Using Virtual Load/Store Queues (VLSQs) to Reduce the Negative Effects of Reordered Memory Instructions

Aamer Jaleel and Bruce Jacob
Dept. of Electrical & Computer Engineering
University of Maryland, College Park
{ajaleel,blj}@eng.umd.edu

Abstract

The use of large instruction windows coupled with aggressive out-of-order and prefetching capabilities has provided significant improvements in processor performance. In this paper, we quantify the effects of increased out-of-order aggressiveness on a processor's memory ordering/consistency model as well as an application's cache behavior. We observe that increasing reorder buffer sizes cause less than one third of issued memory instructions to be executed in actual program order. We show that increasing the reorder buffer size from 80 to 512 entries results in an increase in the frequency of memory traps by a factor of six and an increase in total execution overhead by 10–40%. Additionally, we observe that the reordering of memory instructions increases the L1 data cache accesses by 10–60% and the L1 data cache misses by 10–20%.

These findings reveal that increased out-of-order capability can waste energy in two ways. First, re-fetching and re-executing instructions flushed due to traps require the fetch, map, and execution units to dissipate energy on work that has already been done before. Second, an increase in the number of cache accesses and cache misses needlessly dissipates energy. Both these side effects can be related to the reordering of memory instructions. Thus, to avoid wasting both energy and performance, we propose a 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. We show that VLSQs can reduce trap overhead, cache accesses, and cache misses by as much as 45%, 50%, and 15% respectively when compared to traditional load/store queues. We observe that these reductions yield net power savings of 10–50% with degradation in performance by 1–5%.

1. Introduction

The instruction window or reorder buffer (ROB) is categorized as one of the most important design parameters in modern high performance processors. Previous studies have shown that increasing the size of reorder buffers, issue queues, and load/store queues can lead to increased performance [4, 13, 16, 18, 20]; consequently, much research has looked at the feasibility of increasing the size of these hardware data structures without negatively impacting clock cycle time [5, 11, 13]. However, when one considers "real" effects due to the reordering of memory instructions, the potential performance gains largely disappear.

By varying the aggressiveness of an out-of-order core in terms of reorder buffer sizes, issue queues, load/store queues, and renaming registers, the study in this paper brings to light two potential pitfalls of aggressive out-of-order mechanisms present in real systems that many previous simulation-based studies have not addressed.

- Increasing out-of-order capability conflicts with a processor's memory consistency and ordering model by requiring the processor to take frequent expensive *replay traps*, i.e. flushing the pipeline and re-executing a window of instructions.
- Increasing out-of-order capability can destroy cache locality, thereby causing an application to suffer from a higher number of cache misses than a less aggressive out-of-order mechanism.

This paper shows that even though aggressive out-of-order mechanisms enhance performance, the reordering of memory instructions can cause significant overhead in the system, i.e. the very mechanisms commonly used to improve performance can cause sources of performance degradation in the system. Using the network communication concept of *windowing*, we define a virtual window in the existing load/store queue. The window acts as a virtual load/store queue (VLSQ) and limits the number of available memory instructions at the select and issue logic. We vary the size of the window statically and observe that small VLSQs reduce the frequency of replay traps, cache accesses, and cache misses and provide net power savings with minimal degradation in performance.

1.1. The problem

Due to the increasing processor-memory gap, the long latencies associated with memory requires an abundance of independent instructions to overlap busy work with work that takes a while to do. The most popular mechanism to achieve this is to provide the instruction scheduler with a gigantic window of instructions to schedule from. Larger 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. Though the reordering of ALU instructions poses minimal effects on program execution, the reordering of memory instructions can affect program execution in two distinct ways:

- **Increased Replay Traps:** The reordering of memory instructions can create a variety of hazards that can affect the correct execution of an application. For example, when using load speculation [17, 19], 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

Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)
1530-0897/05 $20.00 © 2005 **IEEE**
IEEE COMPUTER SOCIETY

