WRITE BUFFER PARTITIONING WITH RENAME REGISTER CAPPING IN MULTITHREADED CORES

Allan Diaz and Wei-Ming Lin

Department of Electrical and Computer Engineering, The University of Texas at San Antonio, San Antonio, 78249-0669, TX, USA

ABSTRACT

In simultaneous multithreaded systems, there are several pipeline resources that are shared amongst multiple threads concurrently. Some of these mutual resources to mention are the register-file and the write buffer. The Physical Register file is a critical shared resource in these types of systems due to the limited number of rename registers available for renaming. The write buffer, another shared resource, is also crucial since it serves as an intermediary between the retirement of a store instruction and the writing of its value to cache. Both components, if not configured accurately, can serve as a bottleneck in inefficient usage of the resources and output undesirable performance.

However, when configuring both shared components concurrently, there is potential to all eviate common performance congestion. This paper shows that when implementing a static register capping algorithm (limiting the number of physical register entries for each thread), there is a byproduct of increased variety in source for the write buffer. This also presents an opportunity for the write buffer to have a higher variety to potentially choose for a better suitable thread asit's source at certain clock cycles. With this presented opportunity, this paper proposes a technique to allow the write buffer to both prioritize and enforce the choice for low-latency threads by partitioning the write buffer in two sections; cache-hit priority and cache-hit only partitions, showing that system performance and resource efficiency can be further improved by using this technique in a modified SMT environment.

KEYWORDS

Write Buffer, Simultaneous Multithreading, Physical Register File, Register Renaming

1. INTRODUCTION

A Simultaneous Multi-Threading processor (SMT) is a superscalar processor architecture that provides an enhanced method for improving overall system performance by processing instructions from many threads concurrently, termed thread-level parallelism (TLP) [1], [2]. With the characteristic of having parallel pipeline structures, various threads can occupy through separate pipeline paths in the processor concurrently in the same clock cycle(s), which allows for better resource utilization throughout the system. Subsequently, some pipeline resources are shared and may be occupied by various threads at the same time as well. Figure 1 models the pipeline organization of a 4-threaded SMT system and accurately depicts the visual difference of the parallel and shared components. Shared components, if not configured accurately, can serve as a bottleneck in inefficient usage of the resources and output undesirable performance. Therefore, to maximize the overall system throughput, its shared resources must be properly managed to avoid newly introduced bottle necks due to concurrent execution of threads.

Among the shared components in an SMT system, the shared Physical Rename Register, is regarded as the most crucial of mutual components since these registers would be held for the

longest time until the respective instruction is ultimately committed. Another shared resource to mention is the write buffer, which lies between a core's pipeline and cache memory, serving as an intermediary between the retirement of a store instruction and the writing of its value to cache. These two shared components, when not individually designed correctly, can cause inefficient resource utilization and poor performance. Modifying both components separately can relieve of a probable bottleneck effect, however, later sections will show that there is potential for performance increase when both shared components are configured concurrently.

This paper will introduce a technique known as register capping, the effects it has on the write buffer, and present an opportunity to modify the write buffer into different partitions for better response to the effect. The proposed technique will also rearrange the commit stage allowing a difference in priority amongst the threads at different clock cycles. Thus, the primary focus of this paper is to determine the maximal portion of physical registers that can be allocated, paired with the combination of different partition values in the write buffer for a more latency free and efficient SMT system. This combination in turn will maximize IPC throughput and the proposed method will show very significant improvement up to 70.6% in a 4-threaded SMT system, and 65.9% in an 8-threaded SMT system.

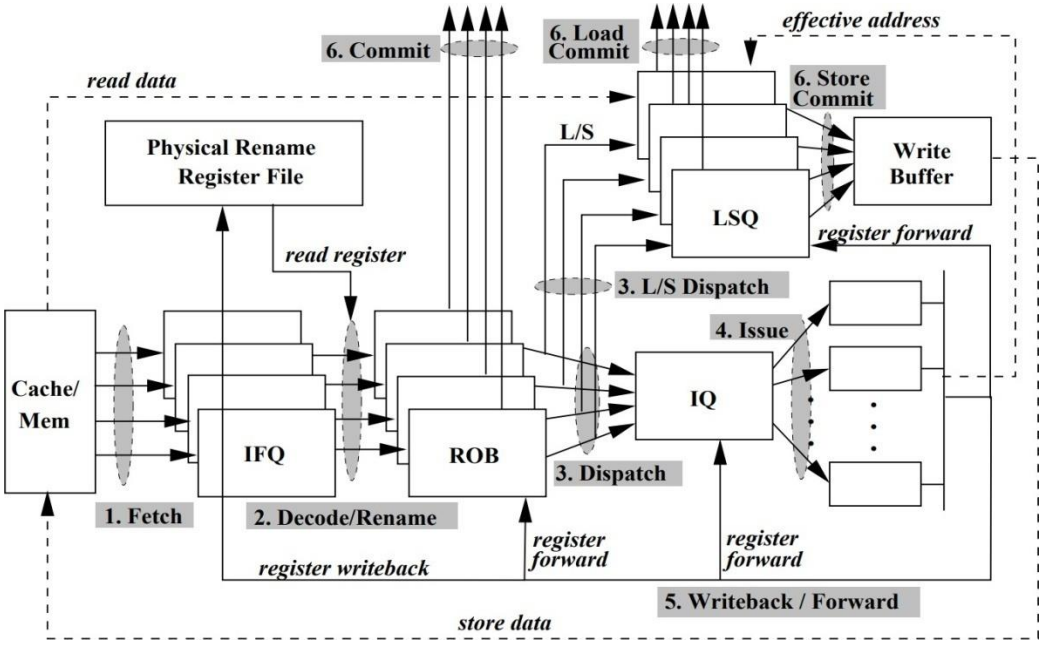

Fig 1. Pipeline stages in a 4-threaded simultaneous multithreaded system.

2. RELATED WORKS

There is ample research on different methods of improving an SMT system and enhancing its components. Some to mention explore several techniques of resource control at various stages of the pipeline. In [3], a resource allocation algorithm is presented that is designed to optimize the performance of the physical register file in a computer system. By providing a real-time cap value based on each thread's activity and effectiveness in occupying the registers, the resource is utilized more efficiently, leading to an overall improvement of system performance. According to Hily and Seznec [4], there is an ideal value pairing for the cache size and workload size in an SMT system. In their work, they demonstrated how several cache parameters, such as block size and size relative to thread count, could exhibit advantageous behavior when configured differently from usual

architectural setups. By employing the technique presented in [5], a dynamic limit is set on how many instructions can be occupied in the issue queue (IQ) by a thread to prevent the queue from becoming clogged by slower threads.

The ICOUNT policy mentioned in [6] is a method of fetching instructions based on how many instructions are in the system before execution, where the thread with the fewest instructions is given higher priority. Having multiple threads being executed concurrently makes this fetch policy ideal for SMT systems. To prevent performance degradation, [7] allows each thread to use a portion of the write buffer rather than all threads having access to all of it, in a sense partitioning. A scheduling algorithm presented in [8], prioritizes the order of write buffer commits among threads to maximize itsutilization of resources, allowing equal opportunity for all threads to retire their instructions.

A multithreading algorithm in [9] prevents some threads from starving while others consume all of the processor cycles by implementing a fairness metric into a multithreading method named Switch-On Event. This method improves processor throughput by switching threads on execution stalls enforced by a fairness ratio of the individual threads' speedups and performance. This technique replying on this ration value guarantees fairness at different levels of strictness and improved the weighted speedup amongst threads. In [10], to diminish the waste of cores resources on instructions that do not need resources that can lead to false ordering dependences, this technique efficiently scales instructions through a hybrid microarchitecture that can dispatch instructions in in-order scheduling mechanisms. This measures the opportunity to better organize a dispatch and evaluate a practical dispatch mechanism. [11] proposes several schemes to improve scalability and increase scheduling throughput with a method called "2OP_BLOCK”. This technique takes another method called "instruction packing" a step further and completely avoids sending an instruction with two source operands that are not ready. This reduces scheduling complexity byeliminating the logic required to support instructions with two unprepared source operands. On [12], a method is introduced based on a memory monitoring framework that centers the concept ofactivity vectors for threads. This allows asystem scheduler to predict cache utilization and inter-thread contention using a dynamically tracking program on phase behavior in an autoregressive model.

A register file capping technique in [13] and [14] is applied on the rename stage that limits the number of additional physical registers that a thread is allowed to occupy at any point of time with a so-called “cap value”. By capping the integer physical register usage of each thread, utilization of this critically shared resource can be vastly improved and consequently leads to a very considerable performance gain.

Regarding the specific impact on commit stage and the shared write buffer component, [15] proposesthat by limiting the number of cache misses that may concurrently occupy the write buffer, it can reduce rejection of resources to cache hits. The paper first dissectsthe effect of cache misses in the write buffer and its behavior, showing a shared write buffer is routinely occupied by many long-latency cache misses in an SMT system. This behavior greatly increases the probability of the write buffer being fully occupied, stalling threads that have cache hits ready to commit. The proposed technique then aims to 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. With specifically concentrating on works [13], [14] and [15], these related works serve as motivation to the proposed technique that works in conjunction in this paper.

3. SIMULATION ENVIRONMENT

3.1. Simulator

M-Sim, a simultaneous multi-threaded microarchitectural simulator, was used as the environment model for the performance analysis of the proposed technique. M-Sim includes accurate models of the pipeline structures of an SMT system, however namely the Register File and Write Buffer will be the focus for this paper. The simulation configuration parameters are outlined in Table 1. In this paper, buffer sizes of both 16 and 32 entries and register file sizes of 160 and 320 will be used for both 4-threaded and 8-threaded workloads respectively.

