LATENCY-AWARE WRITE BUFFER RESOURCE CONTROL IN MULTITHREADED CORES

Shane Carroll and Wei-Ming Lin

Department of Electrical and Computer Engineering, The University of Texas at San Antonio, San Antonio, Texas, USA

ABSTRACT

In a simultaneous multithreaded system, a core's pipeline resources are sometimes partitioned and otherwise shared amongst numerous active threads. One mutual resource is the write buffer, which acts as an intermediary between a store instruction's retirement from the pipeline and the store value being written to cache. The write buffer takes a completed store instruction from the load/store queue and eventually writes the value to the level-one data cache. Once a store is buffered with a write-allocate cache policy, the store must remain in the write buffer until its cache block is in level-one data cache. This latency may vary from as little as a single clock cycle (in the case of a level-one cache hit) to several hundred clock cycles (in the case of a cache miss). This paper shows that cache misses routinely dominate the write buffer's resources and deny cache hits from being written to memory, thereby degrading performance of simultaneous multithreaded systems. This paper proposes a technique to reduce denial of resources to cache hits by limiting the number of cache misses that may concurrently reside in the write buffer and shows that system performance can be improved by using this technique.

KEYWORDS

Simultaneous Multithreading, Write Buffer, Multithreaded Resource Control.

1. INTRODUCTION

Simultaneous multithreading (SMT) is a processor technique which exploits thread-level parallelism (TLP) by executing instructions from multiple threads concurrently [1]. Whereas a fine- or coarse grained multithreaded system switches between individual contexts, SMT architecture allows multiple threads to occupy the pipeline during the same clock cycle. An SMT system resembles that of a scalar processor with the exception of some parallel hardware in the instruction pipeline. With parallel pipeline structures, several threads may advance through separate pipeline paths in the same processor simultaneously. However, some of an SMT system's pipeline resources are shared, rather than parallel, and may be occupied by multiple threads at the same time. A typical pipeline organization in an SMT system is shown in Figure 1.

With the ability to choose from multiple threads in each clock cycle, short-latency hazards (e.g.,an instruction waiting for operands to become ready) which typically stall the pipeline may easily be masked by executing instructions from a different thread [2, 3]. This assists in keeping the pipeline full and improves performance over other architectures. With this technique, individual threads with low instruction-level parallelism (ILP) do not necessary prevent a processor from exploiting parallelism since TLP can be used to fill the delay slots with instructions from other threads [4].

However, simultaneous sharing of resources between all threads inevitably leads to bottlenecks and starvation, especially among storage resources [5, 6]. One shared storage resource in an SMT system is the write buffer. An SMT write buffer lies between a core's pipeline and its cache memory.

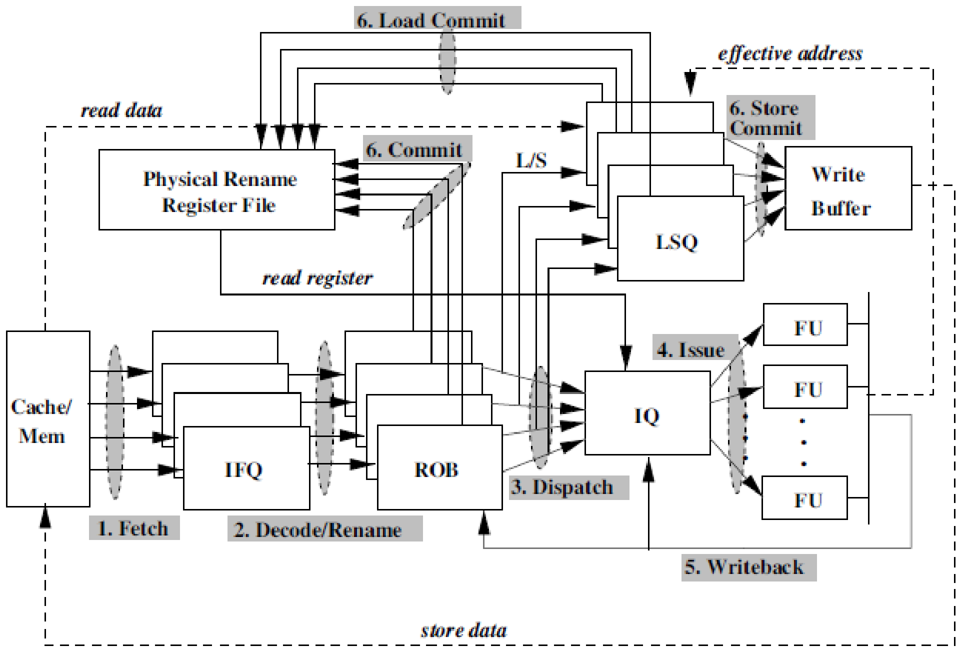

Figure 1: Pipeline Stages in a 4-Threaded SMT System

When a store instruction is retired from the pipeline, the result that is to be written to memory is first sent to the write buffer. Using a write-allocate cache policy, the write instruction's cache line is brought into level-one data (DL1) cache while the value is temporarily stored in a write buffer entry [7]. This buffering time can last between a single clock cycle (in the case of a DL1 cache hit) to several hundred clock cycles (in the case of a cache miss).

This paper analyzes the behavior and effect of cache misses in the write buffer and shows that an SMT write buffer is routinely occupied by a large number of long-latency cache misses. We then show that this situation greatly increases the likelihood of the write buffer being fully occupied, which stalls threads that have cache hits ready to commit. We subsequently show that by limiting the number of cache misses in the write buffer by suspending threads which have a cache miss at the head of their ROB when the write buffer is heavily occupied can significantly improve system performance in a multithreaded environment.

2. Related Work

Previous work has explored several techniques of resource control at various stages of the pipeline in SMT systems. Tullsen and Brown [8] showed that it was beneficial to release resources associated with a thread which has long cache latency pending. Similar to branch prediction flushing, a thread with a long latency memory operation would be flushed from the pipeline, allowing the other threads to consume the resources previously held by the idling thread.

Another resource control method is to prioritize which thread fetches next, rather than rotate in a fashion that is assumed to be fair [9, 10]. In an SMT system, one common technique of execution fairness is the round-robin approach, which chooses a thread in a modular, linear fashion when deciding which thread to execute in the current clock cycle. Rather than allowing each thread to fetch according to a fair rotation, the technique by Luo et al. [9] aims to predict which thread will best use the CPU's resources in the coming clock cycles and choose which thread should fetch more instructions in the current clock cycle.

An additional resource control method monitors the usage of each thread's resources and dynamically limits the resources which are allocated to each thread at runtime [11]. In addition to resource limiting, if one thread is not using resources they may be given to another thread. This technique aims to both limit and optimize the distribution of an SMT system's resources based on efficient use.

Some resource-partitioning methods [6] separate hardware into clusters and allow certain threads to occupy a specific cluster. In this manner, threads of different classifications take different paths through the SMT hardware. This separation of thread classes does not limit a thread within its assigned hardware and still shows improvement in performance via isolation of threads which negatively affect each other.

Thread coscheduling techniques [12, 13] have been shown to increase performance in SMT systems by selectively choosing which threads to run together. Moseley et al. [13] use a technique to predict the combined instructions per clock cycle (IPC) throughput of multiple threads based off their resource utilization and use this prediction as a metric for coscheduling. Tullson et al. [12] dynamically coschedule jobs based off performance, but also consider an assigned priority of each thread. This coscheduling method increases SMT performance while also assisting the operating system in achieving priority goals.

Hily and Seznec [14] determined that cache size and workload size have an optimal value pairing in an SMT system. In their paper, various properties of cache, including block size, associativity, and size relative to thread count were shown to exhibit beneficial behavior if setup differently than that of typical architectural configurations.