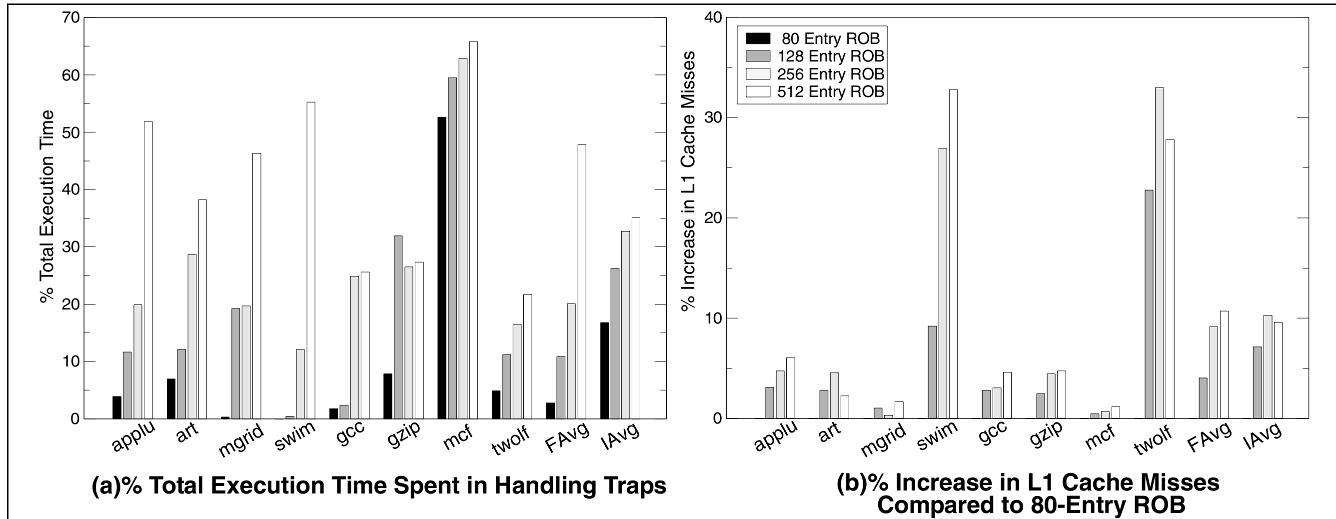

Figure 1: Negative Effects of Increased Out-of-Order Aggressiveness. Increasing the out-of-order capability of a processor can cause (a) 10–60% of total execution time recovering from replay traps (b) and a 10–20% increase in an applications cache misses.

known as a "replay trap." A replay trap can either be handled by flushing the pipeline and restarting execution at the faulting instruction or re-executing only the faulting instruction and all of its direct and indirect dependent instructions. Clearly, the re-execute method is more favorable than the pipeline flush method; however, the complexity in logic required to determine and re-execute an entire dependence chain of the faulting instruction is relatively expensive and can become even more so with increased reorder buffer sizes [19]. Thus, modern systems typically use the pipeline-flush method of handling traps, and, as the frequency of traps increases, significant performance and energy can be wasted in re-fetching and re-executing those instructions flushed.

- **Increased Cache Misses:** Executing memory instructions speculatively or in an order different from actual program order can negatively impact an application's cache locality. For example, a load instruction issued out-of-order can evict data required by both older and future memory instructions that are waiting to be issued. When the older or future memory instruction later executes and misses in the data cache, energy is needlessly wasted in re-fetching and re-filling the recently evicted data cache line. Even more, if the out-of-order load turns out to be only 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 the 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 wastage of energy.

Figure 1 illustrates both these pitfalls as they scale with increased out-of-order capability. By increasing the reorder buffer size from 80 to 512 entries, we observe that 25–60% of total execution time is lost due to the occurrence of replay traps. We also observe that increasing the reorder buffer size from 80 to 512 also negatively impacts an application's cache locality by increasing the number of L1 cache misses by 5–30%. To clarify, a "cache miss" is one that misses both in the data cache and the miss status holding registers (MSHRs) [12]. From Figure 1, 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 [4, 5, 11, 13, 16, 18, 20, 24], with the results from Figure 1 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.

1.2. Proposed solution

A trivial mechanism for reducing the reordering of memory instructions is to use a smaller load/store queue. 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 will need to stall 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 [14], 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.

We are interested in providing a solution that maintains 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 define a *virtual window* within the existing load/store queue. The virtual window acts as a virtual load/store queue (VLSQ) and is maintained using *virtual head* and *virtual tail* pointers that point into the existing load/store queue. To reduce the reordering of memory instructions, during instruction scheduling, the select and issue logic ensures that only memory instructions residing within the VLSQ are selected to be issued. Thus, by varying the size

1530-0897/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

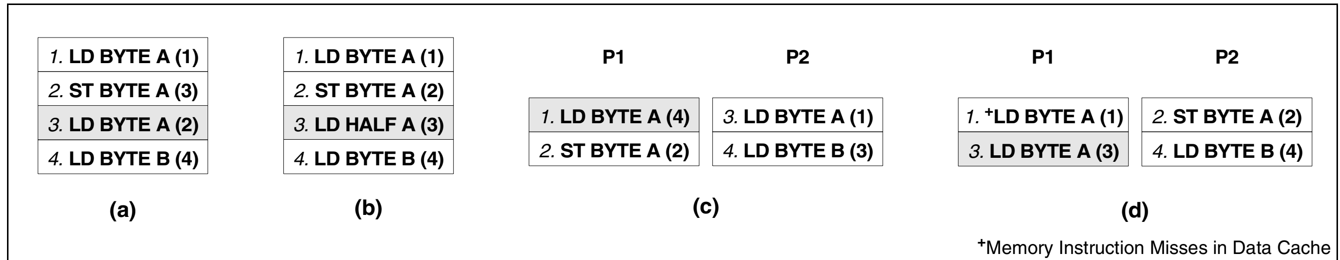

Figure 2: Classification of Replay Traps. The figure illustrates the different types of replay traps that can occur in both uniprocessor and multiprocessor environments. (a) Load-Store Replay (b) Wrong Size Replay (c) Load-Load Replay (d) Load-Miss Load Replay. In the examples, due to a replay trap, re-execution starts from the shaded instruction. Numbers in parenthesis show program execution order and numbers in italics show actual program order.

of the VLSQ, one can throttle the degree by which memory instructions are issued out-of-order.

1.3. Results

We show that VLSQs reduce the reordering of memory instructions, the frequency of traps, and the needless amount of cache accesses and cache misses. As a result, VLSQs eliminate the unnecessary overhead in re-fetching, re-mapping, and re-executing instructions by reducing the frequency of traps by a factor of two to 30; this yields a reduction in total trap overhead by as much as 45%. By reducing the amount of duplicated work, VLSQs reduce the average power dissipated in the fetch, map/rename, and execution units by 10–50%. Furthermore, VLSQs also reduce the number of L1 data cache accesses by 10–60% and the L1 data cache misses by 5–15%. These reductions translate into power savings of 10–50% and 5–30% in the L1 and L2 cache respectively. The reduction in power in each of the different components translates into 10–50% net power savings with performance degradation of 1–5%.

