A REAL-TIME PER-THREAD IQ-CAPPING TECHNIQUE FOR SIMULTANEOUS MULTI-THREADING (SMT) PROCESSORS

APPROVED BY SUPERVISING COMMITTEE:

Wei-Ming Lin, Ph.D., Chair

Eugene John, Ph.D.

Bao Liu, Ph.D.

Accepted:

Dean, Graduate School

DEDICATION

This thesis is dedicated to my beloved parents for their love, encouragement and endless support.

A REAL-TIME PER-THREAD IQ-CAPPING TECHNIQUE FOR SIMULTANEOUS MULTI-THREADING (SMT) PROCESSORS

by

AMIN SAHBA, M.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

MASTER OF SCIENCE IN COMPUTER ENGINEERING

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Engineering
Department of Electrical and Computer Engineering
December 2013

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Wei-Ming Lin who kindly let me work under his supervision. Dr. Lin helped me to learn more subjects in computer architecture and his great advises made me a better researcher. I am thankful to research committee Dr. Eugene John and Dr. Bao Liu for reading and evaluating my thesis.

December 2013

A REAL-TIME PER-THREAD IQ-CAPPING TECHNIQUE FOR SIMULTANEOUS MULTI-THREADING (SMT) PROCESSORS

Amin Sahba, M.Sc.
The University of Texas at San Antonio, 2013

Supervising Professor: Wei-Ming Lin, Ph.D., Chair

Simultaneous multithreading (SMT) provides a platform to improve performance with better resource utilization of superscalar CPUs by sharing key data-path components among multiple independent threads. Effective use of critical resources among threads remains a challenge to SMT due to transient behaviors of threads. As Issue Queue (IQ) is regarded as one of most critical shared resources in the pipeline, putting a limit on its occupation by each thread may easily improves the overall throughput; however, such a limit (cap) should be set properly in real time to accommodate the transient behavior of each thread in order to preclude under-utilization (thus, under-achieving) due to over-capping, or starvation for some threads due to under- capping. We propose a simple dynamic algorithm to adjust the cap value for each thread in real time according to its activeness in terms of its dispatching and issuing activities. The simulation results show that the proposed technique not only achieves a significant improvement in IPC over the regular no-capping technique, but also demonstrates a performance superior to the fixed capping approach.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: INTRODUCTION

Based on the traditional superscalar processors, Simultaneous Multi-Threading (SMT) offers an improved mechanism to enhance overall system performance without having to invest an proportional amount of extra hardware. In a conventional superscalar processor, not only the functional units are not close to be fully utilized with only one thread running at any time, switching from one thread (task) to another involves the intervention of the operating system which leads to extra overhead in CPU power. The main characteristic of SMT processors is the sharing of key datapath components among multiple independent threads, which ensures improved resource utilization. SMT also exploits not only thread-level parallelism (TLP) among the various threads [1], [2], but also equally concentrates on the advantages available at the instruction-level parallelism (ILP) in each thread. Subsequently, due to the sharing of resources, the amount of hardware required in an SMT system is significantly less than employing multiple superscalar ma- chines while achieving similar performance. Most common resources shared in an SMT system include components that are control-complexity-wise easier to share (such as cache memory and physical register bank), and those that are cost- wise better to share (such as Issue Queue (IQ), reservation stations and various functional units). On the other hand, those components which are more thread-specific (such as Re-Order Buffer (ROB)) along the datapath are assumed to remain per-thread ownership. Due to the requirement in resource sharing, these hardware components tend to remain busy in order to accommodate more instructions from all threads. Although allowing these instructions to share these resources ensures the full performance potential afforded by SMT [3], it tends to induce extra control complexities in managing the critical timing path and the processors cycle. To retain the exploitation of both 2 TLP and ILP, a neces- sary solution must be introduced in order to minimize the complexity among these shared resources without affecting the ILP exploitation significantly. At the same time, proper intelligence has to be incorporated into the resource sharing mechanism to ensure that threads share these components in the most efficient and fair manner.