3.2. Workloads

The multi-threaded workloads used for simulations are chosen from the SPEC CPU 2006 benchmark suite that consists of programs with a combination of ILP levels that present a variety of workloads. These benchmarks are chosen in sets of 4 and 8 to simulate the 4-threaded and 8-threaded workloads and can be referred on their combinations to a corresponding "MIX" number identifier. Benchmarks are rated based on their ILP classification also shown in the corresponding Tables. The chosen workloads are based upon having a variety of ILP level combinations and the following Mixes are shown in Table 2 for 4-threaded workloads, and Table 3 for 8-threaded workloads.

3.3. Metrics

To evaluate the performance of this proposed algorithm, combined IPC (Instruction Per Cycle) is a typical metric used to measure the overall performance throughput for each mix, which is defined as the sum of each individual thread's IPC:

$$Overall\ IPC = \sum_{j}^{N} IPC_i \tag{1}$$

where N denotes the number of threads per mix. The new over all IPC of a modified SMT system will then be compared to that of the baseline overall IPC measured in the simulation environment with no modifications or different adjustments from the new. Both baseline and new overall IPC measurements will then serve as variables to find the average IPC improvement defined as function:

$$Avg\ IPC\ Improvement\ Percentage = (\sum_{j}^{M} \frac{IPC_j^{new} - IPC_j^{baseline}}{IPC_j^{baseline}} / M) \times 100 \tag{2}$$

where M is number of mixes. Throughout this paper, these two formulas (1) and (2) will be used to denote the change in performance of each stage in modifying the SMT system.

Table 1. Simulated Processor Configuration Setup

Parameter	Configuration
Bandwidth	8 wide fetch/dispatch/issue/commit
ROB, LSQ, IFQ, IQ Size Entry	128/48/32/32
Functional Units / Issue Latency / Completion Latency	Integer ALU: 4/1/1 Integer Mult: 1/9/3 Integer Div: 1/12/12 Floating Point Add: 4/1/2 Floating Point Mult:1/1/4 Floating Point Sqrt: 1/24/24 Floating Point Div: 1/19/20
Level-1 Instruction Cache	64 KB, 2-way set-associative 64-byte line
Level-1 Data Cache	64 KB, 4-way set-associative 64-byte line write-back, 1 cycle access
Level-2 Unified Cache	512 KB, 16-way set-associative 64-byte line write-back, 10 cycle access
Write Buffer	16 /32 64-byte entries
Branch Predictor	Bimodal 2K entry
Physical Registers (4-thread / 8-thread)	160 / 320 Integer and Floating Point
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

Table 2. 4-Threaded Workloads For Simulation, Chosen From Spec Cpu2006 Suite

Mix	Benchmarks	ILP Classification		
		Low	Medium	High
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, xalancbmk	3	1	0
10	xalancbmk, bzip2, lbm, perlbench	4	0	0

Table 3. 8-Threaded Workloads For Simulation, Chosen From Spec Cpu2006 Suite

Mix	Benchmarks	ILP Classification		
		Low	Medium	High
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, cactus, gromacs, milc	0	5	3
5	star, 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

4. BACKGROUND

4.1. Physical Register File

As a result of the limited number of rename registers available in SMT systems, the Physical Register file is one of the most critical shared resources. Instructions in some threads with high latency block the progress of other fast threads, resulting in inefficient use of resources and poor performance. Which is why, to avoid bottlenecks during the renaming process, SMT systems typically require a much larger physical register file to accommodate multiple threads renaming the registers. Simply adding more registers, however, can be a cost-effective solution.

As mentioned, one specific fixed cap technique is proposed in [13] and [14] that gives each thread in the system an allotted portion of the additional registers to use for register renaming. This limited portion is referred to in this paper as a cap value. Such a cap value, if selected properly, can prevent a long-latency thread from dominating the physical register file; that is, faster threads are still allowed to proceed with their allotted registers. The technique provides a significant performance improvement over a default system. It was found that an optimal fixed cap was usually around 9.

Renaming physical registers will be limited if the size of the physical register file is not much larger than the size of the architectural registers. In an SMT system, this limitation is aggravated even further as resource sharing among several threads is intended to allow for a reduced number of resources than would be required in multiple single-threaded superscalar systems. In our SMT simulation, a 4-threaded system requires a minimum of 32 registers for every thread. This equates to 128 (4 x 32) physical registers as the baseline with which no renaming is possible. For renaming to be possible, allocation of additional registersfor renaming is needed. If a total of 160 physical registers are used, 32 registers (160 - (32 x 4) = 32) are available among the 4 threads for renaming. Applying the [14] technique of register capping with regards to having 32 rename registers available is a reasonable starting parameter. To keep the same ratio of register to threads, the total amount of physical register for 4-threads will be 160 and 8-threads will work with 320.

4.2. Write Buffer

When a store instruction is retired from the pipeline, the result that will be written to memory is first transferred to the write buffer. A write-allocate cache strategy moves the cache line of the write instruction to data level-one (DL1) cache while the value is temporarily kept in a write buffer entry [15]. This buffering period in DL1 might range from a single clock cycle for a cache hit to hundreds of clock cycles for a cache miss. Due to the low latency in a cache-hit scenario, favoring the write buffer to choose a cache-hit will increase performance. The focus is to implement a write buffer partitioning algorithm with a modified fixed cap on resource allocation of the rename registers.

5. PROPOSED METHOD

5.1. Approach

A first step to improving a capping environment would be to find a good cap value that will give us the highest average IPC gain as a starting ground. A simple application of every different value within the range of available registers is shown in Figure 2. As shown, the varying cap values in a 4-threaded workload also vary in average IPC Improvement percentages. The range with the highest improvement is between cap value 7 and 12, but more specifically the highest is when cap equal to 9 with IPC Improvement of 37.75%. Therefore, continuing to analyze the effects of register capping in other resources with a cap value of 9 is an understanding and straightforward approach.

5.2. Analysis

The next phase is to see how capping affects the source of the write buffer, the Re-Order Buffers (ROBs) of each thread and visually depict a difference in variety of resources. Figure 3 and 4 both show a direct comparison in the different number of threads that are ready to commit for their corresponding 9 and 32 cap value. Visually, there is a shift from right to left from having less red and orange sections to more and also almost little to no blue in certain mixes, already inferring there is transfer in number of threads. Particularly, mixes 4, 7, 8, 10 reveal the most shift in color.

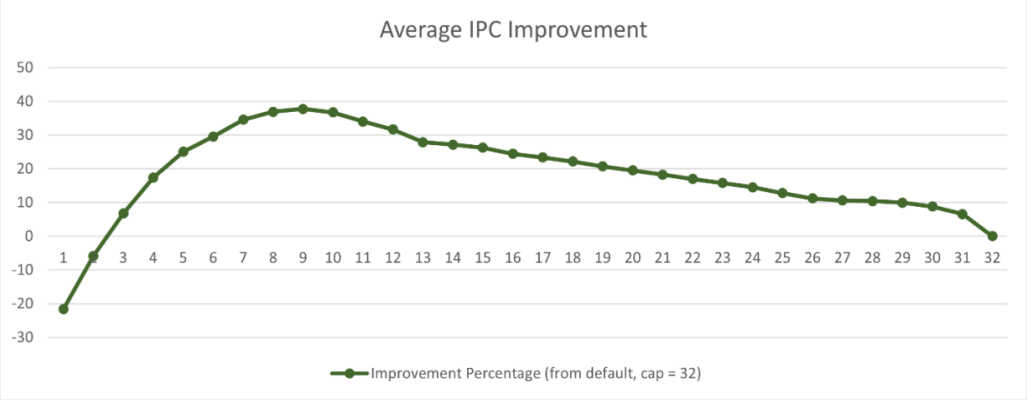

Fig 2. Average IPC Change for 10 4-threaded mixes with varying cap values.

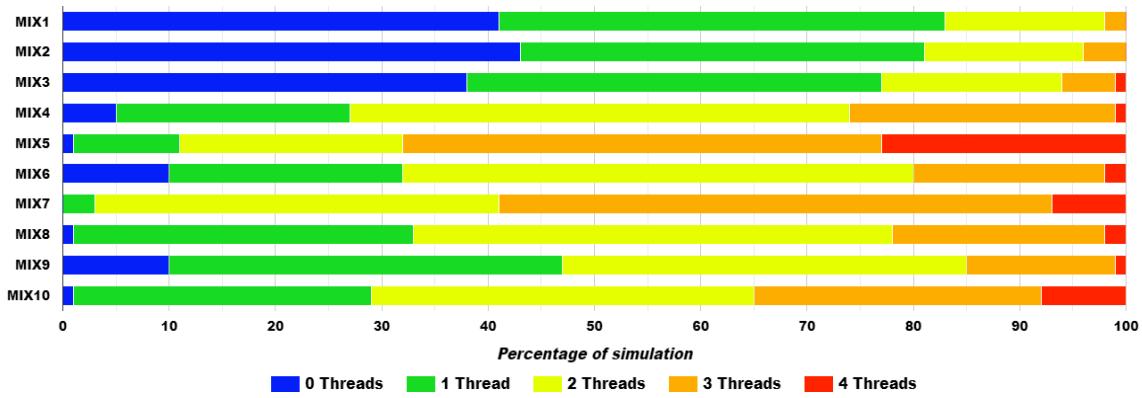

Fig 3. Number of unique threads ready to commit in a default system (Cap = 32)