However, none of these SMT resource control methods attempt to control the write buffer allocation based on the latency of the next store instruction.

3. Simulation Environment

3.1. Simulator

The simulator used in this research is M-Sim [2], an SMT microarchitectural simulator. Based on SimpleScalar 3.0d and extended for SMT functionality, M-Sim is a cycle-accurate simulator which models key pipeline structures in an SMT system, including per-thread instruction fetch queues (IFQs), perthread reorder buffers (ROBs), a shared issue queue (IQ), a shared pool of functional units (FUs), a shared write buffer, and two levels of cache. M-Sim supports the Alpha instruction set, a 64-bit RISC architecture.

The parameters used throughout all simulations will be M-Sim's default configuration, which are outlined in Table 1. During simulation, write buffer sizes of both 16 and 32 entries will be used.

3.2. Workloads

The workloads used throughout the simulations will be chosen from the SPEC CPU 2006 suite of benchmarks. To choose workload combinations for multithreaded simulations, the benchmarks will be partitioned by ILP and chosen in sets of 4 and 8. The benchmarks with the highest IPC will be rated as H, those with lowest IPC will be rated as L, and those between will be rated as M. Workloads will be chosen based on a variety of ILP classification combinations. Chosen workload combinations and their corresponding ILP classes are shown in Tables 2 and 3 for 4-threaded and 8-threaded workloads, respectively.

4. LONG-LATENCYWRITE ANALYSIS

This section aims to analyze long-latency writes (cache misses) and show that they regularly occupy a disproportionately large number of write buffer entries, thereby filling up the write buffer and denying threads with short-latency writes (cache hits) from committing their next instruction, thus

Table 1: Simulation environment parameter setup

Parameter	Description
Bandwidth	8 Fetch / Dispatch / Issue / Writeback / Commit
IFQ/ROB/LSQ/IQ Size	32/128/48/32 entries
Function Units / Completion Latency / Issue Latency	Integer ALU: 4/1/1
	Integer Mult: 1/3/9
	Integer Div: 1/12/12
	Floating Point Add: 4/2/1
	Floating Point Mul: 1/4/1
	Floating Point Div: 1/20/19
	Floating Point Sqrt: 1/24/24
Level-1 Instruction Cache	512 sets / 64-byte blocks / 2-way assoc. / LRU
Level-1 Data Cache	256 Sets / 64-byte blocks / 4-way assoc. / LRU
Level-2 Unified Cache	512 sets / 64-byte blocks / 16-way assoc. / LRU
Write Buffers	16/32 64-byte entries
Branch Predictor	2048-entry Bimodal
Physical Registers (4-thread / 8-thread)	256 / 512 integer and floating point

Table 2: 4-threaded workloads chosen from the SPEC CPU2006 suite

Mix	Benchmarks	ILP Class Count		
		L	M	H
1	namd, calculix, astar, gcc	0	0	4
2	specrand, calculix, astar, leslie3d	0	1	3
3	specrand, soplex, dealII, cactus	0	2	2
4	gobmk, gcc, milc, perlbench	1	1	2
5	astar, leslie3d, povray, lbm	1	2	1
6	soplex, milc, libquantum, xalancbmk	1	2	1
7	gcc, cactus, bzip2, lbm	2	1	1
8	cactus, gromacs, xalancbmk, lbm	2	2	0
9	dealII, sjeng, perlbench, xalancbmkc	3	1	0
10	xalancbmk, bzip2, lbm, perlbench	4	0	0

stalling otherwise efficient threads. This section examines the write buffer when running simulations with 4-threaded workloads and using a write buffer with 16 64-byte entries and M-Sim's default cache configuration and latencies as described in Section 3.

Figure 2 shows the average write buffer usage of the 10 workloads listed in Table 2. As the data show, workloads 1 and 3 use the least number of write buffer entries, with average usage of about 1 and 4 entries, respectively. Workload 2 uses about 10 entries on average, and the rest of the workloads all use between 13 and 16 entries on average. Two notable workloads, 7 and 10, both use nearly 16 on average, suggesting that these workloads contain write-intensive or cache-miss-intensive threads.

Table 3: 8-threaded workloads chosen from the SPEC CPU2006 suite

Mix	Benchmarks	ILP Class Count		
		L	M	H
1	namd, calculix, astar, gcc, specrand, soplex, cactus, povray	0	2	6
2	specrand, calculix, astar, gcc, gobmk, leslie3d, milc, dealII	0	3	5
3	namd, soplex, astar, specrand, dealII, gromacs, cactus, povray	0	4	4
4	specrand, gobmk, namd, libquantum, leslie3d, cacuts, gromacs, milc	0	5	3
5	astar, soplex, dealII, gromacs, leslie3d, cactus, povray, milc	0	6	2
6	gcc, astar, cactus, libquantum, povray, dealII, sjeng, perlbench	2	4	2
7	gromacs, leslie3d, cactus, povray, milc, libquantum, lbm, bzip2	2	6	0
8	dealII, cactus, leslie3d, povray, libquantum, xalancbmk, bzip2, sjeng	3	5	0
9	milc, gromacs, povray, cactus, perlbench, lbm, bzip2, xalancbmk	4	4	0
10	dealII, cactus, libquantum, sjeng, perlbench, lbm, bzip2, xalancbmk	5	3	0

A notable observation is that the 3 workloads which use the least number of write buffer entries are also the workloads with the highest ILP-rated threads (see Table 2).

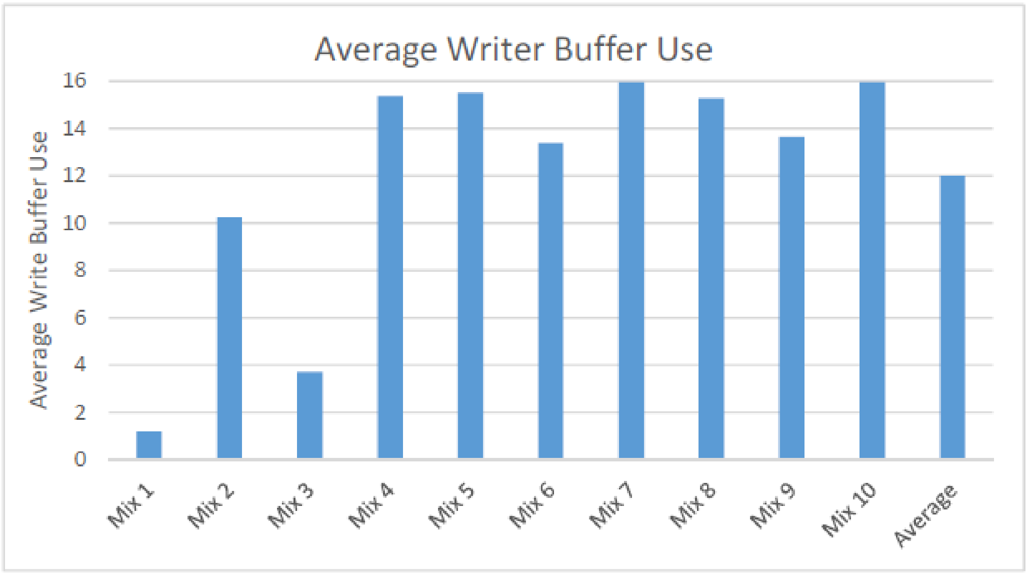

Figure 2: Average use of a 16-entry write buffer by 10 4-threaded workloads in a Simulation environment

To examine what causes high write buffer usage during runtime, the next part of this analysis looks at the contents of the write buffer when it is full. At the completion of each commit stage of the pipeline, if the write buffer is full, the contents are examined and grouped by cache-latency class (level-1 hit, level-2 hit, or cache miss). If, in any clock cycle, the write buffer is not full, its contents are not considered in this analysis since we aim only to see what causes high usage of the write buffer. Figure 3 shows the contents of a full write buffer, partitioned by cache latency, for each workload. Workload 1, the lowest user of the write buffer, never used all 16 entries and is excluded from Figure 3. Workload 2 had an average of about 15 cache misses in the write buffer when it was full, while the other 8 workloads had an average of nearly 16 cache misses in the write buffer when it was full, indicating that a full write buffer is typically caused by cache misses.

