we plot the different benchmarks and the average across all benchmarks (diamonds connected via dashed-line).

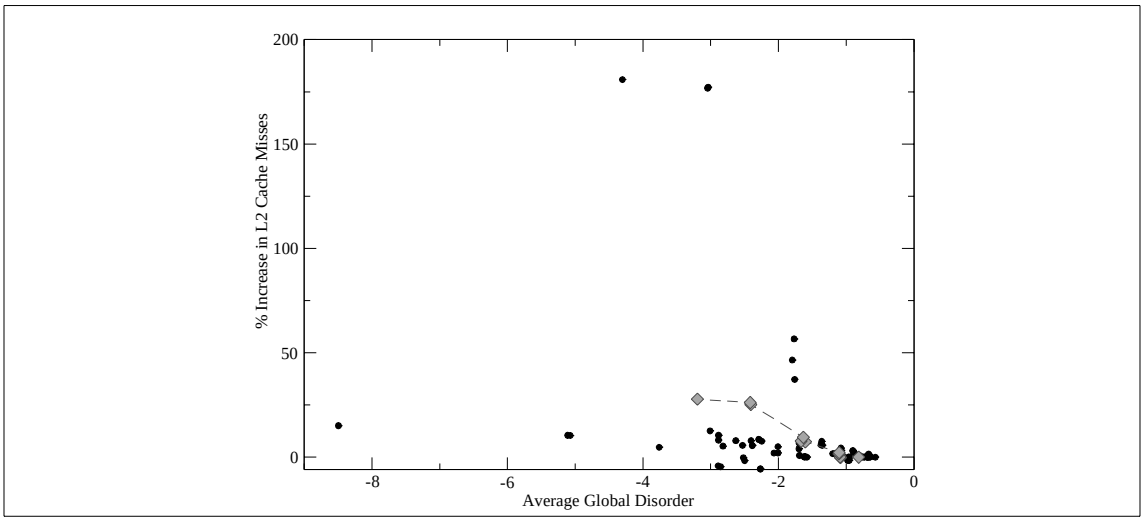


Figure 7.8: L2 Cache Misses Vs. Global Disorder. This figure correlates the increase in the number of L2 cache misses with average global disorder. We observe a direct correlation between the increase in global disorder and the increase in the number of L2 cache misses.

Again, we observe that the degradation in cache performance varies across different benchmarks with similar average global disorders. However, we observe that a general trend shows that increase in the number of L2 cache misses correlates well with an increase in global disorder. Based on the scatter plot, we observe that the "erratic behavior" discussed earlier was essentially varying global disorder across systems with different out-of-order aggressiveness. Thus, we again conclude that the increase in L2 cache misses correlates well with global disorder. Thus, it is imperative that mechanisms to reduce the global disorder be adopted with increasing out-of-order aggressiveness.

7.4 Performance of Aggressive Out-of-Order Mechanisms

Increasing the aggressiveness of the out-of-order core increased the disorder, the total number of cache misses, and the number of replay traps. We know that these trends in normal cases significantly hurt performance. The question however is: *Does the increase in out-of-order execution overcome these hurdles to provide net performance improvements?* Figure 7.9 shows the performance graphs for the five different processor configurations. with the different benchmarks and out-of-order configurations on the x-axis and cycles per instruction (CPI) on the y-axis. CPI is classified into stall cycles where memory instructions could not retire due to memory latency (black), stall cycles where instructions could not retire because they either had not been issued or had not yet finished execution due to ALU latency (medium grey), and overhead cycles due to recovering from branch mispredicts and replay traps (light grey). The ALU and memory components of CPI are computed by measuring the number of cycles the retire stage stalls because it could not retire an ALU or memory instruction. The overhead portion was computed by taking the difference between the total number of cycles and the sum of the ALU and memory instruction stall cycles in the retire stage.

From the figure, we observe that moving from the ALU-in/MEM-in core to the ALU-out/MEM-in core yields performance improvements by 33% or more. These improvements

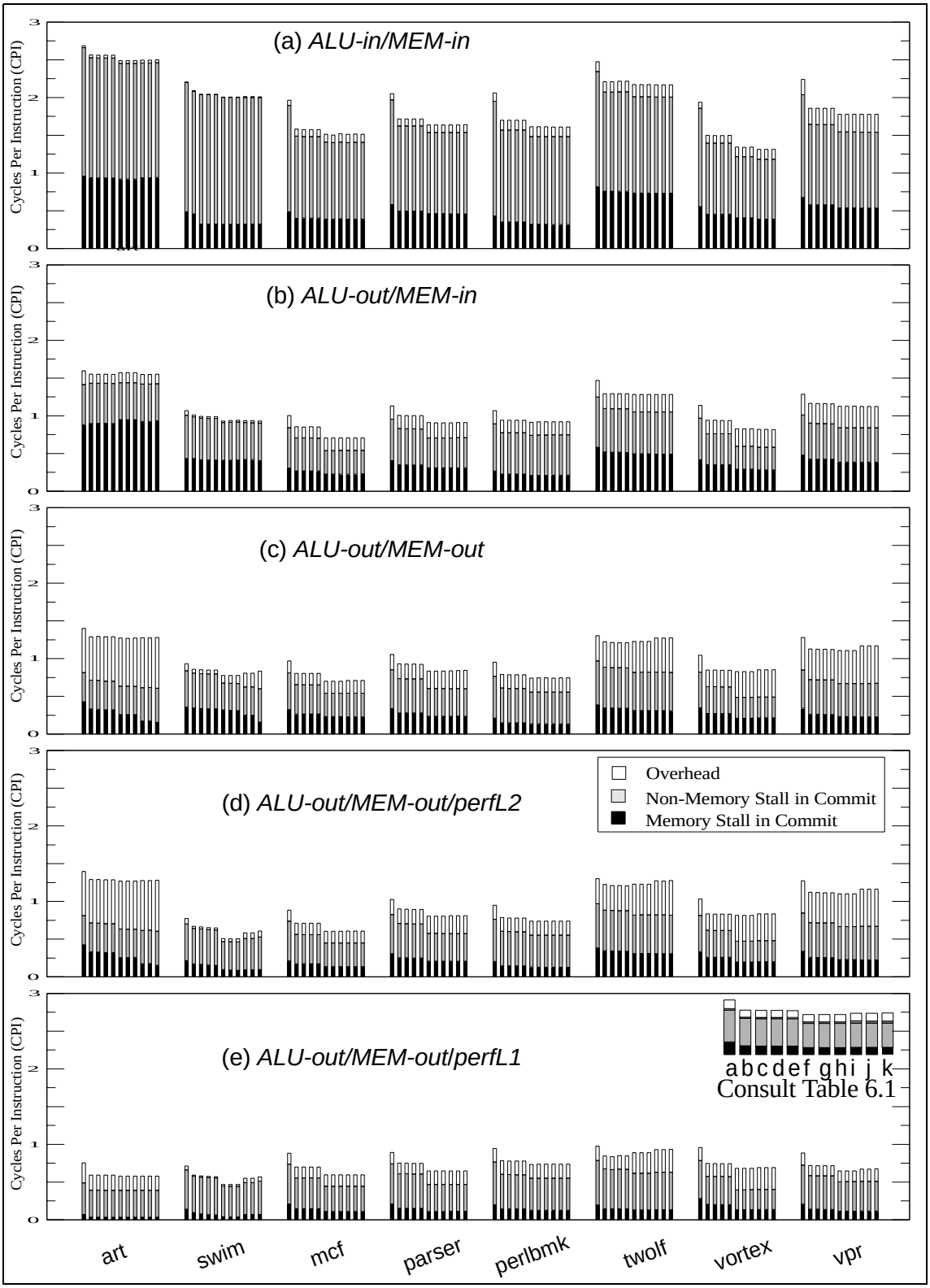


Figure 7.9: Performance. The figure compares the CPI with increased aggressiveness of the out-of-order core. We see two important yet independent scenarios. Firstly, increasing the aggressiveness of the out-of-order cores significantly increases memory consistency traps, thus rather than improving performance, it hurts performance. Secondly, even if we assume zero cycles memory consistency trap recovery, we see that improvements in CPI are minimal.

are largely due to the reduction in stalls while attempting to commit ALU instructions (ALU component of CPI). As expected, the memory component of CPI remains constant in both the systems as they both enforce the in-order issue of memory instructions. Even with improvements in performance, we observe that the effects of the increase in the replay traps is evident by the 3-20% increase in the overhead component of CPI. Thus, even though out-of-order execution of ALU instructions improves overall performance, the degradation due to replay traps reduces maximal possible performance improvement.

Comparing the ALU-out/MEM-in and ALU-out/MEM-out core, a cursory glance of the performance graphs show no remarkable speedups. The performance of these two systems are about the same, within 2-10% (with the exception of art). Moving towards a system that executes both ALU instructions and memory instructions out-of-program order reduces the time spent in waiting for memory operations to finish execution, reduces the time spent waiting for ALU operations to finish execution, however increases the overhead by 200% or more (comparing the white bars of the ALU-out/MEM-in and ALU-out/MEM-out configurations). Thus, all potential gains due to the out-of-order issue of memory instructions are lost in the handling of replay traps.

One would expect that with increased out-of-order aggressiveness comes decreased CPI, however excluding the 2-way issue system, for the remaining configurations, the graphs show performance to be relatively flat. If we overlook the overheads, and assume that the processor would be able to fix the problem with zero overhead, increasing the issue width from 4 to 32 way and the ROB size from 80-512 produces no remarkable improvement in performance. In fact, there are scenarios where it actually hurts performance. We see that increasing the out-of-order aggressiveness is limited by the rate at which memory

instructions retire. Such behavior is perhaps indicative of the fact that commercial out-of-order core processors have stayed stagnant at a maximum 4-way issue core.

If we consider the ALU-out/MEM-out configuration alone, the graphs reveal the tremendous overhead of replay traps with increased issue widths and reorder buffer sizes. The increased out-of-order capability causes memory instructions to be significantly reordered and conflicts with the processor's memory ordering model. Our studies show that a processor can spend as much as 25-40% of total execution time handling replay traps. These results provide further motivation to reduce the frequency of replay traps and reduce wasted work and hopefully gain the performance lost.

7.5 Summary

In this chapter we have correlated increased out-of-order aggressiveness with the negative effects in the memory subsystem. Furthermore, we have also correlated global disorder with an increase in the number of cache misses and replay trap overhead. We showed that aggressive out-of-order mechanisms have the largest amount of global disorder and the most negative effects in the memory subsystem. We conclude that mechanisms to reduce global disorder must be employed in out-of-order aggressive systems to reduce the negative overheads in the memory subsystem.

CHAPTER 8 *Reducing Reordering of Memory Instructions*

We have shown that the increase in the number of replay traps and cache misses can be directly correlated with the reordering of memory instructions. A direct approach to reduce the negative effects would be to reduce the reordering of memory instructions. Besides the approach of issuing memory instruction in program order, an alternative approach to reduce the reordering of memory instructions would be to reduce the total number of memory instructions visible to the select and issue logic. This can either be accomplished by trivial mechanisms such as reducing the size of the reorder buffer itself or alternatively reducing the size of the load-store queue. Reducing these hardware data structures effectively reduces the total number of memory instructions in-flight, thus indirectly reducing the total number of memory instructions visible to the select and issue logic. However, reducing the reorder buffer size reduces the possibility of extracting maximum possible ILP. Alternatively, reducing the size of the load-store queue is a possible option, however, efficient use of all entries in a large reorder buffer directly depends on the size of the load/store queue. This is because the load/store queue not only supports simultaneous searches to find memory dependencies to adhere to memory consistency models, but it also maintains all in-flight memory instructions in program order. In the event that the load/store queue becomes full and a new load/store instruction is fetched, the fetch stage stalls until a memory instruction commits and frees space in the load/store queue. Since memory instructions constitute on average one third of a program's total

instructions, attempting to use a load/store queue that is any less than one third the size of a reorder buffer can under-utilize the reorder buffer.