2. Background

2.1. Reordering of memory instructions

By allowing a processor to exploit instruction level parallelism (ILP) via large instruction windows, both ALU and memory instructions are executed out of program order. Since register renaming maintains the dependencies of ALU instructions, out-of-order issue of ALU instructions poses no threat to functional correctness. On the other hand, since memory dependencies are resolved only after issuing to execute, out-of-order issue of memory instructions can pose threats to functional correctness, especially if two memory instructions issued out-of-order access the same memory location. In such a scenario, the processor may need to initiate a *replay trap*. A replay trap occurs when the processor must roll back the state to force accesses to a particular memory location in order, or to handle different-sized accesses to the same memory location. Replay traps do not require any software support (e.g. interrupt handlers in the operating system); they merely require re-execution of instructions starting from the instruction that caused the replay trap. Figure 2 illustrates the different types of replay traps. Numbers in parentheses signify the order in which instructions are issued to execute, and numbers in italics signify actual program order.

- **Load-Store Replay:** A load-store replay trap occurs when a newer load instruction is issued before all prior store addresses are resolved. In the event that the processor detects a newer load executing out-of-order with respect to an older store that it depends upon, a load-store trap is initiated. This is required so that the newer load acquires data from the store rather than stale data from the cache. For example, in Figure 2(a), if memory instruction number three (a load) executes before the store instruction (both of which access the same memory location A), then the value loaded from the data cache will be incorrect. Microprocessors that use load speculation must handle this replay trap to ensure functional correctness, e.g. Alpha, POWER4, Itanium. [1, 2, 14, 22]
- **Wrong Size Replay:** A wrong-size replay trap occurs when the data for a newer load is partially in the store queue and partially in the data cache. In Figure 2(b), the second load instruction in the program requires reading a half word (two bytes) starting at memory location A; however a prior store writes one byte to the same memory location. When the processor detects this, the load must be re-executed after the older store instruction drains its data from the store queue into the data cache. Note that this replay trap can occur even if memory instructions are issued in program order [1, 2, 22]. As a result, all high performance microprocessors must be able to detect and overcome this hazard.
- **Load-Load Replay:** A load-load replay trap is initiated when two loads to the same memory address are issued out-of-order. In a uniprocessor environment this poses no problems; however, in the case of a multiprocessor environment, out-of-order issue of loads can cause subtle memory consistency problems. For example, in Figure 2(c), if two loads to the same address are issued out-of-order, and a different processor changes the value between the execution of these two loads, then the newer load instruction may obtain the older value, and the older load may obtain a newer value. The 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, processors provide a *memory barrier* instruction that allows the programmer to enforce ordering among memory instructions wherever needed. However, extensive use of memory barriers can negatively hurt performance [18]. Thus, hardware support, via replay traps, is provided by some processors to guarantee load-load ordering to the same address, (e.g., Alpha[1, 2], POWER4[22], and MIPS R10000[3]).

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

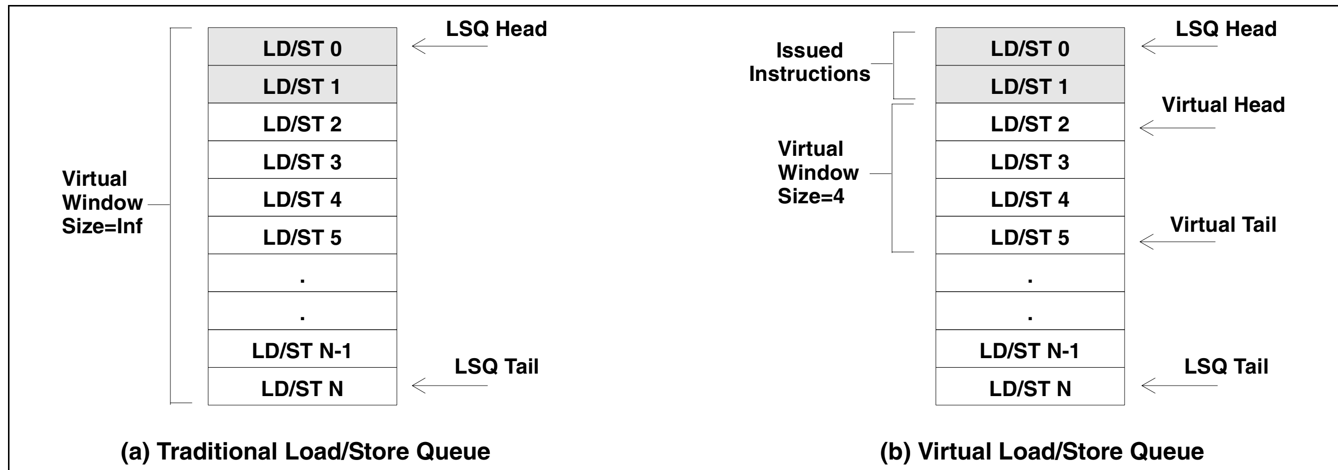

Figure 3: Virtual Load/Store Queue (VLSQ): A mechanism to reduce the reordering of memory instructions. (a) The figure illustrates the traditional implementation of a load-store queue. (b) Using VLSQs, only memory instructions that lie within the virtual head and virtual tail pointers are issued to execute. Other memory instructions must wait till they lie within the virtual window before they can be issued to execute.