1.1 Goals and Contributions

The goal of this research is to develop a dynamic allocation technique which sets the cap value for each individual thread by "predicting" each thread's reasonable demand of IQ entries in the near future according to the recent instruction dispatching and issuing activities. Compared to a fixed- capping approach, this technique should provide a better utilization of IQ and further enhance SMT performance while requiring very minimal additional hardware.

1.2 Related Work

Recently there have been many research activities targeted in improving SMT performance through modifying various stages in the pipeline. These include: improvement of thread co-scheduling by using probabilistic modeling for prediction in [4], avoiding register allocation by predicting transient values from branch misprediction in [5], packing instructions in issue queue to reduce delay and power consumption in [6], an allocation technique on write buffer for efficient resource occupation by limiting the maximal number of write buffer entries that a thread is allowed to have in [7], improving SMT fetching with an estimation of outstanding work in the system for each thread in [8], early deallocation of registers in association with cache misses in [9], dynamically reconfigurable cache design for better IPC in [10], a modified fetch policy with adaptive resource partitioning in [11], and another fetch policy by considering memory-level parallelism in [12]. None of these techniques specifically addresses the contention in the issuing stage and most come with a significant requirement in extra hardware to implement the desired intelligence. There are another set of algorithms that consider how to effectively utilize the limited IQ entries, including capping the usage of IQ entries per thread [13], recalling stalled instructions from IQ [14] and speculative-control to improves threads' utilization of IQ by reducing the amount of flush-out instructions caused by miss-speculation in [15]. However, all these techniques are based on a fixed cap value universal to all threads, which does not take each thread's real-time behavior into consideration for allocating IQ entries.

1.3 Organization of Thesis

The rest of the report is organized in the following way. The workloads and simulation methodologies are introduced in chapter 2. Chapter 3 shows Instruction Dispatching in SMT and briefly describes different techniques in scheduling instructions to be dispatched. Chapter 4 describes fixed IQ-capping method. Chapter 5 explains the proposed approach and provides analytical simulations and finally, the report is concluded with summarizing remarks and future work.

Chapter 2: SIMULATION ENVIRONMENT

The simulation environment including the simulator and the workloads used for our simulations are described in this chapter.

2.1 Simulator

We used M-Sim [16], a multi-threaded micro architectural simulation environment model, to estimate performance of the proposed scheme. M-sim includes accurate models of the pipeline structures such as explicit register renaming, concurrent execution of multiple threads, detailed power estimation using Wattch framework [17], separate Reorder Buffer, Load-Store Queue (LSQ), and register files which are necessary for an SMT model. The Issue Queue, execution functional units are shared among the threads, but branch predictor is exclusive to each thread. The detailed processor's configuration is shown in Table 2.1.

2.2 Workloads

For multi-threaded workloads, we use the mixed SPEC CPU 2000 benchmark suite [17], [18], [19] based on ILP classification. Each of the benchmarks is initialized in accordance with the procedure mentioned in Simpoints tool [20] and then up to 100 million instructions are simulated. Once the first 100 million instructions are committed from any of the threads, simulation is terminated.

In order to categorize multithreaded workloads, each of the benchmarks is simulated individually in a simple scalar environment. Each thread is categorized based its degree of ILP, and three types of ILPs - low ILP (memory bound), medium ILP and high ILP (execution bound) - are identified, A number of mixes of multi-threaded workloads are then used based on different combinations of threads of various ILP types. Both a 4-threaded and an 8-threaded SMT are simulated. Table 2.2 lists the 6 mixes used for the 4-threaded system simulation. Table 2.3 shows the 6 mixes for the 8-threaded case.

