IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 18, Issue 6, Ver. IV (Nov.-Dec. 2016), PP 74-84
www.iosrjournals.org

An Intelligent Resource Sharing Protocol on Write Buffers in Simultaneous Multi-Threading Processors

Yilin Zhang[1] and Wei-Ming Lin[2]
[1]*Advanced Micro Devices Inc. Austin, TX 78725, US*
[2]*Department of Electrical and Computer Engineering University of Texas at San Antonio San Antonio, TX USA*

Abstract: *Simultaneous Multi-Threading (SMT) has been widely studied to lend modern-day CPUs a mechanism to improve resource utilization so as to lead to a higher instruction throughput by allowing concurrent execution of multiple independent threads with sharing of key datapath components. The key to a high-performance SMT is to optimize the distribution of shared resources among temporally competing threads. Allowing any of the threads to overwhelm these resources not only leads to unfair thread processing but also may severely degrade overall system throughput. Write buffer is one of the most critical shared resources in SMT systems due to its size constraint and potentially long occupancy latency from its data. In this paper, we show that, by limiting the number of write buffer entries each thread is allowed to occupy in the commit stage, the overall system throughput is enhanced by a substantial margin. An improvement in IPC of up to 26% and 95% is observed when the proposed technique is applied to a 4-threaded and an 8-threaded SMT system, respectively.*
Keywords: *Simultaneous Multi-Threading, Superscalar, Write Buffer*

I. Introduction

Noting the resource utilization deficiencies in the traditional superscalar processors, Simultaneous Multi-Threading (SMT) offers an improved mechanism by allowing instructions from different threads to be issued in the same clock cycle in order to exploit the full potential of the shared resources. In modern out-of-order SMT processors, this execution model implies the sharing of several resources among threads with a goal to achieve a comparable performance to that from using multiple copies of superscalar processors while saving a significant amount of resources. In order to achieve this, the most significant potential drawback in the SMT execution model in inter-thread blocking due to resource sharing has to be carefully addressed. A typical pipeline organization in an SMT system is shown in Figure 1. Instructions from a thread are *fetched* from memory (and cache) and put into their respective private Instruction Fetch Queue (IFQ). After the stages of *decode* and register-*rename* they are allocated into their respective Re-Order Buffer (ROB) and through the *dispatch* stage into the shared Issue Queue (IQ). Load/Store instructions have their operations dispatched into individual Load Store Queues (LSQ) with address calculation operations also sent into IQ. When the issuing conditions (all operands ready and the required functional unit available) are met, the operations are then *issued* to corresponding functional units and have their results *writeback* to their target locations or forwarded to where the results are needed in IQ. Load/Store instructions, once their addresses are calculated, will initiate their memory operation. Finally all instructions are *committed* from ROB (synchronized with Load/Store instructions in LSQ) in order.

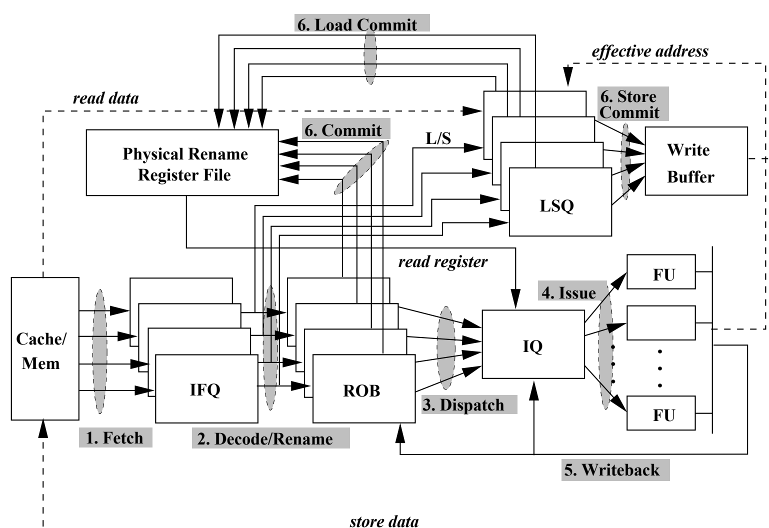

Figure 1. Pipeline Stages in a 4-Threaded SMT System

Essentially SMT improves the overall performance by exploiting Thread-Level Parallelism (TLP) among threads to overcome the limitation of Instruction-Level Parallelism (ILP) presented in a single thread [1][2]. Subsequently, due to the sharing of resources, the amount of hardware required in an SMT system is significantly less than employing multiple copies of superscalar processors while achieving a similar performance.There have been many research efforts aiming at improving the shared resources allocation in SMT systems. Some of them manage the resource usage among threads by modifying the fetch policy. For example, ICOUNT [5] assigns a higher fetching priority to a thread with fewer instructions in pre-issue stages; STALL and FLUSH [6] adopts a fetch policy to address issues from L2 cache misses; a dynamical fetch policy DCRA presented in [7] is a technique based on memory performance of each thread to exploit parallelism beyond stalled memory operations; PEEP [11] controls the fetch unit by exploiting the predictability of memory dependencies; SAFE-T in [12] and a speculation control technique in [13] both give higher fetching priorities to threads with higher branch prediction accuracy. Some other research efforts have also examined the allocation of the shared buffers in the pipeline for a more efficient resource utilization. For example, APRA dynamically assigns resources (IFQ, IQ and ROB) to threads according to changes of threads' behaviour [9] Hill-Climbing [8] is a learning-based algorithm that uses performance feedback to partition the shared hardware resources in the pipeline including IFQ, rename registers, ROB and IQ. Some other research results have enhanced the overall performance by improving the utilization of IQ [10,14,15,16].Note that the common resources in an SMT system shared by threads include various machine bandwidths (e.g., inter-stage bandwidth, read/write ports for register files and memory, etc.), inter-stage buffers (e.g., Issue Queue), functional units, and write buffer, etc. Our analysis in a later section shows that the write buffer proves to be the most critical shared resources in affecting overall performance. No research in the literature so far has investigated issues regarding the sharing of this critical buffer among threads in an SMT system. How to effectively assign the write buffer to competing threads to prevent long-term blocking due to lengthy and dominant occupancy from slower thread(s) is the main theme of this paper. A new technique is developed in this paper to effectively relieve the pressure of the write buffer in an SMT system so as to achieve better resource utilization. Write operations in a program tend to be bursty in nature and thus a sequence of writes from one program could easily overwhelm the whole write buffer. In order to prevent any single thread from disproportionately occupying the write buffer, a simple capping algorithm is proposed in this paper by limiting the number of buffer entries each thread is allowed to occupy at any given time. Our simulation results show that IPC (Instructions Per Clock-cycle) improvement can be as high as 26% for a 4-threaded workload and a whopping 95% for an 8-threaded workload.

