Out-of-Order Memory Accesses Using a Load Wait Buffer

Shelley Chen
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-2862
schen1@ece.cmu.edu

Jennifer Morris
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213-3890
(412) 268-4264
jenm@ece.cmu.edu

Abstract

Many dynamic scheduling techniques take advantage of out-of-order instruction execution to hide memory access latency. However, as the disparity between processor and memory speeds increases, delays in the load-store queue become more of a bottleneck. One way to mitigate these delays is to allow loads and stores to execute and retire from the load-store queue (LSQ) out-of-order. Unfortunately, when the LSQ fills with pending loads, other loads and stores are prevented from entering the buffer to be retired. In addition to out-of-order execution of loads and stores, we propose temporary removal of long-latency, pending loads to a separate load wait buffer (LWB), similar to the waiting instruction buffer (WIB) proposed by Lebeck, et. al. [1]. Simulation results show successive increases in benchmark IPC with out-of-order loads, out-of-order loads and stores, and out-of-order loads and stores with a LWB. The design with the LWB shows up to 303% speedup in IPC.

1 Introduction

As the gap between memory and processor increases, many modern superscalar processors use out-of-order program execution to hide memory access latencies. While instructions with long-latency memory accesses wait for their data to arrive, other ready instructions are allowed to execute, thereby maintaining a high processor utilization. In order to maintain precise interrupts, however, instructions that execute out-of-order must be committed to the architectural state in order. This is especially true of instructions that change the memory state (i.e., store instructions) because it is difficult to retrieve overwritten memory values to restore architectural state.

Traditionally, microprocessor architectures deal with this problem by maintaining strict in-order execution of loads and stores. This method, though effective in maintaining precise interrupts, can lead to decreased system performance, as long-latency loads and stores at the head of the LSQ block all subsequent loads and stores from executing. Additionally, a blocking instruction at the head of the LSQ could cause the queue to saturate, stalling the dispatch of additional load and store instructions.

The efficiency of the LSQ can be improved by allowing loads and stores to complete out-of-order; however, multiple long-latency loads could also saturate the LSQ and lead to similar stalls. One way to remove this bottleneck in the LSQ is to temporarily remove long-latency loads to a LWB until they are ready for execution.

The rest of this paper is organized as follows. Section 2 covers background and related work for our research. Section 3 describes our methodology. Section 4 discusses the implementation details. Section 5 presents our experimental results for out-of-order loads, out-of-order loads and stores, and out-of-order loads and stores with the LWB. Finally, in section 6 we draw conclusions from these results.

2 Background

As mentioned above, Lebeck, et. al. [1], proposed using a WIB to temporarily store long-latency instructions. This removed pending instructions from the issue queue, freeing up entries for other instructions and allowing more instructions free of data dependencies to be issued. Their design with a 32-entry issue window and 2048 entry WIB achieved speedups of 20% to 84% for various SPEC benchmarks. We propose that using a similar wait buffer for the load-store queue will achieve similar results.

Another method of hiding memory access latencies is load forwarding. This technique, which is utilized in our base simulator, was evaluated for performance improvements by Parcerisa and González [2]. Load forwarding is useful when a load instruction has a true data dependency on a store instruction that has not yet been retired. In conventional designs the load must wait for the store to finish writing to memory before the data can be retrieved from memory and the load completed. Load forwarding allows the modified data to be passed directly from the pending store instruction to the load instruction, thus bypassing the latency of the memory write and read. The LWB, which only removes loads that have already accessed the cache, will not affect load-forwarding because it will neither remove stores that could potentially forward values to subsequent loads, nor will it remove loads before they have a chance to receive forwarded values.

The relationship between various load-store buffer retirement rules, including load forwarding, and performance was explored by Hwang, et. al. [3]. In their experiment, all LSQs retired stores in-order, but for loads they modeled and compared four different retirement policies: in-order, bypassing, forwarding, and speculation. The latter three, each with varying degrees of out-of-order load retirement, produced increasingly improved performance over the in-order configuration. Our LWB, combined with load forwarding, not only allows more loads to be retired out-of-order, but it also allows out-of-order store retirement.

