Duplicating and Deconstructing Virtual Load/Store Queues

Vikas Garg
Computer Sciences Department
Univ. of Wisconsin
Madison, WI 53706
vikas@cs.wisc.edu

Sonal Agarwal
Department of Electrical and Computer Eng.
Univ. of Wisconsin
Madison, WI 53706
sagarwal1@wisc.edu

ABSTRACT

Virtual load/store queues (VLSQs) within existing physical load/store queues (LSQ) have been proposed as an effective mechanism for reducing energy losses and increasing performance. The VLSQ restricts reordering of memory operations by limiting the number of memory instructions visible to the issue logic. This decreases the amount of execution time wasted in replay traps and leads to power savings by reducing the number of cache accesses, pipeline flushes, and re-executed instructions.

Our simulation-based evaluation of a VLSQ supports the claim that it reduces power consumption by controlling speculation in the execution of memory instructions, with minimal impact on performance. However, our simulation results also show that simply reducing the physical load/store queue size can be more effective than using a VLSQ. For a range of ROB sizes, reducing the LSQ size to 16 or using a VLSQ of size 16, has a similar effect on performance. However, compared to a VLSQ, a reduced LSQ provides an additional reduction of 1% to 10% in trap overhead, 2% to 10% in L1 cache accesses, 1% to 3% in L1 cache misses, and 2% to 22% in the number of pipeline operations.

Our results show that for conserving power, stalling the pipeline earlier on memory operations is better than delaying the stalls until the subsequent memory dependent ALU operations.

1. INTRODUCTION

Large reorder buffer (ROB), issue queues and load/store queues (LSQ) are used in aggressive out-of-order processors to achieve high performance [2, 9, 10]. Out-of-order issue of memory instructions in a processor can conflict with the processor's memory consistency model and lead to frequent flushing of the pipeline and re-execution of speculatively executed instructions (*replay traps*). Speculative execution of memory instructions can also increase the number of cache accesses and misses. This increase in the number of replay traps and cache accesses/misses due to the reordering of memory instructions leads to wastage of both energy and performance. Results in a recent paper [7] show that increasing the ROB size from 80 to 512 in an 8-wide superscalar processor leads to an increase of 10-40% in the total execution overhead, 10-60% in the number of L1 data cache accesses, and 10-20% in L1 data cache misses. It also increases the number of replay traps by a factor of 6.

Jaleel and Jacob [7] use a virtual load/store queue (VLSQ) within the existing physical load/store queue to reduce memory reordering, while allowing for aggressive speculative execution of instructions that are not dependent on memory operations. This results in a significant reduction in the number of replay traps, cache accesses, and cache misses. They report a net power saving of 10-22% with a performance degradation of only 2-5%.

We implemented VLSQ using a simulation environment and processor configuration similar to that used by Jaleel and Jacob [7]. Our results confirm that the total execution overhead, cache accesses and cache misses increase with an increase in the out-of-order execution capabilities. Our results also validate the effectiveness of using VLSQ to achieve power savings by reducing the negative effects of aggressive speculative execution of memory instructions.

The VLSQ paper [7] does not compare the benefits of virtually reducing the load/store queue size to physically reducing the load/store queue size. In this paper we compare the reduction of the physical load/store queue size to using a comparably sized VLSQ for power/performance management. Our results show that reducing the LSQ size has a bigger impact on power and performance than using a VLSQ. Across different ROB sizes, reducing the LSQ size to 16, or using a VLSQ of size 16, has similar impact on performance. However, reducing the LSQ size results in an additional 2-22% reduction in the number of operations in the front end of the pipeline and 2-10% reduction in the number of L1 cache accesses, as compared to using a VLSQ.

Our results show that although the VLSQ does restrict the reordering of memory operations and prevents the load/store queue from becoming a bottleneck, it is really only shifting the bottleneck to the issue queue for ALU operations. This suggests that throttling the memory operations causes the memory dependent ALU operations to block the pipeline and simply stalling the pipeline in the map stage is no worse than delaying the stalls until the issue stage. Our results further show that for conserving power, stalling the pipeline earlier for memory operations is better than delaying the stalls until the subsequent memory dependent ALU operations. These results corroborate the results of Karkhanis, Smith, and Bose [8]. They show that for achieving maximum energy savings with minimum performance loss, throttling of the pipeline earlier in the fetch stage is more effective than throttling later in the decode or issue stage.

2. VLSQ

Aggressive out-of-order processors use a large issue window and a large load/store queue to exploit instruction level parallelism. Such a design supports high degree of instruction reordering, and specifically, the large load/store queue leads to a significant reordering of memory instructions. This can negatively affect the program execution in two ways:

- **Increased Replay Traps:** Out-of-order issue of memory instructions can cause data dependency violations. For processors that do not support selective re-execution, data dependency violations lead to the flushing of the pipeline and re-execution of instructions that were executed speculatively (*replay traps*). Increase in the reordering of memory instructions increases the number of replay traps and the processor ends up wasting more energy executing instructions that will eventually get discarded.
- **Increased Cache Misses:** Speculative execution of memory instructions leads to an increase in the number of cache accesses and can negatively affect an application's cache locality. This leads to an increase in the frequency of conflict misses and results in wasted energy.