- **Load-miss Load Replay:** A load-miss load replay trap is initiated when two loads to the same memory address are issued, and the first load misses in the data cache and already has a miss information/status holding register (MSHR) allocated to it (Figure 2(d)). An MSHR keeps track of an outstanding memory request to a single cache line [12]. It is used to "merge" multiple requests to the same cache line by keeping track of all destination registers waiting for data from memory. When the data arrives from memory, the MSHR fills all the outstanding destination registers that were waiting for the same cache line. Like the load-load replay trap, subtle memory consistency problems can occur if there is an intervening store from a different processor between two loads to the same memory address. In such a scenario, the MSHR provides the second load instruction with stale data. Thus, to avoid this source of memory inconsistency, the processor waits until the data for the first load is loaded into the destination register and flags the newer load instruction with a replay trap. Note that this replay trap can occur even though memory instructions are issued in program order, (e.g., Alpha[1, 2]).

Irrespective of the type of replay trap, the hardware mechanism currently used to handle a replay trap is identical to those involved in handling branch mispredicts. When an instruction executes and causes a replay trap, the fact is noted in the reorder buffer entry. While committing instructions, if the processor detects that the memory instruction caused a replay trap, the pipeline is flushed, and execution is restarted from the faulting memory instruction. We show that, while these replay traps occur only a fraction of the time with small reorder buffers, increasing out-of-order capability exposes them to be a significant and increasing drain on performance.

2.2. Related work

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

It is also known that larger instruction windows conflict with increasing clock speeds. A good deal of recent effort is aimed at designing efficient and fast issue/selection logic that allows for larger instruction-window sizes while still maintaining high clock speeds. Henry et al. proposed an alternate binary tree circuit implementation for the wakeup logic [11]; Onder et al. proposed explicit wake-up lists associated with executing instructions [16]; Lebeck et al. tackle the instruction window size by proposing an alternate *waiting instruction buffer* (WIB) [13]; and Akkary et al. propose a checkpoint and recovery mechanism to recover from branch mispredicts with larger instruction window sizes [4].

Since larger instruction windows expose aggressive out-of-order processors to more load/store communications, Park et al. propose techniques to scale the load/store queue size using segmentation [18]. Furthermore, to allow for load speculation, Calder et al. tackle the false memory-aliasing problem and propose four different mechanisms for load speculation. Loads predicted not to alias older stores are issued speculatively. If the load is mispredicted, instructions are squashed and re-executed [19].

3. Virtual load/store queues (VLSQs)

To maintain large reorder buffer and issue queue sizes yet reduce the reordering of memory instructions, we propose a virtual load/store queue (VLSQ) using the concept of *windowing* [21]. Windowing is a commonly used technique for implementing flow control while transferring data over 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.

We use the windowing concept to reduce the reordering of memory instructions. We introduce a virtual window into the existing

1530-0897/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

Table 1: Processor Configurations

Configuration Name	ROB Size	Issue Width INT/FP	IssueQ Size INT/FP	# Functional Units INT ALU/INT MULT/FP ALU/FP MULT	LQ/SQ Size
Alpha-80	80	8/4 Way	20/15	4/4/1/1	32/32
Alpha-128	128	8/4 Way	40/30	4/4/1/1	64/64
Alpha-256	256	8/4 Way	80/60	4/4/1/1	128/128
Alpha-512	512	8/4 Way	160/120	4/4/1/1	256/256

load/store queue. The length 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 VLSQ is maintained using two additional 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 VLSQ. The difference between *virtual head* and *virtual tail* is W_{size}, the size of the virtual window. During instruction scheduling, the select and issue logic ensures that only memory instructions residing within the VLSQ are issued to execute. The *virtual head* and *virtual tail* pointers are changed only when the memory instruction at the *virtual head* is issued.

Figure 3(a) illustrates a traditional load/store queue with head pointer at index 0 and the tail pointer at index N. The shaded load/store queue entries indicate instructions that have already been issued. With a traditional load/store queue, the issue logic can schedule any memory instruction (between 2 and N) whose operands are ready. Figure 3(b) illustrates an example of a VLSQ with W_{size} = 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.

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 reducing the reordering of memory instructions, VLSQs can reduce the number of replay traps and cache misses. Second, a VLSQ can also 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 cache accesses. A reduction in the number of speculative memory instructions issued and a reduction in replay traps caused due to the reordering of memory instructions can lead to significant power and energy savings in the data caches and 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.

In this paper, we explore 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 and is part of our ongoing work.

4. Experimental methodology

4.1. Simulation 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. The simulator also models a stride prefetcher 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 the replay traps mentioned in Section 2.1. 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 fact is noted by indexing into the store-wait table using the load's PC. At fetch time, if the processor detects that the PC of the load is set in the store-wait table, the load instruction is not issued until all prior store address are resolved. The store-wait table is cleared unconditionally every 16,384 cycles [1, 2].

We vary out-of-order capability by changing the ROB size, issue and load/store queue size, as shown in Table 1. We use four floating point (*applu*, *art*, *mgrid*, and *swim*) and four integer (*gcc*, *gzip*, *mcf*, and *twolf*) benchmarks from the SPEC 2000 suite [10]. The benchmarks were acquired from the SimpleScalar developers [23] and were warmed up by fast-forwarding the first 2 billion instructions. Data was gathered over the next 500 million instructions. The benchmarks operate on their reference input sets.