Sim-outorder, the base simulator for our design, uses a combined register update unit (RUU)/LSQ structure. Sohi [4] describes the purpose of the RUU and how it operates. The RUU behaves somewhat like an issue queue, in that it receives instructions in order, monitors their dependencies, allocates loads and stores to the LSQ, sends instructions to the functional units for execution, and commits the instructions when possible. Unlike an issue queue, however, the RUU maintains a record of the instruction order, to allow precise interrupts. Our design adds an issue queue to the existing RUU.

3 Methodology

In the base configuration of Sim-outorder no stores are allowed to execute out of program order. Additionally, although loads are issued out-of-order, the load entries in the LSQ are not removed until the corresponding RUU entry is committed and removed. This creates a bottleneck in the LSQ because when the LSQ becomes full, dispatching must stall until instructions are committed and the LSQ empties out. In Sections 3.1 and 3.2 we describe our method of executing loads and stores out-of-order. In section 3.3 we explain how the LWB operates.

3.1 Out-of-Order Loads

Loads are allowed to execute out-of-order when they are not preceded by a store with an unresolved address, however, the LSQ entry is held until the RUU entry commits. This creates a structural hazard by preventing future instructions from issuing due to the LSQ being full.

Profiling was done on LSQs of various sizes to see where the first unresolved store is usually located. If the first unresolved store is usually located near the head of the LSQ, then the potential for performance improvement due to out-of-order loads and stores would be insignificant because the majority of instructions in the LSQ would not be able to issue.

Figure 1 shows that for the mgrid benchmark, the majority of the unresolved stores are located near the end of the LSQ, at least within the second half. This suggests that doubling the effective size of the LSQ with a LWB should result in some performance improvement, because a large number of memory access instructions in the LSQ could potentially be issued out-of-order. Results for the other Spec2000 benchmarks were similar.

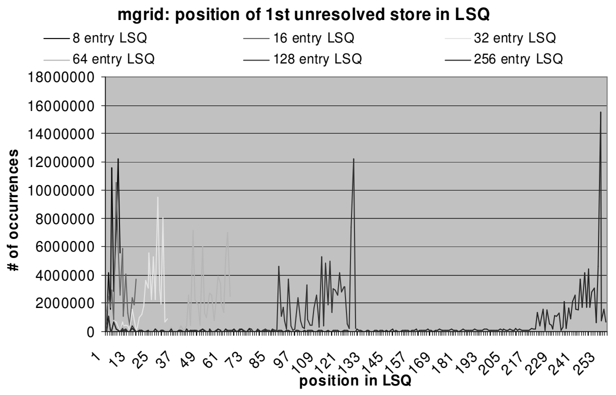

Figure 1: profile of the position of the 1st unresolved store in the LSQ for mgrid benchmark.

3.2 Out-of-Order Stores

Our system allows out-of-order stores by using the following rules to determine eligibility for execution:

1. No stores (or loads) may be executed that are preceded by a store with an unresolved address.
2. No stores may be executed that are preceded by a load with an unresolved address.
3. Stores to the same address must go in program order.

The first rule is necessary because instructions that follow a store with an unresolved address may be trying to access the same memory location. The second rule prevents a load with an unresolved address from receiving data values from a later store executed out of program order. Consider the following code sequence:

```
1   ST X, 10
2   LD ?
3   ST X, 30
```

Instruction 1 is ready, so the store is issued. Instruction 2 is waiting for its address calculation to complete before it can be issued. If instruction 3 is issued before instruction 2 and the unknown address is subsequently determined to be the same address as both 1 and 3, instruction 2 will not receive the correct data. The third rule prevents a later store to the same address from modifying the cache entry for that address before earlier stores have executed.

3.3 Load Wait Buffer

Allowing loads and stores to be retired from the LSQ out of program order frees up entries for more instructions, however loads that have missed in the L2 cache remain in the LSQ until their data has returned. Because the miss penalty in such cases can be very high, these loads remain in the LSQ for extended periods of time. If multiple cache misses occur together, the LSQ may become saturated with long-latency loads.

Rather than remaining in the LSQ and blocking other loads and stores, these loads may be removed and temporarily stored in another buffer, the LWB. This allows other loads and stores to enter and exit the LSQ while long latency loads wait for their data to return from memory.