The motivation for implementing VLSQs is to reduce the speculative execution of memory instructions by throttling the load/store queue size as seen by the issue logic, while at the same time allowing for the use of a large load/store queue to support a large ROB size and speculative execution of non-memory dependent ALU instructions.

VLSQ imposes a virtual window on the physical LSQ to reduce the reordering of memory instructions. This virtual window is similar to the sliding window concept used in networking protocols [12]. The VLSQ provides a sliding window in the physical LSQ, so that during instruction scheduling (issue), only the memory instructions that lie within the sliding virtual window are allowed to be executed/issued. The size of the sliding window (VLSQ size) determines the number of memory instructions that will be considered for execution. Pending memory instructions for which operands are ready, but are not within the virtual window, will not be considered for execution. The window slides forward as the memory instructions at the head of the virtual window are issued. This reduction in the reordering of memory instructions reduces the number of replay traps and cache accesses/misses.

The VLSQ is implemented using two additional queue pointers, *VirtualHead* and *VirtualTail*, that point into the conventional LSQ. The VLSQ size gives the maximum distance allowed between the *VirtualHead* and the *VirtualTail*. The *VirtualHead* always points to the first non-issued memory instruction and moves forward only when the memory instruction in the slot it points to is issued. The *VirtualTail* moves forward when: (1) *VirtualHead* moves forward and *VirtualTail* is not at the end of the queue; (2) a new entry is inserted into the LSQ and the distance between the *VirtualHead* and the *VirtualTail* is less than the VLSQ size. Setting the VLSQ size to infinity means that the virtual window size is same as the physical LSQ size.

Table 1: Processor Configuration

Component	Parameters
I-Cache	64KB, 2-way, 1-cycle
D-Cache	64KB, 2-way, 3-cycle
L2-Cache	2MB, 4-way, 15-cycle
TLB	128 entry, fully associative
Main Memory	1.3 GB/s DDR SDRAM [13]
Branch Predictor	2048 lines bimodal, 2-level
Branch Target Buffer	4096 entry, 4-way
Inst. Fetch Width	8
Inst. Map Width	8
Integer Issue Width	8
FP Issue Width	4
Inst. Commit Width	11
Integer Functional units	4 ALU, 4 MULT
FP Functional units	1 ALU, 1 MULT
Integer Clusters	2
FP Clusters	1
Store Wait Table	1024 entry, reset every 16384 inst.

Table 2: Baseline Out-of-Order Configurations

ROB Size	Issue Queue INT/FP	LSQ Size	VLSQ Size
80	20/15	32/32	Infinite
128	40/30	64/64	Infinite
256	80/60	128/128	Infinite
512	160/120	256/256	Infinite

The VLSQ essentially decouples the LSQ seen by the front end of the pipeline from the LSQ seen by the issue logic. For the instruction fetch and decode stages, VLSQ presents a large traditional LSQ, so that the front end of the pipeline does not stall due to lack of space in the LSQ. At the same time it presents a smaller LSQ to the issue logic to reduce the amount of speculation in the execution of memory operations. This decoupling is based on the assumption that the throttling of memory instructions will not cause other pipeline stages to stall due to memory dependencies.

3. SIMULATION SETUP

We use an execution driven Alpha 21264 simulator *sim-alpha* [6] for all of the simulations. Table 1 gives the processor configuration used in our simulations; as much as possible, these match the parameters used in the original VLSQ study, with the exception that we do not simulate a stride pre-fetcher. A stride pre-fetcher will have some impact on performance numbers [3], but this variation in the absolute numbers will be similar for both VLSQ and LSQ. Also a stride pre-fetcher will not change the relative impact on power and performance of using VLSQ or LSQ to control the out-of-order capabilities of the processor.

To study a range of out-of-order capabilities, we varied the ROB size, issue queue size, LSQ size and VLSQ size. Table 2 enumerates the four sets of baseline out-of-order configuration parameters used in our experiments. An infinite VLSQ

size means that VLSQ is not used and corresponds to a conventional load/store queue. To duplicate the VLSQ results reported in [7], we use the baseline configuration and VLSQ sizes of 2, 4, 8, 16, 32, 64, and Infinity. To study the effect of reducing the LSQ size, we performed the simulations with LSQ sizes ranging from 2 to 64, for each of the baseline configurations.

Like the original study, we used a subset of the SPEC2000 benchmark suite: `art, applu, gcc, gzip, mgrid, mcf, twolf,` and `swim`. The data were gathered for 500 million instructions after fast forwarding the first 2 billion instructions. This simulation setup matches the simulation setup used in [7].