To determine how much out-of-order processor performance is dependent on the out-of-order issue of memory instructions, we investigate a mechanism that uses large reorder buffers and load/store queues yet provides the benefits of smaller load/store queues. Rather than physically reducing the size of the load/store queue, we throttle the degree by which memory instructions are issued out-of-order via a *windowing* mechanism.

8.1 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, we investigate the degree to which out-of-order processor performance is dependent on the out-of-order issue of memory instructions. To study this, we restrict the reordering of memory instructions based on a window of instructions by using the network communication concept of windowing [69]. By using a *sliding window protocol*, we 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 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 communication networks. With typical network communication, a sender normally transmits 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 for the oldest packet in the sender's queue, the window is extended by sliding the window down to allow the transmission of additional packets in the queue.

8.1.1 Virtual Load/Store Queues (VLSQs)

We attempt to reduce the reordering of memory instructions by utilizing the property of windowing. We use the windowing concept to reduce the reordering of memory instructions by introducing a virtual window into the existing load/store queue. The size of the window determines the number of memory instructions available to the select and issue logic. The virtual window essentially acts as a virtual load/store queue (VLSQ). The virtual load/store queue is maintained using two pointers into the existing load/store queue: *virtual head* and *virtual tail*; *virtual head* always points to the oldest non-issued memory instruction and *virtual tail* points to the end of the 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* pointers are changed when the memory instruction at the *virtual head* is issued.

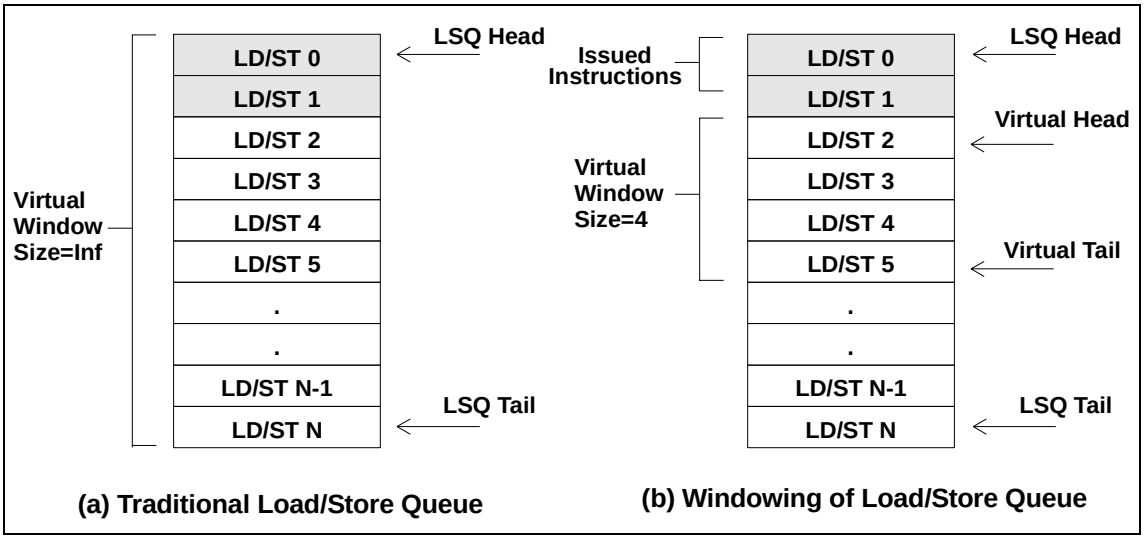


Figure 8.1: Windowing Memory Instructions: A mechanism to reduce the reordering of memory instructions (a) The figure illustrates the traditional implementation of a load-store queue. (b) Using windowing, only memory instructions that lie within the virtual head and virtual tail pointers are issued to execute. Other memory instructions must wait till these lie within the virtual window before they can be issued to the memory system.

Figure 8.1(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 but waiting to retire. With a traditional load/store queue, the issue logic can schedule any memory instruction (between 2 and N) whose operands are ready. Figure 3(b) illustrates an example of a VLSQ with *Wsize* = 4. The *virtual head* pointer points to the first non-issued memory instruction, i.e. memory instruction 2. With a 4-entry VLSQ, the issue logic can only schedule memory instructions 2, 3, 4, or 5. If none of the instructions in the VLSQ have their operands ready, the issue logic stalls the issue of memory operations. When memory instruction two is issued, the virtual window slides down until the *virtual head* pointer reaches the first non-issued memory instruction. Thus, by controlling the size of the VLSQ one can control the degree by which memory instructions are reordered. The larger the size of the VLSQ the more the reordering of memory instructions. The smaller the size of the VLSQ the less the reordering. For example,

by setting the VLSQ to be of size 1, the processor would issue memory instructions in program order, by setting the VLSQ to the size of the LSQ the processor would exploit maximum memory level ILP. Since we vary the size of the LSQ as part of our study, we define a VLSQ that is the size of the LSQ as a VLSQ that is *Infinite* in size.

The benefits of using a VLSQ are two-fold. First, a VLSQ reduces the reordering of memory instructions without affecting instruction fetch bandwidth or the execution of ALU instructions. By controlling the size of the VLSQ, windowing can control the reordering of memory instructions. For example, an infinite VLSQ allows for maximum possible reordering and a VLSQ of size 1 forces memory instructions to be issued in program order. Within these extremes, varying the size of the VLSQ serves as the throttle to control the degree by which memory instructions are reordered.

The second advantage of using VLSQs is that they can reduce the total number of memory instructions executed speculatively. The benefits of reducing speculative memory instructions are: (a) fewer memory disambiguation related load/store queue searches and (b) fewer number of cache accesses and misses. 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 power and energy savings in the data caches and the fetch, map, and execution hardware.

However, a downside associated with using VLSQs is a reduction in the amount of ILP available for memory instructions. Applications that are heavily dependent on the quick execution of memory instructions can suffer from a degradation in performance due to the delayed issue of memory instructions to the memory system. Such *memory-instruction dependent* (or memory intensive) applications may require a larger VLSQ than those

applications that are *memory-instruction independent,* i.e. those that are compute intensive. Characterizing application behavior with different window sizes statically can help determine an optimal virtual load/store queue size.

This dissertation explores the windowing concept by statically varying the size of the VLSQ. We profile applications with different virtual window sizes to determine an optimal VLSQ size. However, a *dynamic* approach of varying the size of the VLSQ based on application run time events such as replay traps and cache misses is also possible for extending the work presented in this dissertation.

8.1.2 Controlling Global Disorder with VLSQs

Using VLSQs to throttle the degree by which memory instructions are issued out-of-order reduces the global disorder in the system. To illustrate the impact of windowing on the issuing of memory instructions and global disorder, Figure 8.2 provides an example of memory instructions issued before and after windowing. For each memory instruction we provide the absolute disorder before and after the use of VLSQs. For the purpose of this example we assume a VLSQ of size 4. Based on the initial example, if we assume that the delay in the issue of older memory instructions are either due to functional unit latency or due to compulsory cache misses, then the time at which memory instructions can be scheduled with VLSQs will follow the same behavior as in the initial case. For example, with an infinite VLSQ, memory instruction ID 2 was issued in cycle 126. With our assumptions, with a VLSQ of size 4, memory instruction ID 2 would only issue in cycle 126. Such behavior is also similar for memory instructions 4, and 6. Thus, with a VLSQ of

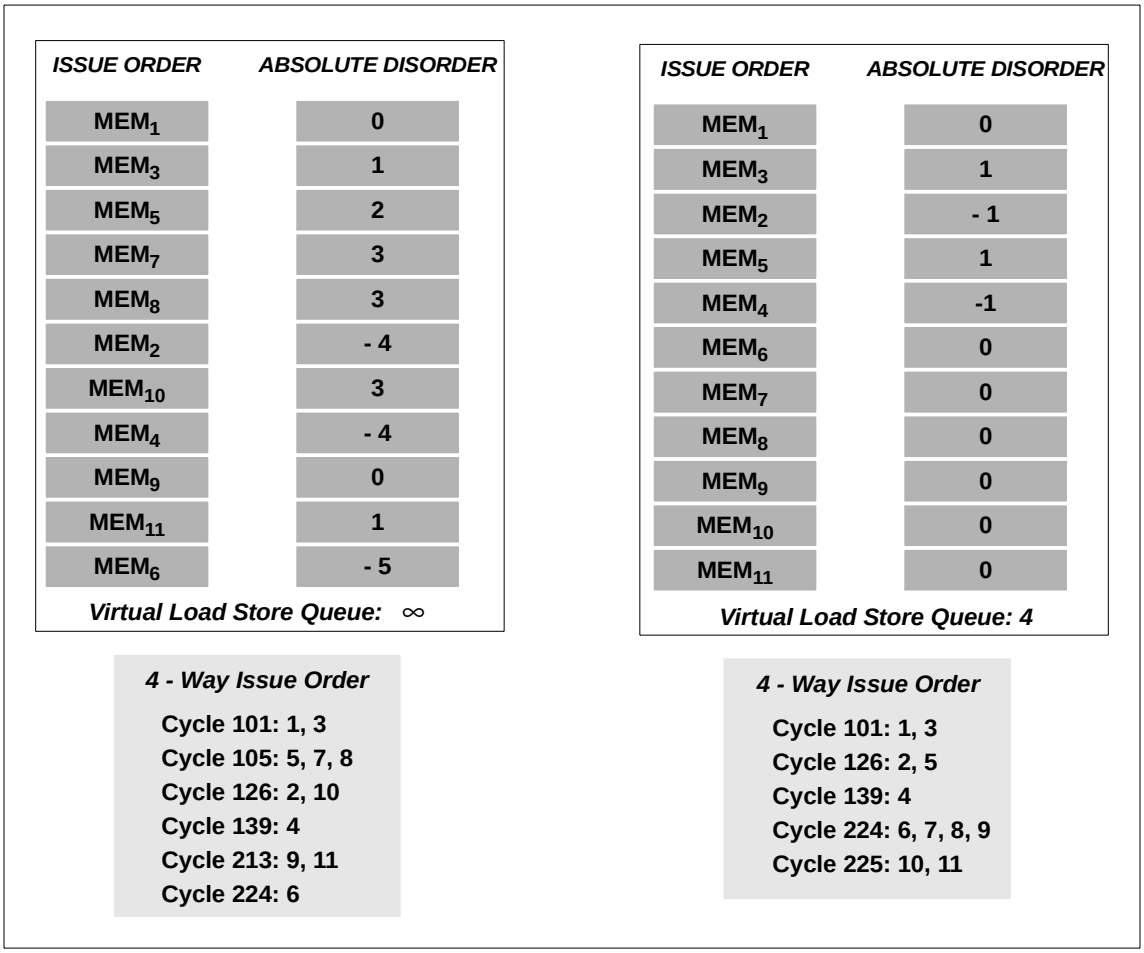


Figure 8.2: Windowing Vs. Absolute Disorder. The figure illustrates the effect of windowing on absolute disorder.

size 4, the order in which memory instructions are throttled based on these limitations, hence, the possible ordering of memory instructions as listed on a cycle-by-cycle based in the figure. Thus, we observe that by the use of VLSQs, in this example, the global disorder of the instructions is reduced drastically. Such reduction in global disorder, as we show later in this chapter, reduces the number of replay traps and cache misses, however it affects performance due to the reduction in memory level parallelism.

From the earlier example, we can see that VLSQs aid in the reduction of global disorder. Based on this, the use of VLSQs can provide us with intuition on the total amount of global disorder that is beneficial in a system. By sizing the VLSQs so that they guarantee a

particular maximum global disorder, we can use VLSQs to identify the maximum useful disorder in a system.

8.2 Windowing Study Simulator Parameters

For this study, we use a validated execution driven Alpha 21264 simulator: sim-alpha [8, 15]. The simulator models a 64KB two-way set associative L1 instruction cache with a single cycle hit latency, 64KB two-way set associative L1 data cache with a 3-cycle hit latency and a 2MB (unified) four-way set associative L2 cache with a 15-cycle hit latency. The caches have a 64-byte line size and also 8 MSHRs per cache. The simulator also models 128-entry fully associative instruction and data TLBs. The simulator also models a 4,096-entry branch target buffer (BTB), and a 2,048-line hybrid gshare-bimodal branch predictor. The simulator uses as its standard back-end DRAM system a detailed DRAM memory and bus model that was developed at the University of Maryland, College Park [6, 7]. For this study we use its 1.3 GB/s DDR SDRAM model.

Unlike previous initial studies in this dissertation, the remainder of the work in this dissertation uses a stride prefetcher (which is now common in modern microprocessors) with a 256-entry 2-way set associative stride table and eight 8-entry stream buffers. The simulator allows for aggressive out-of-order techniques such as load speculation and also detects replay traps mentioned in Section 4.1 of this dissertation. Furthermore, the simulator also maintains a 1024-entry store-wait data structure to avoid recurring store-replay traps. If a load instruction causes a store-load replay trap, the load's PC is stored in the store-wait table. At fetch time, if the processor finds the PC of the load in the store-wait table, the load

Table 8.1: 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	4/2	20/15	4/4/1/1	32/32	41/41
Alpha 21264 x 2	128	4/2	40/30	8/8/2/2	64/64	82/82
Alpha 21264 x 4	256	4/2	80/60	16/16/4/4	128/128	164/164
Alpha 21264 x 8	512	4/2	160/120	32/32/8/8	256/256	328/328

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

Table 8.2: Memory System Configuration

L1 Size	L1 Latency	L1 Line Size	L2 Size	L2 Latency	L2 Line Size
64 KB	3	64 Bytes	2 MB	15	64 Bytes

instruction is not issued until all prior store address are resolved. Note that this mechanism is similar to the store-set mechanism of handling memory dependencies [15]. Store sets keep track of loads and their exact store dependencies, thus they reduce false memory dependencies. Our store-wait structure on the other hand places the load on the youngest oldest store rather than the exact store as in the case of store sets. The mechanism used is more conservative than the store set implementation, but far superior than the blind speculation mechanism used in prior studies of this dissertation.

Furthermore, unlike previous studies where we varied both reorder buffer sizes and issue widths, we now fix the issue widths to 8-way. To increase ILP we vary out-of-order capability by changing the ROB size, issue and load/store queue size, as shown in Table 8.1. As our benchmarks we use the entire SPEC 2000 suite [10]. The benchmarks were warmed up by fast-forwarding the first 2 billion instructions. Data was gathered over the next 250 million instructions. The benchmarks operate on their reference data input sets.

Benchmark	DL1 MSHR Misses/1000	L2 MSHR Misses/1000	Benchmark	DL1 MSHR Misses/1000	L2 MSHR Misses/1000
bzip2	5.927	0.734	ammp	11.267	45.287
crafty	4.207	0.111	applu	17.838	21.699
eon	0.429	0.008	apsi	24.662	9.378
gap	1.019	0.955	art	117.284	22.955
gcc	26.488	1.351	equake	0.024	0.030
gzip	2.999	0.324	fma3d	0.008	0.005
mcf	90.54	107.778	galgel	0.471	0.091
parser	8.411	2.205	lucas	21.637	27.893
perlbmk	0.718	0.983	mesa	1.082	0.685
twolf	19.425	0.120	mgrid	7.898	7.078
vortex	4.67	0.739	swim	20.567	22.979
vpr	17.989	0.313	wupwise	3.484	4.625

Table 8.3: Per Benchmark Cache Miss Statistics

To study the dependence processor performance on the out-of-order issue of memory instructions, we statically vary the size of the VLSQ. We choose the size of the VLSQs such that they provide maximum global disorders of 0, ±2, ±4, ±8, ±16, and ±32. For example, a VLSQ of size of 1 implies in-order issue of memory instructions and ensures a maximum global disorder of 0. A VLSQ of size infinity (labeled *Inf*) is a traditional processor with a VLSQ size equal to the appropriate physical load/store queue size as shown in Table 8.1.

8.3 Effects of Increased Out-of-Order Capability

With realistic issue widths, stride prefetching techniques, and controlled load speculation, Figure 8.3 illustrates the pitfalls in the memory subsystem as they scale with increased reorder buffer sizes. For the SPEC2000 suite of benchmarks, we show the replay trap frequency (in terms of traps/1000 instructions), the replay trap overhead, the increase in the

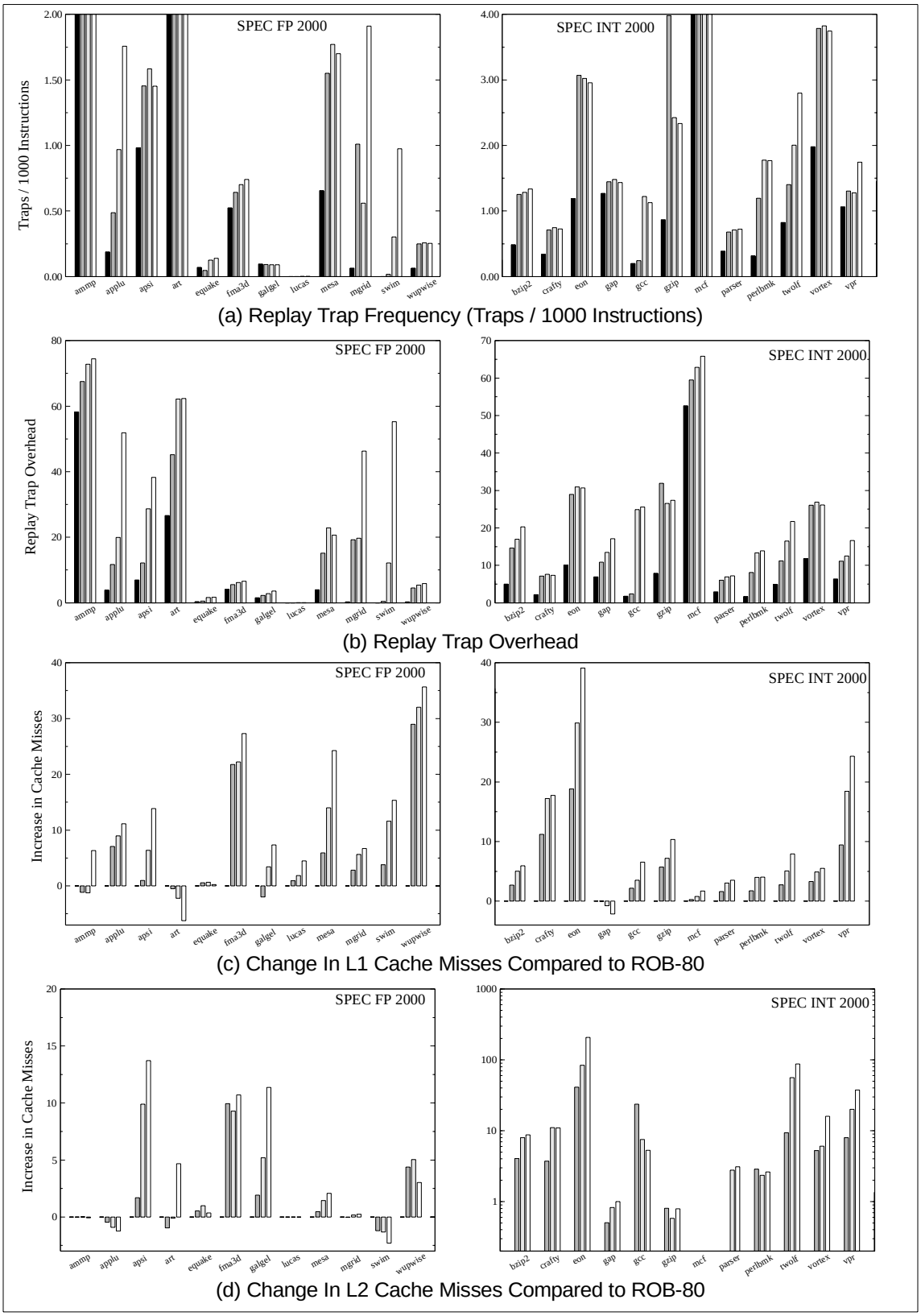


Figure 8.3: Effects of Increased ROB sizes. (a) Replay Trap Frequency (b) Replay Trap Overhead (c) DL1 Cache Misses (d) L2 Cache Misses

L1 cache misses and the increase in the L2 cache misses when compared to an 80-entry reorder buffer. By increasing the reorder buffer size from 80 to 512 entries, we observe a factor of 1-600 increase in the replay trap frequency correlating with an increase in trap overhead by as much as 50%. We also observe that increasing the reorder buffer size from 80 to 512 entries also negatively impacts an application's cache locality by increasing the total number of L1 cache misses by 5–40% and the number of L2 cache misses by 5-120%. For each benchmark we also present the number of cache misses per thousand instructions for an 80-entry ROB in Table 8.3. To clarify, a "cache miss" is one that misses both in the data cache and the miss status holding registers (MSHRs) [37]. From Figure 8.3, we observe that, while the negative effects of out-of-order execution existed for only a small fraction of the time with small reorder buffers, eliminating other sources of stalls by increasing the out-of-order capability exposes these negative effects to represent significant overhead. Since recent research and industry trends are focusing on increasing out-of-order capability [5, 7, 30, 31, 38, 51, 55, 63, 74], with the results from Figure 8.3 in mind, we believe it is imperative that the frequency of traps and the number of cache misses be reduced so that future high performance processors can realize the full potential of more complex out-of-order designs. With these results in mind, we now present the use of memory instruction windowing and its impact on replay traps, cache misses, performance and power consumption.

8.4 Windowing Results

8.4.1 Replay traps

When the reorder buffer is increased from 80 to 512 entries, less than one-third of the total number of memory instructions executed are issued in actual program order. In some benchmarks such as mgrid and swim, less than 10% are issued in actual program order. We observe that the rest of the memory instructions are either issued early or late due to functional unity latency, cache miss latency, or memory latency. This significant degree of reordering suggests that replay traps can (and we show that they do) become a tremendous source of performance and energy overhead with increasing out-of-order capability. To illustrate this, Figure 8.4 shows the total number of traps per 1000 instructions (trap frequency) and the percent of total overhead due to replay traps. We remind the reader that replay trap overhead is tracked as the total amount of work wasted do to the occurrence of replay traps. The data is averaged for all benchmarks of the SPEC2000 suite. The x-axis shows the different Alpha configurations (Alpha-80, Alpha-128, Alpha-256 and Alpha-512), and the different VLSQ sizes (*Inf*-1). We remind the reader that an “infinite” VLSQ is equivalent to the traditional implementation of a load/store queue.

In Figure 8.4(a), considering only the traditional implementation of a load/store queue, i.e. only the bars labeled *Inf*, we observe that replay traps become an important source of performance overhead and wastage of energy with increased out-of-order capability.

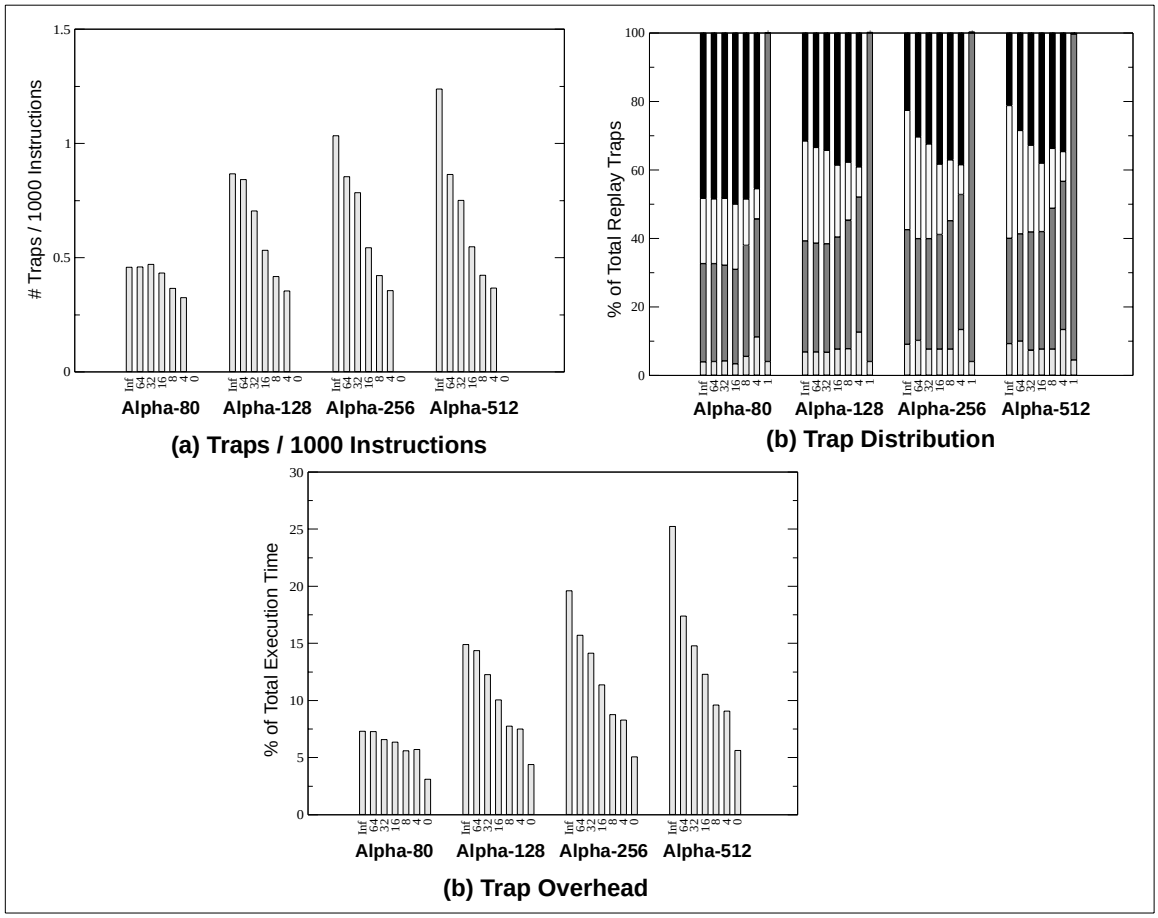


Figure 8.4: Effect of VLSQs on Replay Traps. The figure shows that VLSQs reduce (a) the frequency of traps by a factor of two to 30 and (b) the total execution time lost in traps by 10–45%.

Increasing the ROB size from 80 to 512 entries decreases the frequency of replay traps on average by a factor of 3, meaning that increasing the out-of-order capability can cause an increase in trap frequency by 300%. However, as observed from Figure 8.3, we observe individual benchmarks that suffer from replay traps by more than a factor of 3, in some case as much as a factor of 50 or more. To provide understanding on the replay traps that are most common, Figure 8.4(b) provides the distribution of the occurrences of different replay traps as a percent averaged across all benchmarks. From the figure, we observe that, across all reorder buffer sizes the occurrence of different replay traps is roughly equally distributed. Thus, even though our benchmarks are single-threaded and do not require the memory

consistency replay traps, the equal distribution of these replay trap across all benchmarks implies that they are equally important when increasing out-of-order capability.

As mentioned earlier in the dissertation, the mechanisms for handling replay traps requires the pipeline to be flushed and instructions to be re-fetched and re-executed from the replay trap causing instruction. It is intuitive that the overhead in performance and energy for flushing and re-fetching an entire window of instructions can become extremely high due to the amount of work that needs to be redone. Our studies show that, on average, increasing the out-of-order capability increases the total number of instructions flushed by a factor of two to 300. From Figure 4(c), we observe that the increase in trap frequency translates into on average 7–25% of total overhead due to replay traps. These results reveal that even though a processor can extract maximum possible ILP, too much out-of-order capability can cause the processor to spend an enormous amount of time (and energy) duplicating work that had already been done before.

Clearly, we observe the necessity for reducing the degree by which memory instructions are issued out-of-order. With this in mind, Figure 4(a) also shows that the use of VLSQs can reduce the frequency of traps between instructions by a factor of two to 30. This correlates with a reduction in the total number of instructions flushed by 50–200% and a reduction in total execution time lost by 5–20% on average as shown in Figure 4(c). From the figure, we observe that maximum benefits come from smaller VLSQs. Since smaller VLSQs reduce the reordering of memory instructions, we observe a clear correlation between the reordering of memory instructions and the trap frequency and overhead. Thus, we can conclude that the use of VLSQs can reduce the frequency of replay traps and this can

Benchmark	Trap Frequency	L1 Access Behavior	L1 Miss Behavior	L2 Miss Behavior	CPI Behavior
INTEGER BENCHMARKS					
bzip2	—	↓	↓	U	↑
crafty	↓	↓	↓	U	—
eon	↓	↓	↓	↓	U
gap	—	↓	↑	↓	↑
gcc	↓	↓	↓	↓	↑
gzip	↓	↓	↓	↓	↑
mcf	↓	↓	↓	↑	↑
parser	↓	↓	↓	—	↑
perlbmk	↓	↓	↓	↑	U
twolf	↓	↓	↓	U	U
vortex	↓	↓	↓	↓	—
vpr	↓	↓	↓	↓	U
FLOATING POINT BENCHMARKS					
ammp	—	—	↓	↑	↑
applu	↓	↓	↓	↑	↑
apsi	↓	↓	↓	U	U
art	↓	↓	↑	U	↑
equake	—	↓	↑	↑	↑
fma3d	—	↓	↓	—	—
galgel	—	—	—	—	—
lucas	—	↓	↓	↑	—
mesa	↓	↓	↓	—	↑
mgrid	↓	↓	↓	U	↑
swim	↓	↓	↓	↑	↑
wupwise	↓	↓	↓	↑	U
LEGEND					
Legend:	—: No Change	↑: Increase	↓: Decrease	U: Benefits with medium VLSQs but hurts with small VLSQs	

Table 8.4: Per Benchmark Behavior of Windowing

translate into savings in energy that would otherwise be needlessly spent in re-fetching and re-executing instructions flushed.

To illustrate the behavior of windowing for the different benchmarks, Table 8.4 illustrates the behavior of each benchmark with windowing. For the different metrics of trap frequency, L1 accesses and misses, L2 misses, and performance, the table illustrates the windowing behavior of the benchmark with symbols such as "↑", "↓", "—", and "U". The "↑" symbol implies that the windowing causes the appropriate metric to increase with

smaller virtual load/store queues, the "↓" symbol implies that windowing causes the appropriate metric to decrease with smaller virtual load/store queues, the "—" symbol implies that windowing causes the metric to have no change in behavior with smaller virtual load/store queues, and finally the "U" symbol means that windowing decreases the metric till some "medium" size of the virtual load/store queue beyond which windowing causes the metric to increase with smaller virtual load/store queues. Across all benchmarks, we observe that windowing causes the trap frequency to decrease. Only 7 of the benchmarks present no decrease in the frequency of traps from windowing. We observe that these benchmarks observed no benefit from windowing due to the fact that the bulk of the traps they experienced were those that occurred even with the in-order issue of instructions, i.e. the load-miss-load and wrong-size replay trap. Otherwise, for the remaining applications we observed reductions in trap frequency by factors of 2-600. Thus, we observe that reducing the reordering of memory instructions reduces the frequency of replay traps and the associated trap overhead.

8.4.2 Cache behavior

Figure 8.5 shows the cache behavior in terms of change in L1 cache accesses, L1 cache misses, and L2 cache misses averaged over all benchmarks of the SPEC2000 suite. The data is graphed as the percent change in cache accesses or misses normalized to the Alpha-80 configuration with an "infinite" VLSQ, i.e. each bar graph in the figure is normalized to the first configuration (Alpha-80-Inf). From the figure, considering only traditional load/store queues (first bar in each configuration), we observe that increasing the out-of-

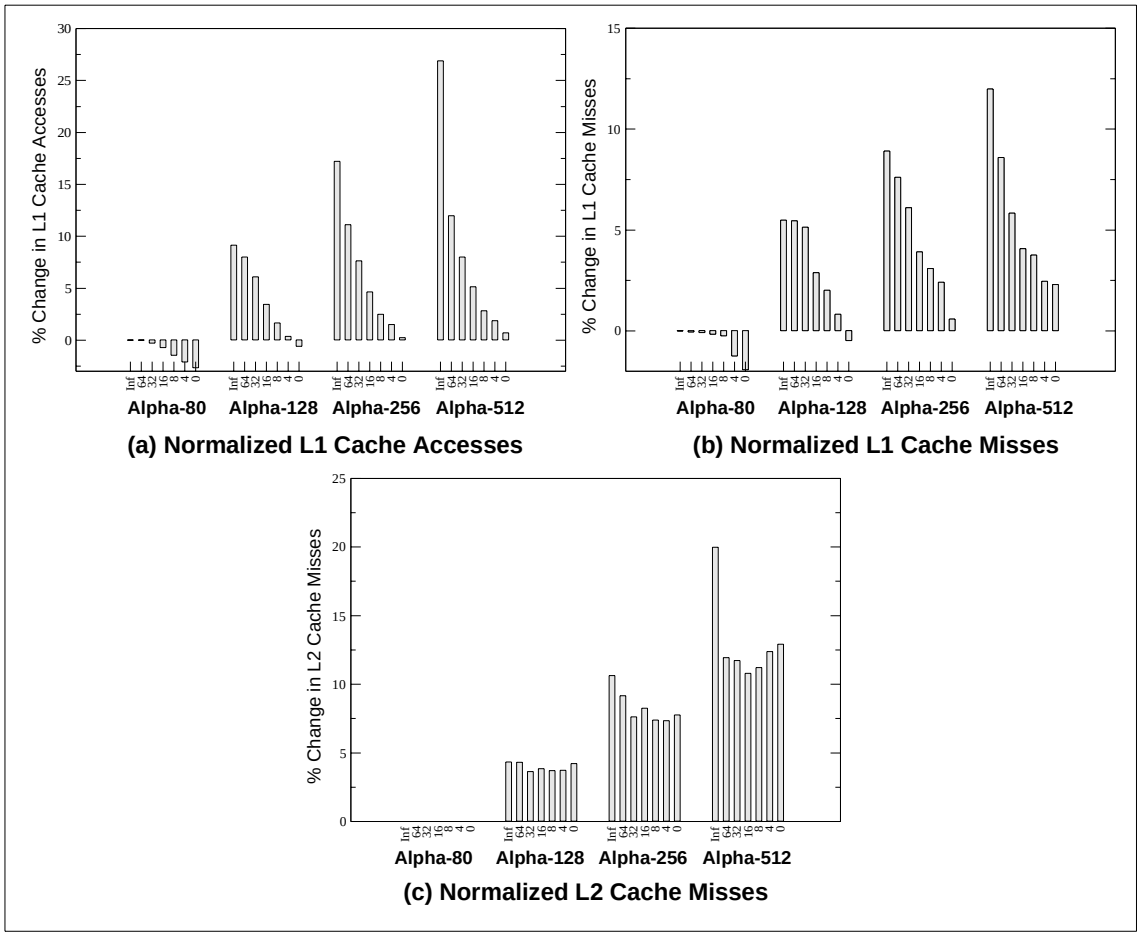


Figure 8.5: Effect of VLSQs on Cache Behavior. VLSQs reduce (a) the number of L1 cache accesses by 5-60% and (b) the number of L1 cache misses by 5-15%.

order capability can increase (on average) the total number of L1 cache accesses by up to 28%, the total number of L1 cache misses by up to 12%, and the total number of L2 cache misses by up to 20%. Again, we observe a direct correlation between smaller VLSQ sizes and cache accesses and misses. We observe that smaller VLSQs can reduce the total number of cache accesses (on average) by 3–30% and the total number of cache misses by 5–15%. These findings reveal that VLSQs can also aid in reducing the unnecessary wastage of energy in the data caches.

An important observation that can be made from the use of VLSQs is the large overhead of speculation. When comparing a VLSQ of size 1 with a VLSQ of size Inf, we observe that

speculation adds on average 5-30% extra cache accesses and about 5-10% additional cache misses in both the L1 and L2 data caches. Similarly, from Figure 8.4(b) we observe that speculation causes a 5-15% increase in overall trap overhead. From the figures we observe that for the different reorder buffer sizes, the overheads are larger in processors with larger reorder buffer sizes. With the large differences between the in-order and out-of-order issue of memory instructions, we observe that the windowing of memory instructions is another approach to reduce the amount of speculative waste that comes with larger reorder buffers. Such reduction in speculative waste is welcomed especially with the growing power envelopes of modern microprocessors.

As before, we refer the reader to Table 8.4 on the behavior of windowing on the L1 and L2 cache for the different benchmarks. Since windowing reduces the total amount of speculation in the system, we observe that all benchmarks observe a decrease in the total number of L1 cache accesses. When comparing the L1 and L2 cache misses, we observe that windowing behaves differently for different benchmarks. Some benchmarks have no effect on cache misses, for most of the benchmarks windowing reduces the total number of cache misses, and for a remaining few benchmarks, windowing causes an increase in the number of cache misses. A decrease in the number of cache misses is because reducing the reordering of memory instructions eliminates the number of conflict misses due to early execution and speculative execution. On the other hand, an increase in the number of cache misses can be explained by the fact that windowing lessens the benefits of speculative data prefetching.

8.4.3 Relating Global Disorder and Negative Effects

Figure 8.6 presents the relationship between the negative effects in the memory subsystem with average global disorder. As with all the metrics in this study, the average global disorder is averaged across all the benchmarks as well. The figure illustrates the average global disorder plotted against trap frequency, increase in L1 cache misses when compared to an infinite sized VLSQ with an 80-entry ROB, and the increase in L2 cache misses when compared to an infinite sized VLSQ with an 80-entry ROB. Each graph presents five line graphs. The dashed line compares global disorder and the different metrics with the increase in ROB sizes. The solid line graphs each represent average global disorders with windowing for the four different ROB sizes: 80, 128, 256, and 512.

Based on the figures, we observe that global disorder correlates well with the trap frequency and the increase in the number of L1 cache misses. From the figure, we observe that a decrease in the global disorder decreases the total amount of degradation in terms of replay traps and total L1 cache misses. However, we observe that this is not true for the L2 data cache. The degradation in cache performance for the different ROB sizes with global disorder of 0 can be by as much as 10% in a ROB-512 when compared to a ROB-80 configuration. Further investigation into the reasoning for this behavior was attributed to the stride prefetcher. The stride prefetcher issues prefetches whenever it detects a stride and there is available bandwidth to the memory subsystem. As memory instructions are serialized and forced to be issued in-order, the available bandwidth to the memory system increases. Thus, the stride prefetcher can continue issuing prefetches, and this can cause

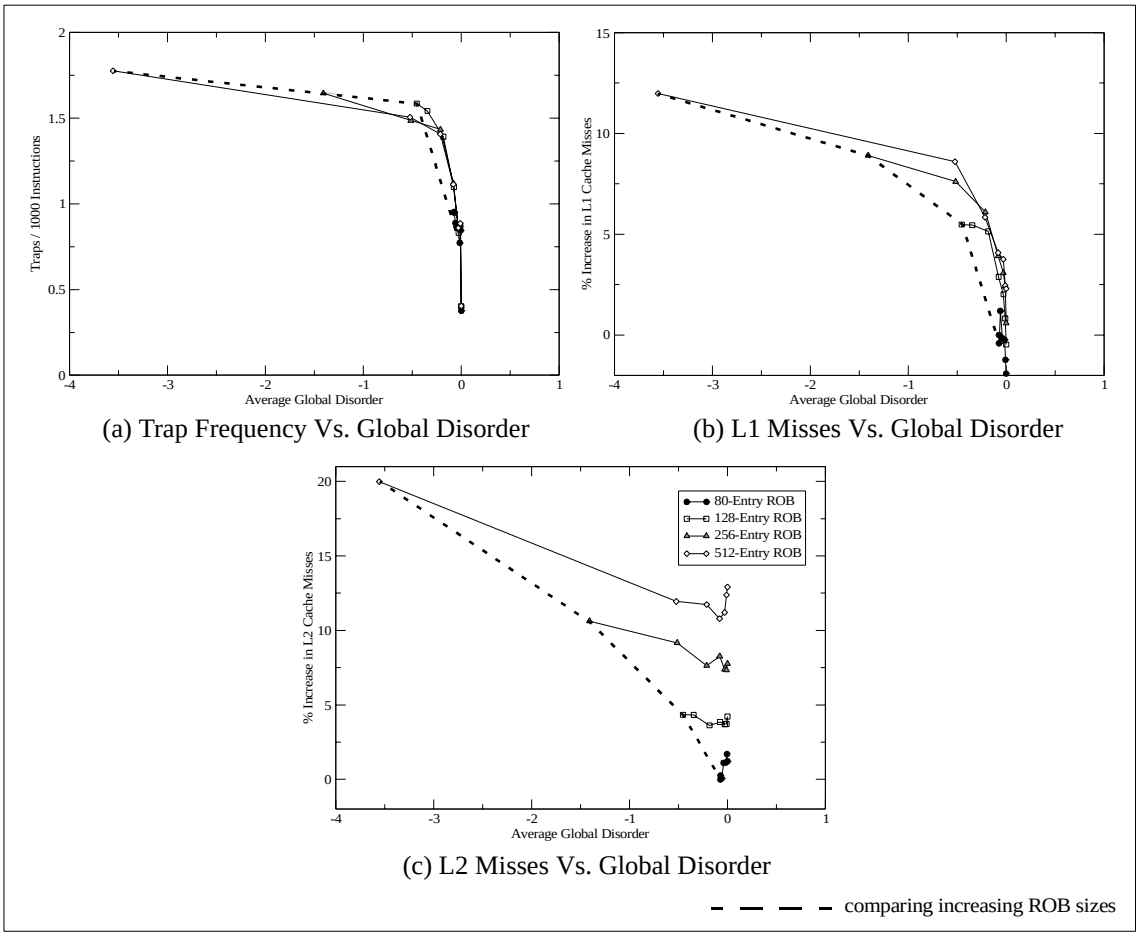


Figure 8.6: Global Disorder Vs. Negative Effects.

data pollution hence causing the increase in the total number of cache misses when comparing global disorder values of 0 across the four different ROB sizes. We observe that for the L2 cache, allowing a certain degree of reordering by using VLSQs of 16/32 controls the stride prefetcher from running ahead.

8.4.4 Power

With increased trap frequency, the components of a processor that are exercised heavily are the fetch, map, and execution units. In a similar manner, increases in cache accesses and misses appropriately require the respective caches to access and fill the required data.

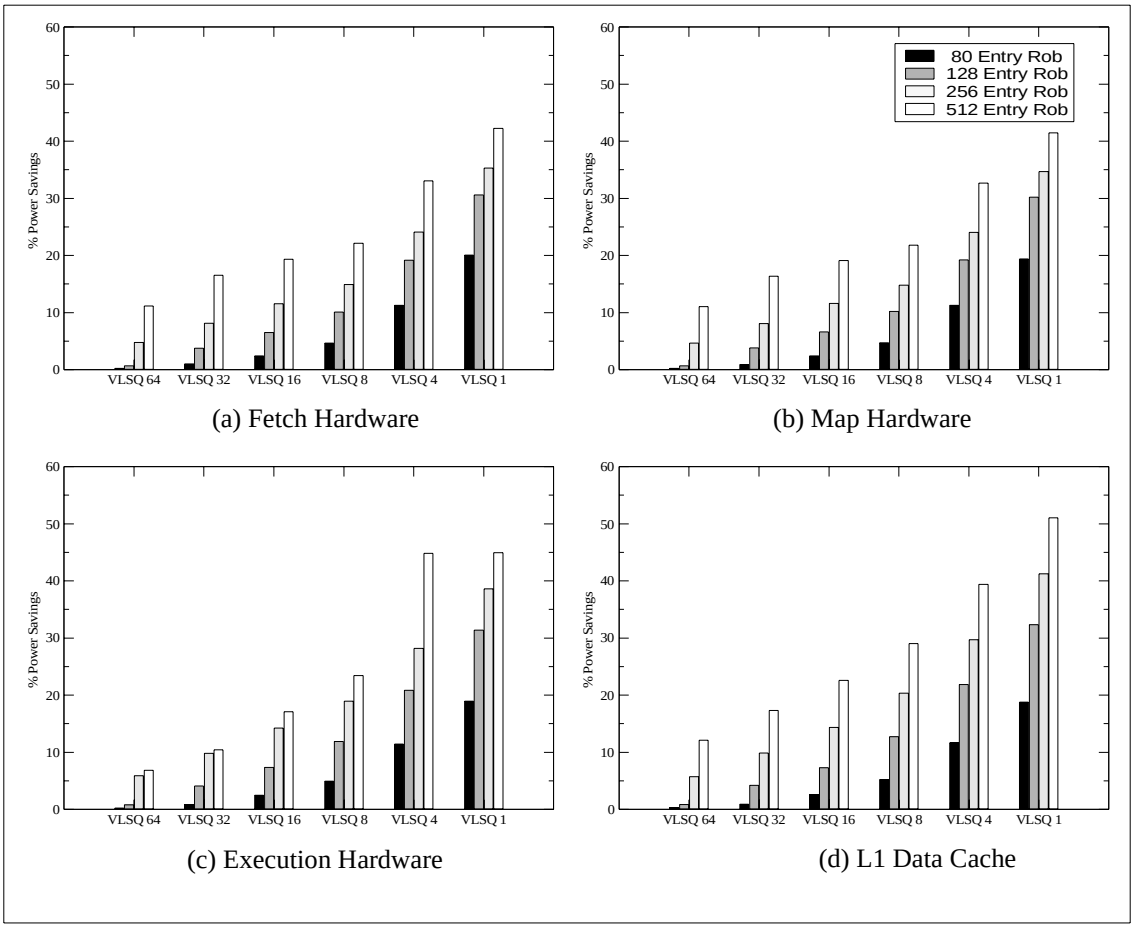


Figure 8.7: Average Power Savings Using VLSQs. By reducing the reordering of memory instructions, VLSQs eliminate the needless amount of energy dissipated in re-fetching and re-executing instructions, and speculative cache accesses. This translates into power savings of 5-50% in the fetch and rename hardware, 10-40% in the execution hardware, and 5-50% in the data cache.

Figure 8.7 shows the savings in average power consumed, normalized to the traditional load/store queue for each of the following components: fetch hardware, mapping hardware, execution hardware, and the L1 cache.

By reducing the reordering of memory instructions we observed a reduction in trap frequency by a factor of two to 30 and a reduction in the total number of instructions flushed by 50–200%, all of which translates into a total reduction in replay trap overhead by 10–45%. This means that the fetch, map, and execute units spend less energy duplicating work that had already been done before. From the figure, we observe that a reduction in the total number of instructions flushed translates into average power savings ranging from 5–50% in

the fetch and rename unit, and 10–40% in the execution unit. Such substantial savings in power are important especially since the total power of all hardware associated with the fetch, map, and execute units contribute to roughly half (46%) of an Alpha 21264's total power consumption [28]. Additionally, we observe that reducing the reordering of memory instructions reduces the average power consumed in the L1 cache by 10–50%. Again, we observe that these savings in the caches are substantial since the on chip data cache contributes about 15% of an Alpha 21264's total chip power [28].

8.4.5 Performance

Figure 8.8(a,b) shows the performance graphs with the different benchmarks and ROB sizes on the x-axis and cycles per instruction (CPI) on the y-axis. As before, CPI is classified into stall cycles where memory instructions could not retire due to memory latency (black), stall cycles where instructions could not retire because they either had not been issued or had not yet finished execution due to ALU latency (medium grey), and stall overhead cycles due to recovering from branch mispredicts and replay traps (light grey). The ALU and memory components of CPI are computed by measuring the number of cycles the retire stage stalls because it could not retire an ALU or memory instruction. The overhead portion was computed by taking the difference between the total number of cycles and the sum of the ALU and memory instruction stall cycles in the retire stage. Note that, due to overlaps between memory and ALU stall cycles, the overhead portion of CPI is not the same as the total execution time lost in replay traps.

From Figure 8.8(a,b), we observe that for some benchmarks, increased out-of-order capability overcame the sources of performance degradation to provide 30–40%

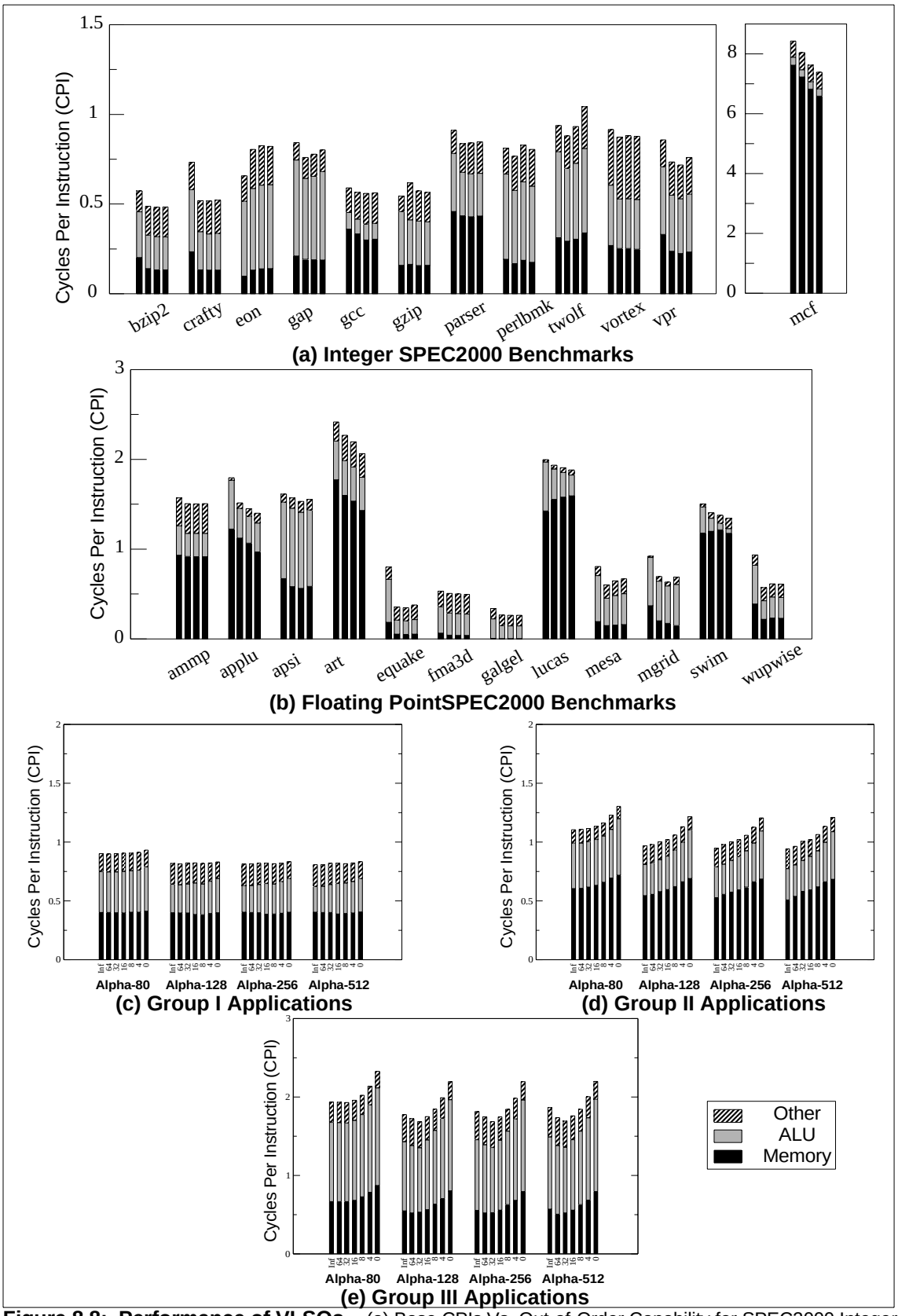


Figure 8.8: Performance of VLSQs. (a) Base CPIs Vs. Out-of-Order Capability for SPEC2000 Integer and Float-Point benchmarks (b,c,d) Effect of VLSQs on processor performance distributed into categories.

Group	Benchmarks
Group I	crafty, vortex, fma3d, galgel, lucas
Group II	bzip2, gap, gcc, gzip, mcf, parser, ammp, applu, art, equake, mesa, mgrid, swim, vpr
Group III	eon, perlbmk, twolf, apsi, wupwise

Table 8.5: Benchmark Categories Based on Performance

improvement in performance. We observe that for such benchmarks the bulk of the performance improvements is achieved by scheduling memory instructions early, thus hiding/overlapping memory latency with useful work. This is evident due to the fact that the memory stall portion of CPI (black) decreases with increased out-of-order capability. On the other hand, we observe that most benchmarks suffer from a performance degradation with reorder buffer sizes of 256 or more. For such applications we observe one or more of the three components of CPI increasing. An increase in the memory portion can be correlated to the increase in cache misses, while the increase in the ALU and overhead portions of CPI can be correlated with an increase in replay traps.

Figure 8.8(c,d,e) show the results of varying the VLSQ size as the average CPI for all benchmarks, categorized as Group I, Group II, and Group III sets. As listed in Table 8.5, the benchmarks included in Group I are: crafty, vortex, fma3d, galgel, and lucas; the benchmarks included in Group II are: bzip2, gap, gcc, gzip, mcf, parser, ammp, applu, art, equake, mesa, mgrid, swim and vpr; finally, the benchmarks included in Group III are: eon, perlbmk, twolf, apsi, wupwise. Group I applications show no remarkable change in performance with reduced VLSQ sizes. This is because the Group I applications are *memory-instruction independent,* that is they are more compute-intensive. We infer this from the fact that the memory stall portion of CPI (black) does not vary with decreased

VLSQ size. Therefore, for such applications, we can gain maximum power savings of 15–50% by issuing all memory instructions in actual program order (as shown in Figure 8.7).

For the Group II and III benchmarks, we observe two different behaviors with smaller VLSQs. First, the memory latency portion of CPI increases. This behavior can be expected because the use of a VLSQ reduces the reordering of memory instructions at the expense of memory ILP. This is apparent because reducing the size of the VLSQ causes an increase in the memory stall portion (black) of CPI. Thus, for applications that are *memory-instruction dependent* (or memory intensive), we observe a 15–30% degradation in performance with decreased VLSQs. However, for such benchmarks (Group II) we observe that VLSQ sizes of 16 and 32 are within 2–5% of the traditional load/store queue. On the other hand, for the group III benchmarks, we observe that medium VLSQs can reduce the negative overheads in the memory subsystem to provide net performance improvements of up to 6% when compared to a traditional load/store queue. For such applications, we observe that the performance improvement is achieved via a decrease in the memory, ALU and the overhead portions of CPI implying that reducing the reordering of memory instructions is successful in reducing the negative overheads in the memory subsystem. As mentioned earlier, a reduction in the memory portion of CPI implies reduction in cache misses and a reduction in the other portions of CPI implies a reduction in the replay trap overhead. Across all three behaviors of windowing, we observe that VLSQ sizes of 16 or 32 are optimal and can not only improve performance but also lead to power savings of 10–22% (as shown in Figure 8.7).

Second, for the Group II and III benchmarks, we also observe that issuing of memory instructions in program order (VLSQ of size 1) can cause a factor of 2 increase in overhead

portions when compared to the traditional load/store queue. We relate this to the occurrence of replay traps. As mentioned in Section 4.1 of this dissertation, replay traps can still occur even if memory instructions are issued in program order. Besides the load-miss-load and wrong-size replay traps, we observed that a load-store replay trap can also occur with the in order issue of memory instructions in the event of a mispredict in the logic that tracks dependencies between loads and earlier stores. For example, a load-store replay trap occurs if a store and its memory-dependent load are simultaneously issued to execute in the same cycle. Since the load and store compute their effective addresses at the same time, store-to-load forwarding cannot occur in the same cycle. Thus, the load instruction must be replayed. The reasoning for the larger overheads with smaller VLSQs is that a replay trap can become expensive if the reorder buffer is full, and this scenario is very likely when combining decreased VLSQ sizes and memory intensive benchmarks. This is because of the latencies associated with the delayed issue of load instructions to the cache and memory subsystem with smaller VLSQs. Thus, in the event of a replay trap the overhead of re-fetching and re-executing an entire window of instructions can become expensive, especially with larger reorder buffer sizes.

Finally, from Figure 8.8(c,d,e), we also observe that out-of-order processors need only a window of 16 or 32 memory instructions to select and issue from. We observe that selecting and issuing to execute memory instructions outside of a window of 32 instructions can unnecessarily waste time and energy recovering from replay traps as well as needless data cache accesses and misses.

8.5 Summary

In this chapter we investigated the use of the network communication concept of *windowing* to determine the degree to which out-of-order execution of memory instructions determines processor and memory performance. By introducing a virtual window into the existing load/store queue, we restrict the out-of-order scheduler to issue only those memory instructions that reside within the virtual window. Those memory instructions that reside outside the virtual window must wait till the virtual window slides over them. The virtual window of instructions within the load/store queue is essentially a virtual load/store queue (VLSQ). By controlling the size of the VLSQ one can control the degree by which memory instructions are issued out-of-order. For example, a VLSQ of size 1 would imply that memory instructions be issued in program order, while a VLSQ that is infinite in size is the same as traditional load/store queues. Our simulations for various VLSQ sizes reveal that the negative effects in the memory subsystem are directly proportional to the size of the VLSQ. The larger the virtual load/store queue, the larger the negative effects in the memory subsystem, and the smaller the load/store queue the smaller the negative effects. We observe that reducing the reordering of memory instructions can reduce the number of cache misses and replay traps to provide power savings. We observe that reducing the reordering of memory instructions can either benefit, provide no change, or hurt performance. We observe that across all benchmarks, a VLSQ of size 16/32 is sufficient to reduce the negative effects in the memory subsystem with degradation in performance of 0-5%.

CHAPTER 9 *Conclusions and Future Work*

9.1 Conclusions

9.1.1 Dissertation In a Nut Shell

The use of 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. The work presented in this dissertation shows that continuing to increase out-of-order aggressiveness does not buy any improvements in processor performance, in fact it can actually degrade processor performance. By varying the aggressiveness of an out-of-order core in terms of reorder buffer sizes, issue queues, load/store queues, and using a realistic DRAM system model, the work presented in this dissertation brings to light problems present in real systems that many previous simulation-based studies have not addressed.

We observe that continuing to increase out-of-order aggressiveness to improve processor performance will come at the cost of a degradation in the performance of the memory subsystem. Specifically, we observe that increased out-of-order capability conflicts with the memory-ordering requirements of a processor, requiring the processor to initiate frequent traps to enforce correct state. Furthermore, we also show that 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 system.

To gain insight on the reason for the degradation in the performance of the memory subsystem, we measured the degree to which memory subsystem performance relies on out-of-order execution By using the network communication concept of *windowing*, we restricted the degree by which memory instructions are reordered. By restricting the reordering of memory instructions, our study revealed that memory instructions issued out-of-order are the primary reason for the increase in the frequency of replay traps. Furthermore, the out-of-order issue of memory instructions is also responsible for both the constructive and destructive references to the data cache. The destructive references as a result of increased speculation is the primary reason for the increase in the number of cache misses when comparing an aggressive out-of-order system to a lesser aggressive out-of-order system.

9.1.2 Detailed Overview

Contrary to existing simulation based studies, this dissertation shows that larger instruction windows and reorder buffers do not necessarily provide significant improvements in performance. With the use of detailed models of the processor and DRAM system, we show that improvements in processor performance saturates beyond a 128-entry reorder buffer. In fact, we observe that it can actually degrade processor performance. Furthermore, this dissertation presents a non-intuitive problem associated with increasing out-of-order aggressiveness—the reordering of memory instructions can cause a degradation in the performance of the memory subsystem. Specifically, we show that increasing out-of-order aggressiveness in terms of reorder buffer sizes increases the frequency of replay traps and the total number of data cache misses. We show that while

these negative effects existed for only a fraction of the time in lesser aggressive systems, removing other sources of stalls by increasing out-of-order capability represents these negative effects to be sources of significant performance loss and unnecessary power dissipation. We believe that reducing these overheads with increasing out-of-order capability is important for future high performance systems to reap the true benefits of increased out-of-order capability.

In efforts to determine the source of the performance loss in the memory subsystem, we conducted studies to determine the degree to which memory subsystem performance relies on out-of-order execution. This was done by varying the issue-logic configurations while keeping the processor constant. Specifically, we defined three issue-logic configurations: ALU-in/MEM-in, ALU-out/MEM-in, and ALU-out/MEM-out. The ALU-in/MEM-in configuration issues all instructions in program order; the ALU-out/MEM-in configuration restricts memory instructions to be issued to the memory system in-order while allowing the out-of-order issue of ALU instructions; and the ALU-out/MEM-out configuration allows for the out-of-order issue of both ALU and memory instructions. When comparing the performance degradation in the memory system for the three issue-logic configurations with increased out-of-order capability, we observe that the transition from the ALU-out/MEM-in issue logic configuration to the ALU-out/MEM-out issue logic configuration causes a significant amount of degradation in the performance of the memory subsystem. Since the only difference between these two issue-logic configurations is the out-of-order issue of memory instructions and a larger degree of speculation, we conclude that both speculation and the out-of-order issue of memory instructions are responsible for the degradation in the performance of the memory subsystem.

Having determined that the degradation in the performance of the memory subsystem is due to the reordering of memory instructions, we defined a metric called *disorder* to measure the degree by which memory instructions are issued out-of-order to the memory subsystem. We define disorder to be of two types: global disorder and local disorder. Global disorder is the degree by which memory instructions are issued out-of-order when compared to fetch-order. Local disorder on the other hand is the degree by which memory instructions are issued out-of-order when compared to other memory instructions issued in the same or previous cycle. Based on our studies, we observe that applications can have a large amount of global disorder with about 10-20% of memory instructions being issued to the memory subsystem on time when increasing reorder buffer sizes. We observe that local disorder with increasing out-of-order aggressiveness is low illustrating the increase in speculation.

Next we showed that there exists a good correlation between the global disorder metric and the negative effects in the memory subsystem. We observe that increase in global disorder is the primary reason for the degradation in the performance of the memory subsystem: the larger the global disorder the worse the performance of the memory subsystem. Based on this correlation, we conclude that reducing the global disorder by throttling the out-of-order issue of memory instructions can mitigate the unexpected negative effects with increasing out-of-order aggressiveness.

To throttle the degree by which memory instructions are issued out-of-order, we use the network communication concept of windowing. With windowing, the out-of-order instruction scheduler is limited to schedule only those memory instructions that lie within a window of instructions. The window is determined by two new pointers into the existing load/store queue, a head pointer that points to the beginning of the window and a tail pointer

that points to the end of the window. The head pointer points to the oldest non-issued memory instruction and the tail pointer points to the end of the instruction window. The window of memory instructions essentially acts like a virtual load/store queue (VLSQ) embedded into the existing load/store queue. The use of the VLSQ limits the number of instructions available to the select and issue logic. Thus, the instruction scheduler can issue only those memory instructions that reside within the virtual load/store queue. The virtual window slides onto younger memory instructions only when the instruction at the virtual head is issued. Thus, younger memory instructions (that are ready to be issued) can only be issued when the virtual window slides onto them. By restricting the number of memory instructions visible to the instruction scheduler, we are able to reduce the reordering of memory instructions. The smaller the size of the virtual window, the smaller the degree by which memory instructions are reordered. The larger the size of the virtual window, the larger the degree of memory instruction reordering.

By using the windowing concept and statically varying the size of the VLSQ, we studied the degree to which memory subsystem performance relies on the out-of-order execution of memory instructions. By changing the size of the VLSQ we vary the degree by which memory instructions are issued out-of-order. We show that the degradation in the memory subsystem is directly proportional to the total global disorder allowed, i.e. the size of the virtual load-store queue. The smaller the size of the virtual load/store queue, the smaller the global disorder, and the lower the frequency of replay traps and cache misses.

We observe that reducing the reordering of memory instructions reduces the frequency of replay traps, cache accesses, and cache misses. These reductions categorize applications into three different behaviors. Applications either benefit from a reduction in the reordering

of memory instructions, they have little or no change in performance, or they hurt from the reduction in the reordering of memory instructions. In the cases where the use of windowing provided no benefits to performance, we observe benefits in power savings due to the reduction in the wasted work due to replay traps and the needless cache accesses and misses. In the cases where the use of windowing hurt performance, we observe that the degradation is primarily due to the fact that the memory subsystem relies heavily relies on the out-of-order execution of memory instructions.

9.2 Significance Of This Dissertation

We believe that this dissertation makes three important contributions to the computer architecture community:

- This is the first study that demonstrates a *non-intuitive* problem associated with the reordering of memory instructions. To our knowledge there is no existing study or published work that explicitly illustrates that the reordering of memory instructions can cause a degradation in the performance of the memory subsystem. The presentation of this problem in itself is of utmost significance: *the very mechanisms commonly used to improve performance are sources of significant performance degradation in the memory subsystem.*
- Existing work on tolerating DRAM latency have proposed novel techniques to implicitly or explicitly scale the size of the scheduling window. However, the effects of increased out-of-order capability on the memory subsystem have been vastly discounted. We observe that while the negative effects of out-of-order execution

existed for only a small fraction of the time with small reorder buffers, eliminating other sources of stalls by increasing out-of-order capability introduces unexpected side effects in the memory subsystem that represent significant overhead.

- The prior statement brings up an important point: we can no longer overlook rarely occurring events in the memory subsystem. Thus, the need for detailed execution driven simulators is of utmost importance. The incorporation of detailed memory subsystem models and a realistic DRAM model into existing simulators, e.g. SimpleScalar, can allow for the problems described in this dissertation to be observed.

Besides the above important contributions, this dissertation also places significance in the following findings:

- Workloads can perform even better once the negative effects in the memory subsystem are reduced. This implies that schemes that reduce the performance degradation in the memory subsystem can allow workloads to reap the true benefits of out-of-order execution.
- Some workloads have little or no benefit from executing memory instructions out-of-order. Since the out-of-order issue logic of high performance microprocessors is complex and consumes large amounts of power, simplification in the issue logic for memory instructions may be able to reduce power.
- Reducing the reordering of memory instructions reduces unnecessary/speculative work. For example, reducing the reordering of memory instructions can reduce unnecessary power consumed in execution units and cache accesses.

9.3 Future Work

In efforts to investigate the degradation in the performance of the memory subsystem, the work presented in this dissertation investigated the degree to which processor and memory system performance is dependent on the reordering of memory instructions. By statically varying the degree to which memory instructions are reordered, we observe that reducing the reordering of memory instructions can reduce the negative effects in the memory subsystem at the cost of processor performance. Thus, we observe that we have two mechanisms that are at odds against each other. Increasing the reordering of memory instructions can improve processor performance at the expense of a degradation in the performance of the memory subsystem. On the other hand reducing the reordering of memory instructions reduces the degradation in memory subsystem performance at the expense of processor performance.

9.3.1 Convert Distant Loads to Useful Prefetches

In attempts to improve processor performance without degrading the performance of the memory subsystem, it would also be possible to convert load instructions that reside outside of the virtual window to be issued to the memory subsystem as early as possible by converting them to prefetch instructions. Since our performance graphs showed that the primary reason for the degradation in processor performance is waiting on memory (due to the late issue of memory instructions), we propose to convert load instructions that reside outside of the virtual window to be sent as early prefetches to the last-level data cache. Once the load memory instruction resides within the virtual window, we propose to send the actual

load to the L1 data cache. Thus, by pushing the load instructions out to the memory subsystem as prefetches as early as possible can reduce processor performance loss without sacrificing memory system performance.

9.3.2 Dynamic Mechanisms for Varying VLSQ Sizes

Based on the intuition that the negative effects in the memory subsystem do not always exists, it would make sense that the reordering of memory instructions only be reduced during the phases in which the negative effects exists. Thus, a dynamic approach to reduce the reordering of memory instructions can provide for improvements in processor and memory subsystem performance. To exploit the benefits of out-of-order execution, it would be desirable that we reap the benefits of out-of-order execution in the phases where there are no negative effects while throttle the degree to which memory instructions are reordered in the phases where the negative effects are prominent. Thus, the work presented in this dissertation can be further extended by a mechanism to dynamically detect the different phases of execution and throttle the reordering of memory instructions at run time.

There are different ways to dynamically throttle the degree by which memory instructions are issued out-of-order. Hardware can monitor the trap frequency and the total number of cache misses periodically. If during a period/phase the hardware detects an increase in trap frequency or cache misses, it can dynamically throttle itself. Alternatively, since we observed a correlation between global disorder and the negative effects in the memory subsystem, the average global disorder metric can be used to throttle the degree by which memory instructions are issued out-of-order. With such a mechanism, the processor must track the average global disorder over time, if at some point in time the average global

disorder increases the processor should start throttling itself and reduce the reordering of memory instructions.

When throttling the reordering of memory instructions dynamically, it is also important to determine dynamically when to let go of the throttle and allow for memory instructions to be re-ordered again. One mechanism could be to let go of the throttle is after a pre-allotted number of memory instructions have executed or a pre-allotted number of cycles have passed.

Thus, the dynamic mechanism of throttling the degree by which memory instructions are issued out-of-order can allow for applications to exploit memory level parallelism during phases where memory instructions do not cause negative effects and control the reordering during phases where the negative effects in the memory subsystem exist.

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] S. Adve and K. Gharachorloo. “Shared Memory Consistency Models: A Tutorial.” Rice University ECE Technical Report 9512 and Western Research Laboratory Research Report 95/7, September 1995.

[5] 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.

[6] J. Baer and T. Chen. “An effect on-chip preloading scheme to reduce data access penalty.” In *Proceedings of Supercomputing*, 1991.

[7] M. D. Brown, J. Stark, and Y. N. Patt. Select-Free Instruction Scheduling Logic. In *Proc. 34th International Symposium on Microarchitecture,* December 2001.

[8] 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.

[9] H. W. Cain and M. H. Lipasti. “Memory Ordering: A Value-Based Approach.” IEEE Micro, vol. 24, no. 6, pp. 110-117, November/December 2004.

[10] B. Calder and G. Reinman. “A Comparative Survey of Load Speculation Architectures.” In Journal of Instruction Level Parallelism.” In Journal of Instruction-Level Parallelism, May 2000

[11] D. Callahan, K. Kennedy, and A. Porterfield. “Software prefetching.” In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[12] M. C. Carlisle, A. Rogers, J. H. Reppy, and L. J. Hendren. “Early Experiences with Olden.” In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 1-20, August 1993.

[13] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. “Simultaneous Subordinate Microthreading (SSMT).” In *Proceedings of the 26th International Symposium on Computer Architecture,* 1999.

[14] T. F. Chen and J. L. Baer. “A Performance Study of Software and Hardware Data Prefetching Schemes.” In *Proceedings of the 21st International Symposium on Computer Architecture.*

[15] G. Chrysos and J. Emer. “Memory Dependence Prediction Using Store Sets” In *Proceedings of the International Symposium on Computer Architecture (ISCA),* 1998.

[16] A. Cristal, J. Martinez, J. Liosa, and M. Valero. “A Case for Resource-conscious Out-of-Order Processors.” In IEEE Computer Architecture Letters, Vol. 2, Oct. 2003.

[17] I. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. “Dynamic speculative precomputation.” In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[18] R. Cooksey. *Content-Sensitive Data Prefetching.* Ph.D. Thesis, University of Colarado, Boulder 2002.

[19] 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.

[20] F. Dahlgren, M. Dubois, and R. Stenstrom. “Fixed and Adaptive Sequential Prefetching Schemes.” In *Proceedings of the 21st International Symposium on Computer Architecture*, 1994.

[21] F. Dahlgren, and R. Stenstrom. “Evaluation of hardware based stride and sequential prefetching in shared-memory multiprocessors.” In IEEE Transactions on Parallel and Distributed Systems, 1996.

[22] 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.

[23] 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.

[24] P. J. Drongowski. “Performance Tips for Alpha C Programmers.” http://h21007.www2.hp.com/dspp/files/unprotected/tru64/tips.pdf

[25] J. Dundas and T. Mudge. “Improving data cache performance by pre-executing instructions under a cache miss.” In *Proceedings of the 1997 International Conference on Supercomputing*, 1997.

[26] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessey. “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.” In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, 1990.

[27] S. Gopal, T. N. Vijaykumar, J. Smith, and G. S. Sohi. “Speculative Versioning Cache.” In IEEE Transactions on Parallel and Distributed Systems, Volume 12 Issue 12.

[28] 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.

[29] J. L. Henning. “SPEC CPU2000: Measuring CPU Performance in the New Millenium”. IEEE Computer, 33(7):28-35, July 2000.

[30] 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.

[31] D. Henry, B. Kuszmaul, and V. Viswanath. “The Ultrascalar Processor.” In *Proceedings of the 20th Anniversary Conference on Advanced Research in VLSI*, 1999.

[32] A. Jaleel and B. Jacob. “Using Virtual Load/Store Queues (VLSQs) to Reduce the Negative Effects of Reordered Memory Instructions.” In *Proceedings of the*

International Symposium on High Performance Computer Architecture (HPCA), San Francisco, CA, 2005.

[33] A. Jaleel and B. Jacob. “In-line Interrupt Handling For Software Managed TLBs.” In *Proceedings of the International Conference on Computer Design (ICCD)*, 2001.

[34] D. Joseph and D. Grunwald. “Prefetching using Markov predictors.” In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA),* 1997.

[35] N. P. Jouppi. “Improving direct-mapped cache performance by the addition of a small full-associative cache and prefetch buffers.” In *Proceedings of the 17th International Symposium on Computer Architecture (ISCA),* 1990.

[36] J. Kin, M. Gupta, and W. Mangione-Smith. “Filtering Memory References to Increase Energy Efficiency.” In IEEE Transactions on Computers, Vol 43, No. 1, January 2000.

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

[38] 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.

[39] C. K. Luk. “Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors.” In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA),* 2001.

[40] C. K. Luk and T. C. Mowry. “Compiler based prefetching for recursive data structure.” In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[41] T. Lyon, E. Delano, C. McNairy, and D. Mulla. “Data Cache Design Considerations for the Itanium 2 Processor.” In *Proceedings of the International Conference on Computer Design*, 2002.

[42] T. Moreshet, and R. I. Bahar. “Effects of Speculation on Performance and Issue Queue Design.” In IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2004.

[43] T. Moreshet and R. I. Bahar. “Complexity-Effective Issue Queue Design Under Load-Hit Speculation.”. In *Proceedings of the Workshop Complexity-Effective Design*, 2002

[44] D. Mosberger. “Memory Consistency Models”. Technical Report 93/11.

[45] M. Moudgill, J. Wellman, and J. Moreno. “An Approach for quantifying the impact of not simulating mispredicted paths or An Approach for quantifying the impact of mispredicted paths.” Presentation at IBM T. J. Watson Research Center, October 1997.

[46] T. C. Mowry, M. S. Lam, and A. Gupta. “Design and evaluation of a compiler algorithm for prefetching.” In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[47] O. Mutlu, H. Kim, D. Armstrong, and Y. Patt. “Cache Filtering Techniques to Reduce the Negative Impact of Useless Speculative Memory References on Processor Performance.” In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing* (SBAC-PAD'04), 2004.

[48] O. Mutlu, H. Kim, D. Armstrong, and Y. Patt. “Runahead Execution: An Effective Alternative to Large Instruction Windows”. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA)*, February, 2003.

[49] O. Mutlu, H. Kim, D. Armstrong, and Y. Patt. “Understanding the Effects of Wrong-Path Memory References on Processor Performance.” In *Proceedings of the 3rd Workshop on Memory Performance Issues (WMPI).*

[50] 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.

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

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

[53] N. Oren. “A Survey of Prefetching Techniques.” July 18, 2000.

[54] V. Pai, P. Ranganathan, and S. Adve. “The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology.” In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 1997.

[55] 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.

[56] Y. Patt. “First, Let’s Get the Uniprocessor Right”, MicroDesign Resources, Microprocessor Report, August, 1996.

[57] J. Pierce, and T. Mudge. “The Effect of Speculative Execution on Cache Performance.” In *Proceedings of the International Parallel Processing Symposium*, APril 1994, Cancun, Mexico.

[58] X. Qiu and M. Dubois. “Tolerating Late Memory Traps in Dynamically Scheduled Processors.” IEEE Trans. Computers, June 2004.

[59] P. Ranganathan, V. Pai, and S. Adve. “Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models.” In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, June 23-25, 1997, Newport, RI, USA.

[60] J. Rivers, E. Tam, E. Davidson. “On Effective Data Supply for Multi-Issue Processors.” In *Proceedings of the International Conference on Computer Design (ICCD)*, 1997.

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

[62] R. Sites. “It’s the Memory Stupid!” Microprocessor Report, pages 19-20, August 5, 1996.

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

[64] A. J. Smith. “Sequential Program Prefetching in Memory Hierarchies.” *Computer*, Dec. 1978.

[65] J. E. Smith and A. R. Pleszkun. “Implementation of precise interrupts in pipelined processors.” In *Proc. 12th Annual International Symposium on Computer Architecture (ISCA’85)*, Boston MA, June 1985, pp. 36–44.

[66] J. Smith. and G. Sohi. “The Microarchitecture of Superscalar Processors.” In Proceedings of the IEEE, December 1995.

[67] E. Sprangle, D. Carmean. “Increasing Processor Performance by Implementing Deeper Pipelines.” *In Proceedings of the International Symposium on Computer Architecture (ISCA 2002)*, 25-29 May 2002, Anchorage, AK, USA

[68] G. S. Sohi and S. Vajapeyam. “Instruction issue logic for high-performance, interruptible pipelined processors.” In *Proc. 14th Annual International Symposium on Computer Architecture (ISCA’87)*, June 1987.

[69] A. S. Tanenbaum. Computer Networks. Third Edition.

[70] 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.

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

[72] L. Vintan, C. Armat, and G. Steven. “The Impact of Cache Organization on the Instruction Issue Rate of a Superscalar Processor.”

[73] M. V. Wilkes. “Slave memories and dynamic storage allocation.” IEEE Transactions on Electronic Computers, 14(2), 1965.

[74] 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

[75] C. Zilles and G. Sohi. “Execution-based prediction using speculative slices.” In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[76] R. Zucker and J. Baer. “A Performance Study of Memory Consistency Models.” Technical Report No. 92-01-02, January 1992.