4 Implementation Details

4.1 The Base Machine

In order to determine the performance of our modified processor, we needed a base machine for comparison. This machine is modeled by sim-outorder from the SimpleScalar toolset [6].

The simulator is an 8-way superscalar out-of-order processor. It contains a 2048 RUU, which handles all of the register renaming and the reordering of the instructions. Our goal was to improve performance by optimizing the LSQ, therefore, we wanted to ensure that the LSQ in the base configuration was indeed the bottleneck of the system. This was accomplished by increasing the size of the RUU until it no longer factored into performance.

Also, the LWB would not show performance benefits unless the LSQ filled up some of the time. In order to allow the LSQ to saturate, we increased the memory access latency to 250 cycles. In addition, we decreased the size of the L2 cache to 256K to ensure that the processor would experience some level 2 cache misses. In a benchmark with 50% memory access instructions, approximately 4 memory access instructions are issued every cycle. Thus, with a 64 entry LSQ, after one L2 cache miss, 250*4 = 1000 memory access instructions could potentially be issued in that time frame (given an infinitely sized LSQ) , which would likely fill up a reasonably sized LSQ.

In addition, the base configuration had a perfect branch predictor. This was done because we wanted to see the affects of the optimizations on the LSQ without influence from the control hazards and mispredictions of the branch predictor. We wanted to ensure that we were only looking at results from the LSQ optimizations.

The base machine has a 64 entry LSQ for holding all of the load and store instructions before they are committed. Note that in this machine, for all memory access instructions, the associated RUU and LSQ entries are held until the instruction commits.

4.2 Issue queue

The base machine described above does not include an issue queue, therefore, we modified the base-version of SimpleScalar to include one. The implementation was fairly straightforward. We maintain two pointers that bounded a subset of the RUU. Within these boundaries only the non-issued instructions are considered part of the issue queue. Instructions that have already been issued, but not committed, are ignored. The issue queue head pointer always points to the oldest instruction that has not been issued. Each time an entry is issued, the issue queue head pointer is incremented, effectively removing the instruction from the issue queue. The issue queue tail is incremented every time a new instruction is dispatched, modeling the addition of entries into the issue queue. Before any instruction is dispatched, the issue queue must be checked. If the issue queue is full, then the system needs to stall until some instructions are issued.

4.3 Out-of-order Loads

In the baseline SimpleScalar simulator, loads are allocated an entry in the LSQ when the instruction is dispatched. Each cycle, the LSQ is traversed from head to tail in search of loads and stores whose operands are ready. The search is stopped when a store with an unresolved effective address is encountered, because all later loads and stores are blocked by that store.

As soon as the effective address is ready, unblocked loads proceed to the issue stage, where the memory access takes place. From there, the load is set to complete in the writeback stage, and finally removed from the LSQ and RUU in commit.

Our implementation also blocks loads and stores from issuing in the LSQ when there is a previous store with an unresolved address, therefore, we did not modify the original implementation of that logic. We did, however, change the design to remove the LSQ entry during the writeback stage, when the instruction is completed. This allows the completed load to exit the LSQ earlier.

4.4 Out-of-order Stores

Originally, stores are completed as soon as they are issued, but their cache accesses are done in order, when they are completed. The baseline simulator ensures that stores are executed in program order by stalling their cache accesses until the instructions are committed. Unfortunately, this also guarantees that the stores hold their LSQ entries until they are committed, which is long after they have completed.

We modify this so that stores are issued only if no other unresolved memory access instructions exist before them in the LSQ. In addition, no other instructions with the same effective address exist before them in the LSQ either. This means that stores that are issued can still complete right away, but they are also free to access the cache right after being issued. Thus, we moved the cache access from the commit stage to the issue stage of the pipeline, which allows us to remove the associated LSQ entry from the LSQ as quickly as possible. In turn, this early removal of the LSQ entry allows another instruction to be issued.

The instructions in the RUU are still committed in program order to ensure precise exceptions.