As mentioned earlier, we statically vary the size of the VLSQ to analyze the effect of different VLSQ sizes with increasing out-of-order capability. A VLSQ of size 1 implies in-order issue of both loads and stores. A VLSQ of size infinity (*Inf*) is a traditional processor with a VLSQ size equal to the appropriate physical load/store queue size shown in Table 1.

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

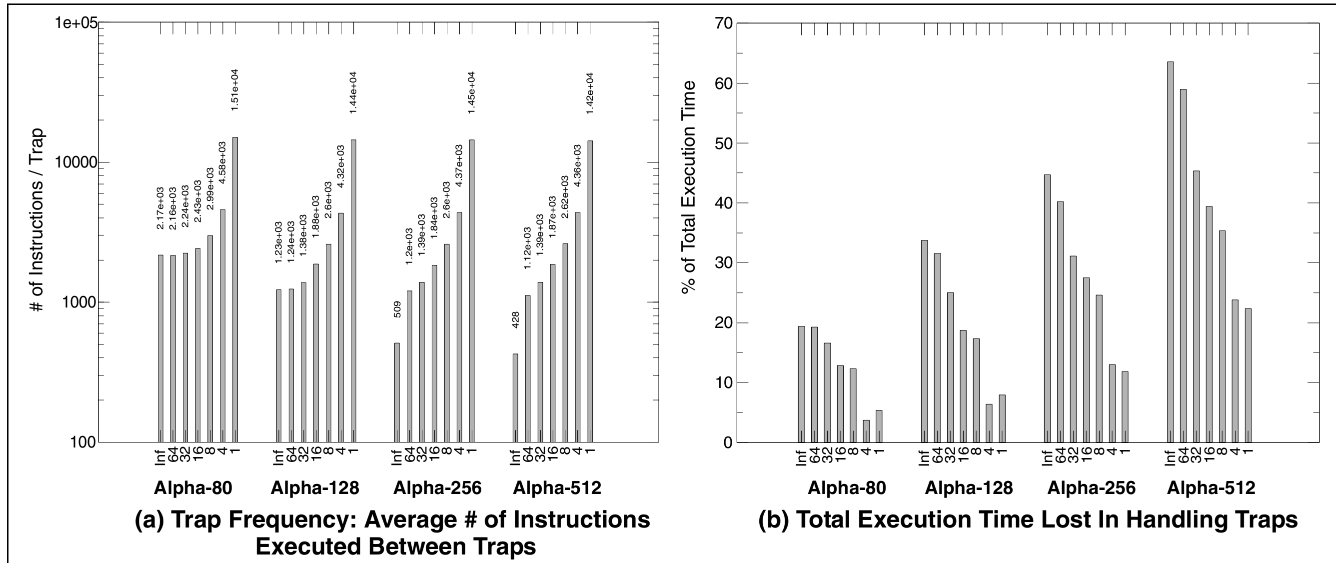

Figure 4: Effect of VLSQs on Replay Traps. The figure shows that VLSQs reduce (a) the frequency of traps by a factor of two to 30 and (b) the total execution time lost in traps by 10–45%.

5. Effects of increased OoO aggressiveness

5.1. Replay traps

When the reorder buffer is increased from 80 to 512 entries, less than one-third of the total number of memory instructions issued 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 4 shows the total number of instructions executed between traps (trap frequency) and the percent of total execution time lost in replay traps. Execution time lost is found by tracking the difference in cycles between when an instruction was originally fetched and when the instruction is re-fetched due to a replay trap. The data is averaged for all eight benchmarks. 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 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. Increasing the ROB size from 80 to 512 entries decreases the average number of instructions executed between traps by a factor of 6, meaning that increasing the out-of-order capability can cause an increase in trap frequency by 600%. For benchmarks like mgrid and swim, we observe an increase in trap frequency by factors of 50 or more. As mentioned earlier in the paper, the mechanisms for handling replay traps requires the pipeline to be flushed and instructions to be re-fetched and re-executed from the faulting 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. For example, if at the time of a replay trap the reorder buffer is a 100% full, on an 8-wide processor with a 512-entry ROB, it will take 64 cycles plus memory latency, functional unit latency, and dependency stalls to restore the state of the reorder buffer to what it was before the trap. 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(b), we observe that the increase in trap frequency translates into 20–65% of total execution time lost due to replay traps. These results reveal that even though a processor can extract maximum possible ILP, too much reordering of memory instructions 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 10–45% as shown in Figure 4(b). From the figure, we observe that maximum benefits come from smaller VLSQs, clearly correlating the reordering of memory instructions to trap frequency and overhead. Thus, we conclude that the use of VLSQs can reduce the frequency of replay traps and this can translate into savings in energy that would otherwise be needlessly spent in re-fetching and re-executing instructions flushed.