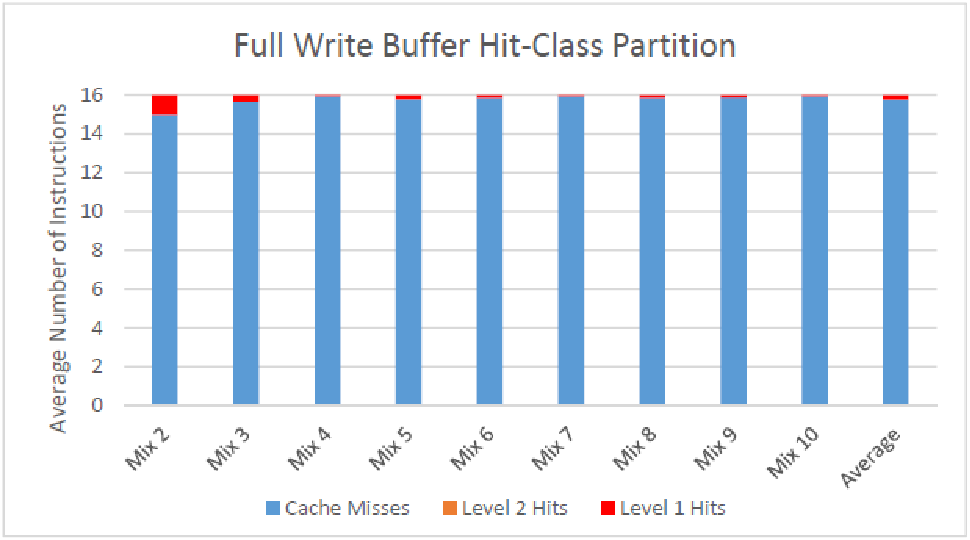

Figure 3: Occupancy of a full, 16-entry write buffer sorted by each instruction's cache-access latency in a 4-threaded simulation environment

To see how this impacts performance, we next look at the implications of a full write buffer. To determine if a full write buffer, likely caused by cache misses (see Figure 3), negatively affects system performance the following criteria will be sought.

1. The write buffer is full
2. Bandwidth to the write buffer is available
3. A store instruction is ready to be committed

These criteria describe a scenario in which a thread has a store ready to be committed, yet the thread cannot commit solely due to a lack of resource availability in the write buffer. When this condition arises, the store instruction mentioned in Criteria 3 will be referred to as a "blocked store". Since the write buffer is likely full because of cache misses (as shown in Figure 3), blocked stores which are cache misses will offer little implication of performance impact (as cache misses waiting for other cache misses are an uninteresting case). However, blocked cache hits would suggest that cache misses in the write buffer may be preventing non-offending threads from continuing (cache hits waiting for cache misses).

The next part of this analysis examines how often there is a blocked store instruction in the pipeline. Figure 4 shows two data plots. The leftmost plot shows the frequency at which Criteria

1 and 2 hold, shown as a percentage of clock cycles. That is, the left plot shows how often the write buffer fills up before bandwidth is depleted between the LSQ and the write buffer in any clock cycle. On the right, the frequency at which Criteria 1, 2, and 3 hold is shown as a percentage of clock cycles. That is, the right plot shows how often a blocked store occurs.

Figure 4 shows that workloads 1, 2 and 3 rarely or never exhibit blocked stores. Most other threads have a moderate amount of blocked stores, with the exception of workloads 7 and 10. These two workloads, which also has the highest rate of write buffer use, create a blocked store instruction in about 90% of all clock cycles. As a workload average, a blocked store instruction occurs during about 40% of clock cycles.

With the frequency of blocked stores known, their cache-latency class will be examined to determine what types of store instructions are being blocked. Figure 5 shows the partition of the blocked stores quantified in Figure 4. As the data show, the majority of blocked instructions are cache hits. Two notable exceptions are, again, workloads 7 and 10, whose blocked store latency distribution is less biased than the rest. These two workloads also have the highest ratio of write buffer usage and the highest frequency of blocked stores. These two workloads' high blocked-instruction rate and less biased distribution could both be attributed to these workloads having many more cache misses than the rest. With more cache misses, the write buffer is full more often, there are more blocked stores, and more of the blocked stores are cache misses, which is consistent with the data.

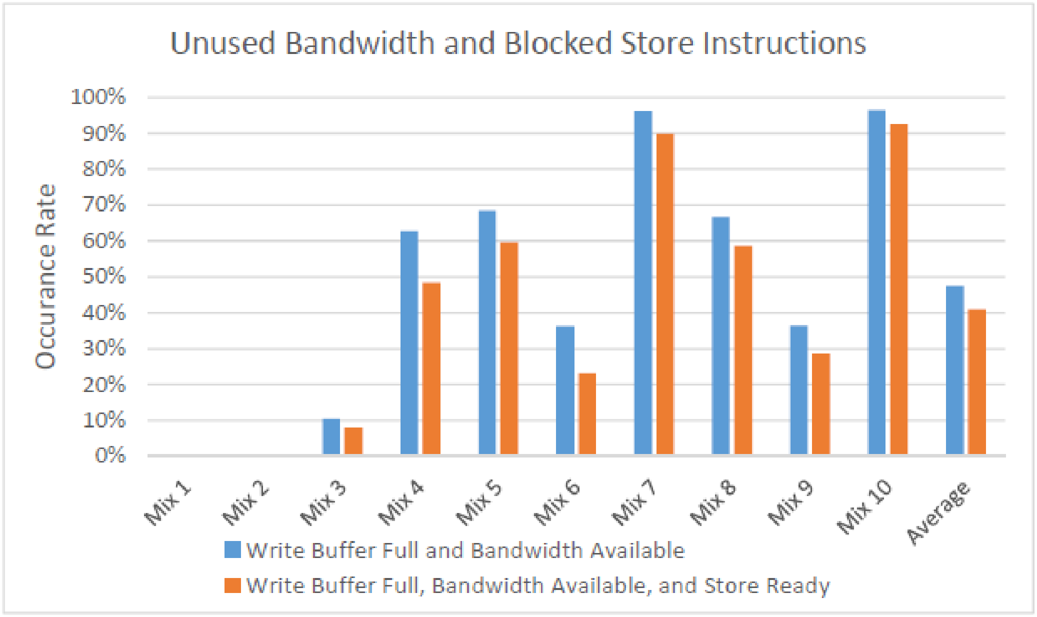

Figure 4: Frequency of store instructions which are blocked by a full write buffer in a 4-threaded simulation environment

We conclude this section with the observation that, with the simulation environment described in Section 3 and with 4-threaded workloads described in Table 2, during 40% of clock cycles there is a completed store instruction which is blocked from being committed solely by a full write buffer. In about 32% of clock cycles there is a blocked DL1 cache hit and in about 8% of clock cycles there is a blocked cache miss.

5. Proposed Write Buffer Allocation Algorithm

In Section 4 it was shown that short-latency cache hits are habitually blocked from being committed to the write buffer due to long-latency cache misses filling the write buffer. In some workloads, in nearly 60% of clock cycles a cache hit gets blocked, despite available bandwidth to the write buffer. This algorithm aims to allow these cache hits to have a reserved place in the write buffer to prevent blocking and improve system performance.