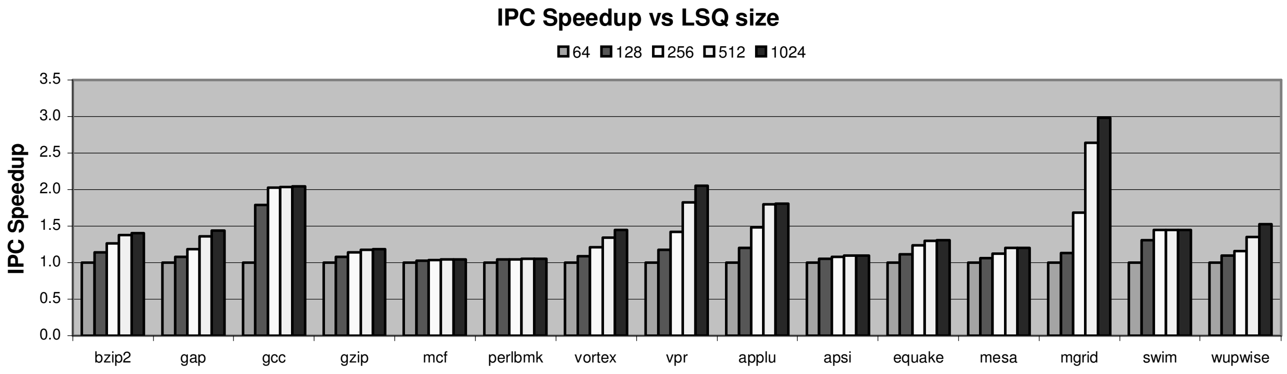

Figure 2: IPC speedup vs. LSQ size for 15 SPEC2K benchmarks

4.5 Load Wait Buffer

The LWB in our design is a 64-entry, fully-associative table comprised of entries identical to those found in the LSQ. When an L2 cache miss occurs on a load, the corresponding entry in the LSQ is moved to the LWB to wait for the data to arrive. During the writeback stage, when the long latency load has received its data and is ready to commit, the LWB entry is recovered.

4.6 Benchmarks

Our design was tested using integer and floating point benchmarks from the SPEC2K suite. First, we ran simulations on all of the benchmarks, with different sizes of LSQ (64, 128, 256, 512, and 1024 entries). Figure 2 shows the results of these preliminary simulations. The five benchmarks that showed the largest performance increase from increasing the LSQ size were gcc, vpr, applu, mgrid, and swim.

5 Experimental Results

Figures 3, 5, 7, 9 and 11 show the performance improvements of each optimization to the LSQ. The simulations were done on the ref inputs for the second billion instructions for each benchmark. For comparison purposes, the base machine described in Section 4.1 is included in the graphs as well. With the baseline machine, optimal performance is achieved with an LSQ size of 256 entries for gcc and swim, and 512 entries for vpr, mgrid, and applu. This means that increasing the effective size of the LSQ past 256 (through the addition of a LWB) will not significantly improve the performance of the system.

The first optimization that we implemented was the out-of-order loads (OOL). For this optimization, we merely moved the cache access up to the writeback stage, and we removed the LSQ entry there as well. With OOL, the effective size of the LSQ seems to have doubled in most cases, with the exception of gcc. For vpr and applu, the optimal LSQ size has not reduced to 256 entries. For swim and mgrid, the optimal LSQ size is now 128 entries. Even though gcc has a large percentage of load instructions, it does not seem to improve with the out-of-order loads. This could be because most of these loads are long latency loads, which take up entries in the LSQ, blocking ready memory accesses from being issued.

The second optimization was the implementation of out-of-order stores on top of the out-of-order loads from the previous optimization (OOL&S). With the exception of gcc, the other four benchmarks have a slight increase in performance increasing the LSQ size from 64 entries to 128 entries. After that, increasing the LSQ size only brings negligible performance improvements to the processor.

The last optimization to the LSQ was the addition of a 64 entry load wait buffer to the out-of-order loads and stores (OOL&S, 64 LWB). From the graphs, we can see that the IPC from running the processor on each benchmark has just about reached it maximum performance.

Notice that the optimal speedup for this subset of benchmarks was significant with the addition of the LWB. The maximum speedup was for the mgrid benchmark at 303%. This may have been due to the fact that running mgrid on the baseline simulator resulted in an LSQ that was full about 89% of the time. The minimum speedup was 50% for the swim benchmark. This is due to the fact that running swim on the baseline simulator, the LSQ is full only 49% of the time. Thus, even with the LWB and OOL&S there is not much room for a performance improvement.