Table 2.1: Configuration of the Simulated Processor

Parameter	Configuration
Machine Width	8 wide fetch/issue/commit
Buffer size	32-entry Issue Queue, 48-entry Load/Store queue, 128-entry ROB
Function Units & Latency (total/issue)	8 Int Add (1/1), 4 Int Mult (3/1) / Div (20/19), 4 Load/Store (2/1), 8 FP Add (2), 4 FP Mult (4/1) / Div (12/12), Sqrt (24/24)
Physical registers	256 integer and floating point
L1 I-cache	64KB, 2-way set-associative, 128-byte line
L1 D-cache	32 KB, 4-way set-associative, 256-byte line
L2 Cache unified	2 MB, 8-way set-associative, 512-byte line
BTB	2048-entry, 2-way set-associative
Branch Predictor	2K-entry gshare, 10-bit global history per thread
Pipeline Structure	5 stages: fetch, dispatch, execution, write back & commit
Memory	64-bit wide, 300 cycles access latency

Table 2.2: Simulated Multi-threaded Workload on A 4-thread SMT

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	perlbmk, mesa, swim, crafty	1	0	3
Mix 2	mcf, equake, vpr, lucas	4	0	0
Mix 3	applu, gcc, mgrid, galgel	0	4	0
Mix 4	mcf, equake, mesa, crafty	2	0	2
Mix 5	perlbmk, lucas, galgel, gcc	1	2	1
Mix 6	mesa, swim, apsi, mgrid	1	2	1

Table 2.3: Simulated Multi-threaded Workload on A 8-thread SMT

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	perlbmk, mesa, swim, crafty, galgel, mgrid, lucas, apsi	2	3	3
Mix 2	mcf, equake, vpr, lucas, applu, gcc, perlbmk, swim	5	2	1
Mix 3	applu, gcc, mgrid, galgel, apsi, vpr, crafty, mesa	1	5	2
Mix 4	mcf, equake, mesa, crafty, apsi, swim, lucas, perlbmk	4	1	3
Mix 5	perlbmk, lucas, galgel, gcc, mcf, equake, mesa, crafty	3	2	3
Mix 6	swim, apsi, mgrid, equake, mcf, vpr, applu, gcc	4	4	0

Chapter 3: INSTRUCTION DISPATCHING IN SMT

There have been many different terminologies adopted for instruction pipelining stages (e.g. issue, dispatch, etc.) in a superscalar system, and their references became even more ambiguous in an SMT system in which more resource sharing is required than in a superscalar system. For example, instruction "issuing" has been referred to different stages of processing by different articles. Throughout this paper, we choose to adopt the terminologies used by most SMT articles.

In a typical single-thread superscalar system, instructions are "dispatched" from ROB into the reservation stations (either centralized or functional unit-specific) when space is available and then "issued" to the corresponding functional unit whenever the issuing conditions are met, i.e. operands are ready and the requested functional unit becomes available. However, in a basic SMT system, each thread has its own ROB and instructions from these thread-specific ROBs have to "compete" for a shared Issue Queue (IQ) through a dispatching scheduling algorithm. This IQ can be considered as Centralized Reservation Station not only shared among the functional units but also shared among the threads in real time. A basic functional block diagram of this basic design is depicted in Figure 3.1.

Due to the significantly large size of each IQ entry and the complex circuitry for its control, the number of entries in this shared resource usually is much smaller than the number of ROB entries. Having its output sent into a tightly shared resource, the instruction dispatch stage is considered one of the most critical stages that dearly affect the overall system performance. There have been many different techniques in scheduling instructions to be dispatched from ROB intending to exploit ILP. Most techniques rely on simple instruction-level readiness to reduce the instructions' waiting time in IQ [21]. To better utilize the shared IQ entries, instructions from different threads should be given different priorities depending on the then threads' "activeness" in real time. Threads have been shown to demonstrate various types of transient behaviors throughout their lifespan, including stretches of time durations with fluctuating ILP. Instructions from thread(s) of lower ILP (or simply "slower" in instruction issuing) would clog the IQ if they are allowed to continue to