Unless stated otherwise, all the results presented in this paper are the arithmetic mean of the results obtained for each of the eight individual SPEC2000 benchmarks used. Instead of reporting power savings, we report the reduction in the number of pipeline operations. There is a direct correlation between the power consumption numbers reported in the original VLSQ study and the number of operations in various stages of the pipeline [4, 11]. So the percentage reduction in number of pipeline operations is equivalent to the percentage reduction in power consumption. In addition to the power savings due to reduction in pipeline operations, a smaller load/store queue will consume less power compared to VLSQ because of the reduced physical size of the load/store queue.

4. RESULTS

We present the results for the impact of variation in VLSQ and LSQ on performance, stalls, traps, cache accesses/misses, and pipeline operations. For all of the results presented, when the VLSQ size is varied, the LSQ size is set to the physical LSQ size used in the baseline configuration. The data point labeled "Inf" in the VLSQ graphs means that an infinite VLSQ size is used *i.e.* the virtual load/store queue is disabled. When the LSQ size is varied, the VLSQ is set to infinity. The data point labeled "Base" in the LSQ graphs corresponds to the LSQ size used in the baseline configuration. For a given ROB size the data points corresponding to "Base" for LSQ and "Inf" for VLSQ represent the identical baseline configuration.

4.1 Performance

Figure 1 shows the variation in CPI for different VLSQ sizes. For a VLSQ size of 2 there is a performance degradation of 17-27% as we increase the ROB size from 80 to 512. These results closely match the numbers presented in the original study, where they show a performance degradation of 15-30% for small VLSQ sizes.

As the VLSQ size is increased, the amount of throttling decreases, and the performance approaches that of the baseline configuration. For a VLSQ size of 16 or greater, the CPI is within 2% of the baseline CPI. This indicates that we can use a VLSQ size of 16 without having a significant impact on performance. These numbers also match the results presented in the original study where the authors show that for VLSQ sizes of 16 and 32, the performance is within 2-5% of the traditional load/store queue.

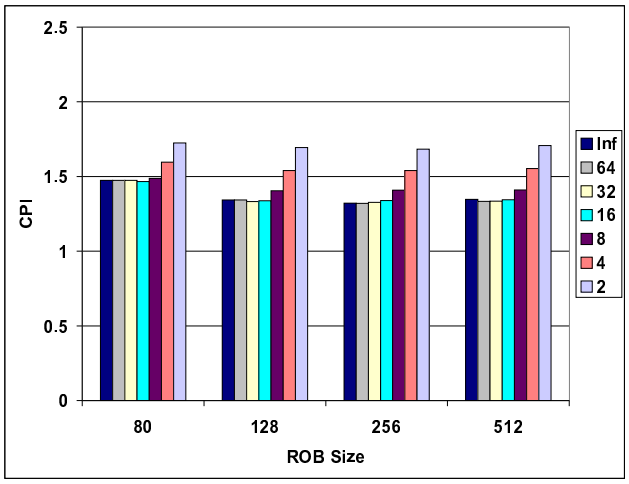

Figure 1: VLSQ Performance

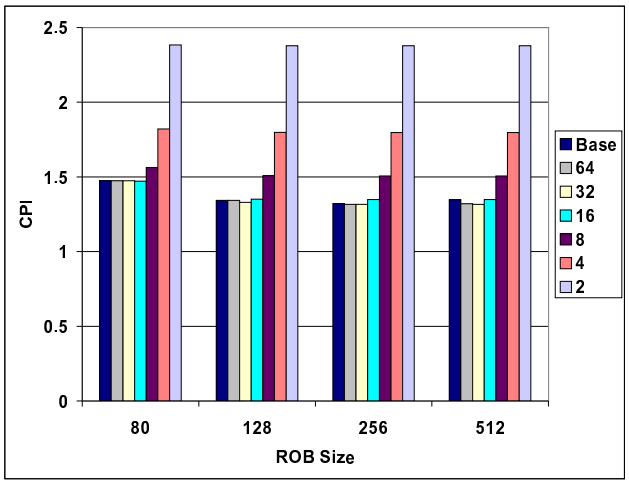

Figure 2: LSQ Performance

Figure 2 shows the variation in CPI for different LSQ sizes. For an LSQ size of 2 there is a performance degradation of 61-80% for various ROB sizes. We show in the following sections that the bigger drop in performance for LSQ is due to the reduced amount of speculation and an increase in the number of cycles that the processor is stalled in the map stage waiting for empty load/store queue entries. As the LSQ size is increased, performance approaches the baseline performance. For an LSQ size of 16 or greater, there is almost no performance degradation, and the CPI numbers match the CPI numbers for VLSQ.