Figures 4, 6, 8, 10, and 12 show the percentage of time the LSQ was full during the total simulation time for the five different benchmarks analyzed. From the results of the simulations, we observed that the LSQ became full for an increasingly smaller percentage of the total simulation time with the addition of each optimization. This confirms the fact that the optimizations are successfully emptying out the LSQ as they were intending to do.

There is an extreme fall at 128 entries for vpr for the simulator with the implementation of the LWB and the OOL&S. We are not sure why this happens. It does not reflect in the results of the IPC trends and could just be an anomaly.

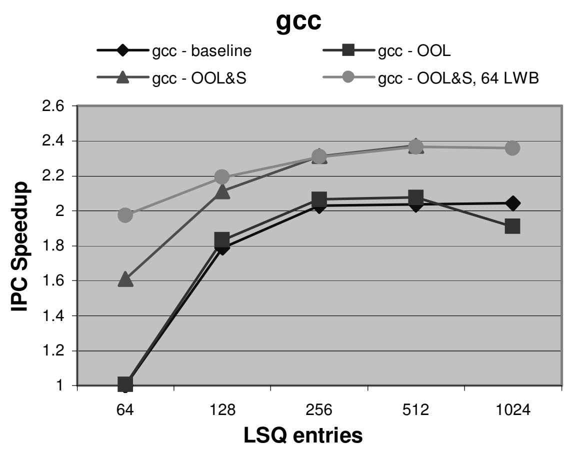

Figure 3: IPC Speedup for gcc: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

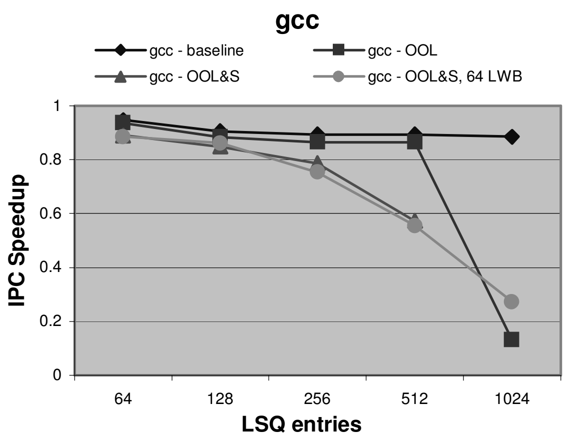

Figure 4: Percentage of time LSQ is full for gcc: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

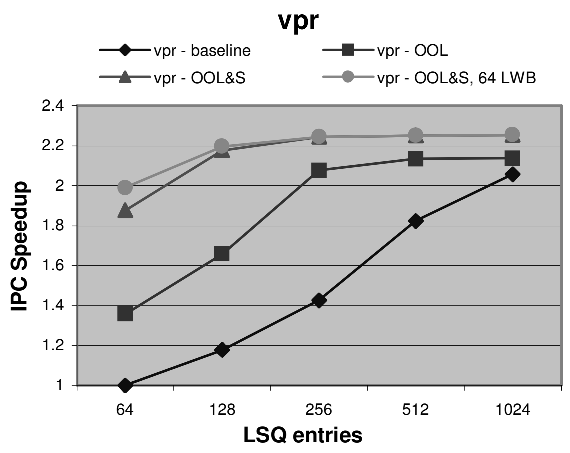

Figure 5: IPC Speedup for vpr: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

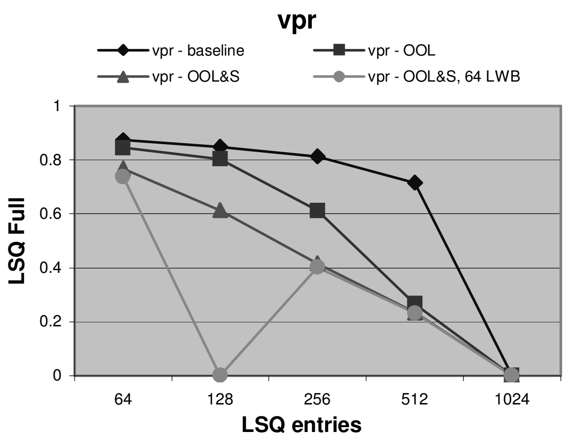

Figure 6: Percentage of time LSQ is full for vpr: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