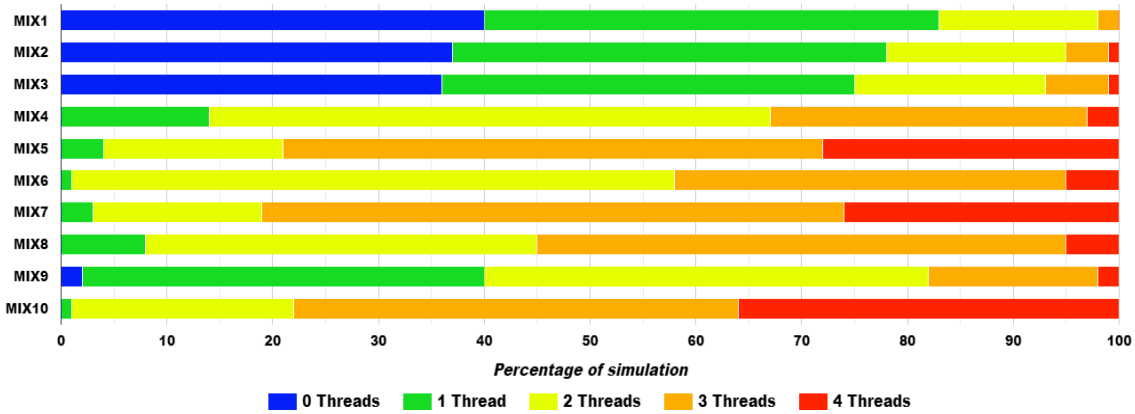

Fig 4. Number of unique threads ready to commit in a capped system (Cap = 9)

Comparing the combined average number of threads for all mixes would tell us the overall average of what the write buffer would encounter throughout the simulation. In that perspective, in a default, no capping environment, there is 14.8% of the simulation having 0 threads ready to commit, 27.3% having only 1 thread ready to commit, 31.8% for 2 threads, 21.2% with 3 threads, and only 4.9% for 4 threads. When applying cap value to 9, there are 0, 1, 2, 3, and 4 threads ready to commit through the whole simulation with 11.6%, 19.4%, 29.2%, 29.2%, and 10.7% respectively. Comparing both scenarios with having a no capping as a baseline, there is a decrease of 3.2% of 0 threads ready to commit, 7.9% decrease of 1 thread, 2.6% decrease for 2 threads, a healthy 7.9% increase for 3 threads, and great 5.8% increase on 4 threads which is a significant difference considering any change on 4 threads supersedes other number of threads. The shift from a default variety of threads to a larger one throughout the simulation infers that the write buffer now has a bigger variety pool source of ROB heads to choose from. Having a higher variety provides not only a higher chance to encounter a cache hit entry among the threads, but also an opportunity to filter out threads at any given time. This choice of exclusion comes at a smaller risk of inefficiency compared to default, again, due to the higher variety and number of entries flowing into the write buffer. When capping to 9, rather than having a write buffer have a first come first serve through a round robin selection, the write buffer is allowed to choose which thread to choose from to commit.

This gives the opportunity to prioritize threads having potential cache hit instruction over their non-cache hit thread counter parts. A thing to note on how capping affects the source of a write buffer is the actual size of the ROBs for each thread in every mix. Analyzing the ROB size in Figure 4 and 5, with no capping, the average for each mix barely reaches 20, the max size with around 50-55 occurred twice, and the highest total number of entries for a mix was 94 on mix 7. In comparison, with cap value 9, the average size exceeds 20 numerously, the max size exceeding 55 occurred 8 times, and the highest total number of entries for a mix was 145 on mix 6. Although the proposed algorithm does not manipulate ROB size as a factor, it directly affects the potential of improvement.

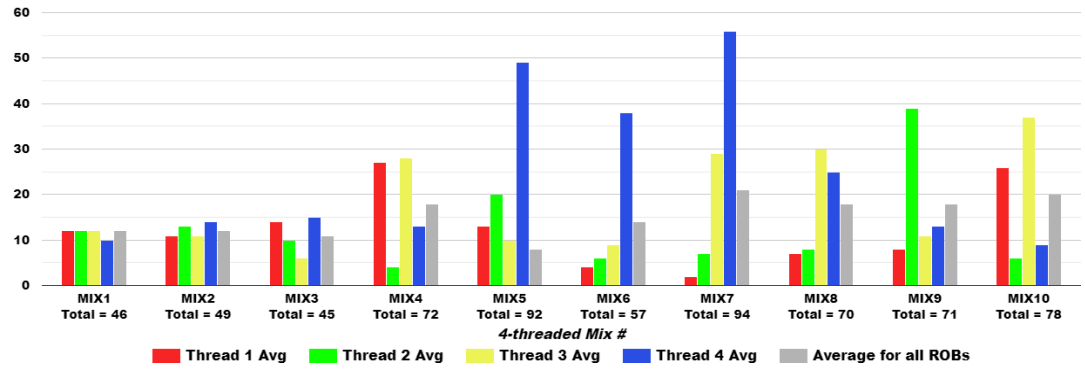

Fig. 5. Size of ROB for each thread and average size per Mix for a default system. (Cap = 32)

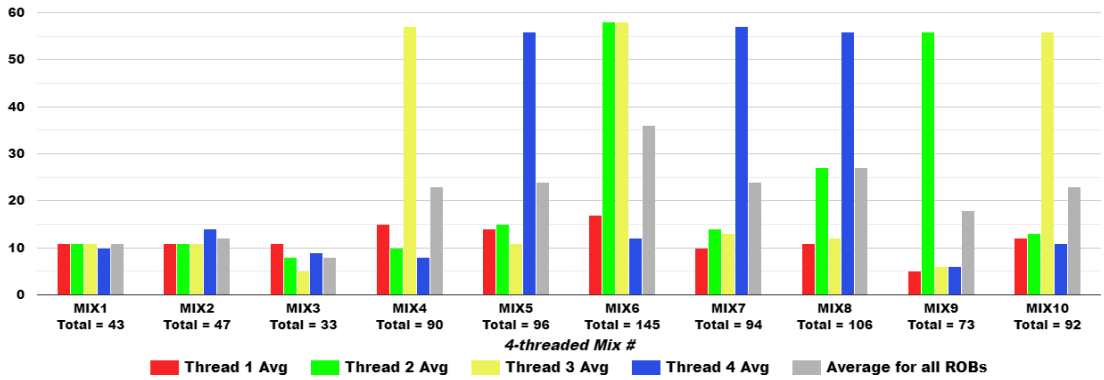

Fig .6. Size of ROB for each thread and average size per Mix for a capped system. (Cap = 9)

6. PROPOSED ALGORITHM

Previously, it has been shown that when register capping with 9, both the variety of the threads and average size of ROB increase. Therefore, the proposed algorithm aims to filter in the higher variety as the most suitable threads with a cache-hit at the head of ROB. The approach to this is to first depict the process of a default commit process.

Figure 7 visually depicts the normal process used in an SMT system during the commit stage where a store instruction gets committed in a round robin-ordered fashion for each thread. Figure 8 also shows the pseudo code associated with Figure 7 and additional detailed actions associated in the commit phase of an SMT system. In this commit portion, a thread is selected to be the new current thread as a source for the rest of the commit cycle, chosen by round robin formula, c modulo N, having c depict the current clock cycle of commit initiation and N representing the number of concurrent threads. Once a thread is selected, the first entry (head) of the ROB of the selected thread must be committed first, assuming this entry is ready to be committed. If the current thread does not have an ROB head ready to be committed, the thread is immediately excluded for consideration for that clock cycle and checks to see if a next thread is available to repeat the process of checking if the ROB head is ready to be committed. Once a ROB entry is verified to be ready and commit bandwidth is available, the next step is ensuring the ROB head is a store is crucial as it is the only way the entry can be entered into the write buffer. If the entry is not a store, there is no need for a write buffer, so the instruction immediately gets committed to commit stage. Now, available commit bandwidth is decremented, and the next entry in the same thread that is now the new ROB head is checked. Finally, if ROB instruction has been verified to be a store and if the write buffer is not full to even have the availability to accept an entry, the store value will be written to the write buffer and the workflow continues until commit bandwidth is used up or there are no more threads remaining to choose from. One thing to emphasize is that if the write buffer is full, the thread is completely disregarded and no longer in consideration for the rest of the commit process cycle.

With the presented byproduct of increased variety or number of different threads ready to commit when register capping to the most efficient cap value, the current default algorithm in the commit stage does not take full advantage to be able to choose which thread is best to choose from. In other words, with this default algorithm, the write buffer does not discriminate if the current ROB entry of the thread is a cache hit/miss and regardless if ROB head of the thread will potentially degrade performance of by having a long latency write buffer entry for several hundred clock cycles, the write buffer will always choose the first entry when it gets to that section of the commit process.

The goal of this technique is to allow the write buffer to choose what ROB head entry from a certain threadto prioritize cache hit entries, prevent potentially blocked cache hits, and maximize commit bandwidth, before filling up and decreasing efficiency. Therefore, modifying the commit stage and partitioning the write buffer into prioritized sections along with finding the optimal prioritizing level values for each partition will be the key with this technique.

Applying the proposed method, Figure 9 in illustration form now shows the modified process during the commit stage where a store instruction still gets committed in a round robin ordered fashion for each thread. Figure 10 again, also shows the associated pseudo code with Figure 9, with detailed actions in incorporating modifications. In this redesigned commit stage, a thread is still selected by round robin formula, c modulo N. The process of checking to see if the ROB head of a thread is a store and requirements before that are still the same as default, but the algorithm starts to change once the process starts to involve the write buffer. Instead of the non-full write buffer simply allowing the instruction to commit once it gets to that point, it checks to see if the entry is a data cache level 1 hit. If a cache-hit is true, then entry will be committed to write buffer and written to cache within the next few clock cycles. If entry is not a cache-hit, this is where the

partition values come into play. Condition functions 3 and 4 below are the conditions of the proposed partitioned write buffer and the M and N values that trigger different paths in the new modified commit stage.