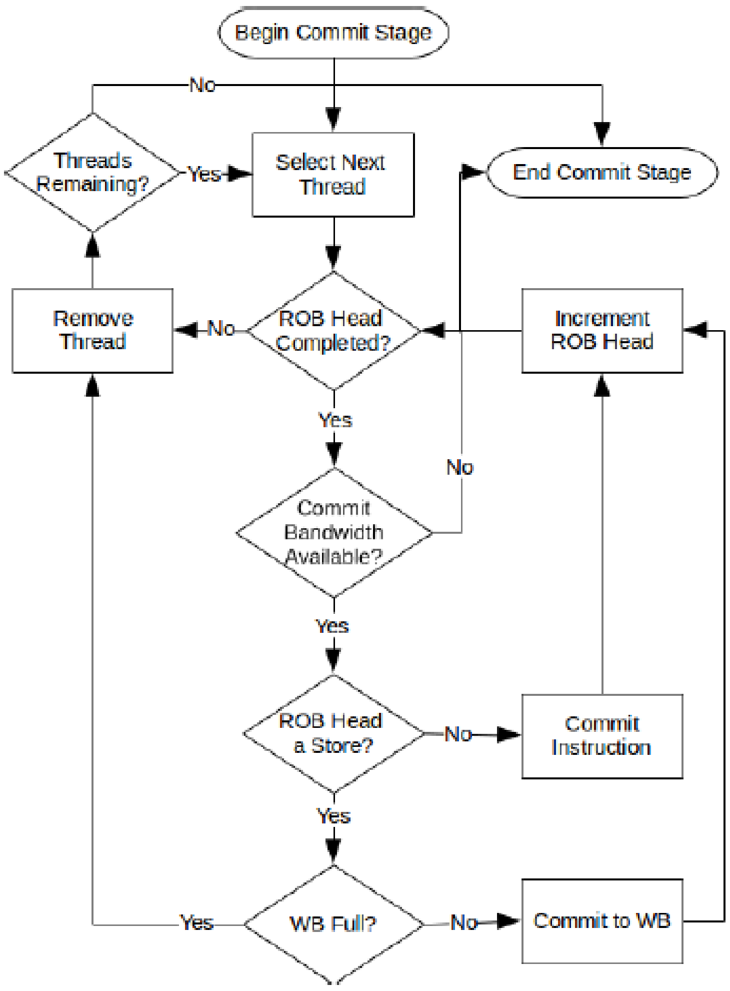

Figure 6: Typical SMT round robin commit algorithm

Figure 6 describes the procedure normally used by an SMT system when committing store instructions with a round-robin dispatch. During the commit portion of the pipeline, the system starts committing instructions from a current context, chosen by the formula c mod *N*, where *N* is the number of active threads and c is the current clock cycle. Instructions are committed starting from the head of the ROB of the current context. If the current context does not have an instruction ready to commit at the head of its ROB, the context is removed from consideration for the rest of the clock cycle and the next thread is checked in the same manner. Otherwise, the head of the ROB is committed and the head is decremented by one index. If the committed instruction was a store, its value will be written to the write buffer, given that the write buffer has available bandwidth and at least one unused entry.

It is the goal of the proposed algorithm to predict and prevent future blocked cache hits. To accomplish this, the events that cause blocked stores must be prevented, i.e., an excessive number

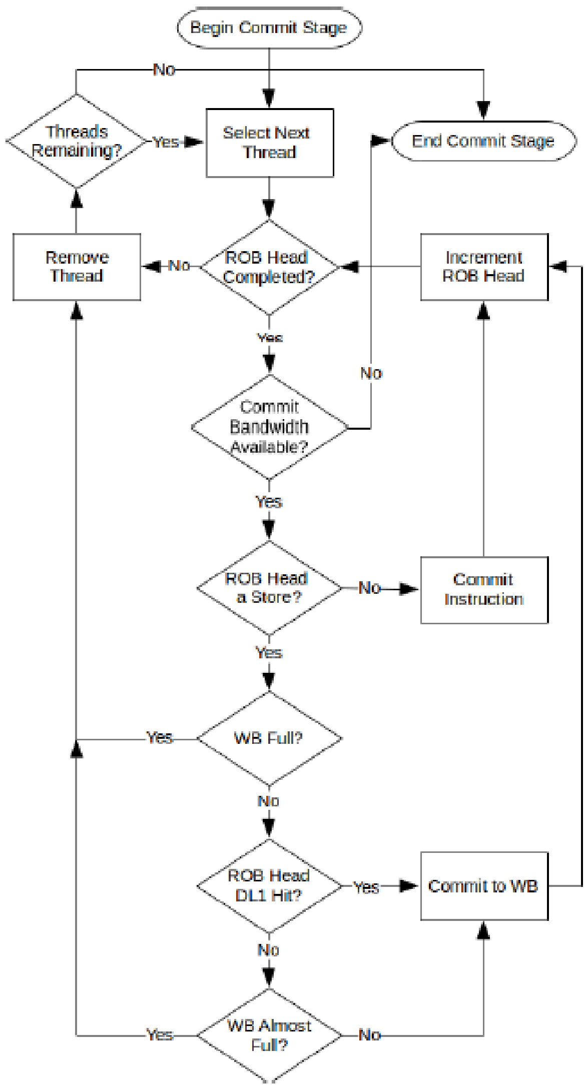

Figure 7: Modified SMT round robin commit algorithm

of cache misses should not be allowed to occupy the write buffer. The logic to control this will be executed within the commit algorithm to selectively choose when it is acceptable to send a store to the write buffer. Specifically, it shall not be acceptable to commit a store when the following two criteria hold.

1. The next store is a cache miss
2. The write buffer is near capacity

When this condition arises under the normal algorithm, the cache miss would be written to the nearly-full write buffer. Allowing these cache misses to be committed may subsequently contribute to a higher number of blocked cache hits, as shown in Section 4. Therefore, not letting this cache miss get committed may lower the frequency of blocked cache hits. Figure 7 shows the modifications to the original commit algorithm needed to implement the logic to prevent blocked cache hits.

This method effectively partitions the write buffer into two sets: general purpose entries and cache-hit-only entries. The former will be referred to as unreserved entries and the latter reserved entries. In other words, there is a subset of the write buffer to which any store may be written to (unreserved) and a subset to which only cache hits may be written to (reserved). When a store is written to the write buffer, the unreserved entries are used first, regardless of the cache latency. When only reserved entries remain, no more long-latency writes may be sent to the write buffer and any threads whose ROB head is a cache miss will be suspended from committing until unreserved write buffer entries are available.

6. Simulation Results

In this section the results of implementing the proposed algorithm in the simulation environment are examined. 4-thread and 8-thread workloads will be used with both 16-entry and 32-entry write buffers. All other simulation environment parameters will be held as described in Section 3. To gauge the performance of the algorithm, workload throughput will be measured in IPC. The IPC will be compared to that of the baseline as measured in the simulation environment with no modifications to the commit algorithm. In these results, there will be two types of IPC measured: arithmetic sum and harmonic mean. The arithmetic sum will gauge the overall throughput and the harmonic mean will gauge the fairness of thread execution. We define the arithmetic sum as

$$\sum_{i=1}^{N} IPC_i \tag{1}$$

and define the harmonic mean as

$$\frac{N}{\sum_{i=1}^{N} \frac{1}{IPC_i}} \tag{2}$$

where N is the number of concurrent threads and $I\ PC_i$ is the throughput of thread i. To evaluate the range of the algorithm's effect, the number of unreserved write buffer entries will be swept through avast range fine enough to find the value for optimal system performance improvement. At each value of unreserved entries, the change in IPC metrics will be computed as compared to the baseline. Each simulation will run up to 1 million instructions, meaning when the fastest thread completes 1 million instructions the simulation will end. Subsections 6.1 through 6.4 show the results of evaluating the algorithm with 16- and 32-entry write buffers and with 4- and 8-threaded workloads.