4.2 Stalls

To get a better understanding of the impact of variation in VLSQ and LSQ on the processor pipeline, we measured the distribution of stall cycles. Stall cycles are the the cycles for which, due to lack of resources, the number instructions mapped is less than the map width. We classify these stall cycles into three categories:

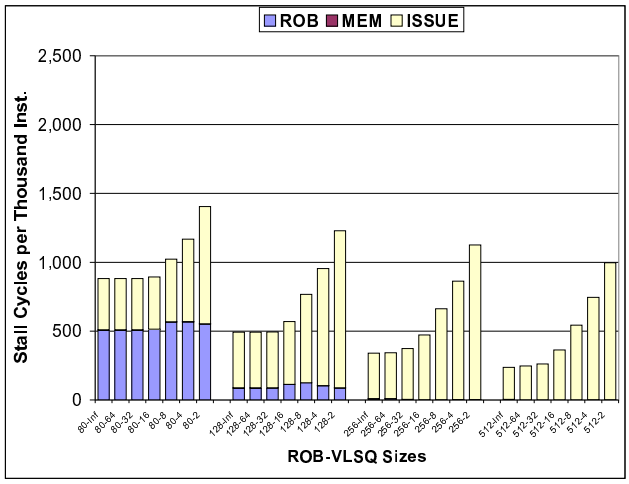

Figure 3: VLSQ Stall Cycle Distribution

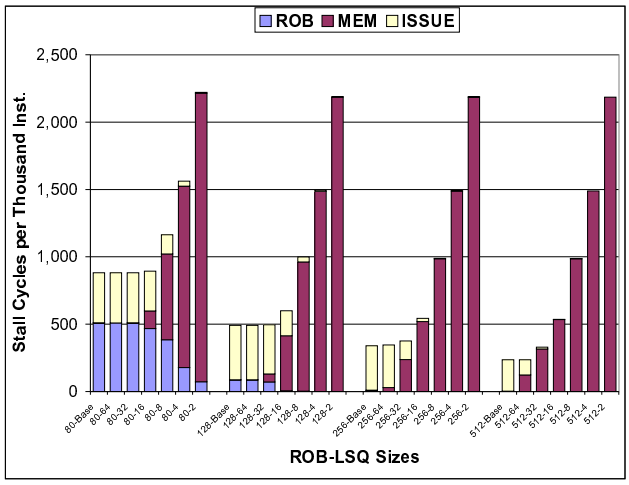

Figure 4: LSQ Stall Cycle Distribution

- **ROB:** Stalls due to lack of entries in the reorder buffer and the physical register file.
- **MEM:** Stalls due to lack of entries in the load/store queues
- **ISSUE:** Stalls due to lack of entries in the Integer/FP issue queues.

Figure 3 shows the variation in the total number of stall cycles per thousand committed instructions, broken into the three categories for different VLSQ sizes; Figure 4 plots the total number of stall cycles, and the stall distribution for various LSQ sizes. For the baseline configurations, the total number of stall cycles per thousand committed instructions decreases from 881 to 236 as we increase the ROB size from 80 to 512. For large VLSQ and LSQ sizes, both the LSQ and VLSQ have a similar number of stall cycles across all the different ROB sizes. For a given ROB size, as we decrease the VLSQ and LSQ sizes, the total number of stall cycles increases significantly. Reducing the LSQ size causes a bigger increase in the total number of stall cycles compared to using VLSQ. For a ROB size of 512, reducing the LSQ size to 2 causes the number of stall cycles per thousand instructions to go up by 1950, compared to an increase of 761 for a VLSQ of size 2.

Beyond a ROB size of 128, the reorder buffer and the physical register file are no longer a bottleneck. For large ROB sizes, all the stalls are caused by the issue queue or the load/store queue. For a given ROB size, as we reduce the LSQ size, the number of MEM stalls increase and overshadow the ROB and ISSUE stalls. For a ROB size of 128, as the LSQ size is reduced from 64 to 2, the contribution of MEM stalls to the total number of stall cycles, goes from less than 10% to almost 100%. This shows that reducing the size of LSQ causes it to become a bottleneck. The small load/store queue fills up with instructions that miss in the L1 cache. Due to lack of space in the load/store queue, no more instructions can be mapped, and this stalls the entire pipeline.

For a given ROB size, as we reduce the VLSQ size, the number of ISSUE stalls goes up, while there is no increase in the number of MEM stalls. For a ROB size of 128, decreasing the VLSQ size from 64 to 2 results in the number of ISSUE stalls to increase from around 80% to more than 90% of the total number of stalls, while the number of MEM stalls stays at less than 1%. This shows that a large physical load/store queue helps to avoid stalls due to instructions that miss in the L1 cache by providing adequate buffering for the delayed instructions. Although there is a large pool of pending memory instructions, the VLSQ throttles the execution of these instructions to avoid memory reordering. This slows down the effective rate of completion of memory instructions, and the ALU instructions that are dependent on these memory instructions have to wait longer in the issue queue before they can be issued. This causes the issue queue to back up and become the bottleneck. So, although the physical load/store queue is not a bottleneck anymore, the reduced speculation in memory instructions is becoming a bottleneck indirectly, by causing the memory dependent ALU operations to stall. The increase in the issue queue size as the ROB size is increased helps in alleviating the issue bottleneck, as indicated by the reduction in the number of stalls for a ROB size of 512.