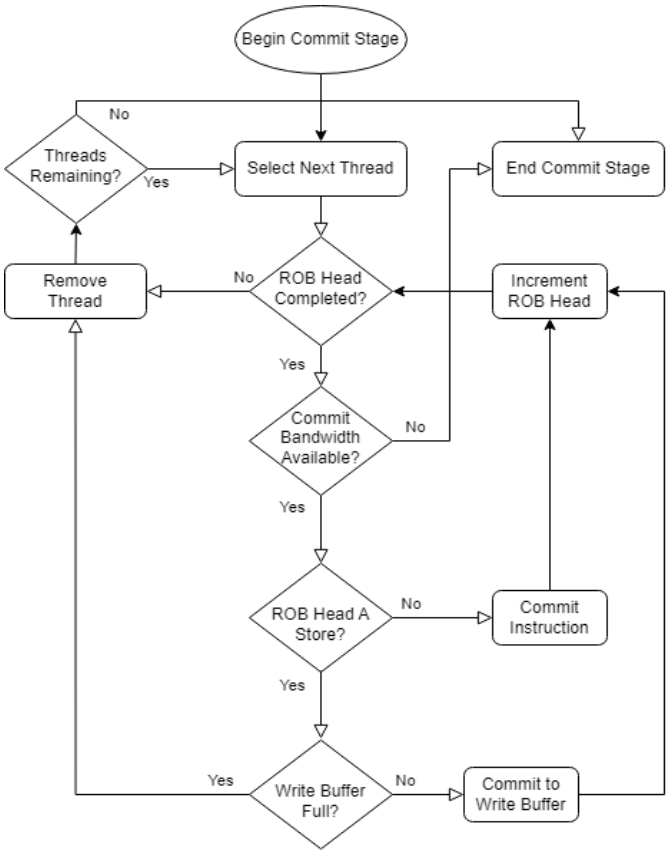

Fig. 7. Unmodified round robin commit stage algorithm.

```
function commit()   //round robin
{
     Collect current thread_list
     Check if any threads are stalled or if no instructions have committed for a while
     current thread = current_time % thread_list size;
     WHILE commit bandwidth is still available and committed instruction limit is not met
          IF no threads are left from thread_list
          .    BREAK out of WHILE loop
          IF current thread has no ROB entries
          .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          ROB_Entry = current thread's ROB head entry
          IF ROB_Entry NOT completed (not ready to commit)
          .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          IF ROB_Entry is a store or load instruction
          .    //NOTE: ROB and LSQ are in sync, both head entries, if not, stop simulation
          .    IF load/store instuction is NOT completed on LSQ-side (not ready to commit)
          .    .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          .    IF write buffer is full
          .    .    Remove any/all entries in write buffer that have already been committed/past due
          .    IF Functional Unit AND write buffer are both available
          .    .    IF Data Level 1 (DL1) cache is available
          .    .    .    Commit and write store value to DL1-cache
          .    .    .    Determine if cache hit or miss, Record Timestamp
          .    .    IF Data-Translation Lookaside Buffer(D-TLB) is available
          .    .    .    Check and Read Load value in D-TLB
          .    .    .    Determine if lookup hit or miss, Record Timestamp
          .    .    Insert a write buffer entry with recorded timestamp
          .    ELSE IF Write buffer is still full and/or Functional Unit NOT available
          .    .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          .    .    .    .
          <COMMIT THE INSTRUCTION>
}
```

Fig. 8. Pseudocode for unmodified commit stage.

$$\left.\begin{array}{ll} (3) & Cache\ Hit\ Priority : WB_{current} \geq WB_{limit} - M \\ (4) & Cache\ Hit\ Only : WB_{current} \geq WB_{limit} - N \end{array}\right\} \quad where\ M \geq N$$

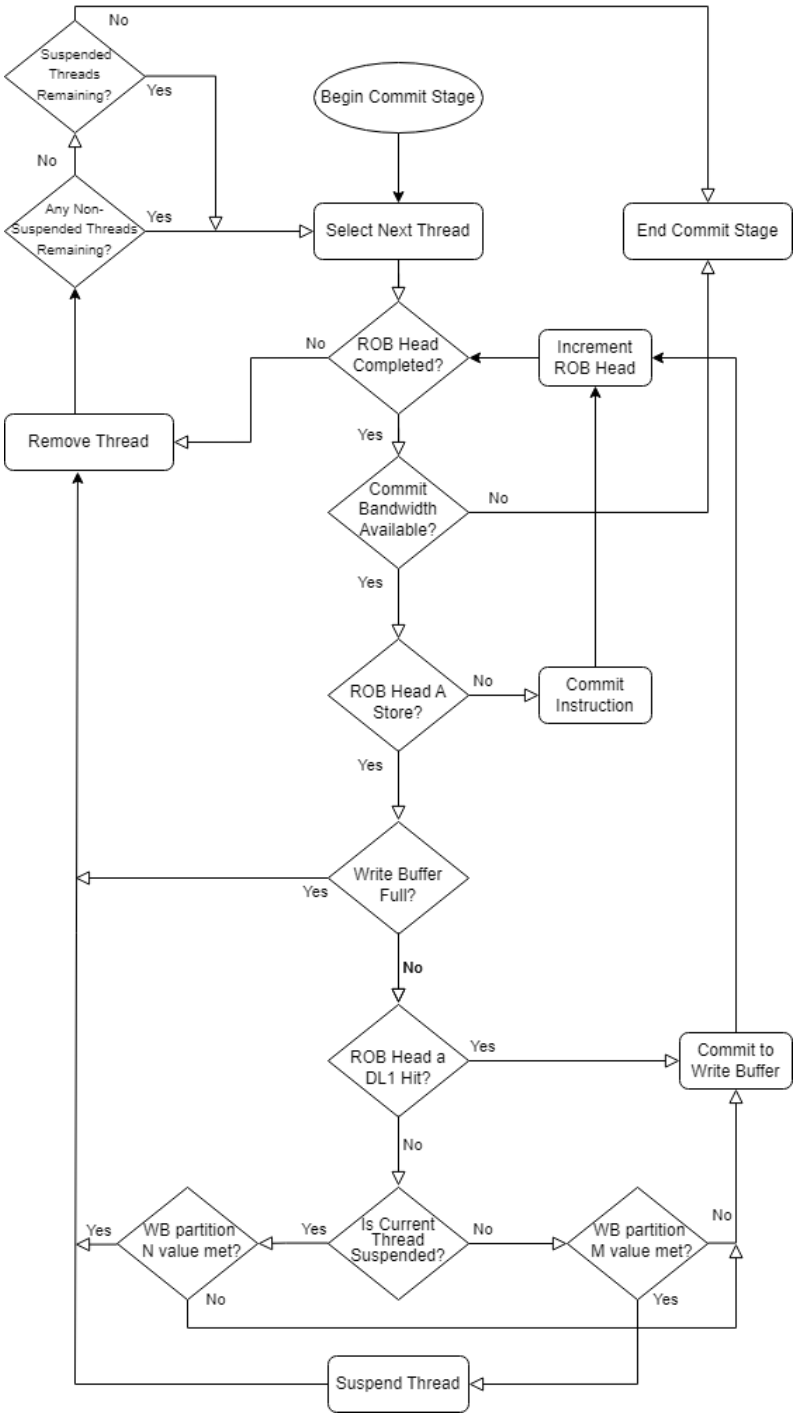

Fig. 9. Modified round robin commit stage algorithm with write buffer partitioning.

```
function commit()   //round robin
{
     Collect current thread_list
     *Create new suspended_thread_list*
     Check if any threads are stalled or if no instructions have committed for a while
     current thread = current_time % thread_list size;
     WHILE commit bandwidth is still available and committed instruction limit is not met
          IF no threads are left from thread_list
               *IF suspended_thread_list is also empty*
          .         BREAK out of WHILE loop, no further instructions committed
               *ELSE*
                    *thread_list = suspended_thread_list*
                    *clear/empty suspended_thread_list*
          IF current thread has no ROB entries
          .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          ROB_Entry = current thread's ROB head entry
          IF ROB_Entry NOT completed (not ready to commit)
          .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          IF ROB_Entry is a store or load instruction
          .    //NOTE: ROB and LSQ are in sync, both head entries, if not, stop simulation
          .    IF load/store instuction is NOT completed on LSQ-side (not ready to commit)
          .    .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          .    IF write buffer is full
          .    .    Remove any/all entries in write buffer that have already been committed/past due
          .    IF Functional Unit AND write buffer are both available
          .    .    *IF ROB_Entry's LSQ equivalent is NOT in Data Level 1 (DL1) cache AND current thread has NOT been given second chance*
          .    .    .    *IF Partition Value 'M' has not been met*
          .    .    .         *FLAG that current thread has now been given second chance*
          .    .    .         *ADD current thread to suspended_thread_list*
          .    .    .         *REMOVE current thread from thread_list, SELECT next thread and start from top of WHILE loop*
          .    .    *ELSE IF ROB_Entry's LSQ equivalent is NOT in Data Level 1 (DL1) cache AND current thread has been given second chance*
          .    .    .    *IF Partition Value 'N' has not been met*
          .    .    .         *UN-FLAG that current thread has now been given of second chance*
          .    .    .         *REMOVE current thread from thread_list, SELECT next thread and start from top of WHILE loop*
          .    .    IF Data Level 1 (DL1) cache is available
          .    .    .    Commit and write store value to DL1-cache, Determine if cache hit or miss, Record Timestamp
          .    .    IF Data-Translation Lookaside Buffer(D-TLB) is available
          .    .    .    Check and Read Load value in D-TLB, Determine if lookup hit or miss, Record Timestamp
          .    .    Insert a write buffer entry with recorded timestamp
          .    ELSE IF Write buffer is still full and/or Functional Unit NOT available
          .    .    REMOVE thread from thread_list, SELECT next thread and start from top of WHILE loop
          .    .    .    .
          <COMMIT THE INSTRUCTION>
}
```

Fig. 10. Modified round robin commit stage algorithm with write buffer partitioning