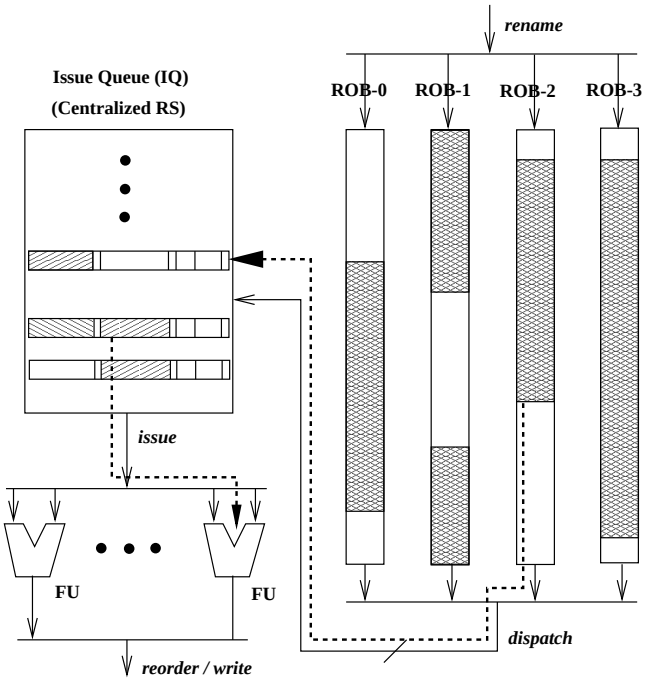

Figure 3.1: A Simple Four-Thread SMT Instruction Processing Flow

be dispatched into IQ. On the other hand, instructions from a thread with higher ILP (or simply "faster" in instruction issuing) should be given a higher priority in utilizing the IQ. There have not been many research results on how to share the IQ in a more time-adaptive manner allowing different threads to utilize (occupy) the IQ in an "on-demand" basis. In [22], an adaptive technique is proposed to allow all ROBs to be "shared" among threads in order to accommodate threads that are more active with extra ROB entries borrowed from the ROBs from less active threads, in which "activeness" of a thread is based on a ratio between number of issue-bound and commit-bound instructions in a thread's ROB. Control complexities involved in sharing the ROBs may be the most inhibitive factor in justifying the performance gain from such an approach. Some other more advanced scheduling techniques, such as the one presented in [23], combine more information from different stages in the pipeline to optimize the scheduling/dispatching result, albeit requiring significantly more hardware and control logic.

In this research, we choose to retain the per-thread ROBs without any sharing among them, and rely on a simple scheduling algorithm in dispatching instructions from different threads. Our

analysis shows that activeness of a thread can be fluctuating very unexpectedly in time, due to several factors, e.g. cache misses, branch miss-predictions, write port latencies, etc. A thread that has been active can suddenly becomes "inactive" (or, more precisely, much less active) due to any of these reasons and stays "idle" in the pipeline for a long duration of time. A write cache miss can easily delay the in-order commit stage while a read miss can significantly slow down the pipeline due to data dependencies. A branch miss-prediction requires all speculative instructions to be flushed out and subsequently the activeness of the thread suffers before the correct path of instructions is re-established. To make it worse, these threads that have been just recently more active (than other threads) tend to occupy more shared resources (for example, the IQ, reservation stations, among others) than others. When these threads suddenly become inactive, the shared resources typically cannot be released for other threads to use, mainly due to various system or operating limitations inhibiting it from doing so. Consequently, there could be only a very limited amount of shared resources for other threads to share, and thus limiting the potential performance. While there could be many different intelligent ways to redistribute the resources, they tend to be either very costly in hardware or simply infeasible in implementation for real-time operation due to the excessive logic required to realize the intelligence.