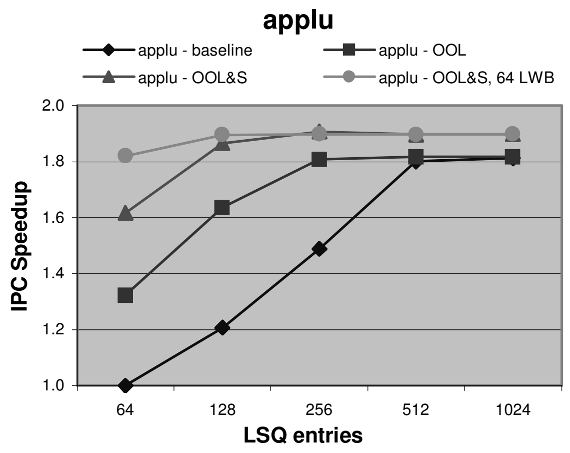

Figure 7. IPC Speedup for applu: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

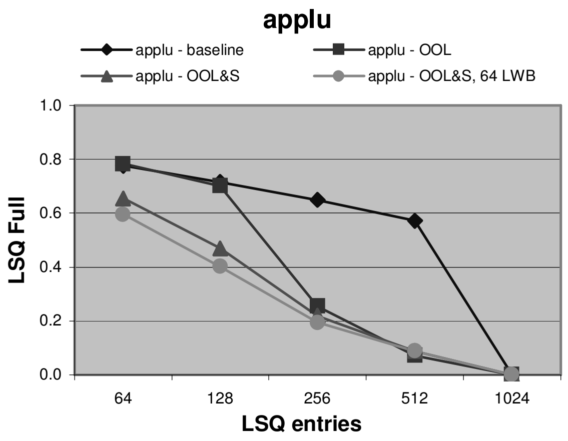

Figure 8: Percentage of time LSQ is full for applu: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

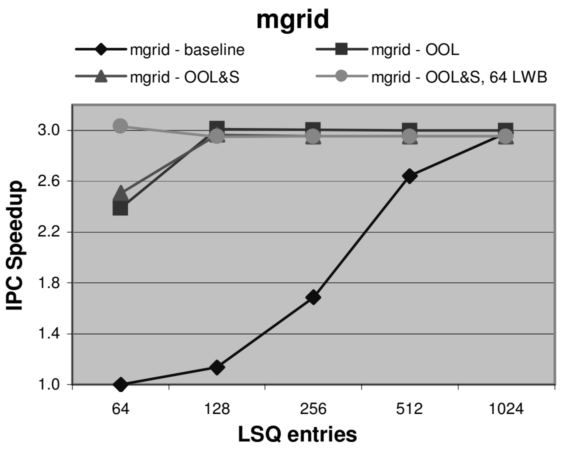

Figure 9: IPC Speedup for mgrid: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

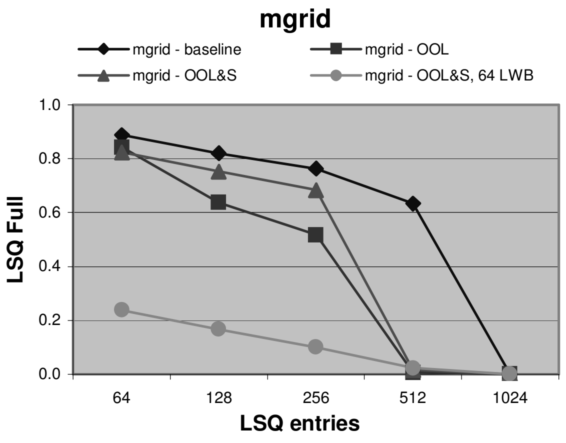

Figure 10: Percentage of time LSQ is full for mgrid: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

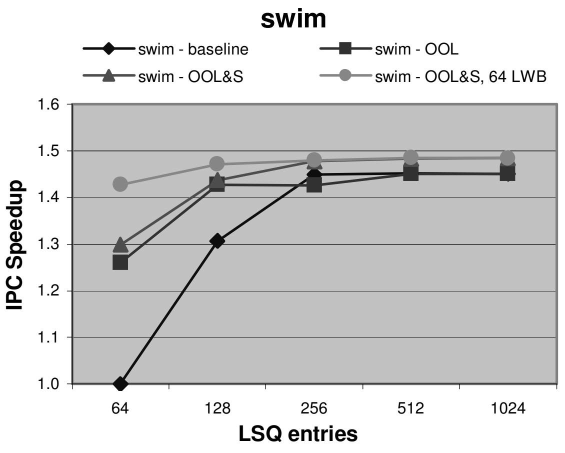

Figure 11: IPC Speedup for swim: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

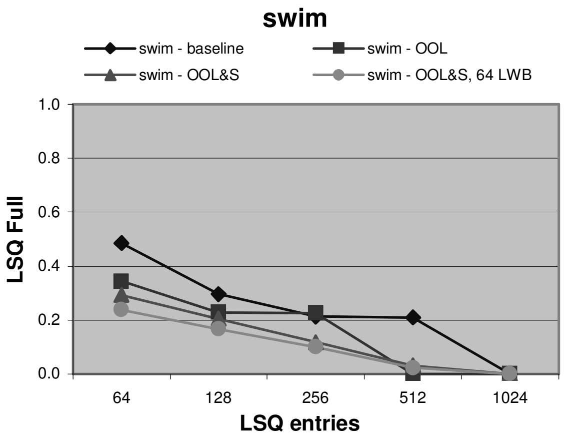

Figure 12: Percentage of time LSQ is full for mgrid: baseline, out-of-order loads, out-of-order loads & stores, and out-of-order loads & stores with a load-wait-buffer

6 Conclusion

One of the largest impedances in performance for high-end processors is the time to service a memory access. Even though most high end processors these days can execute instructions out or order, if the buffer holding the memory access instructions fills up (the LSQ), then the processor still needs to wait for the memory system to service the cache miss before it can continue dispatching instructions. In this paper, we proposed several different optimizations to help reduce occupancy of LSQ. The first method we analyzed was to reduce the amount of time that a load instruction stayed in the LSQ. A LSQ entry for a load is essentially not needed once it has passed the writeback stage, therefore, we can free the LSQ entry after this stage. The second optimization was to implement a similar idea concerning stores. This was a bit more complicated due to data integrity issues of writing to the cache, but the data integrity of the program can be ensured by following a small set of rules. Thus, stores can actually free their associated LSQ entry after they are issued. Finally, we also implemented a Load Wait Buffer (LWB). While loads are waiting for the memory system to service their cache misses, they are sitting idle in the LSQ, impeding the processor from dispatching another memory access instruction. Our LWB allows long latency loads to be removed from the LSQ while their cache misses are serviced. A LWB of 64 entries, combined with the out-of-order load and store execution, can provide performance improvements up to 303%.

Our simulations show that reducing the occupancy of the LSQ does indeed improve the performance of the processor. All three of the optimizations focused on emptying out the LSQ. The largest performance increase came when the 64 entry LWB caused a significant decrease in LSQ occupancy, which shows that the size and structure of LSQ is indeed a large bottleneck of the system.

References

[1] R. Lebeck, J. J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. “A Large, Fast Instruction Window for Tolerating Cache Misses”. *29th International Symposium on Computer Architecture*, May 2002.

[2] J.M. Parcerisa and A. Gonzalez, “The Latency Hiding Effectiveness of Decoupled Access/Execute Processors”. *24th. Euromicro Conference*, Aug. 1998.

[3] H.-Y. Hwang, R.-M. Shiu, Jean Jyh-Jiun Shann. “An X86 Load/Store Unit with Aggressive Scheduling of Load/Store Operations”. *International Conference on Parallel and Distributed Systems,* 1998.

[4] G. S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers”. *IEEE Transactions on Computers*, Mar. 1990.

[5] Platt, J.R. Strong inference. *Science*, Oct. 1964.