Initially, if the write buffer encounters a non-cache hitting thread when the cache hit priority condition is met, meaning the current size of the write buffer is greater or equal to the maximum size a write buffer can be minus value of M, the thread in question starts a process. Rather than completely disregarding the thread in a default commit stage, a thread is first suspended to a different pool of suspended threads and removed from the original pool of non-suspended threads. Suspended threads are still accounted for but are held at a lower priority than non- suspended (untouched or unvisited) threads. To ensure every thread is considered, a suspended thread, as shown in the Figure 9, is only considered when either all non-suspended threads have been suspended, or there are no more non-suspended threads to choose from. Although this M partition gives priority to thread with ROB head as cache hit, if no cache-hit thread is available or all have already been committed, suspended threads have another opportunity to be considered. Assuming bandwidth is available and all threads have been accounted for and suspended, the pool of suspended threads can now re-enter the commit stage at with the condition that they now follow a different path. Since they are marked as suspended and have been filtered before, they now follow the cache hit only partition that is dependent on the value N. This partition is comparable to [15], in which this section forces the write buffer to only accept cache-hit threads if capacity and condition is met. If the N partition however has not been met but a thread has is currently suspended and filtered by the M partition, the thread is committed regardless as to not waste commit bandwidth on enforcing a cache-hit that is not available at the current clock cycle. This methodology in turn prioritizes threads with cache hit but also allows threads with no cache-hit at a given clock cycle and second opportunity to be able to commit the store instruction. As explained, the only way a thread is suspended is when it encounters the write buffer at M partition value. Therefore, the only viable values for M and N are that M must always be greater than N. In other words, a thread must

first be given a second chance in order to be considered in the second phase. The difference between M and N is a inevitable buffer on how many threads will be given a second opportunity.

The focus is to determine the maximal number of rename registers that can be issued to each thread, paired with the combination of M and N partitioning values that can maximize IPC. The process to maximal M and N values involves simulating the fixed capped system and extrapolating every possible M and N value. With the established cap value of 9 for 4-threaded workloads to combine this algorithm with, Figure 11-14 plot all the possible combinations of partition M and N values as the x and y axis, and the average increase percentage as z-axis.

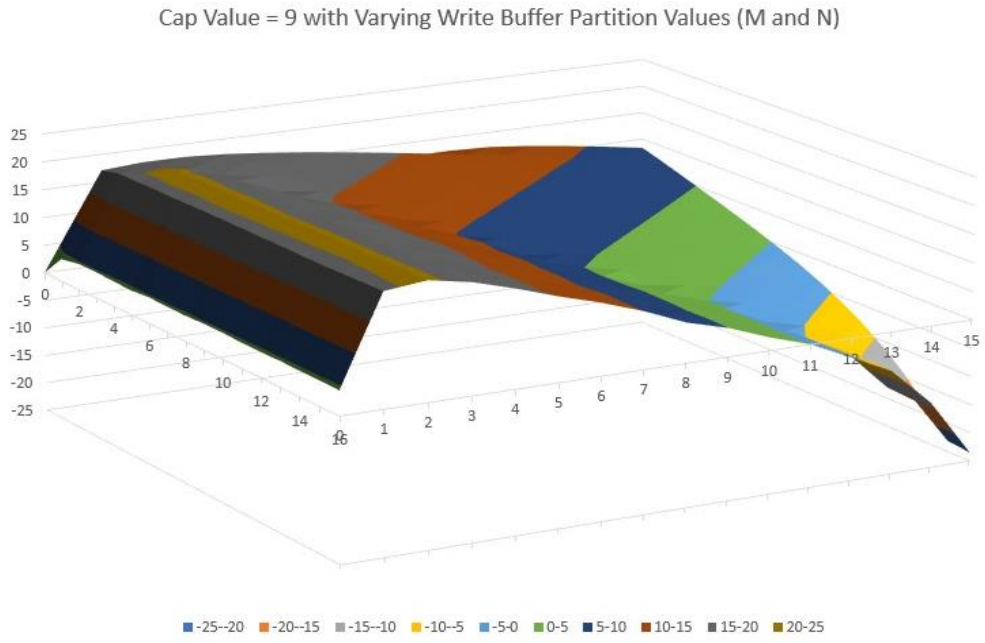

Fig. 11. 3D Representation of the varying partition values M and N when cap = 9

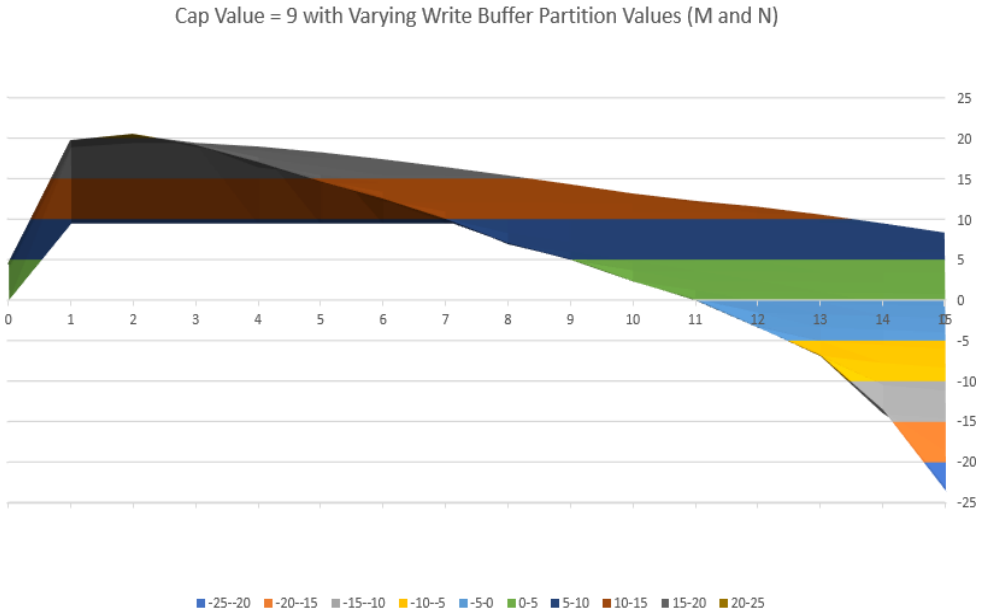

Fig. 12. Cache-hit only partition N as x-axis, & percentage improvement as y-axis.

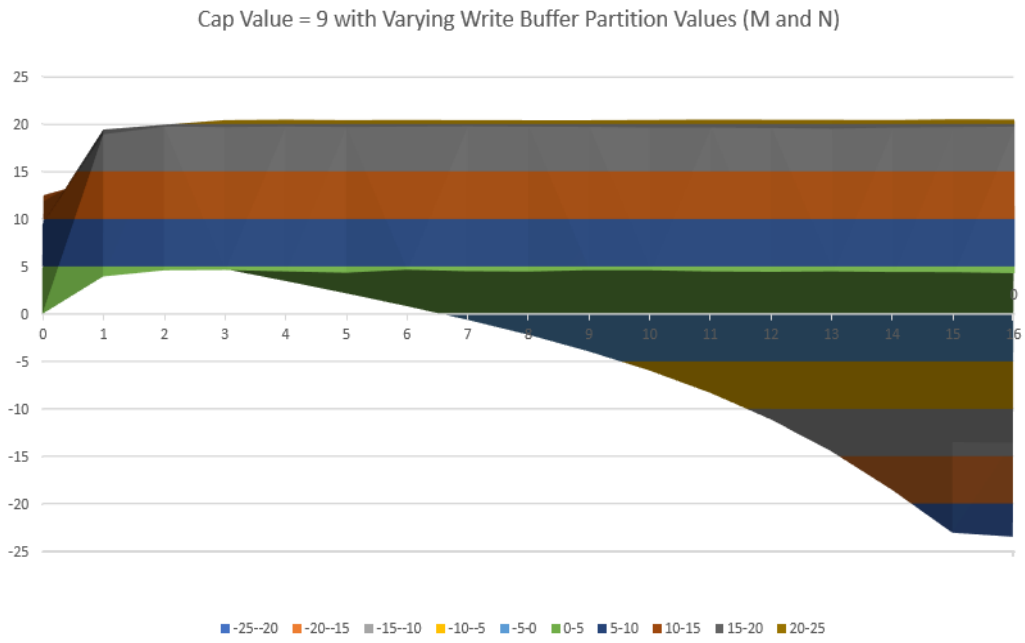

Fig. 13. Cache-hit priority partition M in the x-axis, & percentage improvement as y-axis.

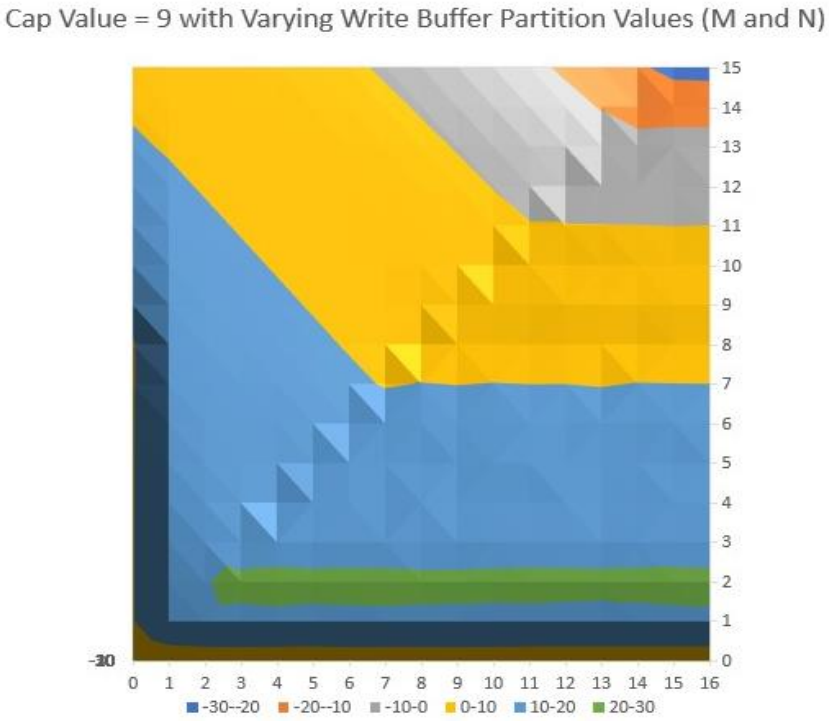

Fig. 14. Cache-hit priority partition M as x-axis, & cache-hit only partition N as y-axis

What can be inferred for the 3-D surface graphs (Figure 11-14) is that the usual peak at to get the highest percentage is when the cache-hit only partition value N reaches 2 with having the cache-hit priority partition M value range between 3 and 16. According to the actual raw data, the actual

combination that gave the highestaverage IPC value was M = 15 and N = 2. In other words, in a 4-threaded SMT system with a 16-entry write buffer, the highest performance increase when implementing the modified commit and write buffer partitioned algorithm with a static when 15 out of the 16 entries of the write buffer are prioritized for cache-hit threads (not enforced), and 2 entries are reserved only for cache-hit threads (enforced).

7. SIMULATION RESULTS

The proposed algorithm will be tested with the following different parameters of an SMT system. As mentioned before, the register file size will remain at 160 registers for 4-threaded workloads, and 320 for 8-threaded workloads, keeping the same ratio of number of registers per thread.

The same guided process of choosing a cap value for the system and finding M and N partition values in the write buffer will be done for each system configuration. The following results will be listed showing the following data, in order. First, a per-mix IPC change percentage between a capping plus write buffer algorithm, and mixes with a capping adjustment only (no algorithm). Second, an average IPC change percentage, with a default system with no capping (cap = 32) as baseline, for varying cap values applying the best combination of M and N in the write buffer partitioning algorithm (tested with the best cap value).

7.1. 4-Threaded Workloads | 16 Write Buffer Entry

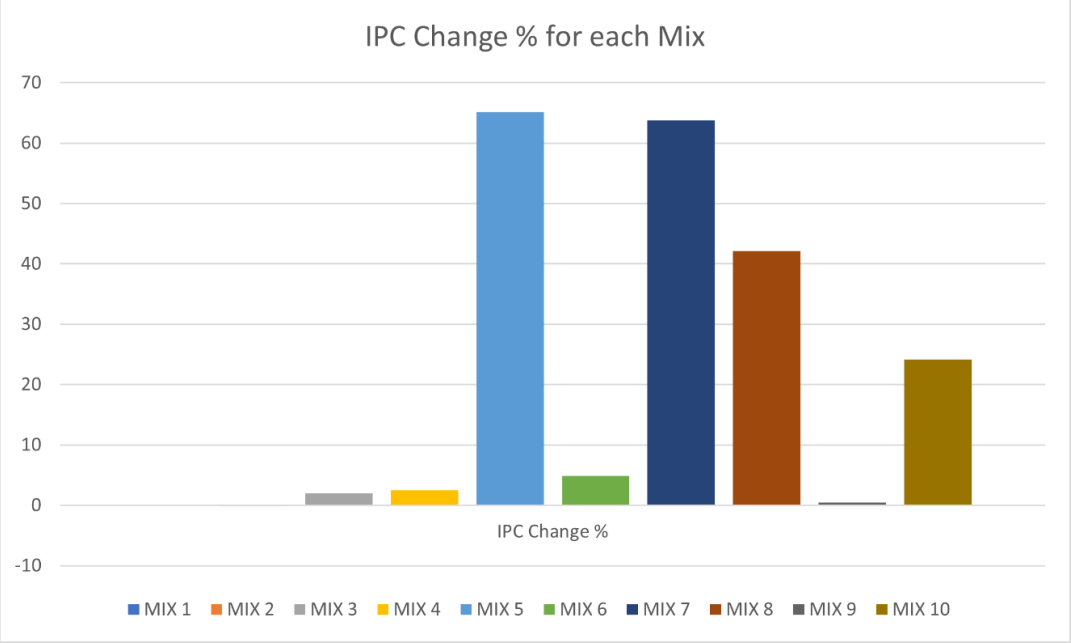

Fig. 15. Per-mix IPC change with capping = 9, M=15, and N=2, compared to just capping = 9.

As guided, the best cap value that was tested to bring the highest IPC average was 9, and the best partition values were M=15 and N=2. The average IPC (improvement) change for the above figure was 20.5% when in contrast to a just capping system with value 9. In Figure 15, visual analysis

tells the specific mixes that had the most change, namely 5, 7, 8, and 10. These mixes are also depicted in Figure 3 and 4 to have the most "shift" on the number of different threads ready to commit which corresponds to our initial inference.

With Figure 16, as the cap value nears a default state, the write buffer algorithm does not perform very well and does not produce an additional improvement. This is entirely because as the cap value digresses from the optimal value (cap = 9), the variety of threads and ROB size decrease. Since the source of the algorithm suffers depletion, so does the algorithm itself. Together with capping and the proposed algorithm, there was a peak 70.6% increase compared to default with no modification.

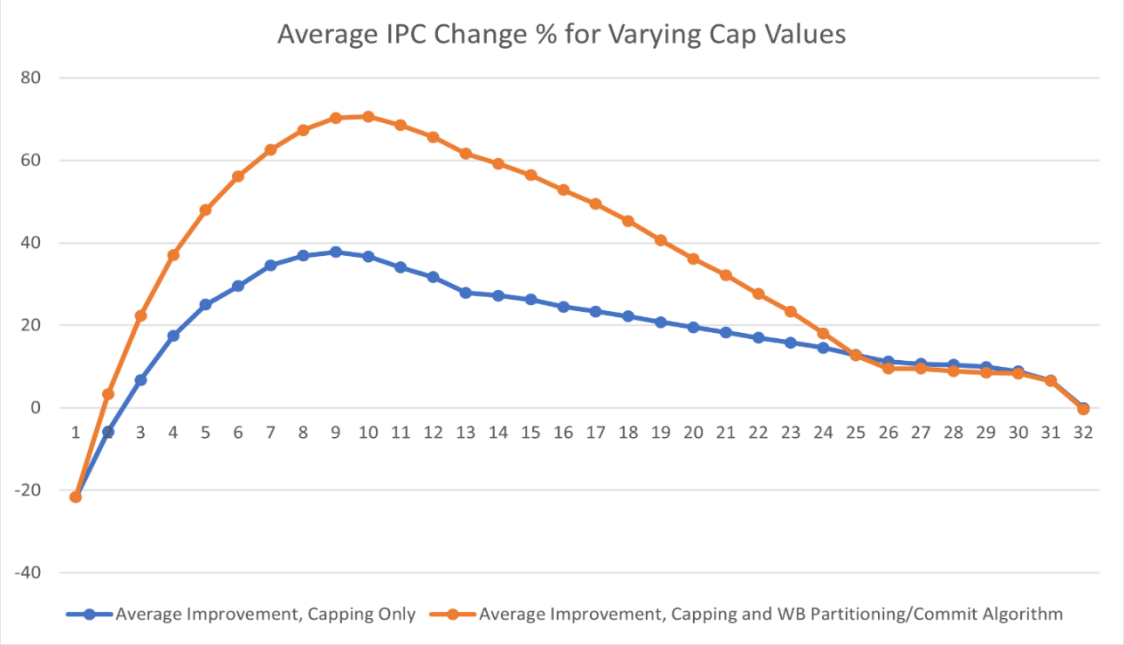

Fig. 16. Both capping only and capping with algorithm compared to default system

7.2. 4-Threaded Workloads | 32 Write Buffer Entry

The best cap value tested to bring the highest IPC average was 9. After, it was tested that the best values of M and N were 30 and 2 respectively. The average IPC improvement for the above figure was 7.25% compared to just capping with value 9. In Figure 17, the same specific mixes that had the most change (5, 7, 8, and 10), also showing around the same ratios as before. However, compared to its 16-entry counterpart with the same 4-threaded workload, the percentage for each significant thread has decreased significantly, by a bouthalf.

As reflected from the per-mix analysis previously, Figure 18 shows how much of a difference in the size limitof the write buffer can affect the algorithm results. With double the write buffer entries with same 4 threaded workload, the algorithm gains little throughout the varying cap values, but barely significant enough in certain cap range.

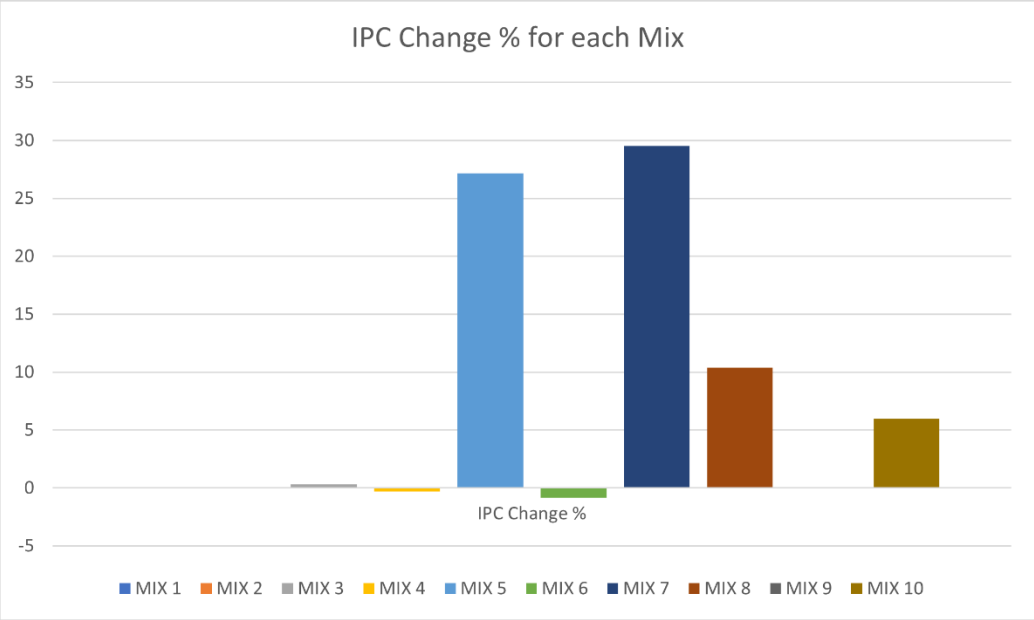

Fig. 17. Per-mix IPC change with capping = 9, M=30, and N=2, compared to just capping = 9.

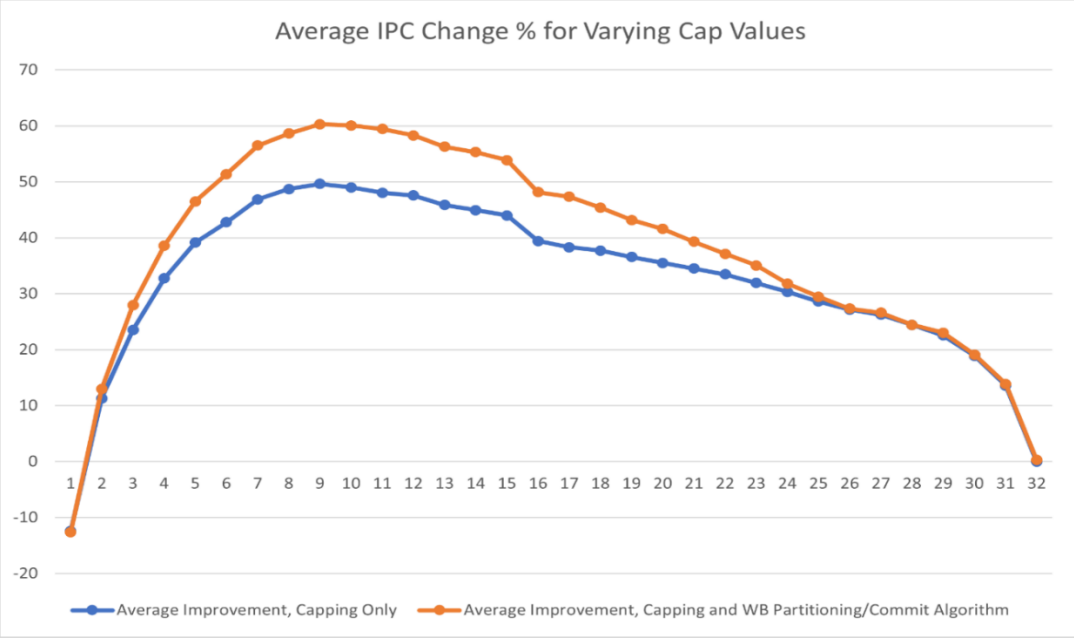

Fig. 18. Both capping only and capping with algorithm compared to default system.

7.3. 8-Threaded Workloads | 16 Write BufferEntry

The best cap value tested to bring the highest IPC average was 9. After, it was tested that the best values of M and N were 16 and 2. In Figure 19, there is a significant increase in IPC percentage among all Mixes. In the previous scenarios for a 4-threaded system, only 3 Mixes were able to exceed a minimum percentage increase of 40%, but in this result, there is nearly 50 percent of all mixes reaching or exceeding 40%. The average IPC improvement for this section was 35.8% which is a huge bump in gain so far.

Figure 20 shows the biggest margin of difference between only capping and capping plus algorithm. Although it is the biggest increase due to the proposed algorithm being implemented, it does not hold to be the highest increase percentage with capping and write buffer algorithm combined when compared to default c= 32.

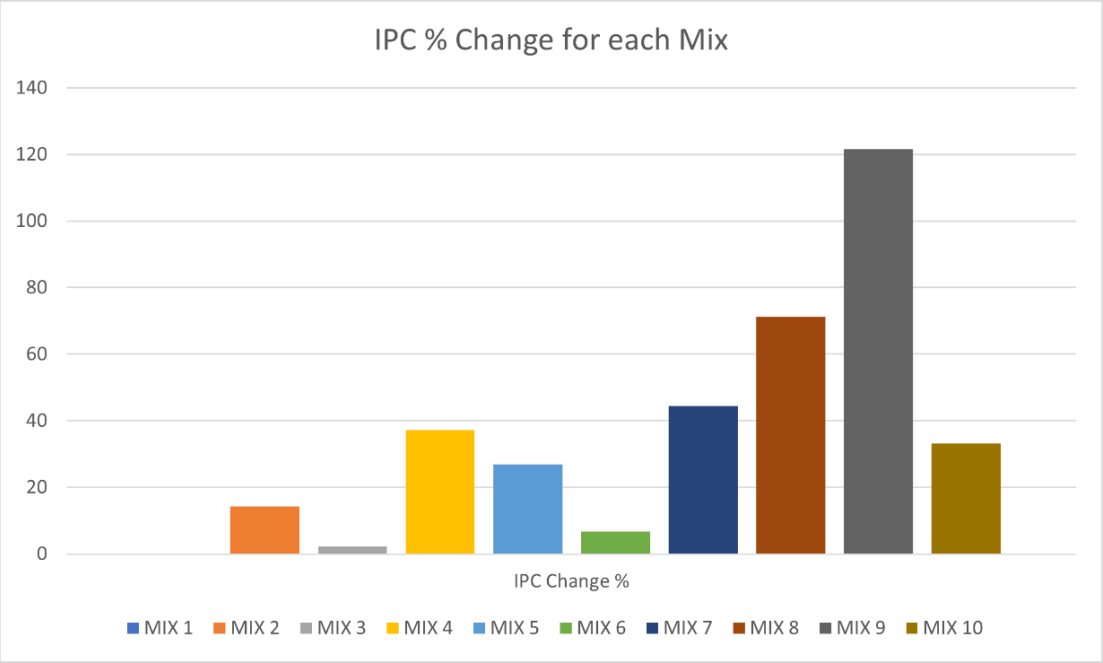

Fig. 19. Per-mix IPC change with cap=9, M=16, and N=2, compared to just capping c=9.

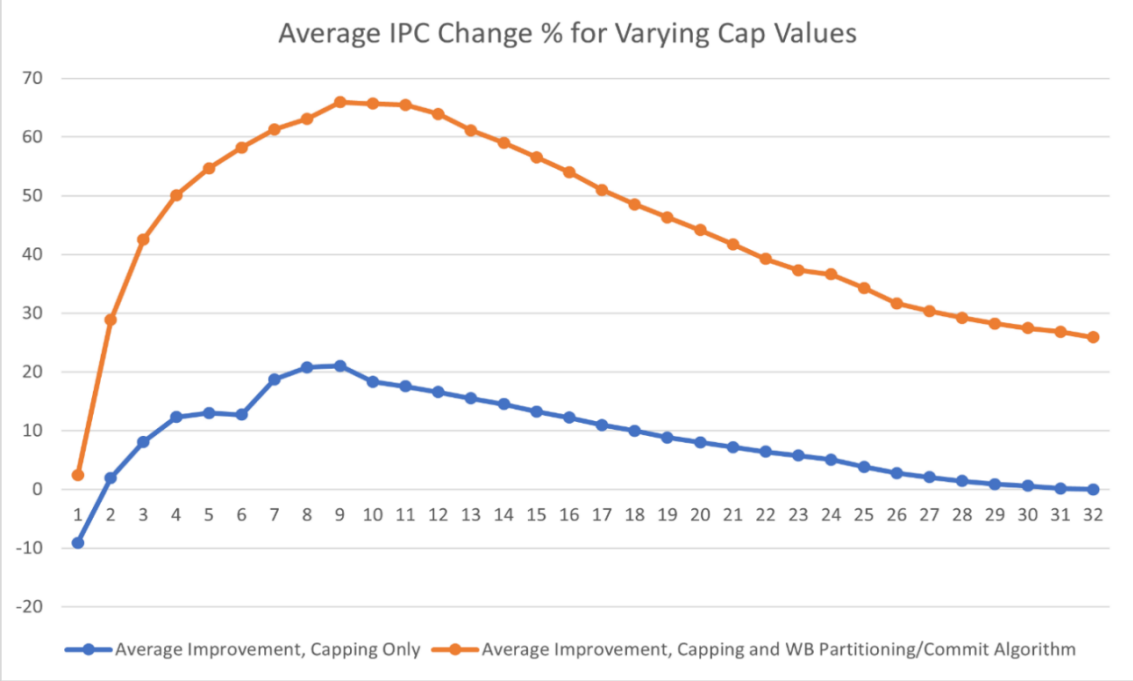

Fig. 20. Both capping only and capping with algorithm compared to default system

7.4. 8-Threaded Workloads | 32 Write Buffer Entry

The best cap value tested to for the highest IPC average was again, 9. It was then tested for the best values of M and N, which was found to be 32 and 2.

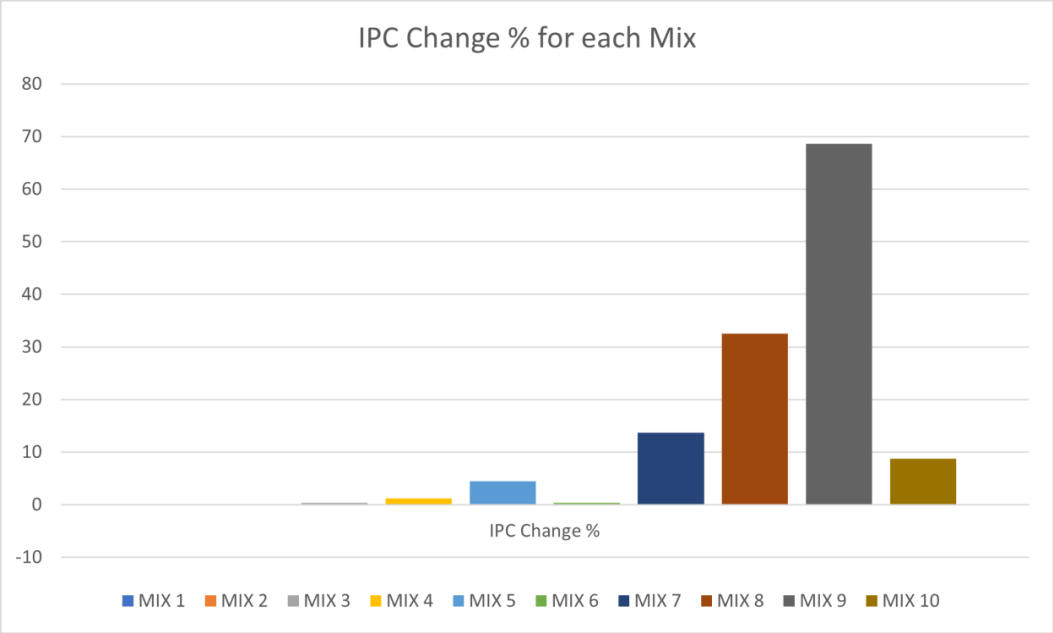

Fig. 21. Per-mix IPC change with cap=9, M=32, and N=2, compared to just capping c=9.

In Figure 21, the average IPC improvement above was 13.02%. Figure 22 shows that even though ourpreliminary testing on deciding which cap value is best, when applying M = 32 and N = 2, even though the values were tailored for a cap value of 9, there is a higher IPC percentage gain on cap value number 10, having 28.2% and 28.35% for 9 and 10, respectively.

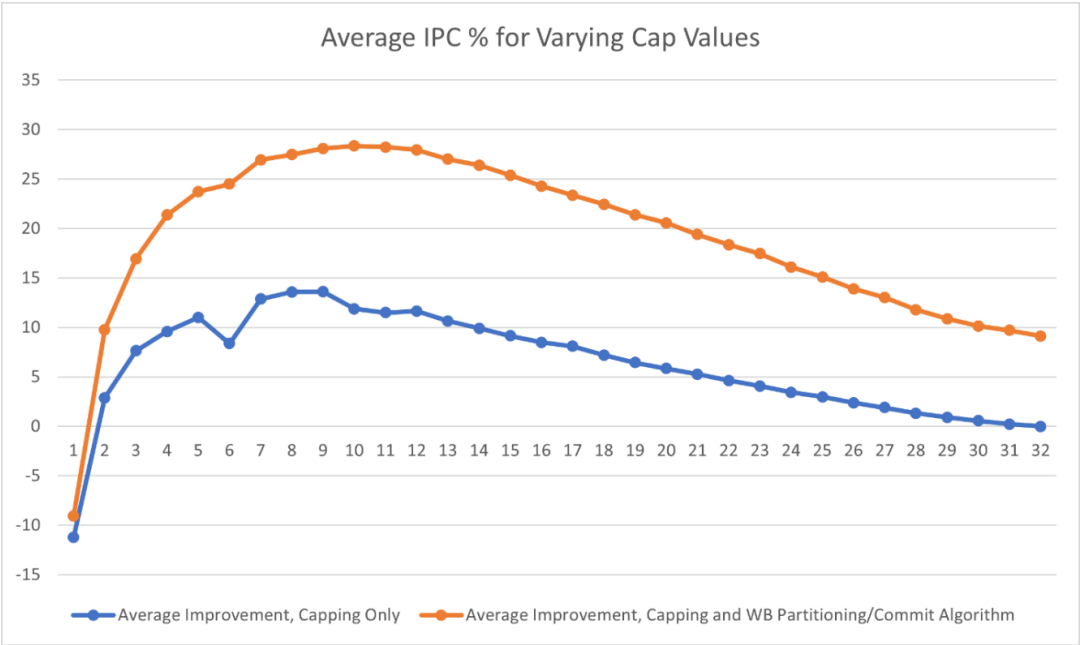

Fig. 22. Both capping only and capping with algorithm compared to default system.

7.5. Results Summary

Table 4. Highest peak performance compared to default for each system configuration.

Workload 1,2,3,4 (in order)	Write Buffer	Capping Value	M (cache-hit priority)	N (cache-hit only)	Average IPC-Improvement %	
					From Default	From Cap = 9
4-threaded	16 entry	9	15	2	70.6%	20.49%
4-threaded	32 entry	9	30	2	60.3%	7.75%
8-threaded	16 entry	9	16	2	65.93%	35.8%
8-threaded	32 entry	10	32	2	28.35%	13.02%

In Table 4, the different results for each configured system are shown. In workload 1, comparison from default shows the highest overall IPC improvement percentage with 70.6%. This includes both capping and write buffer/commit modifications combined and working concurrently. However, when showing the marginal difference from just capping, there is a decent 20.49%. This infers that in this environment, the capping algorithm does most of the improvement and the proposed algorithm adds a decent improvement as well. In workload 2, overall IPC improvement percentage was 60.3%, but only 7.75% with capping as baseline. This marginal increase is the lowest in the bunch in terms of just capping comparison. In workload 3, the overall IPC improvement percentage with 65.93%, and a healthy and highest marginal difference compared to cap only of 35.8% This infers that in this environment, the capping algorithm, and the proposed algorithm work cohesively and efficiently together and both add around the same effort and gain. In workload 4, IPC improvement percentage with 28.35% and marginal difference from just capping at 13.02%. Although the worst in overall gain from default, the marginal percent difference is not the lowest.

8. CONCLUSION

In simultaneous multithreaded systems, there are several pipeline resources shared among multiple threads con- currently. If these shared resources are not configured correctly, they can serve as a bottleneck in inefficientresource usage and generate undesirable performance. As shown in this paper, there is the potential to alleviate common performance congestion by configuring the mutual resources, register file and write buffer, concurrently.

This paper has shown that when implementing a static register capping algorithm, that is limiting the number of per-thread physical register entries, a byproduct of increased variety in the source to the write buffer. This allowed for an algorithm for the write buffer to have a higher variety to potentially choose a more suitable thread as it's source at certain clock cycles. Partitioning the write buffer into two sections; cache-hit priority and cache-hit only, has shown that system performance can be further improved by using this technique. Essentially having all of the write buffer entries to prioritize cache hit and leaving one or two entries for cache-hits only resulted in the best improvement in a capped system. The one drawback of this proposal, however, is that depending on the parameters of the system, the algorithm may be limited to improvement. In comparison to a default SMT system, peak overall IPC was increased by nearly 71% with this proposed method.

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," the Proceedings of the 19th Annual International Symposiumon Computer Architecture, 1992.

[2] D. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," the Proceedings of the 22nd Annual International Symposium on Computer Arch., 1995.

[3] M. Sheikh and W.-M Lin, "Dynamic capping of rename registers for SMT processors," Journal of Systems Architecture, vol. 99, 2019

[4] Hily, Sébastien and André Seznec. "Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading." (1997).

[5] A.Sahba, Y. Zhang, M. Hays, and W. Lin, "A Real-Time Per-Thread IQ Capping Technique for Simultaneous Multi-threading (SMT) Processors," in 11thInternational Conference on Information Technology: New Generations, 2014,pp.413–418.

[6] 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," the Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 191–202,1996.

[7] Y. Zhang and W. M. Lin, "Write buffer sharing control in SMT processors," in PDPTA'13: The 19th International Conference on Parallel and Distributed Processing Techniques and Applications,2013.

[8] S. Lawal, Y. Zhang, and W. M. Lin, "Prioritizing write buffer occupancy in simultaneous multi-threading processors," J. Emerg. Trends Computer. Inf. Sci, vol.6, no. 10, pp.515–522,2015.

[9] Gabor, R., Weiss, S., & Mendelson, A. "Fairness enforcement in switch on event multithreading." ACM Transactions on Architecture and Code Optimization. Association for Computing Machinery (ACM), vol. 4, no. 3, 2007

[10] Sleiman, Faissal M., & Wenisch, T. F. "Efficiently Scaling Out-of-Order Cores for Simultaneous Multithreading." ACM SIGARCH Computer Architecture News, vol. 44, no. 3, Association for Computing Machinery (ACM), June 2016

[11] J. J. Sharkey and D. V. Ponomarev, "Exploiting Operand Availability for Efficient Simultaneous Multithreading," in IEEE Transactions on Computers, vol. 56, no. 2, Feb. 2007

[12] Kihm, J.L., Janiszewski, A., & Connors, D.A. "Predictable Fine-Grained Cache Behavior for Enhanced Simultaneous Multithreading (SMT) Scheduling." International Conference on Communication Control and Computing Technologies (2004).

[13] Y. Zhang and W. M. Lin, "Efficient physical register file allocation in simultaneous multi-threading CPUs," in 33rd IEEE International PerformanceComputingandCommunicationsConference,2007.

[14] Y. Zhang and W.-M Lin, "Efficient resource sharing algorithm for physical register file in simultaneous multi-threading processors," Microprocess and Microsystems, 2016.

[15] S. Carroll and W. M. Lin, "Latency-aware write buffer resource control in multi-threaded cores," Int. J. DistributedParallel Syst. (IJDPS), vol. 1, pp. 7–7, 2016

AUTHORS

Allan Diaz received his B.S. and M.S. degrees in Computer Engineering from the University of Texas at San Antonio. He studied integrated design, and his research interests include computer architecture, parallel and distributed computing, artificial intelligence, and machine learning.

Wei-Ming Lin received the Ph.D. degree in Electrical Engineering from the University of Southern California in 1991. He is a professor of Electrical Engineering, and also the Associate Dean for Graduate Studies of the College of engineering in the University of Texas at San Antonio. His research interests include distributed and parallel computing, computer architecture, computer networks and internet security.