There have been many non-adaptive dispatching algorithms proposed, including simple round-robin, withinclock- cycle-round robin [24], two-op-block [21], etc. None of these algorithms are assigned with a priority among threads, instead all are based on a pre-assigned thread index order. The basis for performance comparison in this paper will be the traditional simple round-robin dispatching which simply starts dispatching all dispatchable instructions from a thread according the index order and moves to the next thread if the dispatching bandwidth allows for more instructions; the index order is then rotated naturally to the next thread to start in order in the next clock cycle.

Chapter 4: FIXED IQ-CAPPING

The proposed technique is based on the fixed IQ-capping method [13] which presents a simple mechanism to better manage IQ sharing among threads. In the default SMT system, when a thread is stalled, its instructions already dispatched into IQ will stay in IQ until the thread becomes active again. Such a stall is due to either a simple data dependency which is usually resolved in a few clock cycles or a more resource-straining cache miss which can last over a few hundred clock cycles. If such a thread has already occupied a significant portion of IQ entries before becoming stalled (inactive), these entries will remain occupied which may seriously impede the progress of other active threads and thus the overall performance. By limiting the maximal number of IQ entries that a thread is allowed to occupy, the fixed capping method reduces the risk that most of IQ entries are occupied by inactive threads.

Figure 4.1 and Figure 4.2 show the IPC (Instruction PerClock cycle) values of each mix with the fixed capping method applied with different cap values in 4-threaded and 8-threaded SMT systems, respectively, where R and q denotes the ROB size and IQ size. The workload and system settings are as described in Section 3. In the 4- threaded system as depicted in Figure 4.1, with the cap value adopted varying from 12 to 26 (for q = 32), compared to the default (non-capping) setting, each mix shows a very different performance spectrum as the cap value changes. A similar result is observed in the 8-threaded case; however, the cap value is set to be within a much smaller range than that of 4-threaded case. This different setting is needed because 8 threads concurrently running in the system (versus 4 threads), with the same size of IQ, create a higher level of competition for IQ entries among them, and thus a tighter capping is required to ensure occurrences of IQ-starvation are mitigated.

The average IPC improvement using the fixed capping method is shown in Figure 4.3 and Figure 4.4 – up to 3.7% is observed in a 4-threaded system and 23.5% for an 8-threaded system. It is obvious that the fixed capping technique has achieved a significant improvement over a non-capping system; however there are a few intrinsic shortcomings in this approach that need to be

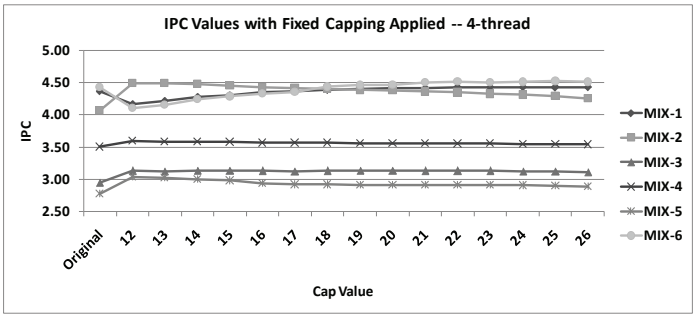

Figure 4.1: IPC Performance Comparison Using Fixed Capping on a 4-threaded SMT System under (R, q) = (128, 32)

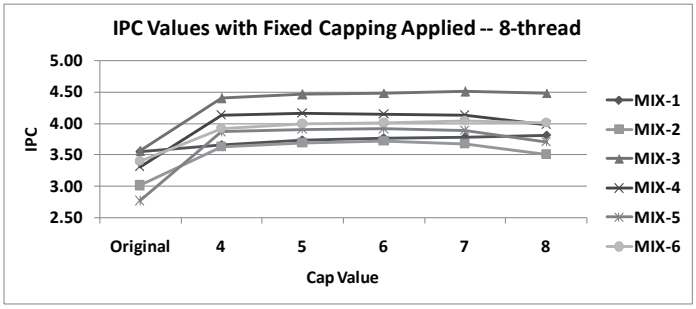

Figure 4.2: IPC Performance Comparison Using Fixed Capping on an 8-threaded SMT System under (R, q) = (128, 32)

carefully addressed to deliver its full potential.