5.2. Cache behavior

Figure 5 shows the cache behavior in terms of change in L1 cache accesses and L1 cache misses averaged over all eight benchmarks. The data is graphed as a percent change in cache accesses or misses normalized to the Alpha-80 configuration with a full-size ("infinite")

IEEE COMPUTER SOCIETY

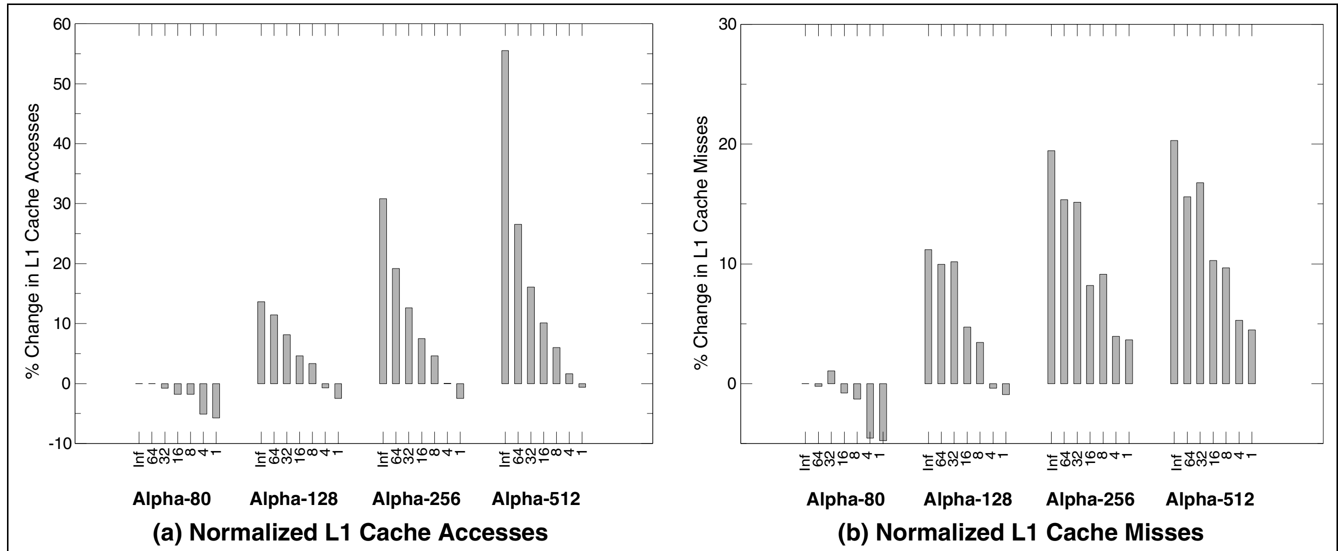

Figure 5: Effect of VLSQs on Cache Behavior. VLSQs reduce (a) the number of L1 cache accesses by 5-60% and (b) the number of L1 cache misses by 5-15%.

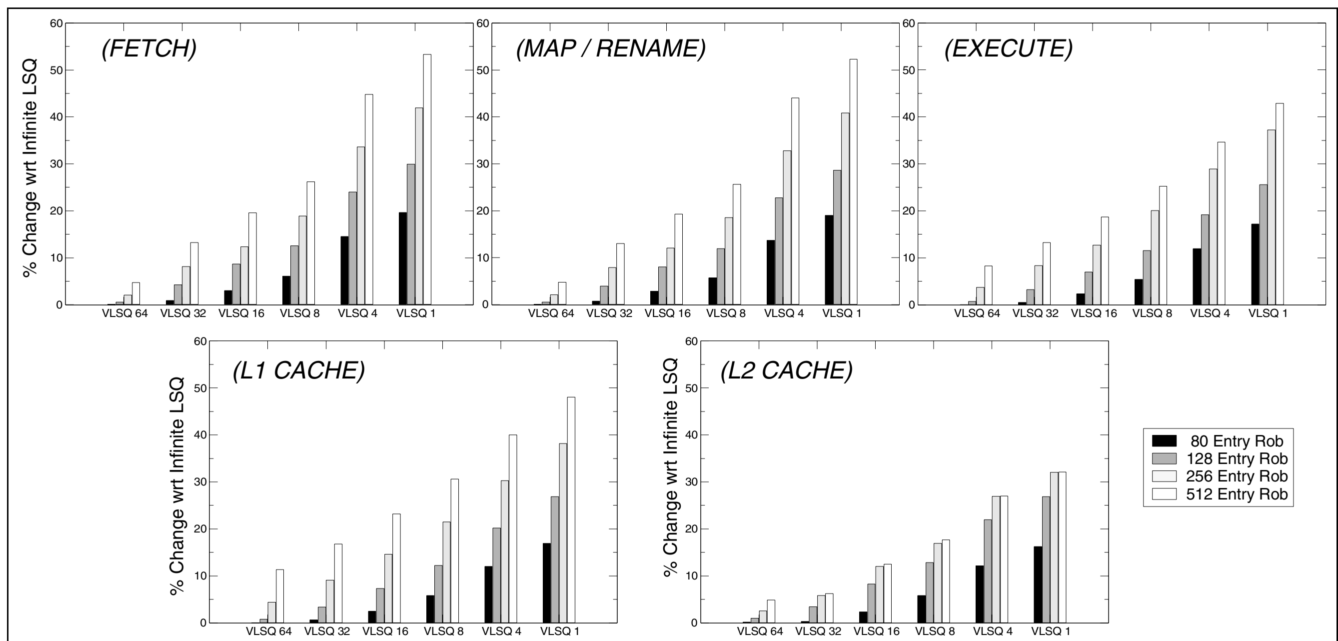

Figure 6: Average Power Savings Using VLSQs. By reducing the reordering of memory instructions, VLSQs eliminate the needless amount of energy dissipated in re-fetching and re-executing instructions, and speculative cache accesses. This translates into power savings of 5-50% in the fetch and rename hardware, 10-40% in the execution hardware, and 5-50% in the caches.

VLSQ. From the figure, considering only traditional load/store queues, we observe that increasing the out-of-order capability increases the total number of cache accesses by 10–60% and the total number of cache misses by 10–20%. We observe a direct correlation between reduced VLSQ sizes and a reduced number of cache accesses and misses. We observe that smaller VLSQs can reduce the total number of cache accesses by 5–60% 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 cache.