6.1. 16-Entry, 4-ThreadWorkloads

We begin the performance analysis with a write buffer size of 16 and using 10 4-thread workloads. Figure 8 shows that the average improvement of the 10 workloads peaks at an unreserved entry value of 15 with about a 21% gain in arithmetic IPC and a 11% gain in harmonic mean.

Figure 9 shows the arithmetic IPC change by workload.We see that the 21% increase in average arithmetic IPC was contributed to primarily by four workloads: 5, 7, 8, and 10. Referring back to

Figure 5, we see that these four workloads also had the highest ratio of blocked DL1 hits, all with rates of over 50% of clock cycles. The rest of the workloads, despite some having significant blocked Dl1 hit percentages, showed little to no response to the algorithm.

Figure 10 shows the harmonic IPC change per workload. We see that the 10% gain in average harmonic IPC was also contributed to primarily by four workloads: 5, 7, 8, and 10. These are the same workloads that heavily contributed to the average arithmetic IPC improvement. Similarly, the other 6 workloads showed little to no response in harmonic IPC change at the optimal unreserved entry count.

It is interesting to note that with as many as 12 reserved entries we see IPC improvement, but performance peaks at just 1 reserved entry. This is likely due to DL1 cache hits having a 1 clock cycle

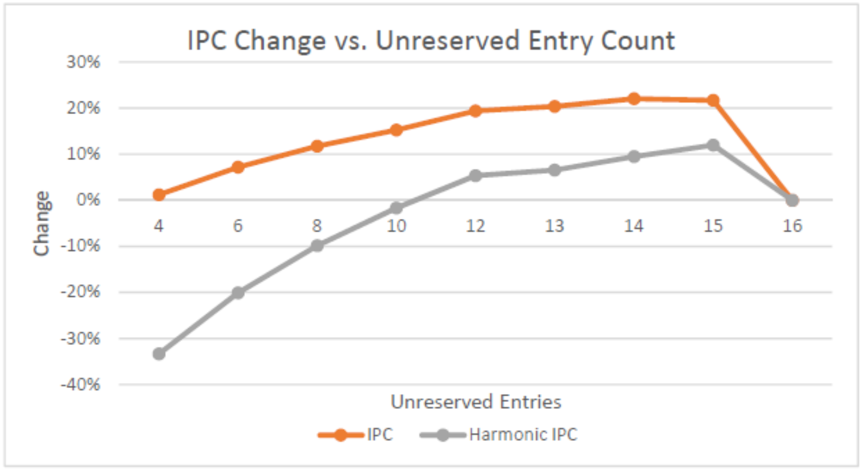

Figure 8: Average performance change vs number of unreserved entries for a 16-entry write buffer and 4-threaded workloads

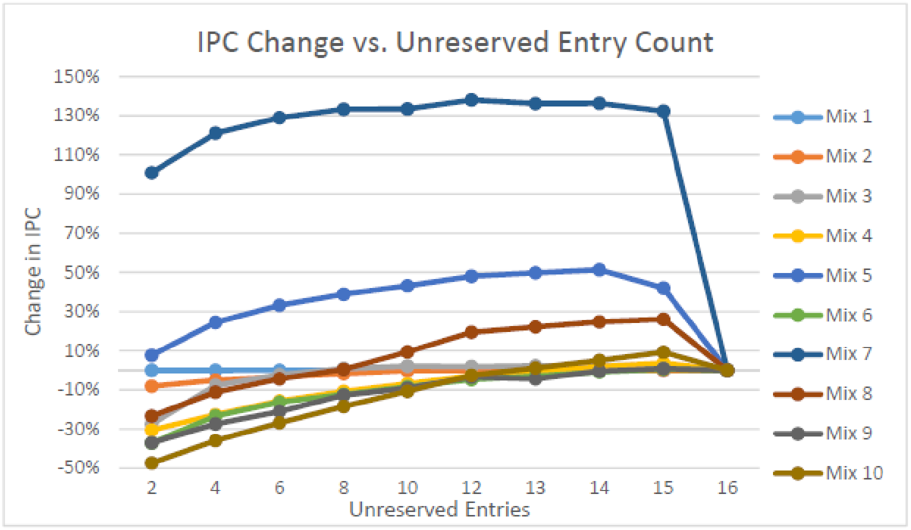

Figure 9: Per-mix arithmetic IPC change vs number of unreserved entries for a 16-entry write buffer and 4-threaded workloads

latency in our environment and a much higher prevalence of DL1 cache hits than DL2 cache hits. With such a short latency, cache hits can make efficient use of a small number of reserved entries.

6.2. 16-entry, 8-ThreadWorkloads

The next part of the simulation results looks at the same parameter setup (including a 16-entry write buffer) but with 8-thread workloads instead of 4-thread. Figure 11 shows the results of applying the algorithm in the simulation environment with 8-thread workloads. A maximum improvement in average arithmetic IPC of about 37% was achieved by having 14 of the 16 entries serve as unreserved entries and 2 reserved. An optimal harmonic IPC change came at a slightly different value, achieving

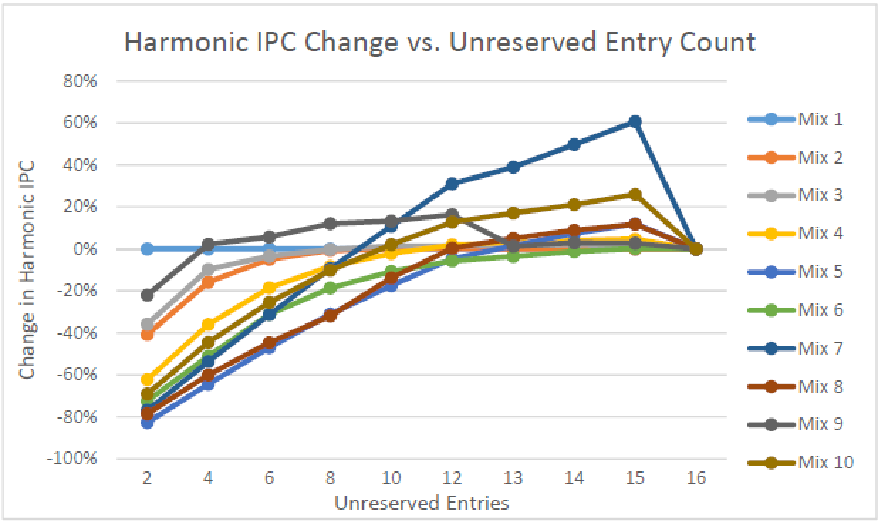

Figure 10: Per-mix harmonic IPC change vs number of unreserved entries for a 16-entry write buffer and 4-threaded workloads

about a 25% gain at 15 unreserved entries and just 1 reserved.

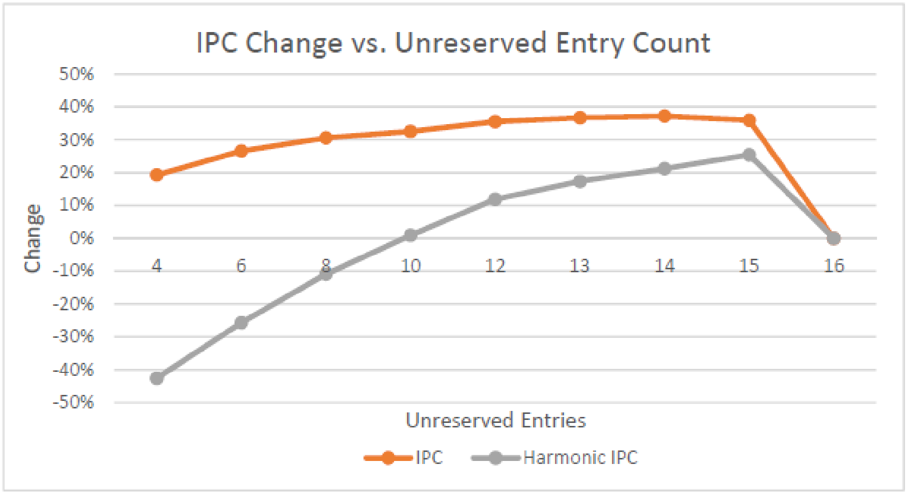

Figure 11: Average performance change vs number of unreserved entries for a 16-entry write buffer and 8-threaded workloads

Figure 12 shows the arithmetic IPC changes by workload after applying the algorithm. As in Section 6.1, we see that some threads had significant responses while others showed little to no response. In the 4-threaded case, 4 workloads were the main contributors to arithmetic IPC gain. In this case, 6 of the 10 workloads showed significant improvement. Workloads 4, 5, 7, 8, 9, and 10 all had arithmetic IPC gains of between 15% and 95% at their optimal values. With more threads per workload and a similarly-sized write buffer, it was expected that more workloads would respond to the algorithm in this more competitive environment.

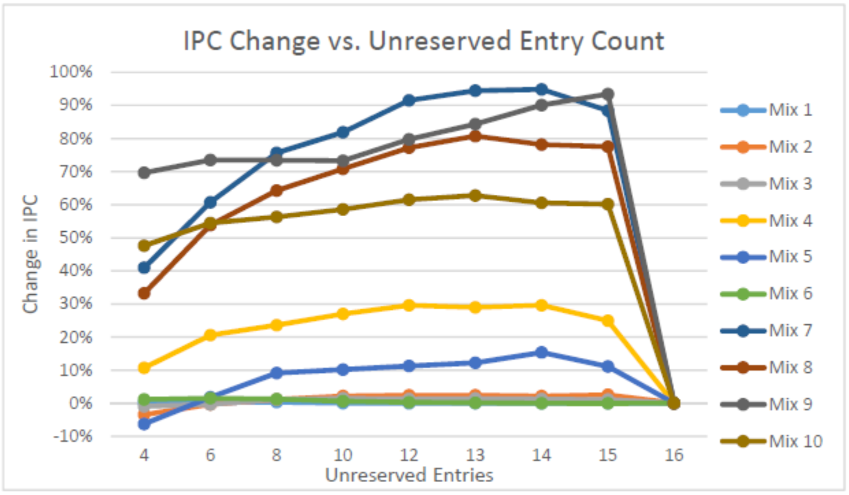

Figure 12: Per-mix arithmetic IPC change vs number of unreserved entries for a 16-entry write buffer and 8-threaded workloads

Figure 13 shows the harmonic IPC change per workload after applying the algorithm. We see that the 6 primary contributors to the arithmetic IPC gain were the same 6 contributors of the harmonic IPC gain, workloads 4, 5, 7, 8, 9, and 10, all with increases between 10% and 120% at their optimal values. Two workloads, 8 and 9, dominated the results with gains of about 60% and 120%, respectively.

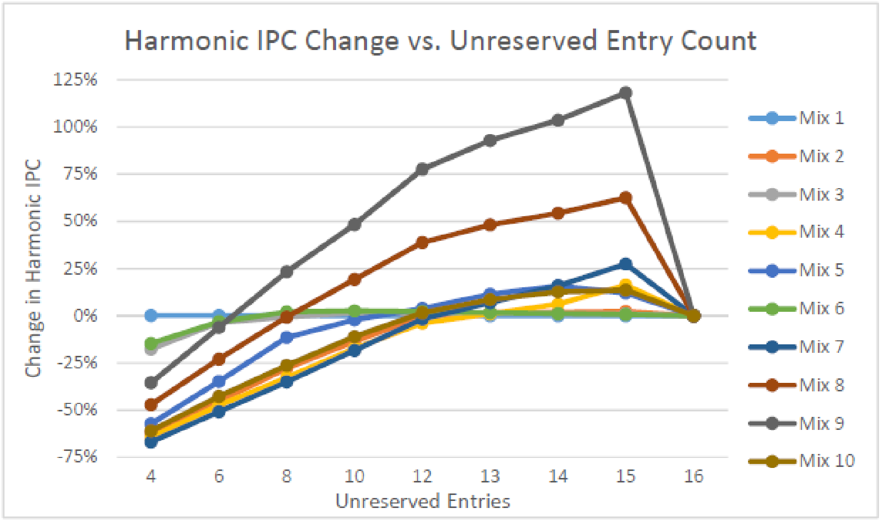

Figure 13: Per-mix harmonic IPC change vs number of unreserved entries for a 16-entry write buffer and 8-threaded workloads

6.3. 32-entry, 4-ThreadedWorkloads

The next two sections show the results of the algorithm applied with the same workloads but with a 32-entry write buffer, twice the size of the previous setup. As in the preceding section, 4-threaded and 8-threaded workloads are evaluated. Figure 14 shows the results of the overall average of IPC change with 4-threaded workloads. Expectedly, the gains are considerably more moderate with a write buffer which is double in size. Still, we see an increase in arithmetic IPC of about 8% and a gain in harmonic IPC by about 6% when just one write buffer entry is reserved for cache hits. The change in performance becomes positive at 20 reserved entries and peaks at 31. Again, the small number of reserved entries needed to maximize performance gains is likely due to the relativity of DL1 hits compared to DL2 hits and the speed at which DL1 hits are committed to cache (1 clock cycle).

Figure 15 shows the arithmetic IPC change by workload for each stage of the parameter sweep. Similarly to the 16-entry case, workloads 5 and 7, the workloads with the highest rate of blocked cache hits, dominate the performance gains. With the 16-entry case, the results were primarily split between 4 workloads. With a larger write buffer and more moderate overall gains, the number of workloads which saw significant benefits was halved. Most workloads saw little, none, or negative performance change throughout the parameter sweep.

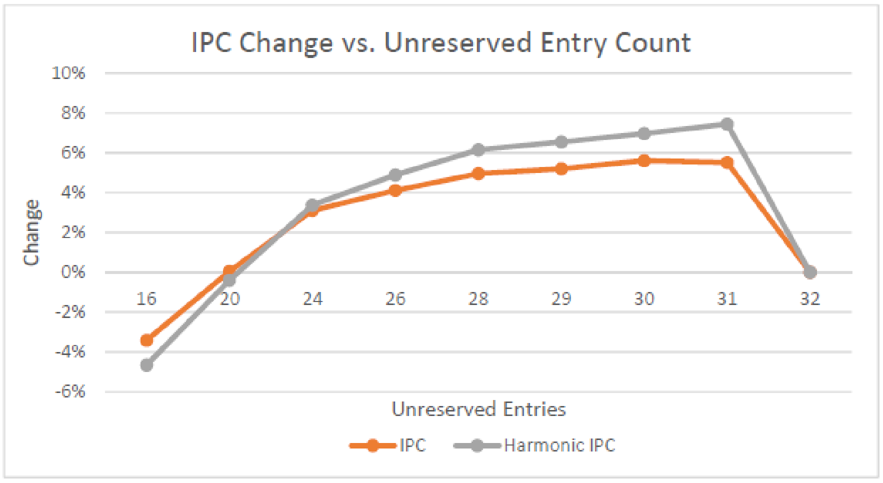

Figure 14: Average performance change vs number of unreserved entries for a 32-entry write buffer and 4-threaded workloads

Figure 16 shows the harmonic IPC change for each workload during the parameter sweep. In this case, workloads 5, 7, 8, and 10 received the most benefit from applying the algorithm. These 4 workloads were also the biggest benefiters of the algorithm when using a 16-entry write buffer (see Figure 10) and had the highest rate of blocked cache hits in the analysis (see Figure 5).

6.4. 32-entry, 8-ThreadedWorkloads

Finally, Figure 17 shows the average IPC change in the 8-threaded workloads with a 32-entry write buffer with the algorithm applied. Similar to the 4-threaded, 32-entry results, there were more modest gains in both arithmetic and harmonic IPC of about 8%. Positive change came with as little as 16 reserved write buffer entries and peaked at 31 unreserved write buffer entries.

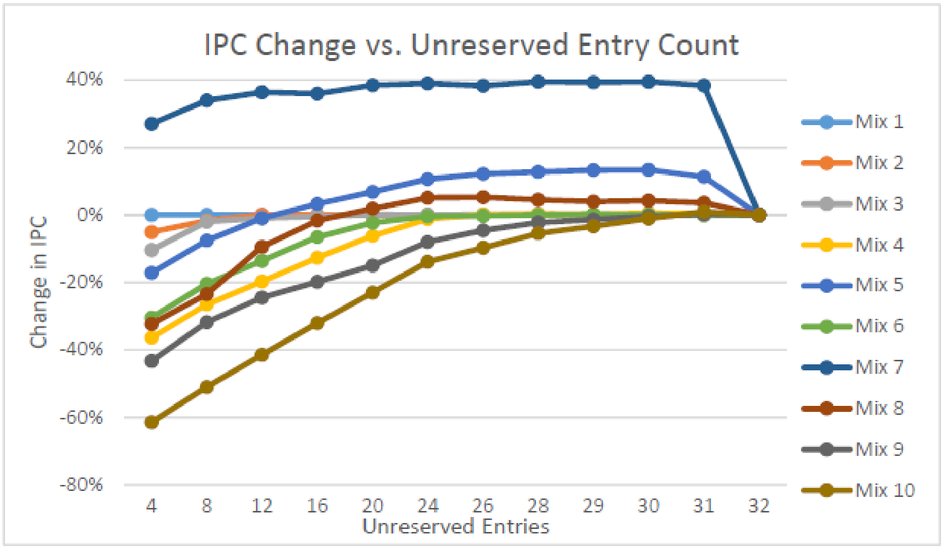

Figure 15: Per-mix arithmetic IPC change vs number of unreserved entries for a 32-entry write buffer and 4-threaded workloads

Figure 18 shows the arithmetic IPC change per 4-threaded workload when using the algorithm with a 32-entry write buffer. Mixes 7, 8, 9, and 10 dominated the improvements with gains ranging from 12% to 26% at the optimum overall unreserved write buffer entry count. These 4 workloads also saw the most improvement with a 16-entry write buffer. The other 6 workloads saw almost no change, although 2 of these unresponsive mixes had individual gains of 10% and 25% with a 16-entry write buffer.

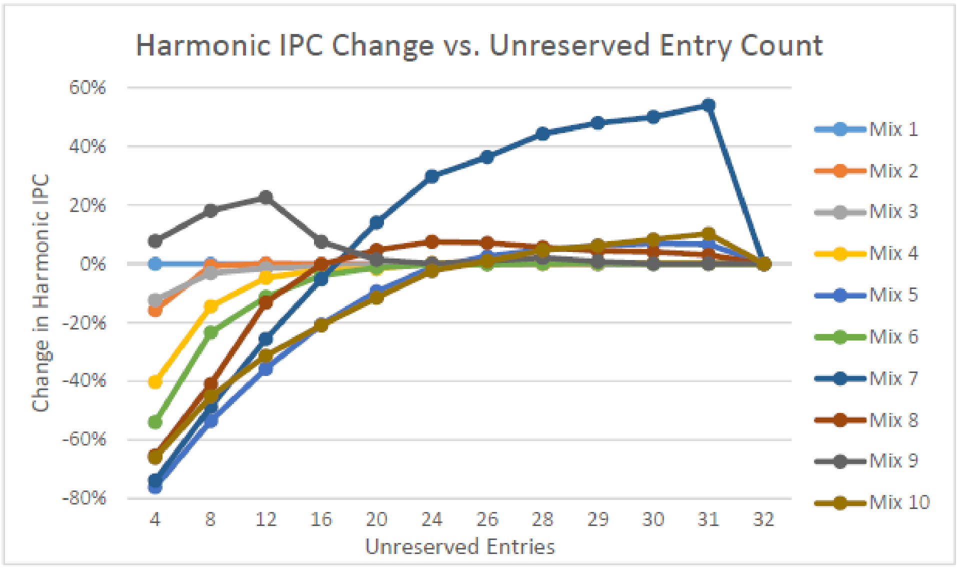

Figure 16: Per-mix harmonic IPC change vs number of unreserved entries for a 32-entry write buffer and 4-threaded workloads

Figure 19 shows the harmonic IPC change of each 8-threaded workload with a 32-entry write buffer. Mixes 7, 8, and 9 showed the greatest increase, all with improvements between 20% and 30%. These workloads were 3 of the 4 highest gainers of arithmetic IPC as well. The fourth workload that showed significant arithmetic IPC improvement showed a modest improvement of about 8%, while the other 6 workloads showed virtually no change.

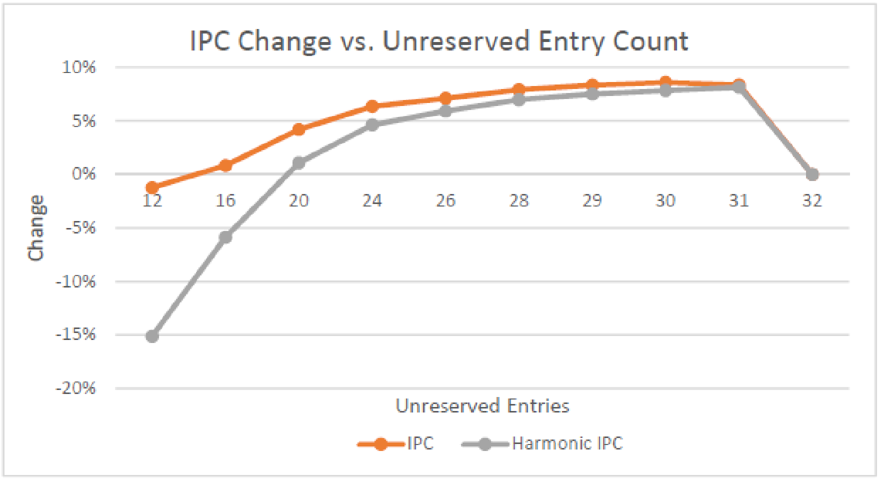

Figure 17: Average performance change vs number of unreserved entries for a 32-entry write buffer and 4-threaded workloads

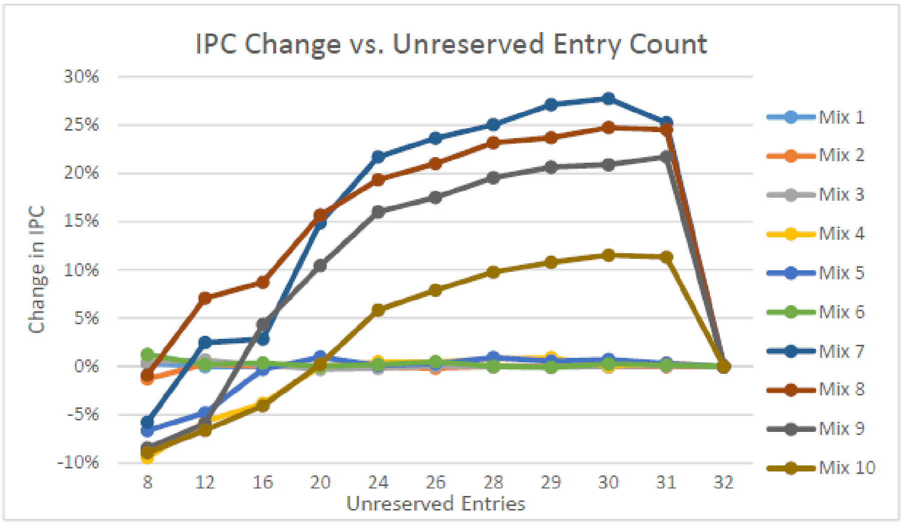

Figure 18: Per-mix arithmetic IPC change vs number of unreserved entries for a 32-entry write buffer and 8-threaded workloads

6.5. Results Summary

Tables 4 and 5 summarize the maximum performance gains and the corresponding number of reserved writes buffer entries for the algorithm for every combination of 4- and 8-threaded workloads and 16- and 32-entry write buffers in the simulation environment.

7. CONCLUSION

This paper showed that store instructions with low cache latency are routinely blocked from being committed due to a full write buffer in an SMT system. It was shown that the write buffer was typically full due to an excessive number of long-latency store instructions being committed unchecked.

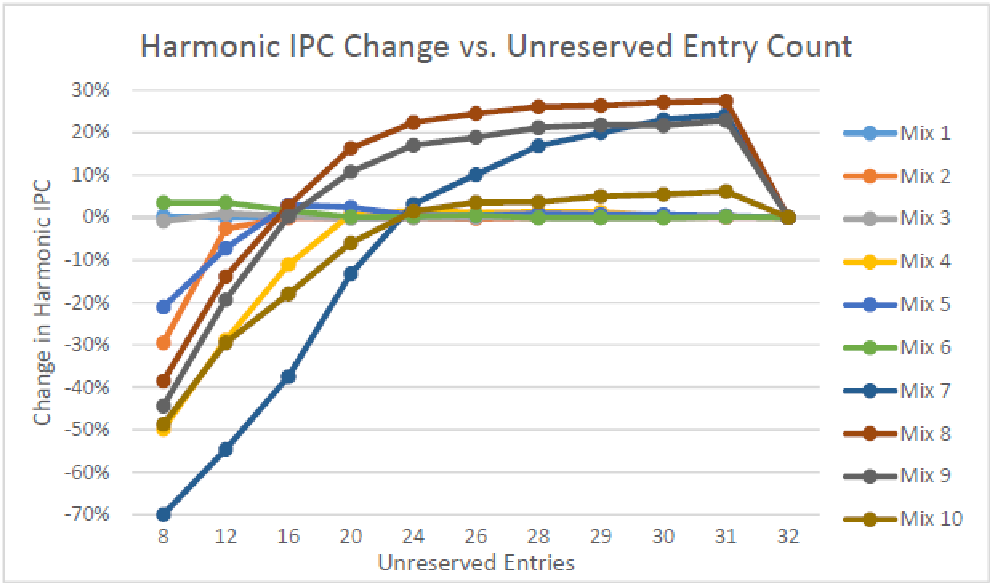

Figure 19: Per-mix harmonic IPC change vs number of unreserved entries for a 32-entry write buffer and 8-threaded workloads

Table 4: Summary of maximum arithmetic and harmonic IPC change for 4- and 8-thread workloads with a 16-entry write buffer

	Arithmetic IPC	Harmonic IPC	Reserved Entries
4-Threaded Workloads	21.7%	11.9%	1
8-Threaded Workloads	35.9%	25.4%	1

Table 5: Summary of maximum arithmetic and harmonic IPC change for 4- and 8-thread workloads with a 32-entry write buffer

	Arithmetic IPC Gain	Harmonic IPC Gain	Reserved Entries
4-Threaded Workloads	5.5%	7.4%	1
8-Threaded Workloads	9.3%	9.0%	1

This paper proposed an algorithm to reduce the number of long-latency store instructions in the write buffer by skipping threads with long-latency store instructions when the write buffer was nearly full. It was shown that this technique increases system performance by keeping some write buffer entries reserved for threads which have low-latency store instructions ready to be committed. The proposed algorithm improved system performance on average by 5.5% to 35.9%, depending on workload size and system configuration, in a simulation environment.

REFERENCES

[1] S.J.Eggers D. M. Tullsen and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In ACM SIGARCH Computer Architecture News, volume 23, pages 392–403. ACM, 1995.

[2] P.B.Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In ACM SIGPLAN Notices, volume 21, pages 11–16. ACM, 1986.

[3] H.M.Leby J. L. Lo R. L. Stamm S. J. Eggers, J. S. Emer and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. Micro, IEEE, 17(5):12–19, 1997.

[4] H.M.Levy R. L. Stamm D. M. Tullsen J. L. Lo, J. S. Emer and S. J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. ACM Transactions on Computer Systems (TOCS), 15(3):322–354, 1997.

[5] Ulrich Sigmund and Theo Ungerer. Identifying bottlenecks in a multithreaded superscalar microprocessor. In Euro-Par'96 Parallel Processing, pages 797–800. Springer, 1996.

[6] Steven E Raasch and Steven K Reinhardt. The impact of resource partitioning on smt processors. In Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on, pages 15–25. IEEE, 2003.

[7] N.P.Jouppi. Cache write policies and performance, volume 21. ACM, 1993.

[8] D.M.Tullsen and J.A.Brown. Handling long-latency loads in a simultaneous multithreading processor. In Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture, pages 318–327. IEEE Computer Society, 2001.

[9] Kun Luo, Jayanth Gummaraju, and Manoj Franklin. Balancing thoughput and fairness in smt processors. In ISPASS, volume 1, pages 164–171, 2001.

[10] Kun Luo, Manoj Franklin, Shubhendu S Mukherjee, and Andre Sezne. Boosting smt performance by speculation control. In Parallel and Distributed Processing Symposium., Proceedings 15th International, pages 9–pp. IEEE, 2001.

[11] Francisco J Cazorla, Alex Ramirez, Mateo Valero, and Enrique Fernandez. Dynamically controlled resource allocation in smt processors. In Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on, pages 171–182. IEEE, 2004.

[12] Allan Snavely, Dean M Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In ACM SIGMETRICS Performance Evaluation Review, volume 30, pages 66–76. ACM, 2002.

[13] Tipp Moseley, Joshua L Kihm, Daniel Connors, Dirk Grunwald, et al. Methods for modeling resource contention on simultaneous multithreading processors. In Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on, pages 373–380. IEEE, 2005.

[14] Sébastien Hily and André Seznec. Contention on 2nd level cache may limit the effectiveness of simultaneous multithreading. 1997.

AUTHORS

Shane Carroll is a Ph.D. student of Electrical Engineering at The University of Texas at San Antonio (UTSA) and works as a software engineer. He received his M.S. degree in Computer Engineering from UTSA and his B.S. degree in Computer Engineering Technology and Mathematics from Central Connecticut State University. His research interest is computer architecture.

Dr. Wei-Ming Lin is currently a professor of Electrical and Computer Engineering at the University of Texas at San Antonio (UTSA). He received his Ph.D. degree in Electrical Engineering from the University of Southern California, Los Angeles, in 1991. Dr. Lin's research interests are in the areas of distributed and parallel computing, computer architecture, computer networks, autonomous control and internet security. He has published over 100 technical papers in various conference proceedings and journals. Dr. Lin's past and on-going research has been supported by NSF, DOD, ONR, AFOSR, etc.