1) A fixed cap value for all systems? From the results shown in Figure 4.3 and Figure 4.4, one can easily see that the best cap value changes when the number of threads concurrently running in the system changes. In a 4- threaded system, the mixes have their optimal cap values varying from 12 to 25, while in an 8-threaded system, the optimal cap value decreases dramatically to around 6. Figure 4.5 shows the optimal cap values for a 4-threaded and an 8-threaded workload when the fixed capping method is applied, from which we see that the optimal cap value also varies dramatically in different workload situations. Thus, it is not reasonable to assume a fixed cap value can always lead to improvement in performance, let alone achieving its full potential. In addition to the change in the number of threads, other changes in system settings can also easily lead to an unexpected outcome with a fixed cap value, including changes in the running threads, changes in the inter-pipeline-stage bandwidth, sizes of various buffers in the system (e.g. IQ, ROB, IFQ, write buffer, etc.), number of functional units, etc.

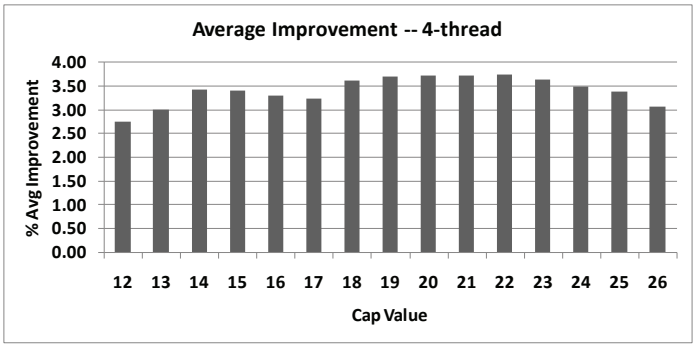

Figure 4.3: Average IPC Improvement by Using Fix Capping Method on a 4-threaded SMT System under $(R, q) = (128, 32)$

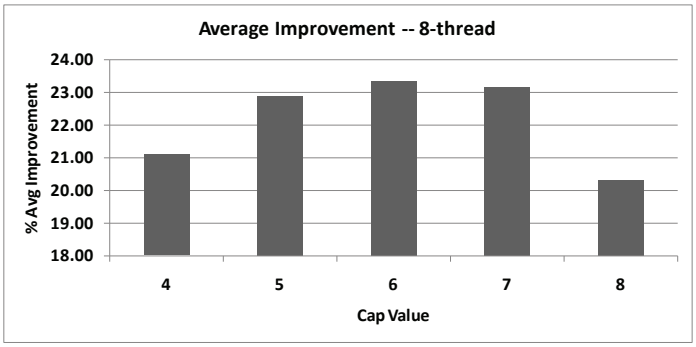

Figure 4.4: Average IPC Improvement by Using Fix Capping Method on an 8-threaded SMT System under $(R, q) = (128, 32)$