5.3. 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 6 shows the savings in average power consumed, normalized to the traditional load/store queue for each of the following components:

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY

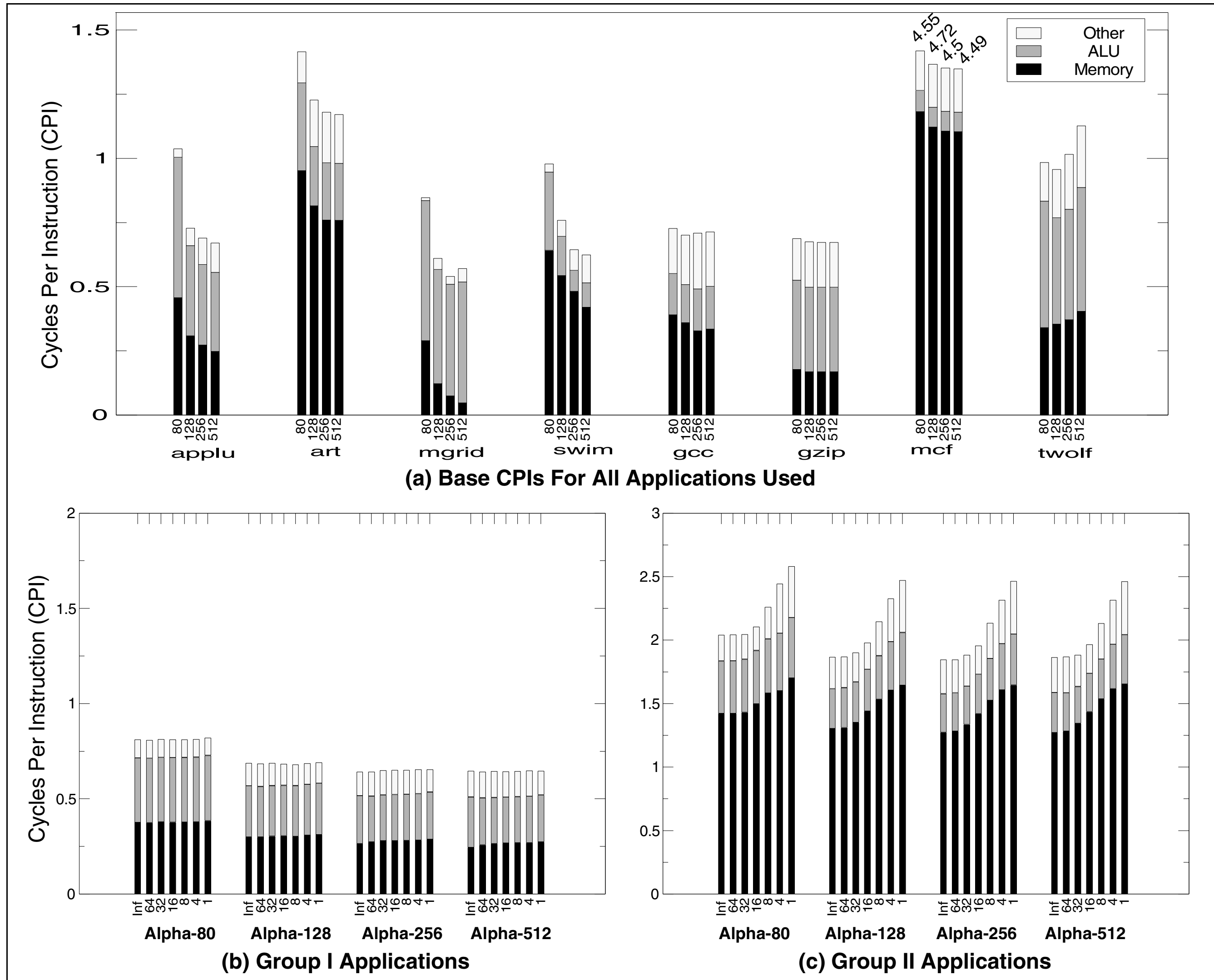

Figure 7: (a) Base CPIs Vs. Out-of-Order Capability (b)(c) Effect of VLSQs on Processor Performance.

fetch hardware, mapping hardware, execution hardware, L1 cache and L2 cache.

With the use of VLSQs, we observe 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 [9].

Additionally, we observe that the use of VLSQs reduces the average power consumed in the L1 cache by 10–50% and in the L2 cache by 5–30%. Again, we observe that these savings in the caches are substantial since the caches contribute about 15% of an Alpha 21264's total chip power [9].

5.4. Performance

Figures 1, 4 and 5 revealed that increasing the aggressiveness of the out-of-order core in general increased the number of replay traps and cache misses. We know that such trends normally hurt performance; the question, however, is: *Does the increase in out-of-order capability overcome these hurdles to provide net performance improvements*? Figure 7(a) shows the performance graphs with the different benchmarks and ROB sizes on the x-axis and cycles per instruction (CPI) on the y-axis. CPI is classified into stall cycles where memory instructions couldn't retire due to memory latency (black), stall cycles where instructions couldn't 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 couldn't 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

IEEE COMPUTER SOCIETY

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 7(a), we observe that for four benchmarks, increased out-of-order capability overcame the sources of performance degradation to provide 30–40% 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 applications such as mgrid, gcc, and twolf 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 and replay traps, while the increase in the other two portions of CPI can be correlated with an increase in replay traps.