4.3 Replay Traps

Figures 5 and 6 give the percentage of total execution cycles spent in handling replay traps for different VLSQ and LSQ configurations respectively. Just as in [7], the execution cycles lost in handling traps is determined by keeping track of the difference in number of cycles between the original instruction fetch and the subsequent re-fetch due to a replay trap. The number of cycles spent in trap handling corresponds to the extra work that the processor performs and then discards. For the baseline configuration, as the ROB size increases from 80 to 512, the trap overhead increases from 24% to 45% of the total execution time, compared to a 25-60% increase in the original VLSQ study. This confirms the results that as the amount of speculation is increased, the percentage of wasted execution cycles goes up. Reducing LSQ size is more effective in decreasing the number of cycles spent handling traps. For a ROB size of 512 and a queue size of 16, VLSQ reduces the percentage of time wasted in

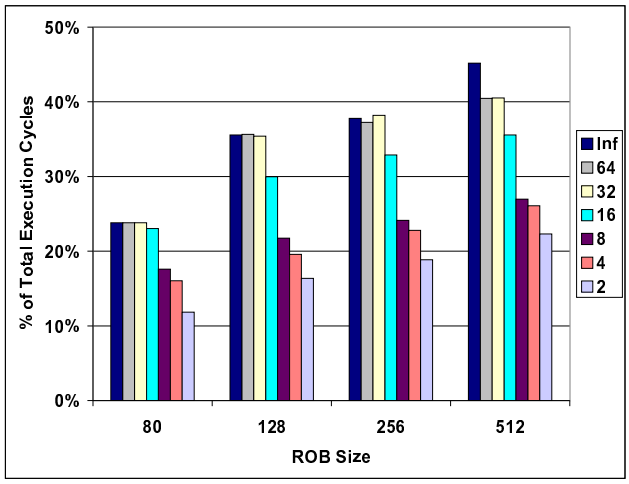

Figure 5: VLSQ Replay Trap Overhead

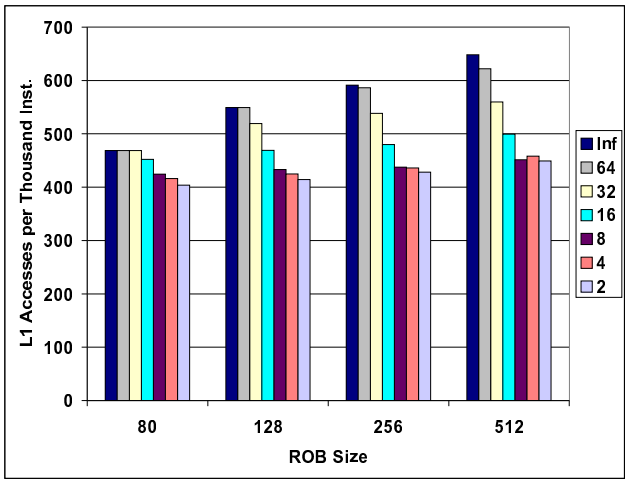

Figure 7: VLSQ L1 Accesses

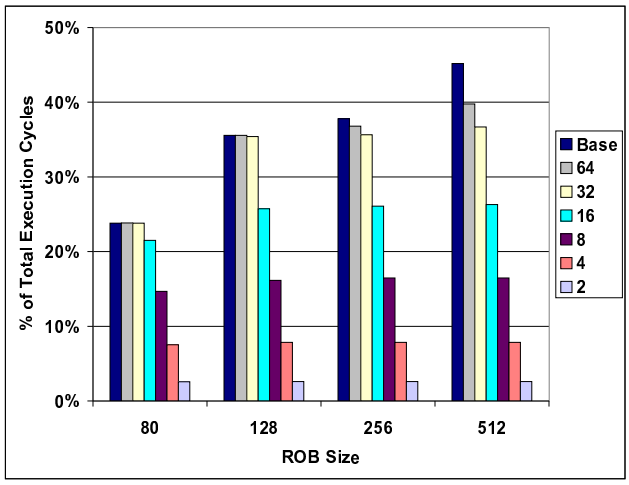

Figure 6: LSQ Replay Trap Overhead

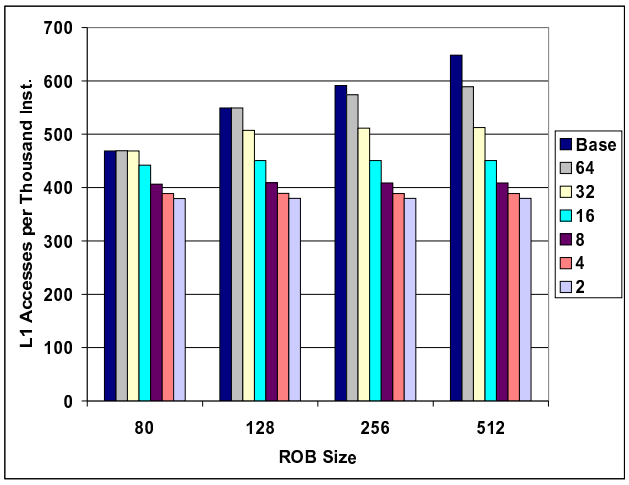

Figure 8: LSQ L1 Accesses

replay traps by 9%, compared to a reduction of 19% for LSQ. This indicates that the VLSQ is allowing more speculative execution and causing more wasted computation than a small LSQ. This corroborates the results presented in the previous section where we show that reducing the LSQ size results in a larger increase in the number of stall cycles and hence a bigger reduction in speculative execution, compared to a VLSQ of the same size.

Both the study presented here and [7] use a 1024-entry store-wait table that is cleared after every 16K instructions [5, 9]. The store-wait table is indexed using the PC of the instruction causing the load-store replay trap. If during instruction fetch the store-wait table indicates a load-store dependency, the load is not issued until all the prior stores are resolved. The store-wait table is effective in avoiding load-store replay traps, our results show that the store-wait eliminates 95% of the load-store replay traps. The simulation setup also causes replay traps on load-load dependencies. Load-load dependencies are not a problem in uni-processor machines, but need to be handled in multiprocessor systems to avoid memory consistency violations [1]. Even in multiprocessor systems, load-load dependencies should cause a replay trap only if there is an intervening commit from another processor. As a result, both the original paper and our study overestimate the reduction in replay traps by blindly flushing the pipeline on all load-load dependencies. Our simulation results show that disabling the load-load replay traps reduces the trap frequency from one trap per 328 committed instructions to one trap per 1584 committed instructions, for a ROB size of 512. This reduction in replay trap frequency reduces the replay trap overhead by 53% and results in a 16% improvement in the CPI for a ROB size of 512. Even with load-load traps disabled, both the VLSQ and a small LSQ provide similar relative reductions in the trap overhead.

4.4 Cache Behavior

Figures 7 and 8 give the number of L1 data accesses per thousand committed instructions for the different VLSQ and LSQ configurations. For the baseline configuration, as the ROB size is increased from 80 to 512, the number of L1 accesses increases by 38%. This matches the increase in the

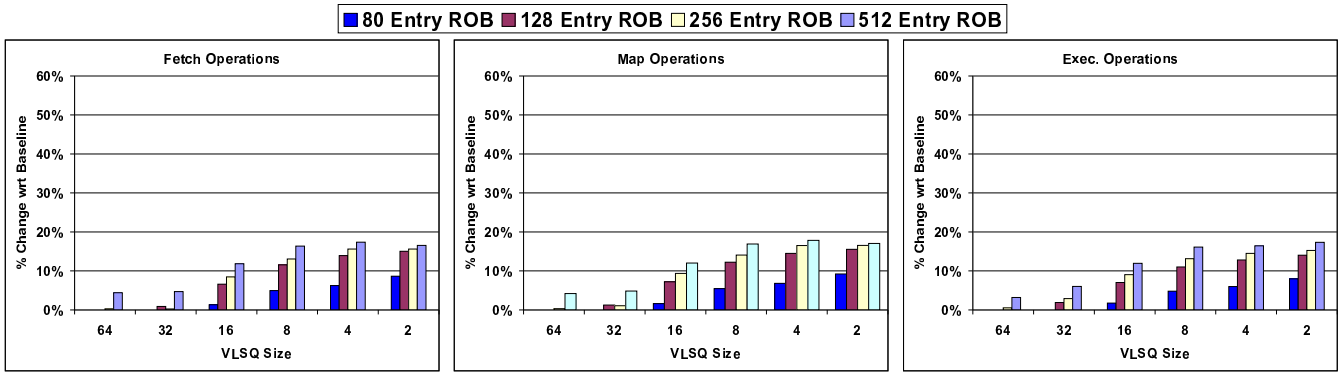

Figure 9: VLSQ Reduction in Pipeline Operations

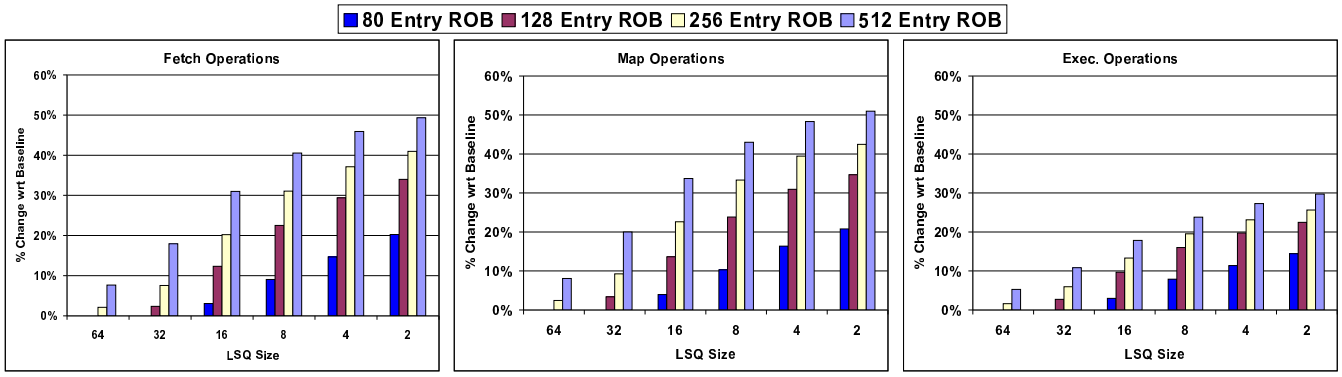

Figure 10: LSQ Reduction in Pipeline Operations

time spent handling traps as the ROB size is increased, and corresponds to the speculatively executed instructions that are discarded because of pipeline flushes. Using a VLSQ, or a smaller LSQ, reduces the number of replay traps and this translates directly into a reduction in the number of L1 cache accesses. Just as with stalls and replay traps, reducing the LSQ size has a bigger impact on L1 cache accesses, compared to using a VLSQ. A VLSQ of size 16 achieves a reduction of 15% in the number of cache accesses per thousand committed instructions, compared to a reduction of 18% with an LSQ of size 16. The number of cache misses per thousand committed instructions increases from 91 to 96, as the ROB size is increased from 80 to 512. Using a VLSQ size of 16 leads to a reduction of 0-2% in the number of cache misses, compared to a reduction of 1-5% with an LSQ of size 16.

4.5 Pipeline Operations

Figure 9 gives the reduction in the number of operations in the various stages of the pipeline for VLSQ and Figure 10 gives the corresponding results for LSQ. These numbers are normalized to the baseline configuration for each of the ROB sizes. For a ROB size of 512, using a VLSQ of size 16, reduces the number of operations in the various stages of the pipeline by an average of 12%. With an LSQ of size 16, the number of operations goes down by an average of 27% in the various pipeline stages. The results show that decreasing the amount of speculation (smaller VLSQ and LSQ sizes) reduces the number of pipeline operations, with LSQ being more effective than VLSQ in reducing the number of operations, and hence power consumption, in the various pipeline stages.

4.6 Discussion

To better understand the impact of using VLSQs and to compare their use with simply reducing the physical LSQ size, we compare the simulation results for the baseline configuration, a VLSQ size of 16 and an LSQ of size 16, across all the different ROB sizes. We present data for VLSQ and LSQ sizes of 16 because the performance for a VLSQ size of 16 is comparable to the performance for an LSQ of size 16, and for both of them performance is within 2% of the performance for the baseline configurations.

In Table 3, the "Base" columns correspond to the baseline configurations as defined in Table 2. The "VLSQ" configurations are derived from the "Base" configurations by changing the VLSQ size from infinity to 16, and the "LSQ" configurations are derived from the "Base" configurations by changing the LSQ size to 16.

The total number of stall cycles per thousand committed instructions goes down as we increase the ROB size. Both the VLSQ and LSQ have a greater number of total stalls compared to the baseline configurations. The LSQ and VLSQ have a different distribution of stall cycles across the three categories (ROB, MEM and ISSUE). LSQ has many more MEM stalls, due to lack of entries in the load/store queue,

Table 3: Comparison of Baseline, VLSQ size of 16 and an LSQ of size 16

Rob Size	80			128			256			512		
Configuration	**Base**	**VLSQ**	**LSQ**	**Base**	**VLSQ**	**LSQ**	**Base**	**VLSQ**	**LSQ**	**Base**	**VLSQ**	**LSQ**
CPI	1.47	1.47	1.47	1.34	1.34	1.35	1.32	1.34	1.35	1.35	1.34	1.35
ROB Stall Cycles	507	513	467	86	113	6	8	2	0	1	0	0
MEM Stall Cycles	3	2	131	2	0	409	2	0	519	3	0	534
ISSUE Stall Cycles	371	379	295	404	455	185	330	470	24	233	363	1
Total Stall Cycles	881	893	893	492	569	600	340	473	543	236	364	536
Traps	24%	23%	22%	36%	30%	26%	38%	33%	26%	45%	36%	26%
L1 Accesses	469	452	442	549	469	451	591	480	451	648	499	451
L1 Misses	91	91	90	94	93	91	96	93	91	96	94	91
Fetch Ops	0%	1%	3%	0%	7%	12%	0%	8%	20%	0%	12%	31%
Map Ops.	0%	2%	4%	0%	7%	14%	0%	9%	23%	0%	12%	34%
Exec Ops.	0%	2%	3%	0%	7%	10%	0%	9%	13%	0%	12%	18%

and has almost no stalls in the issue queue. The VLSQ, on the other hand, has almost no stalls in the load/store queue, and most of the stalls are in the issue queue because the issue queues fill up with memory dependent ALU instructions.

The percentage of execution cycles lost due to replay traps goes up as the ROB size is increased. This corresponds to the increased out-of-order execution and greater misspeculation. This increase in trap overhead represents wasted computation cycles. Both the VLSQ and LSQ help reduce the replay trap overhead, with the LSQ being more effective than the VLSQ.

Similar to the replay trap overhead, L1 accesses per thousand instructions go up with an increase in the ROB size and LSQ is more effective than VLSQ in reducing the number of L1 accesses. The pipeline operations are measured as the reduction in the number of operations, relative to the baseline configuration, for a given ROB size. Consequently, the baseline numbers are all zero. Overall, both the VLSQ and LSQ provide a significant reduction in the number of pipeline operations, with LSQ having a bigger impact than VLSQ.

The results shows that the VLSQ is achieving its goal of preventing the negative effects of reordered memory instructions, without causing the load/store queue to become a bottleneck. The results also show that some amount of stalling in the front end of the pipeline is useful for reducing the amount of speculative execution and helps to reduce the unnecessary execution and flushing of misspeculated instructions. Physical reduction in the LSQ size causes the pipeline to stall sooner rather than later, and is more effective in reducing the negative effects of reordered memory instructions.

5. CONCLUSIONS

Simulation results using our implementation of a VLSQ confirm the results presented by Jaleel and Jacob [7]. Using a VLSQ within a conventional LSQ can reduce the negative effects of reordered memory instructions. It helps to reduce the speculative execution of memory instructions without making the LSQ a bottleneck. Using a VLSQ of size 16 has almost no impact on performance, and can lead to reduced replay trap overhead, less cache activity, and significant power savings.

Our results also show that reducing the physical LSQ size results in a performance drop that is similar to the reduction in performance with a comparably-sized VLSQ. However physically reducing the load/store queue size results in lower trap overhead, reduced cache activity, and higher power savings compared to using a VLSQ. This analysis does not take into consideration the power savings because of the reduced physical size of the LSQ. Factoring in these savings will further increase the effectiveness of using a smaller LSQ to conserve power, compared to using the VLSQ scheme proposed in [7].

The VLSQ provides an easy mechanism for dynamic runtime control over the power/performance trade-off by reducing the reordering of memory instructions at the issue stage. However, for dynamically controlling the degree of speculation, it will be advantageous to dynamically restrict the number of load/store queue entries available during the mapping stage. Restricting speculation at the map stage will provide more power saving, for a similar drop in performance, compared to restricting the speculation at issue stage. In fact it might be useful to just shut down a section of the LSQ, instead of virtually reducing the load/store queue size. This will result in more power savings because of the reduction in the active LSQ size.

Our results show that stalling earlier in the pipeline to control the amount of out-of-order execution is an effective way to reduce the number of pipeline flushes and wasted execution power. For power savings, it will be useful to provide some kind of a predictive feedback loop that stalls the front end of the pipeline in a manner similar to the Just In Time Instruction Delivery proposal [8]. If speculative execution results in a large number of replay traps, then it makes sense to stall pro-actively in the front end of the pipeline, instead of waiting for the pipeline flushes.

6. ACKNOWLEDGEMENTS

We would like to thank Jim Smith for his invaluable support and guidance. The authors will also like to thank Saisanthosh Balakrishnan, Aamer Jaleel, and the anonymous reviewers for their feedback and help in calrifying some of the underlying concepts.

7. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: An Efficient, Scalable Alternative to Reorder Buffers. *IEEE Micro*, 23(6):11–19, 2003.

[3] J.-L. Baer and T.-F. Chen. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Trans. Comput.*, 44(5):609–623, 1995.

[4] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[5] G. Z. Chrysos and J. S. Emer. Memory Dependence Prediction Using Store Sets. In *ISCA*, pages 142–153, 1998.

[6] R. Desikan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: a validated execution driven Alpha 21264 simulator, 2001.

[7] A. Jaleel and B. Jacob. Using Virtual Load/Store Queues (VLSQs) to Reduce the Negative Effects of Reordered Memory Instructions. In *HPCA '05*, pages 191–200, 2005.

[8] T. Karkhanis, J. E. Smith, and P. Bose. Saving energy with just in time instruction delivery. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 178–183, New York, NY, USA, 2002. ACM Press.

[9] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[10] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 59–70, Washington, DC, USA, 2002. IEEE Computer Society.

[11] K. Natarajan, H. Hanson, S. W. Keckler, C. R. Moore, and D. Burger. Microprocessor pipeline energy analysis. In *ISLPED '03: Proceedings of the 2003 international symposium on Low power electronics and design*, pages 282–287, New York, NY, USA, 2003. ACM Press.

[12] RFC793. Transmission Control Protocol. September 1981. DARPA Internet Program Protocol Specification.

[13] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *SIGARCH Computer Architecture News*, 33(4):100–107, 2005.