II. Write Buffer

A write buffer (or referred to as “store buffer” in some articles) is designed to hide long write latency when CPU encounters cache misses [3,4,17,18]. A typical two-level cache organization with write-back policy at both levels is shown in Figure 2. Modern-day processors usually have a multi-level cache architecture and may adopt different write policies, write-through and write-back, at different levels, which in turn may require a different design for the write buffer.

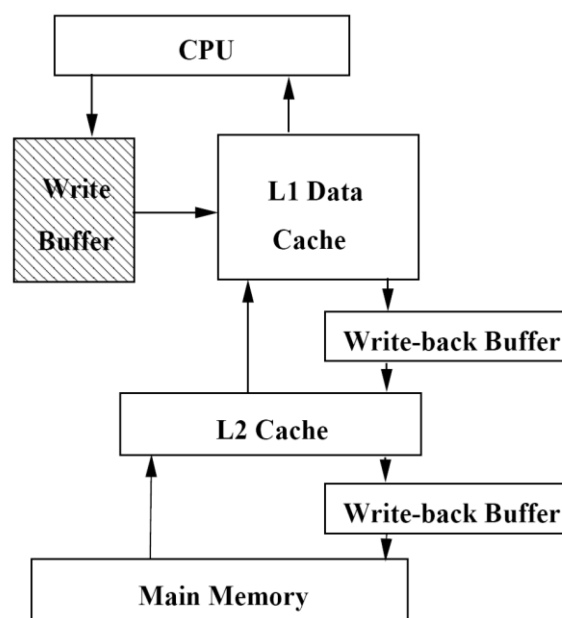

Figure. 2. A Typical Two-Level Cache Organization With A Write Buffer

Write-merging (also known as write-coalescing) is a technique aggregating writes to the same cache block (line) to reduce miss penalty as well as the buffer size requirement, which is prevalent in most modern processors [4]. In a write-coalescing buffer, a data read operation retrieves its target data from either the L1 data cache or the write buffer (if the target data is still residing in the buffer from a previous write yet to finish); a write operation will be merged into an existing write in the buffer if they belong to the same block (cache line), i.e. they will share the same entry in the buffer until the write operation finishes. When no merging exists, a write will not be allowed to commit if the buffer is full. Some researchers have focused their work on designing more efficient write buffer management for a generic superscalar CPU system including the study of effects from write buffer depths, retirement and load-hazard policies [17], and a new write option update policy in [18], etc.In a superscalar system, all instructions have to be committed in order to ensure precise exception and correct speculative processing. Thus, if a write instruction cannot commit due to a full write buffer, all subsequent instructions will be blocked as well. In an SMT system where multiple threads are processed concurrently, blocking of one thread due to a full write buffer may very well mean blocking of another thread if it also has a write operation waiting to commit. What makes the difference between superscalar and SMT systems is that in the superscalar one if such a blocking happens it is an inevitable one since the order of committing instructions in the only program cannot be changed. On the other hand, in an SMT system, the order of committing instructions from different threads does not have to be fixed and therefore the order to committing write instructions from different threads may affect the overall system flow dramatically.

Since the write buffer is an on-chip component, to retain a large write buffer can be both cost and clock-timing prohibitive. For example, the 3rd generation of Intel XScale microarchitecture has a write buffer with only 12 entries [19]. Without employing a larger buffer, exploitation of TLP among threads in SMT will be thus severely hampered -- contention among threads in this buffer may prevent “faster” threads (with faster writes) from committing their writes. To fully exploit both TLP and ILP, proper intelligence has to be incorporated into this resource sharing mechanism to ensure that threads share this component in an efficient and fair manner. The cache design shown in Figure 2 will be assumed in our discussion with both levels of cache using the write-back policy. Note that there is very minimal effect on the write buffer performance from adopting a different write policy. In this paper, our proposed algorithm is tested in such a system with the feature of write coalescing.

III. Simulation Environment

The simulation environment adopted by our research, including the simulator and the workloads used are described in this section.

3.1. Simulator

We use the M-sim [20], a multi-threaded microarchitectural simulation environment to estimate the performance of an SMT system. M-sim explicitly models the key pipeline structures such as the Reorder Buffer (ROB), the Issue Queue (IQ), the Load/Store Queue (LSQ), separate integer and floating-point register files, register renaming, and the associated rename table. M-sim supports both single threaded execution and the simultaneous execution of multiple threads. In SMT mode, the physical register file, IQ, functional units and write buffer are shared among threads. Table 1 gives the detailed configuration of the simulated processor.