Figure 7(b) and 7(c) show the results of varying the VLSQ size as the average CPI for all benchmarks, categorized as Group I and Group II sets. The benchmarks included in Group I are: mgrid, swim, gcc, gzip; and the benchmarks included in Group II are: applu, art, mcf and twolf. 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 rather 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 6).

On the other hand, for the Group II 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 we observe that VLSQ sizes of 16 and 32 are within 2–5% of the traditional load/store queue. This indicates that VLSQ sizes of 16 or 32 are optimal and can lead to power savings of 10–22% (as shown in Figure 6).

Second, for the Group II benchmarks, we 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 an increase in the occurrence of replay traps. As mentioned in Section 2.1, replay traps can still occur even though memory instructions are issued in program order. Besides the load-miss-load and wrong-size replay traps, we observe that a load-store replay trap can also occur with the in order issue of memory instructions. 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. 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. 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 7(b) and 7(c), 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 unnecessary data cache accesses and misses.

6. Conclusions and future work

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 study in this paper shows two pitfalls of aggressive out-of-order mechanisms. First, increased out-of-order capability conflicts with the memory-ordering requirements of a processor, resulting in frequent traps to maintain correct state. Second, out-of-order execution of memory instructions destroys an application's cache locality, causing it to suffer from a higher number of cache misses than a less aggressive out-of-order mechanism. We observe that both these side effects of out-of-order execution are due to the reordering of memory instructions. We observe that increasing the reorder buffer size from 80 to 512 entries increases the trap frequency by a factor of six or more. The increased trap frequency translates into an application spending 25–60% of total execution time recovering from traps. Furthermore, we observe that increasing the out-of-order capability increases the average L1 data cache accesses by 10–60% and the average L1 data cache misses by 10–20%. These results show that the very mechanisms commonly used to improve performance cause sources of performance degradation in the system.

We show that the use of virtual load/store queues (VLSQs) reduces the reordering of memory instructions, the frequency of traps, and the needless amount of cache accesses and cache misses. As a result, a VLSQ eliminates the unnecessary overhead in re-fetching, re-mapping, and re-executing instructions by reducing the frequency of traps by a factor of two to 30 and also reducing the total trap overhead by as much as 45%. By reducing the amount of duplicated work, VLSQs reduce the average power dissipated in the fetch, map/rename, and execution units by 20–50%. Furthermore, VLSQs also reduce the number of L1 data cache accesses and misses by 5–60% and 5–15% respectively. Both these savings translate into power savings of 10–50% and 5–30% in the L1 and L2 cache respectively. The reductions in power dissipation in each of the different components translates to 5–50% net power savings with performance degradation of 1–5%.

We are currently exploring the *dynamic* approach of windowing. Rather than statically restricting the size of the VLSQ (which requires profiling), it is possible to dynamically increase or decrease the size of the VLSQ based on application runtime events such as replay traps and cache misses. Unlike the static approach, the dynamic approach of windowing can allow for applications to exploit memory ILP during application phases where replay traps and cache misses occur infrequently.

IEEE COMPUTER SOCIETY

7. Acknowledgements

The initial phase of this study, specifically the effects of out-of-order execution on cache performance, occurred while Aamer Jaleel was interning at the Compaq Western Research Labs, Palo Alto, California. The authors are grateful to Norm Jouppi, Partha Ranganathan, and Keith Farkas for their initial thoughts and guidance. The authors would also like to thank Raj Desikan, Donald Yeung, Brinda Ganesh, Sadagopan Srinivasan, Bharath Iyer, Heather Hanson, and the anonymous reviewers for their valuable support and feedback.

The work of Aamer Jaleel was supported in part by NSF CAREER Award CCR-9983618, NSF grant EIA-9806645, and NSF grant EIA-0000439. The work of Bruce Jacob was supported in part by NSF CAREER Award CCR-9983618, NSF grant EIA-9806645, NSF grant EIA-0000439, DOD MURI award AFOSR-F496200110374, the Laboratory of Physical Sciences in College Park MD, the National Institute of Standards and Technology, and Cray Inc.

8. References

[1] Compaq Computer Corporation. “Alpha 21264 Microprocessor Hardware Reference Manual.” June 1999.

[2] Compaq Computer Corporation. “Compiler Writer’s Guide for the Alpha 21264” June 1999.

[3] Silicon Graphics, Inc. *MIPS R10000 Microprocessor User’s Manual version 2.0*, October 1996.

[4] H. Akkary, R. Rajwar, and S. T. Srinivasan. “Checkpointing Processing and Recovery: Towards Scalable Large Instruction Window Processors.” In *Proc. 36th International Symposium on Microarchitecture,* December 2003.

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

[6] 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.

[7] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. “High-Performance DRAMs in Workstation Environments.” *IEEE Transactions on Computers,* 50(11):1133-1153, November 2001.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] T. Lyon, E. Delano, C. McNairy and D. Mulla. “Data Cache Design Considerations for the Itanium 2 Processor.” In *IEEE International Conference on Computer Design (ICCD)*, Freiburg, Germany, September 2002.

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

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

[17] S. Onder and R. Gupta. “Dynamic Memory Disambiguation in the Presence of Out-of-Order Store Issuing.” In *Proc. 32nd International Symposium on Microarchitecture*, 1999.

[18] 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.

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

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

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

[22] 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.

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

[24] 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

1530-0897/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY