International Journal of Computer Systems (ISSN: 2394-1065), Volume 02, Issue 9, September, 2015
Available at http://www.ijcsonline.com/

Dynamic Issue Queue Allocation Based on Reorder Buffer Instruction Status

Shane Carroll and Wei-Ming Lin

Department of Electrical and Computer Engineering, The University of Texas at San Antonio,
1 UTSA Circle, San Antonio, Texas 78249, United States

Abstract

In a simultaneous multithreaded system, a shared instruction issue queue is a common pipeline bottleneck. If left unchecked, it is not uncommon for some threads to occupy a disproportionately large number of issue queue slots, which has a negative impact on overall performance of the system as measured by the number of completed instructions per clock cycle [1]. Previous research has shown that imposing a permanent limit, or cap, on the number of instructions an individual thread may have in the instruction queue at any time can increase performance of a system [1]. This paper proposes a method that, in addition to a permanent issue queue cap, seeks to identify threads whose performance does not justify an equal number of issue queue entries and dynamically give these threads a lower cap in order to increase system performance beyond the benefit of a permanent cap.

I. Introduction

Simultaneous Multithreading (SMT) is a processor technique which exploits thread-level parallelism (TLP) by executing instructions from multiple threads concurrently [5]. Whereas a coarse- or fine-grained multithreaded system switches among individual contexts in sessions or in clock cycles, SMT architecture allows multiple threads to occupy the pipeline during the same clock cycle. An SMT system resembles that of a coarse- or fine-grained multithreaded processor with the exception of some shared resources for the purpose of concurrent processing which are occupied by multiple threads at the same time.

With the ability to choose from any of multiple threads in each clock cycle, short-latency hazards (e.g., unresolved input dependencies) which typically stall the pipeline may easily be masked by executing instructions from a different thread [6][7]. 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 ILP since TLP can be used to fill the delay slots with instructions from other threads [8].

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 (FU) 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.

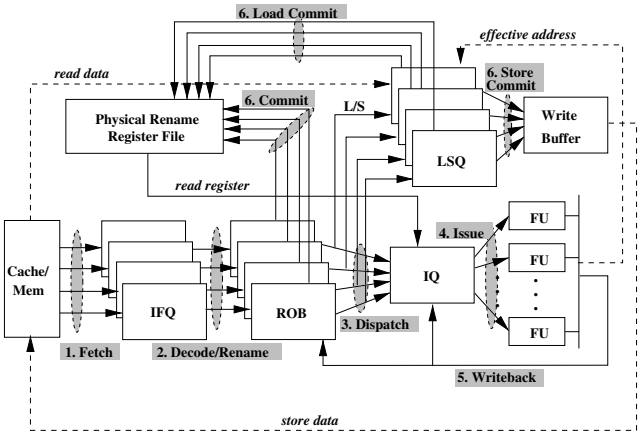

Fig. 1. Pipeline Stages in a 4-Threaded SMT System

By exploiting TLP and ILP simultaneously, SMT systems have been shown to have better performance than other architectures [7]. However, sharing individual resources among multiple threads concurrently inevitably leads to bottlenecks and starvation [9]. One shared resource is the IQ, which serves as a collection of buffers for the functional units that ultimate execute on the operands of each instruction. This shared resource has been shown to be prone to overconsumption by some threads, leading to starvation in other threads and degraded overall system performance [1]. This paper proposes a dynamic technique to reduce overconsumption of the IQ to improve overall system performance in an SMT system.

II. Previous Work

Several techniques have been proposed to limit bottlenecks and resource starvation in SMT systems. One technique is thread co-scheduling, which aims to group concurrent threads in such a way that minimizes conflicts between threads [10][11]. For example, pairing a thread which heavily uses integer computations with a thread that primarily uses floating-point computations may produce less resource conflict and starvation than a pair of threads which both use floating-point computations. With this technique, threads are effectively partitioned into sets whose threads minimally interfere with each other.

Other methods include resource control at various stages of the pipeline. One method of resource control is to limit the number of IQ entries a thread may occupy but not necessarily partitioning the IQ into parallel hardware resources [1]. In other words, given an IQ with N entries and M concurrent threads, an individual thread may occupy a maximum of k IQ slots, and it may be such that $k > N/M$. With this method, each thread's IQ entry allocation is limited, but still greater than if the IQ were partitioned.

Another resource control method is to prioritize which thread fetches next, rather than rotate in a fashion that is assumed to be fair [12][13]. In an SMT system, one common technique of execution fairness is the round-robin approach, which chooses a thread in a linear modular fashion when deciding which thread to execute in the current clock cycle. Rather than choose fairly, such as round robin, this resource control technique 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. Another technique chooses to monitor the usage of each thread's resources and dynamically limit the resources which belong to each thread at runtime [2]. In addition to limiting, if one thread is not currently using some resources, those resources may be reallocated to another thread. This technique both limits and optimizes the distribution of an SMT system's resources.

Other techniques adopt a less flexible resource-partitioning approach, including the technique in [4], which 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.

However, none of these techniques seek to dynamically limit IQ resources based on some measure of thread performance. This paper proposes a technique to control the IQ bottleneck to improve system performance.

III. Simulation Environment

A. 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 private IFQs, private ROBS, a shared IQ, and a shared pool of functional units (FUs). The parameters used throughout all simulations will be M-Sim's default configuration, which is outlined in Table I.

TABLE I. Simulation environment parameter setup

Parameter	Description
Bandwidth	8 Fetch / Dispatch / Issue / Writeback / Commit
IFQ/ROB/LSQ/IQ Size	32/128/48/32 Entries
Function Unit: Number of Units / Completion Latency / Issue Latency	Integer ALU: 4/1/1
	Integer Mult: 1/3/1
	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 64-byte Entries
Branch Predictor	2048-entry Bimodal
Physical Registers (4-thread, 8-thread)	256 / 512 Integer and Floating Point

B. 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 individual ILP and chosen in sets of 4 and 8 by varying ILP classes. Those with highest IPC will be rated as H, those with lowest IPC will be rated as L, and those between will be rated as M. Chosen workload combinations and their corresponding ILP classes are shown in Tables II and III for 4-threaded and 8-threaded workloads, respectively.

TABLE II. 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

TABLE III. 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

C. Performance 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 (sometimes referred to as Arithmetic IPC):

$$\text{Overall\_IPC} = \sum_{i}^{n} \text{IPC}_i \tag{1}$$

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,

$$\text{Harmonic\_IPC} = n / \sum_{i}^{n} \frac{1}{\text{IPC}_i} \tag{2}$$

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 (instead of the improvement percentage on IPC averaged over all mixes), which is applied to both Overall_IPC and Harmonic_IPC, namely,

$$\%\text{\_Improvement} = (\sum_{j}^{m} \frac{\text{IPC}_j^{\text{new}} - \text{IPC}_j^{\text{baseline}}}{\text{IPC}_j^{\text{baseline}}} \times 100\%)/m \tag{3}$$

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

IV. DYNAMIC ISSUE QUEUE RESOURCE ALLOCATION

A. Thread Classification

As discussed in Section II, there has been a proposed method to cap all thread's maximum IQ entry allocation at the same predetermined cap value [1]. This method is easy to implement and has been shown to improve system performance. However, treating all threads equally and without using any dynamic information may not be optimal. It is possible that using more than one cap value simultaneously, based on ranking threads by how efficiently they use the IQ, could improve system performance beyond the static approach. In this section, we define a metric for separating threads into classes and assigning each class a unique IQ cap.

B. Defining Inefficient Threads

To dynamically classify threads, we will use the status of their instructions currently residing in the ROB to extract information about the efficacy of that thread's IQ usage. To gauge a thread's ROB status, the number of instructions in each distinct state of an ROB entry will be tallied. An instruction in the ROB can take one of five general states:

1) **Awaiting dispatch**
   An instruction awaiting dispatch is ready to advance to the next pipeline stage as soon as bandwidth and buffers are available. An instruction with this status is dependent only on system resources for advancement through the pipeline.
2) **Dispatched with operands not ready**
   An instruction which is dispatched to the IQ with operands not ready is blocked from advancing within the pipeline by its input dependencies. An instruction with this status will occupy IQ resources with no chance to proceed to a functional unit until another instruction produces its input operands.
3) **Dispatched and ready for execution**
   An instruction dispatched to the IQ and ready for execution has no unresolved input dependencies. As soon as a functional unit and bandwidth is available, this type of instruction is free to advance to the next stage of the pipeline.
4) **Executing**
   An instruction which is executing (in a functional unit) will be released to the next stage of the pipeline after a predetermined number of clock cycles.

5) **Completed but not committed**
An instruction which is completed but remains in the ROB for more than one clock cycle could be in this state for multiple reasons. Two conditions must be satisfied in order for a completed instruction to advance in the pipeline (commit): the instruction must be at the head of the ROB/LSQ and there must be buffers and bandwidth available. The latter condition is a property of the system, while the former is a property of the thread. If an instruction spends multiple clock cycles completed but not yet at the head of the ROB, it could indicate a stoppage of the thread such as having an unresolved control instruction or other long-latency operation ahead in the ROB. This condition may result in inefficient pipeline resource usage by the offending thread.

With the above instruction-state categorization, states 2 (dispatched with operands not ready) and 5 (completed but not committed) may indicate that a thread is not efficiently using its pipeline resources efficiently. We refer to the instructions in one of these two states as "***blocked***" since they cannot advance in the pipeline due to thread-specific conditions.

C. Quantifying Issue Queue Use Efficiency

It is proposed that threads which exceed a number of blocked instructions should be given less IQ resources than other threads. To show that threads with many blocked instructions should be capped below other threads, it shall suffice to show that these threads are not using their IQ resources as efficiently as the other threads. For this purpose, we define a measure of IQ efficiency in terms of a thread's performance and IQ usage. This measure will be evaluated over some period of time, measured in clock cycles, which we define as a *window*. We define IQ efficiency as the ratio of IQ usage to IPC over some window. That is,

$$\text{IQ Efficiency} = \frac{\sum_{i=k}^{k+n-1} N_{C_i}}{\sum_{i=k}^{k+n-1} N_{IQ_i}} \tag{4}$$

where n is the window length in clock cycles, k is the starting clock cycle such that $k \mod n = 0$, N_{IQ_i} is the number of instructions in the IQ at clock cycle i and N_{C_i} is the number of instructions committed at clock cycle i.

In a pipelined system, such as SMT, an instruction must spend at least one clock cycle in each stage of the pipeline before advancing to the next. Thus, an instruction must spend a minimum of one clock cycle in each stage's hardware, including the IQ. However, there is no upper bound on the number of clock cycles an instruction may spend in a buffered stage's hardware such as the IQ. To advance from the IQ to a functional unit, two hardware conditions and one software condition must be met. To satisfy the hardware requirements, there must be an available, appropriate functional unit and there must be bandwidth between the IQ and the functional unit pool to transport the instruction. The software requirement for an instruction to advance from the IQ to a functional unit is that all input operands for that instruction are ready.

Since each instruction can commit at most once but must spend at least one clock cycle in the IQ, it should be expected that a thread's total number of IQ-usage clock cycles should be greater than the number of committed instructions from the same thread. Since both of these metrics are positive, we have that

$$0 \leq \sum_{i=k}^{k+n-1} N_{C_i} \leq \sum_{i=k}^{k+n-1} N_{IQ_i} \tag{5}$$

$$\Rightarrow 0 \leq \frac{\sum_{i=k}^{k+n-1} N_{C_i}}{\sum_{i=k}^{k+n-1} N_{IQ_i}} \leq 1 \tag{6}$$

$$\Rightarrow 0 \leq \text{IQ Efficiency} \leq 1 \tag{7}$$

For a thread to achieve an IQ Efficiency of 1, it must be the case that

$$\sum_{i=k}^{k+n-1} N_{C_i} = \sum_{i=k}^{k+n-1} N_{IQ_i} \tag{8}$$

In this case, each instruction must spend exactly one clock cycle in the IQ and eventually commit (not be squashed due to mis-speculation), which represents the most efficient use of IQ resources. On the other hand, for a thread to achieve an IQ Efficiency close to 0, it must be the case that

$$\sum_{i=k}^{k+n-1} N_{C_i} \ll \sum_{i=k}^{k+n-1} N_{IQ_i} \tag{9}$$

In this case, for each committed instruction there is a much greater average number of clock cycles of IQ use. This is the result of a thread's instructions spending too much time waiting in the IQ or too many squashed instructions due to mis-speculation.

Note that equations (5), (6), and (7) assume little to no overlap between windows of measurement. Given that some instructions that spent time in the IQ during the previous window but committed during the current window, there may be some variation to the implications of this formula based on the window length chosen, which can be considered minimal if the window length is sufficiently long.

D. Real-Time Criteria for Dynamic Capping

According to the ROB instruction-state categorization discussed in Section IV-B, threads will be separated into two classifications for applying IQ cap values: "fast" and "slow" threads. That is, a "slower" thread, which has a larger number of blocked instructions, will be more heavily capped than a "faster" thread which has a smaller number of blocked instructions.

In order to justify a more stringent cap on threads with many blocked instructions, a correlation between "high blocked instruction count" and "low IQ Efficiency" will be shown. This correlation is a suggestion that threads with many blocked instructions are not using their IQ resources efficiently and that it may help overall system performance by limiting these threads more than others. To measure IQ Efficiency as described in Equation 4, 10 4-thread workloads will be simulated to 10 million instructions. IQ Efficiencies will be measured over a window of 256 clock cycles as described in Equation 4, and the "blocked instruction count" is the average number of blocked instructions per clock cycle during the window.

Figures 2 and 3 show the results of measuring IQ Efficiency against blocked instruction count, with each curve from each of the 10 workloads. The horizontal axis displays the number of blocked instructions. The vertical axis shows the average IQ Efficiency for all data points collected at a particular number of blocked instructions. A minimum of 10 data points were required for each plotted point.

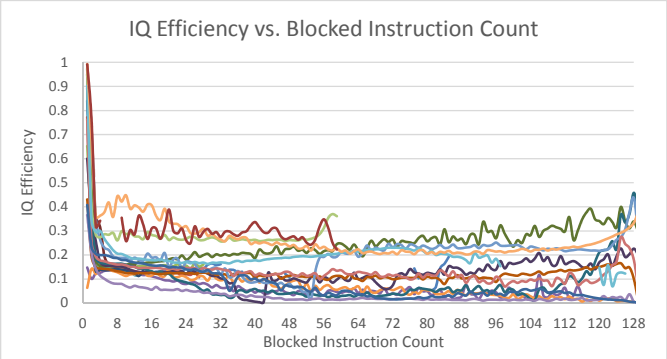

Fig. 2. IPC efficiency vs blocked instruction count, mixes 1-5

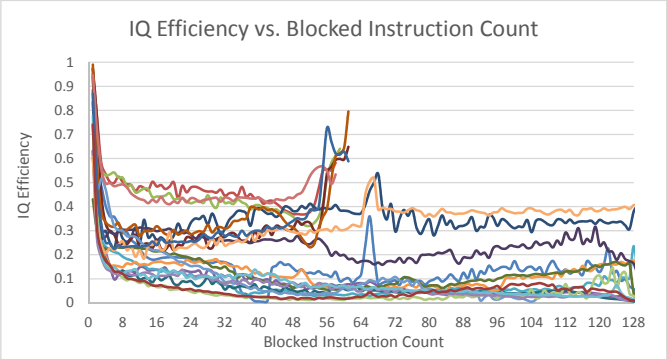

Fig. 3. IPC efficiency vs blocked instruction count, mixes 6-10

In these two figures, it is clearly shown that there is a general decline in IQ efficiency from 0 to 40 blocked instructions. At fewer than 40 blocked instructions, no thread shows near-zero IQ efficiency. Beyond this point, several threads' efficiency drops to near zero and does not recover. As threads approach this point, it may be beneficial to an SMT system to limit how many IQ entries these threads may occupy.

Figure 4 further shows the average distribution of blocked instruction counts for all threads of all mixes for the 20 workloads presented in Tables II and III. The number of

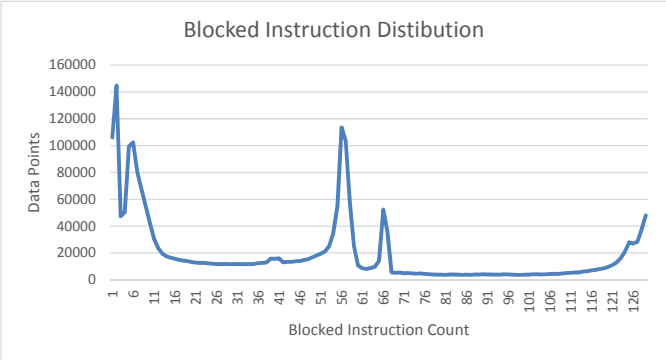

Fig. 4. Distibution of the number of blocked instructions per thread with a window size of 256 clock cycles over a simulation of 10 million instructions and a 128-entry ROB

data points collected during 10 million simulated instructions with a window size of 256 clock cycles is plotted on the vertical axis for each number of blocked instructions a thread may have. As the data show, a thread having fewer than 16 blocked instructions is very common. There is an almost even distribution between 16 and 50 blocked instructions. Sharp peaks occur at the points of 58 and 66 blocked instructions, which are caused by a few threads with repetitive behavior involving long-latency data-cache writes, the discussion of which is beyond the scope of this paper. Beyond these peaks, there are few data points until nearly 128 blocked instructions, where the points reflect a full or near-full ROB occupied by blocked instructions.

V. Proposed Algorithm

A. *Capping IQ-Inefficient Threads*

This algorithm proposes the use of two different IQ capacity values: a "static cap" and a "slow cap". The static cap is a permanent limit on how many instructions a thread may have in the IQ at any time regardless of that thread's dynamic status. This is synonymous to the static capping method as shown in previous research [1].

The slow cap is a limit on how many instructions a "slow" thread may have in the IQ at any time, as judged by the proposed algorithm. A slow cap is not intended to starve a slow thread in exchange for arithmetic IPC improvement. The placement of a slow cap should be the result of a thread which poorly utilizes its given IQ space. The slow cap is intended to best distribute the limited space in the IQ for the best, but still fair, system performance. In order for this algorithm to function well, a proper threshold value on the number of blocked instructions for a thread to be classified as "slow" needs to be predetermined.

B. Finding Optimal Parameters

There are three parameters that need to be optimized for the proposed algorithm: (1) the threshold on the number of blocked instructions for a thread to be classified as slow, (2) the slow cap value to be imposed on the slow thread(s), and (3) the static cap value for all others. Order of parameter optimization may significantly affect the overall effect of the algorithm. We choose the threshold to be the first to optimize due to its less significant influence on the overall outcome.

This will be accomplished by first choosing an arbitrary, but objectively small, slow cap. This cap will be chosen such that it is less than the size of the IQ divided by the number of threads being run. With this parameter fixed, the performance change of the workloads will be measured against the workload's base IPC for several tests, sweeping through various threshold values for an optimal selection. The threshold value will be swept across a range that exceeds the average number of blocked instructions as shown in the hypothesis. The threshold that produces the highest improvement in arithmetic IPC will then be used for further parameter optimization.

After an optimal threshold is found, the previously-arbitrary slow cap will be swept through in a similar manner. The optimal threshold will be used to decide which threads to apply the slow cap to during execution. At each value of the slow cap, the performance changes of each mix will again be measured against their base IPC. The value of the slow cap that produces the highest increase in arithmetic IPC will be used to find further parameters.

Once the threshold and slow cap are optimized, the final optimization step turns to the static cap value. Holding the threshold and slow cap at their respective optimal values found previously, the static cap will be swept through similar to the preceding steps. At the end of this sweep, all parameters should be optimal, though in the selected order of optimization.

VI. Simulation Results

A. Four-Threaded Workloads

1) Static Capping: Figure 5 shows the results of applying the static capping algorithm to the given 4-thread workloads in the given simulation environment. The optimal improvement using a static cap with these workloads is a 4.31% gain in arithmetic IPC and a 2.42% gain in harmonic IPC at a cap of 12.

2) Finding an Optimal Threshold: The first necessity of this algorithm is to find the number of blocked instructions at which to impose a cap on the offending thread. To do this, an arbitrary slow cap of 6 was chosen. With a 32-entry IQ and 4-thread mixes, equally dividing the IQ among the threads would result in 8 slots per thread. A cap of 6 will ensure that threads deemed slow will have a significantly smaller portion of the IQ than the other threads but should not be starved of resources, given that a simple cap of 6 results in positive gain (see Figure 5).

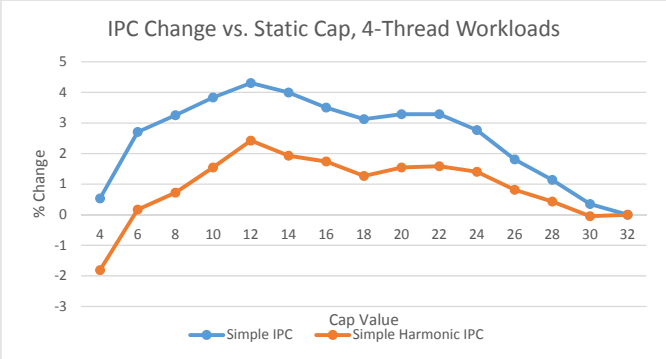

Fig. 5. % Change in IPC and Harmonic IPC using a static capping algorithm

No static cap will be used to find the optimal threshold since an optimal static cap with slow capping is not known at this point. Although the static capping algorithm shows an optimal value of 12 without any other cap, no assumptions can be made about the behavior of the system with a slow cap imposed. The arithmetic and harmonic IPC will be measured as an average of all 10 mixes at each increment of the threshold from 4 to 44. Figure 6 shows the results of the threshold sweep, indicating an optimal threshold value of 28.

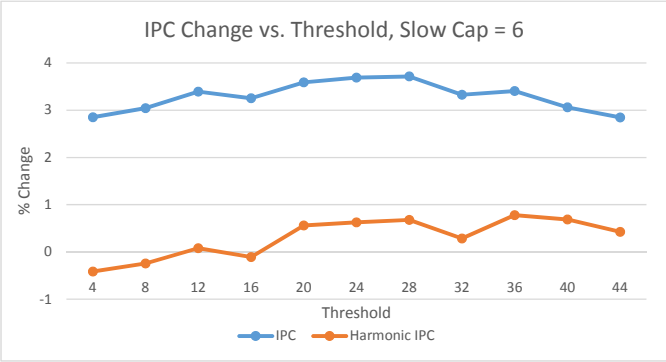

Fig. 6. % Change in IPC and Harmonic IPC vs threshold with a slow cap of 6

3) Finding an Optimal Slow Cap: With an optimal threshold found, the arbitrary slow cap can be removed and an optimal slow cap can now be sought. For this process, we hold the threshold at its optimal value, 28, and sweep through all the possible values of the slow cap. The slow cap will be swept from 2 to 9 in increments of 1. This range begins at a very stringent value and approaches the optimal value of a static cap with the same workload. The static cap will again be fixed at 32 since it is not yet optimized and no assumptions can be made on the static cap under these conditions. Again, the arithmetic and harmonic IPC values will be measured as an average of the 10 workloads. Figure 7 shows the results of the sweep of the slow cap. The optimum slow cap occurs at a value of 5 with an arithmetic IPC increase of 3.71% and a harmonic IPC increase of 0.68%.

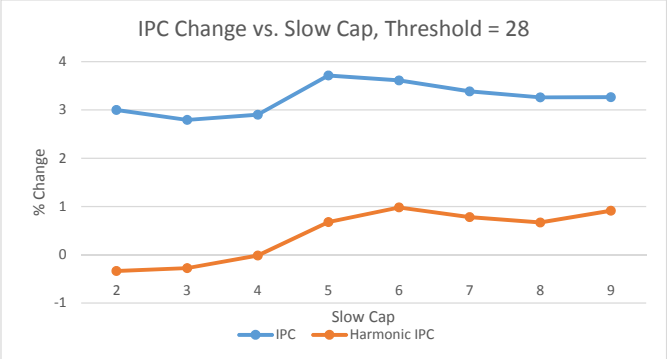

Fig. 7. % Change in IPC and Harmonic IPC vs slow cap with a threshold of 28

4) Finding an Optimal Static Cap: So far, the optimal values of the threshold and slow cap have been found to be 28 and 5, respectively. In addition to these parameters, the static cap will need to be optimized to cap every thread, not just the threads which are deemed slow. To find the optimal static cap, the threshold and slow cap will be fixed at their previously-found optimal values. The static cap will be swept from 6 to 32 in increments of two. At each increment, the arithmetic and harmonic IPC will be measured as averages of the 10 workloads as in the other parameter sweeps. Figure 8 shows the results of the static cap sweep with the optimized threshold and slow cap values. A maximum improvement of arithmetic IPC occurs with a static cap value of 20.

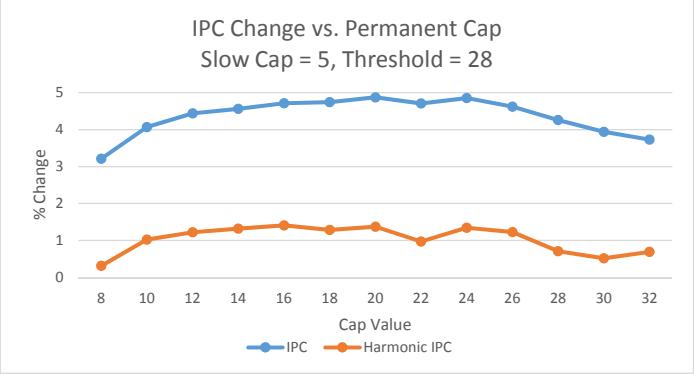

Fig. 8. % Change in IPC and Harmonic IPC vs static cap with a slow cap of 6 and threshold of 28

5) Comparison to Static Capping: With the threshold, slow cap, and static cap all optimized, the proposed algorithm can now be compared to the original static capping algorithm. The results will be compared at each value of the static cap measured within the measured ranges for each algorithm. Figure 9 shows the dynamic and static capping algorithms plotted against each other for 4-thread workloads. The optimal value for the original static capping algorithm occurs at 12 with an arithmetic IPC improvement of 4.3% and a harmonic IPC improvement of 2.42%. The optimal value of the dynamic algorithm occurs at 20 with an improvement of 4.87%. The harmonic IPC also peaks at this static cap value with an improvement of 1.37%.

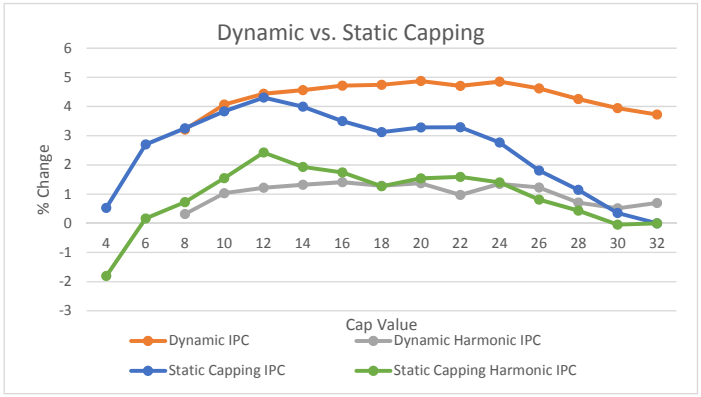

Fig. 9. A Comparison between static capping and dynamic capping

The proposed dynamic capping approach not only delivers a higher improvement than the static one, but it also can sustain the improvement even when the static cap value is raised higher. Performance improvement from the static approach quickly dips down to close to zero when the static cap is raised to above 28, while the dynamic one still maintains an improvement at above 4%. This scenario clearly demonstrates the intended ability of thread-based adjustment in the proposed dynamic technique.

B. Eight-Threaded Workloads

1) Static Capping: Figure 10 shows the results of 8-thread workloads with a static capping algorithm. The optimal improvement using a static cap with these workloads is a 3.15% gain in arithmetic IPC and a 1.25% gain in harmonic IPC at a cap value of 6.

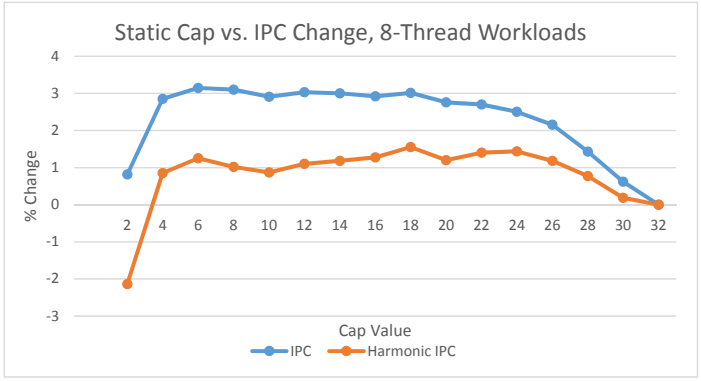

Fig. 10. % Change in IPC and Harmonic IPC vs threshold using a static capping algorithm

2) Finding an Optimal Threshold: As in the 4-threaded case, an arbitrary slow cap will be chosen in order to sweep through the range of reasonable thresholds to determine an optimal starting point. However, in addition to the slow cap, an arbitrary static cap will also be used for the 8-threaded case. With 8 threads and just 32 IQ entries, disproportionate IQ use and thread starvation is much more likely than the 4-thread case. When this occurs, the optimum threshold may be 0 for our arbitrarily-chosen slow cap, denying the ability to find optimum parameters. To ensure that an optimum threshold can be found, the 8-thread workloads will use a static cap large enough to provide sufficient IQ space but small enough to prevent these side effects.

Thus, arbitrary values of 3 and 16 for the slow and static caps, respectively, will be used to find an optimum threshold value. A slow cap of 3 ensures that a thread deemed slow receives less than an equal share of IQ space. The static cap value of 16 allows a thread to occupy up to half of the IQ, which will allow plenty of resources while not allowing any thread to hide an optimum threshold.

The result of the threshold sweep is shown in Figure 11. The optimum threshold value for the given cap values is 28, with an arithmetic IPC gain of 3.03% and a harmonic IPC gain of 1.3%.

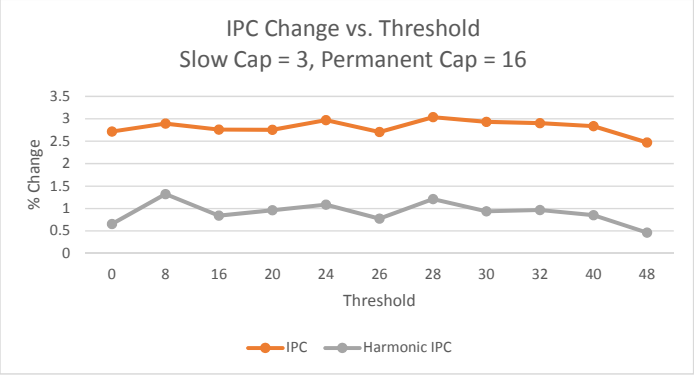

Fig. 11. % Change in IPC and Harmonic IPC vs threshold with a slow cap of 3 and static cap of 16

3) Finding an Optimal Slow Cap: The optimal slow cap of the 8-thread workload will be sought through a process similar to the 4-thread workloads. The threshold will be fixed at the previously-found optimum, 28, and the simple cap will remain at 16.

Since the peaks of performance improvements are less-pronounced than those of the 4-thread workloads, a wider range of values will be swept for the possible cap values of the 8-thread workloads. The cap will be swept from a low value of 2 to a high value of 16 in increments of two. Once a peak is found, the cap will be swept with finer granularity on both sides of the peak to allow for more precise results. This range will cover all values from 2, which was the lower-bound of the 4-thread workload, to 16, the value of the static cap. From Figure 12, it is shown that the maximum improvement occurs with a slow cap value of 7 with an arithmetic IPC increase of 3.39% and a harmonic IPC increase of 1.45%.

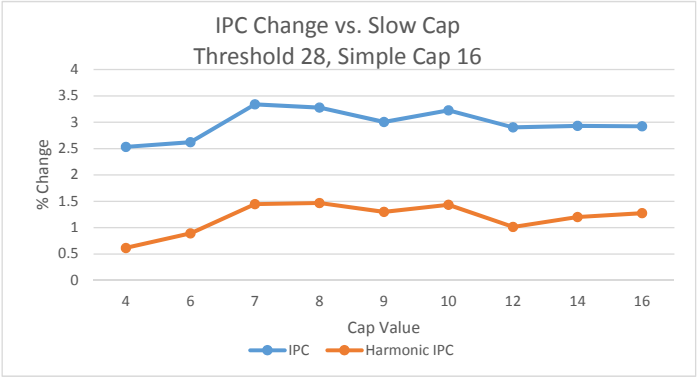

Fig. 12. % Change in IPC and Harmonic IPC vs slow cap with a threshold of 28 and static cap of 16

4) Finding an Optimal Static Cap: As with the 4-thread case, the optimum slow cap and thresholds will be fixed as the value for the simple cap is swept over a range of values to find an optimum. Thus, the slow cap will be fixed at 7 and the threshold will be fixed at 28. Results of this sweep are shown in Figure 13 A static cap value of 16 is found to be the optimum from this process.

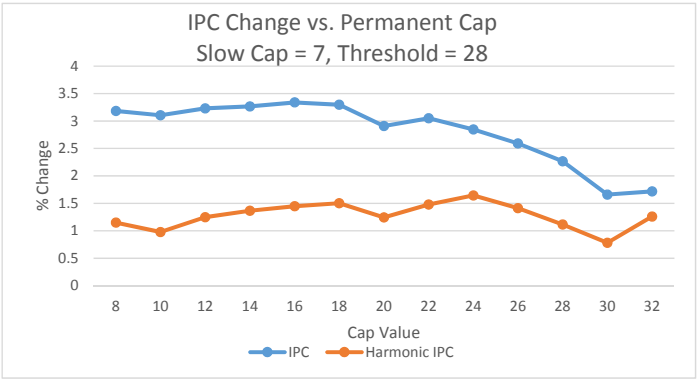

Fig. 13. % Change in IPC and Harmonic IPC vs static cap with a slow cap of 7 and a threshold of 28

5) Comparison to Static Capping: Figure 14 shows the dynamic and static capping algorithms plotted against each other for 8-thread workloads. The optimal value for the original static capping algorithm occurs at 6 with an arithmetic IPC improvement of 3.14% and a harmonic IPC improvement of 1.25%. The optimal value of the dynamic algorithm occurs at 16 with an arithmetic IPC improvement of 3.33% and a harmonic IPC improvement of 1.45%. A similar trend for the proposed dynamic approach in better sustaining the improvement under higher static cap values is also illustrated in this figure.

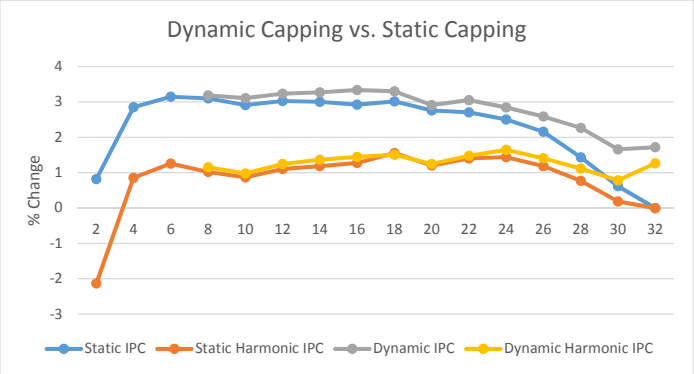

Fig. 14. Comparison between static capping and dynamic capping

VII. Future Work

Listed in this section are several potential directions of extension from this research that may lead to further performance improvement.

A. Parameter Optimization

The parameters chosen as optimal were derived using a greedy algorithm, i.e., local optimizations were chosen at each sequential step and the set of collected parameters were declared optimal. This greedy heuristic has produced parameters which are supportive of the hypothesis, but are not necessarily the best due to the selection order in the optimization process. Giving each parameter additional sweeps while fixing the other two at their last-known optimum could refine the parameter set further, for example.

B. Criteria Expansion

The algorithm proposed in this paper classified threads as poorly-performing based on the number of ROB entries which were issued and awaiting operands or completed and not committed. The sum of these two types of instructions was used as a performance indicator, with equal weight given to each type. Accuracy of this indicator may be further expanded in two ways: different parameters or weighted parameters. Although the correlation shown in the hypothesis suggests that the two chosen instruction types are indicators of thread performance, there may be other better or similarly correlating parameters. In addition, a weighting system may further improve the effectiveness of the algorithm. For example, in the system tested, the maximum number of instructions completed and not committed is 128, while the maximum of instructions dispatched and awaiting operands cannot exceed 32. The vast differences in capacity should call for a weighting system to better reflect their respective degree of correlation, instead of a simple sum of the two.

C. Reducing Hardware Cost

The method proposed in this paper keeps a running sum of how many instructions are in the IQ awaiting operands or completed and not yet committed, and this value needs to be updated in each clock cycle. At the ending edge of each window the average of all values is then calculated for the algorithm to impose the cap. Such a process obviously requires a computation overhead which may be undesirable cost-wise or time-wise.

Possible variations to the algorithm include sampling only on window edges. For example, one can check the current status of each thread at each window edge; that is, using a one-clock cycle status to impose caps for the next window. Another possible variation is a sparser sampling method which effectively falls between the original method and sampling only on the window edges. This method would sample each thread periodically throughout each window, but not every clock cycle. The average would then be used as in the original method, but with less data taken and less computation required.

VIII. Conclusion

This paper presents an algorithm for dynamically choosing a limit to the number of IQ slots a thread may occupy based on its collective instruction status. It has been shown that that dynamic capping can improve system performance beyond that of a static cap. The potential improvements described in VII suggest the algorithm has greater potential with less overhead. These alternatives are being investigated.

References

[1] Y. Zhang et al., "Autonomous Control of Issue Queue Utilization for Simultaneous Multi-Threading Processors,

[2] F. J. Cazorla, A. Ramirez, M. Valero and E. Fernandez, "Dynamically Controlled Resource Allocation in SMT Processors", *In the Proceedings of the 37th International Symposium on Microarchitecture*, pp. 171-182, Dec. 2004

[3] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor", *Proceedings of the 23th Annual International Symposium on Computer Architecture*, pp. 191-202, May 1996

[4] S. E. Raasch and S. K. Reinhardt, "The impact of resource partitioning on SMT processors", *Proceedings of the 12th international Conference on Parallel Architectures and Compilation Techniques*, pages 15âĂŞ25, Sept. 2003

[5] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism", *ACM SIGARCH Computer Architecture News*, pages 392-403, 1995

[6] P. B. Gibbons and S. S. Muchnick, "Efficient instruction scheduling for a pipelined architecture", *ACM SIGPLAN Notices*, pages 11-16, 1986

[7] S. J. Eggers, J. S. Emer, H. M. Leby, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors", *Micro, IEEE*, pages 12-19, 1997

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

[9] U. Sigmund and T. Ungerer, "Identifying Bottlenecks in a Multithreaded Superscalar Microprocessor", *Euro-Par'96 Parallel Processing*, pages 797-800, 1996

[10] A. Snavely, D. M. Tullsen, and G. Voelker, "Symbiotic Jobscheduling with Priorities for a Simultaneous Multithreading Processor", *ACM SIGMETRICS Performance Evaluation Review*, pages 66-76, 2002

[11] R. Jain, J. Hughes, and S. Adve, "Soft real-time scheduling on simultaneous multithreaded processors", *ACM SIGMETRICS Performance Evaluation Review*, pages 134-145, 2002

[12] K. Luo, J. Gummaraju, and M. Franklin, "Balancing Thoughput and Fairness in SMT Processors", *ISPASS*, pages 164-171, 2001

[13] K. Luo, M. Franklin, S. S. Mukherjee, A. Sezne, "Boosting SMT Performance by Speculation Control ", *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 9, 2001