Table 1. Configuration of Simulated Processor

Parameter	Configuration
Machine width	8-wide fetch/dispatch/issue/commit
L/S Queue Size	48-entry load/store queue
ROB & IQ size	128-entry ROB, 32-entry IQ
Functional Units & Latency (total/issue)	4 Int Add (1/1) 1 Int Mult (3/1)/Div (20/19) 2 Load/Store (1/1), 4 FP Add (2/1) 1 FP Mult (4/1)/Div (12/12) Sqrt(24/24)
Physical registers	Integer and floating point as specified in the paper
L1 I-cache	64KB, 2-way set associative 64-byte line
L1 D-cache	64KB, 4-way set associative 64-byte line write back, 1 cycle access latency
L2 Cache unified	512KB, 16-way set associative 64-byte line write back, 10 cycles access latency
BTB	512 entry, 4-way set associative
Branch Predictor	bimod: 2K entry
Pipeline Structure	5-stage front-end (fetch-dispatch)

	scheduling (for register file access: 2 stages, execution, write back, commit)
Memory	32-bit wide, 300 cycles access latency

3.2. Workload

Simulations on simultaneous multi-threading use the workload of mixed SPEC CPU2006 benckmark suite [17] with mixtures of various levels of ILP. ILP classification of each mix is obtained by initializing it in accordance with the procedure mentioned in Simpoints tool and simulated individually in a simplescalar environment. Three types of ILP classification are identified, low ILP (memory bound), medium ILP, high ILP (execution bound). Table 2 and Table 3 give the selected 4-threaded and 8-threaded workload for our simulation with various mixtures of ILP classification types respectively (each number in the table denotes the number of programs with the corresponding classification in the respective mix).

Table 2. Simulated SPEC CPU2006 4-Threaded Workload

Mix	**Benchmarks**	**Classification (ILP)**		
		Low	Med	High
Mix1	libquantum, dealII, gromacs, namd	0	0	4
Mix2	soplex, leslie3d, povray, milc	0	4	0
Mix3	hmmer, sjeng, gobmk, gcc	0	4	0
Mix4	lbm, cactusADM, xalancbmk, bzip2	4	0	0
Mix5	libquantum, dealII, gobmk, gcc	0	2	2
Mix6	gromacs, namd, soplex, leslie3d	0	2	2
Mix7	dealII, gromacs, lbm, cactusADM	2	0	2
Mix8	libquantum, namd, xalancbmk, bzip2	2	0	2
Mix9	povray, milc, cactusADM, xalancbmk	2	2	0
Mix10	hmmer, sjeng, lbm, bzip2	2	2	0

Table 3. Simulated SPEC CPU2006 8-Threaded Workload

Mix	**Benchmarks**	**Classification (ILP)**		
		Low	Med	High
Mix1	libquantum, dealII, gromacs, namd, soplex, leslie3d, povray, milc	0	4	4
Mix2	libquantum, dealII, gromacs, namd, lbm, cactusADM, xalancbmk, bzip2	4	0	4
Mix3	hmmer, sjeng, gobmk, gcc, lbm, cactusADM, xalancbmk, bzip2	4	4	0
Mix4	libquantum, dealII, gromacs, soplex, leslie3d, povray, lbm, cactusADM	2	3	3
Mix5	dealII, gromacs, namd, xalancbmk, hmmer, cactusADM, milc, bzip2	3	2	3
Mix6	gromacs, namd, sjeng, gobmk, gcc, lbm, cactusADM, xalancbmk	3	3	2

3.3. Metrics

For a multi-threaded workload, total combined IPC is a typical indicator used to measure the overall performance, which is defined as the sum of each thread's IPC:

$$Overall_IPC = \sum_i^n IPC_i \quad (1)$$

where n denotes the number of threads per mix in the system. However, in order to preclude starvation effect among threads, the so-called Harmonic IPC is also adopted, which reflects the degree of execution fairness among the threads, namely,

$$Harmonic_IPC = n / \sum_i^n IPC_i \quad (2)$$

In this paper, these two indicators are used to compare the proposed algorithm to the baseline (default) system. The following metric indicates the improvement percentage averaged over the selected mixes, which is applied to both $Overall_IPC$ and $Harmonic_IPC$, namely,

$$IMP_\% = \left(\sum_j^m \frac{IPC_j^{new} - IPC_j^{baseline}}{IPC_j^{baseline}} \times 100\%\right)/m \quad (3)$$

where m denotes the number of mixes of the workload in our simulation.

IV. Motivation

The technique proposed in this paper is based on the conjectures that a write buffer of limited size tends to be the bottleneck for the pipeline operation, especially in an SMT system. What contributes to this bottleneck is a combination of high buffer occupancy and imbalanced occupancy among the threads. This section is devoted to the discussion to support these conjectures. The simulation results in this section are based on the system configuration with the ten mixes of 4-threaded workload as described in Section 3.

4.1. Write Buffer Size Analysis
We first analyze how critical the size of write buffer is to the overall throughput. Figure 3 shows the average of overall IPC from using different sizes of write buffer. When the size of the buffer

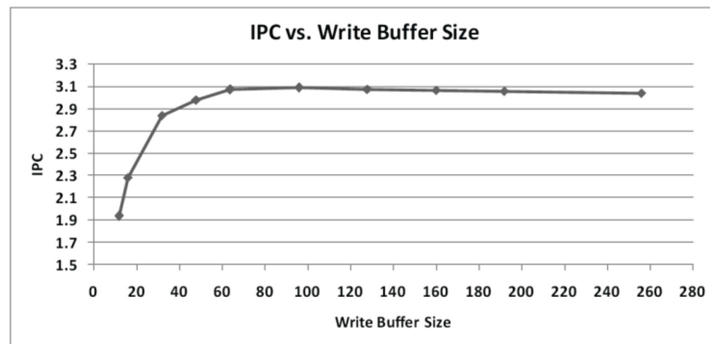

Figure 3.IPC vs. Write Buffer Size

(denoted as *B*) increases from 12 to 64 the overall IPC increases by almost 60%. In order to reach the highest possible IPC, a minimum of 64 entries are required, which is beyond nowadays-acceptable size for such an on-chip buffer. This minimum size becomes even larger when the number of threads further increases. This clearly indicates that the write buffer could easily become the performance bottleneck if its size is kept within the practical range.

4.2. Write Buffer Occupancy
The next analysis is to determine how often the write buffer and how much of it is occupied. Figure 4 shows the percentages of clock cycles during which the given percentage of entries are occupied in the write buffers with *B*=12 and *B*=16. In almost 70% (55%) of time the buffer is completely occupied for *B*=12 (*B*=16), and for 90% of time more than 80% of entries are occupied for either case.

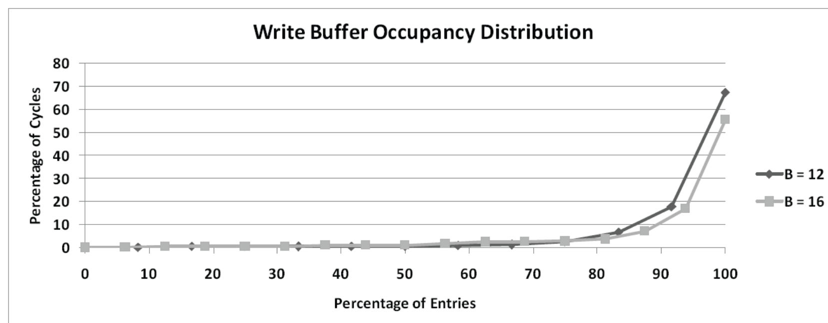

Figure 4. Write Buffer Occupancy Distribution

An explanation to the write buffer's high-occupancy rate would be the long latency of some write instructions staying in the buffer, which in turn leads to the investigation on this latency. Figure 5 shows the average buffer latency distribution of the ten workload mixes. Note that there are two peaks in this distribution – one at one clock cycle and the other at about 330. The first peak which

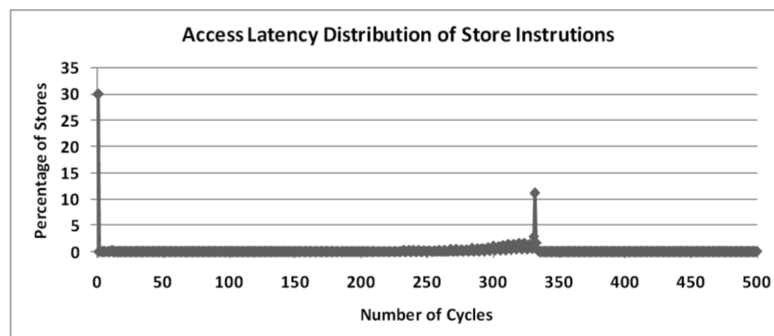

Figure 5. Access Latency Distribution of Store Instructions

Accounts for about 30% of all is from the store instructions with L1 cache hit which requires only one clock cycle as the “default delay” (meaning no waiting for the block that may be in the process of loading from L2). The second peak at 330 obviously corresponds to the instructions that encounter an L2 cache miss which incurs a default delay including the memory access latency (300 clock cycles according to simulator parameter setting) and the access latency to the two levels of cache plus some extra overhead for block transfer. All other occurrences in the distribution are from writes with various levels of hit/miss which, in addition to the corresponding default delay, take on additional delays from bus contention with other read and/or write operations. The delay from the bus contention varies depending on the severity of the competition with other cache misses. Due to this, some of the buffer latency values can go beyond 500.

Another even more intriguing and damaging factor to the SMT's performance is that the write buffer can be completely overwhelmed by a single thread. Figure 6 shows the percentages of time that at least one thread is occupying at least the given percentage of the write buffer entries. As expected, the smaller the write buffer is the more prominent the said dominance becomes. For example for B=12 or B=16, in about 90% of time there is at least one thread occupying half of the write buffer entries, and in over 40% of time at least one thread uses at least three quarters of entries, clearly indicating the imbalance of the resource usage. Even when the write buffer size increases to 32, there is still at least one thread occupying half of entries for over 50% of time. Such a single-thread usage dominance may easily lead to performance degradation when the dominating thread has mostly long-latency write operations, leaving very few precious entries for other threads to compete for.

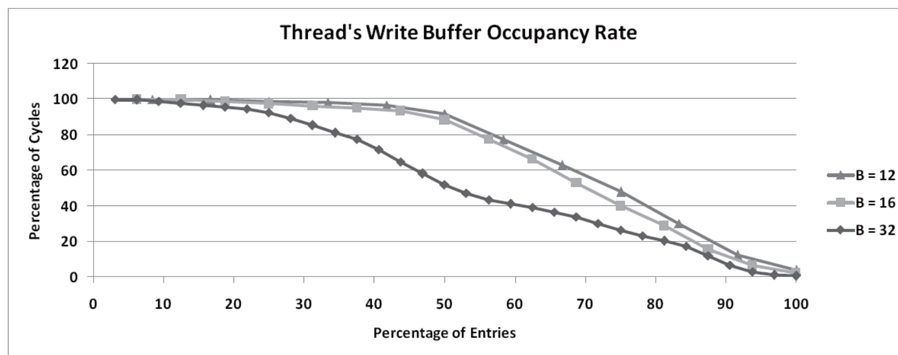

Figure 6. Write Buffer's Single-Thread Dominance Rate

4.3. A Large Write Buffer?

As discussed above, the write buffer is obviously a bottleneck in an SMT system, and to retain a larger write buffer does not seem to be an economical or practical solution. Control logic required to support a larger buffer can be difficult to justify or simply becomes infeasible timing-wise. Even if a larger-size buffer is attainable, its utilization can be discouraging due to the intrinsic nature of write operations. Statistics from the benchmark programs employed in this study show that store instructions account for only about 10% to 18% of all instructions and not only their occurrences are mostly bursty in time but the distribution of their buffer latency (i.e. their hit/miss behaviour) is also egregiously bursty, which easily leads to a very uneven occupancy level in time. This is clearly illustrated by Figure 7 where, under a large write buffer (B=96), for at least a given number of entries occupied the percentage of cycles is tallied. Very much to our amaze, about three quarters of the buffer entries are actually left unused in 50% of time, a strong testament to the decision in not using a large buffer. Instead, one should resort to developing a better allocation algorithm to utilize the limited buffer space in a more intelligent manner.

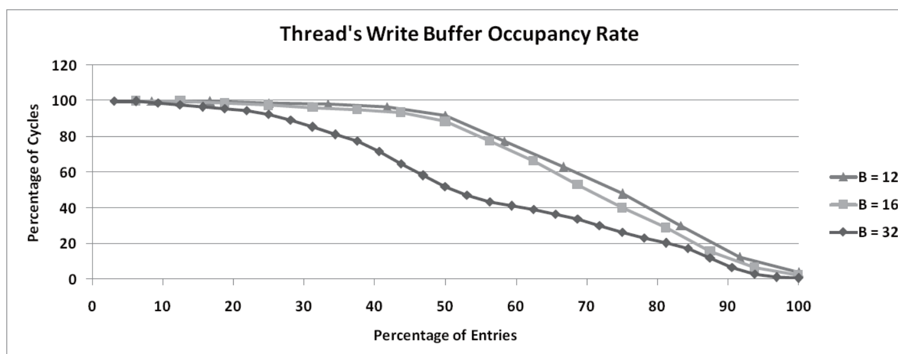

Figure 7. Write Buffer Occupancy Rate

V. Proposed Method

The proposed allocation technique is considered a modification to the *commit* stage of the default algorithm by imposing a very simple control mechanism on assigning the write buffer entries. This technique is based on a simple intention to prevent the buffer from being overwhelmed by any single thread. In this technique, a “cap” value (denoted as (*C*) is set to limit the number of write buffer entries any thread is allowed to occupy at any time. In order to have a simpler and more systematic comparison among using different buffer sizes, instead of using the absolute cap value, we adopt the ratio between the cap value and the buffer size (*B*), denoted as the “cap fraction (*F*)” for our simulation where *F=C/B*. A thread will stop committing any store instruction (at the head of ROB) once this thread has reached its cap value. The complete commit stage algorithm is shown in Figure 8 modified with the proposed technique (the shaded region in flowchart).A write instruction from a thread when reaching the head of its ROB may come across three separate delays waiting for each of the following shared resources: (1) commit bandwidth (2) write port and (3) write buffer, as depicted in the three condition checking steps in the flowchart. Most of the delays are from the third condition waiting for the write buffer, if left uncontrolled. The newly imposed fourth condition is added to reduce this delay. Most of the delays are from the third condition waiting for the write buffer, if left uncontrolled. The newly imposed fourth condition is added to reduce this delay. The main compromise between the benefits and drawbacks from the techniques depends on the setting of the cap fraction value. If the cap fraction is set too high, the intended function of this technique in suppressing single thread's dominating occupancy will not be effective. On the other hand, if the cap fraction is set too low, overall performance may suffer if concurrent writes from multiple threads do not happen often enough. Our simulation to be presented in the next section will be used to study the effect of this compromise and to lend us a good indication of where a good range of cap fraction should be.

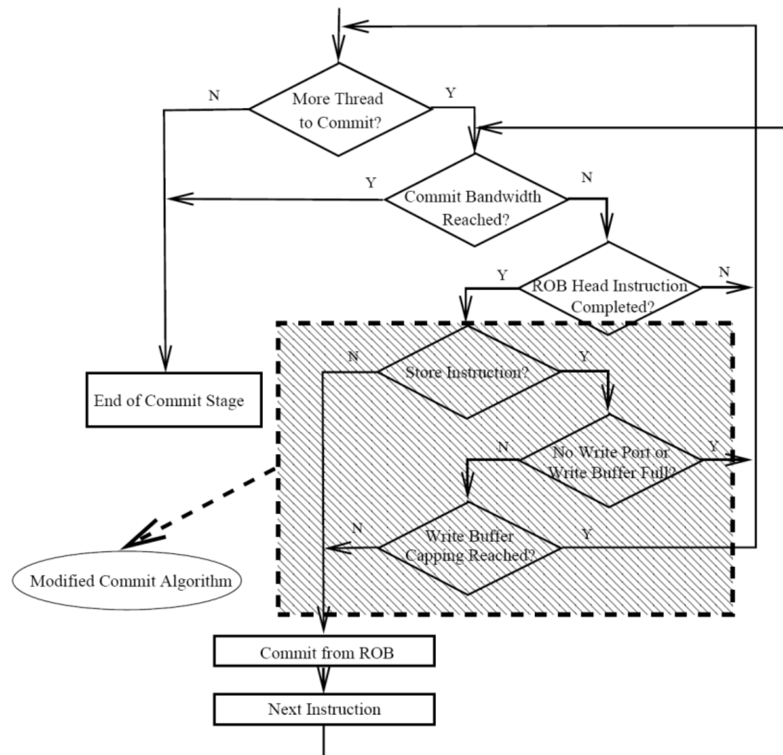

Figure 8. Flow Chart of Modified Commit Algorithm

VI. Simulation results

Based on the simulation environment and the workloads described in Section 3, the proposed technique is tested compared to the default system. As aforementioned, throughout the simulation, the notion of cap fraction will be adopted for a systematic comparison when buffer size is varied, and the fraction value is adjusted with a fixed increment of fraction of 1/16 no matter what value of *B* is chosen. That is, for each simulation run the cap fraction is set to *d*/16 where *d* varies from 1 to 16 (the cap fraction value is always rounded down to the next integer). Figure 9 shows the result of IPC improvement when the proposed technique is applied to 4-threaded and 8-threaded workloads. Improvement from this technique can go up to 12.5% and 66.4% for the 4-threaded and 8-threaded one, respectively. This result in general solidifies all our aforementioned claims. First of all, the effectiveness of our technique is more prominent for an 8-threaded system than a 4-threaded one due to higher

competition. Secondly, optimal setting of the cap fraction is above the point of $1/n$ – the optimal cap fraction happens at 7/16 for the 4-threaded system and 3/16 for the 8-threaded system. Thirdly, there exists an obvious compromise between the ensuing benefit and drawback from setting a different cap fraction value. When the cap fraction is set higher than the optimal point, the higher the cap is the less benefit this technique can produce, and there is virtually no more improvement once the cap fraction is set to be at least 14/16. On the other hand, once the cap fraction is set lower then the optimal value, the tighter (lower) the cap is, the less flexible the buffer allocation becomes and the benefit of sharing becomes less. In the 4-threaded case, when the cap fraction falls below 2/16 any benefit from the technique is completely offset by its ensuing detrimental effect.

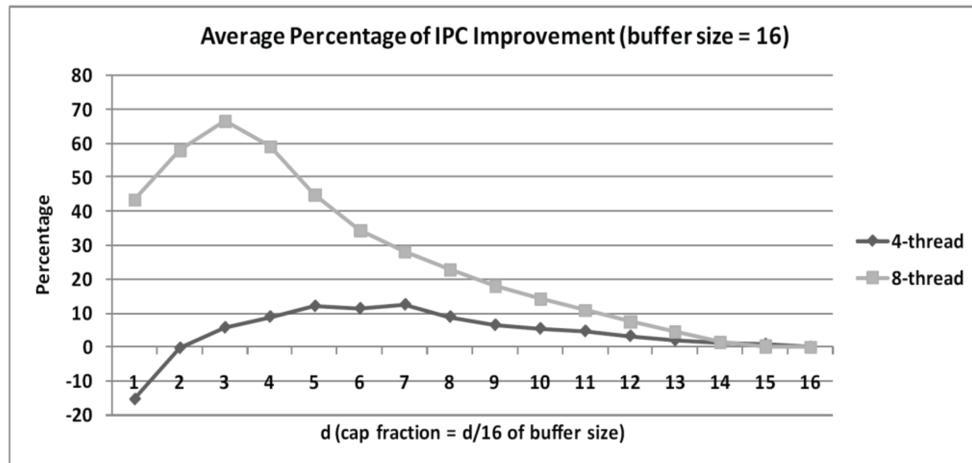

Figure 9.Average Percentage of IPC Improvement (B=16)

However, in the 8-threaded case, very much to our surprise, there is still a very significant performance improvement (43%) even when the cap fraction is set to the lowest 1/16 (i.e. a cap of "one" entry). This scenario clearly indicates that, in such a high competition environment: 8 threads for a buffer size of 16, any capping is better than no capping and, if left uncontrolled, constant dominating occupancy situation from one or very few threads. Figure 10 further shows per-mix IPC improvement in a 4-threaded system, from which we can see that most of the mixes have the similar trend of IPC improvement and the highest IPC improvement gained by a mix can be up to 87%.

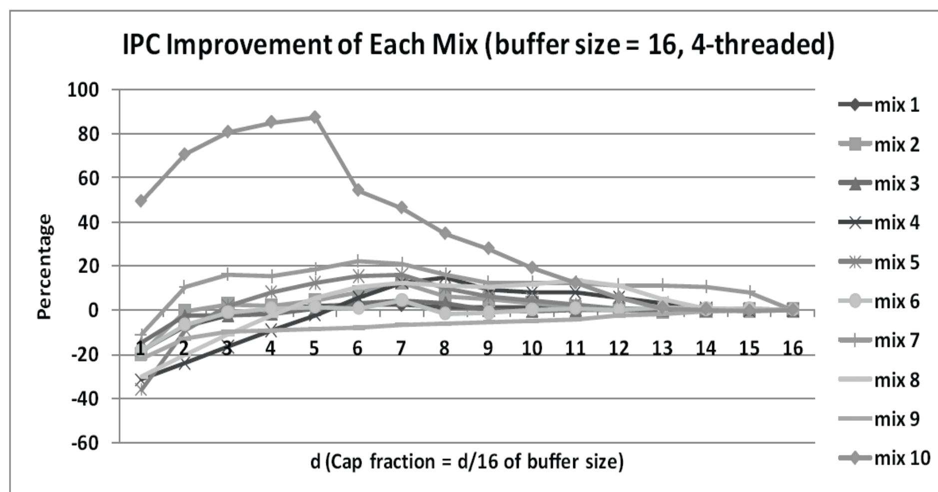

Figure 10. Per-Mix IPC Improvement vs. Capping Value for 4-threaded Workload (B=16)

Figure 11 demonstrates the proposed technique's influence on the performance with different write buffer sizes varying from 12 to 64. This result further solidifies our claim in how the effective of our technique can be swayed by the buffer size, achieving a maximum improvement of 26% and 95% with B=12 on 4-threaded and 8-threaded workloads, respectively. On the other hand, in the 4-threaded case, there is minimal gain from this technique when the buffer size exceeds 32. In the 8-threaded case, the technique somehow still maintains a discernible amount of improvement (up to 3.6%) even the buffer is increased to 64.

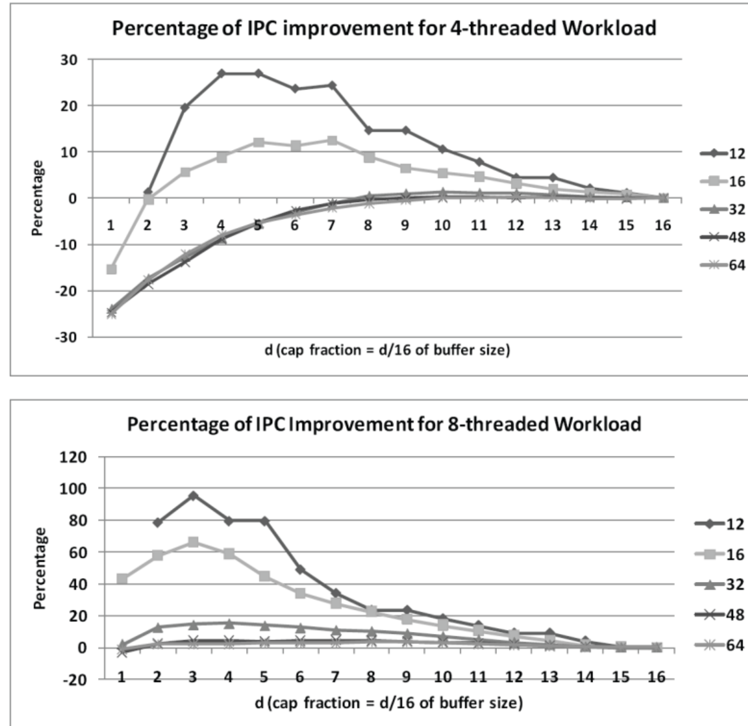

Figure 11. Performance Comparison with Varying Write Buffer Size for 4-threaded and 8-threaded Workloads

In order to reflect the degree of execution fairness among the threads when the proposed technique is applied, Figure 12 shows the percentage of improvement of Harmonic IPC for 4-threaded and 8-threaded workloads. As the result shows, when cap fraction is set too small, the harmonic IPC tends to suffer even when the system sees an improved IPC. In a system with more threads, the effect in balancing resource among threads brought by the proposed technique is much more significant. For example, in a 4-threaded system, our algorithm gains a harmonic IPC improvement of up to 10.8%, while in a 8-threaded system, the maximal improvement can be as high as 46.7%. One would also notice that the optimal cap fraction for harmonic IPC usually is not the same value as that of IPC, which means in the practical systems, one needs to choose the best cap fraction by compromising the improvement between overall IPC and harmonic IPC. Similar to the trend in overall IPC, this result also indicates that the proposed technique leads to more improvement in harmonic IPC when more threads are involved and a smaller write buffer is employed, due to the additional fairness it provides to the system.

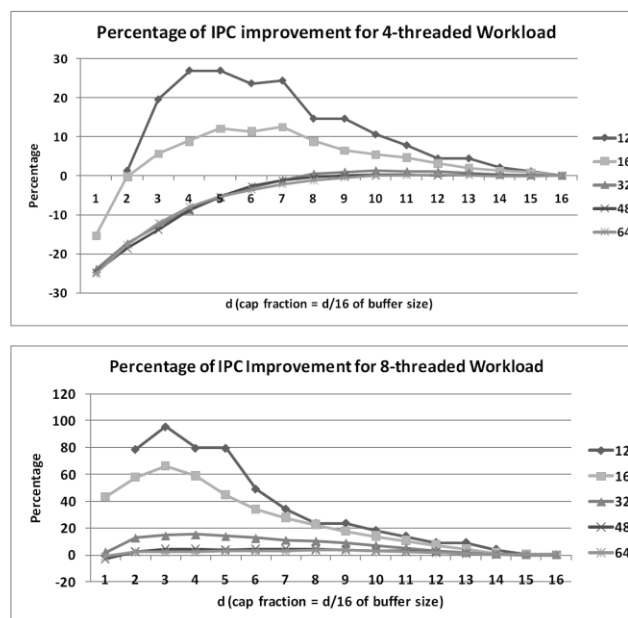

Figure 12.Harmonic IPC Comparison with Varying Write Buffer Size for 4-threaded and 8-threaded Workloads

An analysis that can directly reveal how the proposed technique is capable of effectively relieving the pressure on the write buffer is the comparison on write buffer occupancy. Figure 13 shows the comparison of the write buffer occupancy rate between the default system before and after the modified commit algorithm. The displayed result is for a 4-threaded system with the write buffer

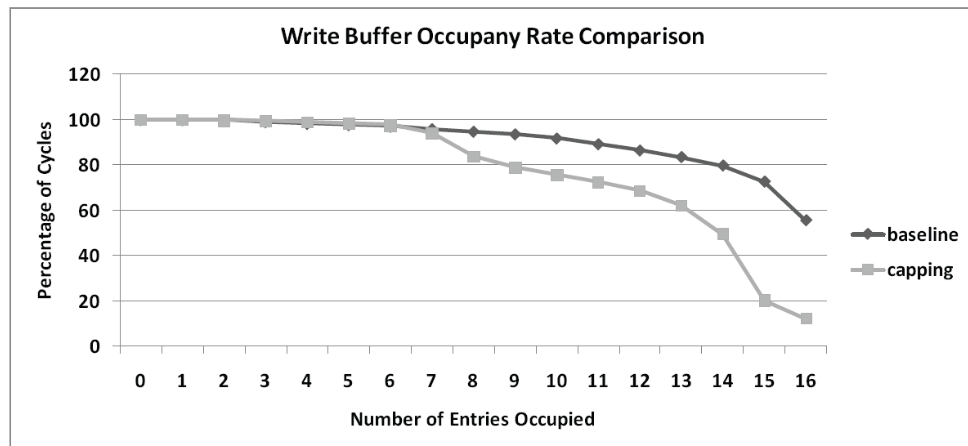

Figure 13.Write Buffer Occupancy Rate Comparison (*B*=16)

Size equal to 16 and the capping fraction set at 7/16. With the proposed method applied, the percentage of cycles with a high-occupancy write buffer shrinks dramatically. The percentage of time when the buffer is completely full is reduced from 56% to 14%, a huge improvement leading to the necessary buffer space to sustain a less disrupted write traffic.

VII. Conclusions

This paper clearly demonstrated that uncontrolled utilization of resource shared among the threads in an SMT system could significantly affect the overall performance. Due to the limited size of the write buffer and long latency from some write operations, write buffer easily becomes a bottleneck that severely limits performance of an SMT system. By capping the write buffer usage of each thread, utilization of this critically shared resource can be vastly improved and consequently leads to a very considerable performance gain. Another noteworthy aspect in this technique is that such an improvement is achieved without having to invest much extra hardware nor imposing extra constraints on the clock rate and can be incorporated with further intelligence into such a control technique for even more performance improvement.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Number 1538418. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] H. Hirate, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads", *Proc. of 19th Annual International Symposium on Computer Architecture*, 1992, May, 136-145.

[2] D. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *Proc. of 22nd Annual International Symposium on Computer Architecture*, 1995, May, 392-493.

[3] P. P. Chu and R. Gottipati, "Write Buffer Design for On-Chip Cache", *In the Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pp. 311-316, October 1994.

[4] J. L. Hennessy and D. A. Patterson, "Computer Architecture:A Quantitative Approach", Morgan Kaufmann Publishers, 2007.

[5] D. M. Tullsen, S. J. Emer, H. M. Levy, J. L. Lo and R. L. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multi-Threading Processor", *Proc. of 23rd Annual International Symposium on Computer Architecture*, 1996, May, 191-202.

[6] D. M. Tullsen and J. A. Brown, "Handling Long-latency Loads in a Simultaneous Multithreading Processor*", In the Proceedings of the 34th International Symposium on Microarchitecture*, pp. 318-327, December 2001.

[7] F. J. Cazorla, A. Ramirez, M. Valero and E. Fernandez, "Dynamically Controlled Resource Allocation in SMT processors", *Proc. of 37th International Symposium on Microarchitecture*, 2004, Dec, 171-192.

[8] S. Choi and D. Yeung, "Learning-Based SMT Processor Resource Distribution via Hill-Climbing", *the Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pp. 239-251, June 2006.

[9] H. Wang, I. Koren and C. M. Krishna, "An Adaptive Resource Partitioning Algorithm for SMT Processors", *the Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 230-239, October 2008.

[10] T. K. D. Nagaraju, C. Douglas, W.-M. Lin and E. John, "Effective Dispatching for Simultaneous Multi-Threading (SMT) Processors by Capping Per-Thread Resource Utilization", *The Computing Science and Technology International Journal*, Vol. 1, No.2, pp. 5-14, December 2011.

[11] S. Subramaniam, M. Prvulovic and G. H. Loh, "PEEP: Exploiting Predictability of Memory Dependences in SMT Processors*", the Proceedings of the 14th International Symposium on High Performance Computer Architecture*, pp. 137-148, February 2008.

[12] D. Kang and J. L. Gaudiot, "Speculation Control for Simultaneous Multithreading", *the Proceedings of 18th International Parallel and Distributed Processing Symposium*, April 2004.

[13] K. Luo, M. Franklin, S. S. Mukherjee and A. Seznec, "Boosting SMT performance by Speculation Control*", the Proceedings of the International Parallel and Distributed Processing Symposium*, April 2001.

[14] Y. Zhang, C. Douglas and W.-M. Lin, "On Maximizing Resource Utilization for Simultaneous (SMT) Processors by Instruction Recalling", *The 18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, July 16-19, 2012, Las Vegas, NV.

[15] Y. Zhang and W.-M. Lin, "Capping Speculative Traces to Improve Performance in Simultaneous Multi-Threading CPUs", *The 18th International Conference on Parallel and Distributed Processing Techniques and Applications, Workshop on Multithreaded Architectures and Applications,* May 20-24, 2013, Boston, MA.

[16] Y. Zhang, M. Hays, W.-M. Lin and E. John, "Autonomous Control of Issue Queue Utilization for Simultaneous Multi-Threading Processors", *The 22nd High Performance Computing Symposium (HPC 2014)*, April 2014.

[17] K. Skadron and D. W. Clark, "Design Issues and Tradeoffs for Write Buffers", *the Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pp. 144-155, February 1997.

[18] S. Kim and J. Lee, "Write Buffer-Oriented Energy Reduction in the L1 Data Cache of Two-level Caches for the Embedded System", *the Proceedings of the 20th symposium on Great lakes symposium on VLSI*, pp. 257-262, May 2010.

[19] "3rd Generation Intel XScale Microarchitecture--Developer's Manual", *http://www.intel.com*.

[20] J. Sharkey, "M-Sim: A Flexible, Multi-threaded Simulation Environment", *Tech. Report CS-TR-05-DP1*, Department of Computer Science, SUNY Binghamton, 2005.

[21] Standard Performance Evaluation Corporation (SPEC) website, http://www.spec.org/.