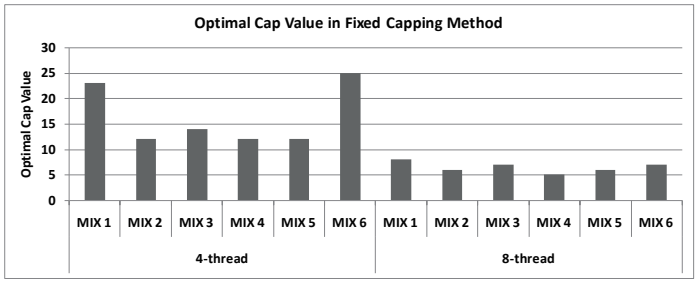

Figure 4.5: Optimal Cap Value

2) A temporally fixed cap value? Similar to the first concern, it is simply unfathomable to assume one fixed cap value is capable of delivering a sustainable performance throughout the lifetime of a mix of threads, considering the fact that most programs may exert very different behaviors at different time durations throughout their lifespan. Combination of behaviors from threads concurrently running poses another level of variation, which may easily prompt for a totally different cap value. This is true even the number of threads supported by the system remains unchanged. Threads can be stalled at arbitrary time points for an arbitrary length of duration and the number of stalled threads concurrently happening again can vary, all of which may demand a different cap value for a best transient performance.

3) One cap value for all threads? The third concern originates from the fact that all threads running the system can be very different from one another not only in their ILP categorization but also in their transient behavior in terms of activeness. That is, it may not be the best practice to impose the same cap value for all threads at the same time – capping an inactive thread makes sense while capping an active thread may easily impede its progress and thus the overall system performance.

To address these concerns, this report proposes a technique to instead dynamically adjust the cap value for each thread in real time based on some performance indicators observed.

Chapter 5: PROPOSED DYNAMIC PER-THREAD CAPPING

As mentioned in previous section, the fixed capping approach does show very promising results while an arbitrarily assigned fixed cap value does not always lead to a desirable gain for different mixes or different system settings. Setting an identical fixed cap for all threads throughout their lifespan also limits further performance improvement to be achieved. The proposed method is based on the idea that if activeness of a thread in the coming clock cycles can be somewhat predicted, a better per-thread cap value can be set for it. One simple mechanism for such a prediction can be based on number of instructions "progressing" in the stages before and/or after the IQ, which directly affects the use of IQ entries. Accordingly, the two proposed techniques in this paper choose to monitor the number of instructions dispatched and issued in the current clock cycle as indicators of a thread's activeness. The level of activeness thus observed is then used to adjust the cap value of the thread for the next clock cycle. The proposed adjustment algorithm is to simply set a smaller cap value for a less active thread, based on a notion that the instructions dispatched from an inactive or less active thread tends to occupy IQ slots for a longer time than those from the more active threads. Setting a tighter cap for less active threads would allow more entries to be utilized by threads that are more active, delivering a higher throughput.

If the cap adjustment algorithm is implemented properly, two obvious advantages of this technique over the fixed-capping one can be realized: (a) a cap value individually tailored for each thread, and (b) the individual cap value adjusted according to the real-time behavior of the respective thread. However, in order to implement this algorithm, one has to take into consideration its feasibility in real-time. That is, the adjustment mechanism cannot be too complicated involving too much control logic to satisfy timing constraint per clock cycle. The first advantage just mentioned requires that each thread has it own independent control logic, and the second advantage calls for a logic to set the cap value in real time based on the observed performance. To simplify the process, two main components are specified as in the following:

1. Simple performance indicators are used

2. Only a few cap value settings are adopted

For (1), as aforementioned, the number of instructions dispatched and issued in the current clock cycle are used as indicators of a thread's activeness. As for (2), our methods choose to limit the cap value to be in one of only 4 states in the control state machine, and a change of activeness of a thread will trigger a transition from one state to another, setting a different cap value. Note that using more states would most likely have a higher potential for better performance, while at the same time rendering the system too costly or too slow for real-time implementation. A diagram of a typical 4-state machine is shown in Figure 5.2.

5.1 Base Cap Value

Among the 4 states in our state machines, one corresponds to the "base" cap value, which represents the upper limit of the cap values that be set at, and each of the other three states allows for a lower cap value when the thread shows a sign of being less active. In order to have a reasonable "base" cap value (denoted as α) for various systems with different number of threads and IQ sizes, α is pre-selected to be a portion of the IQ size (denoted as q) and this upper limit value decreases as the number of threads increases. The rationale behind such a thread-number-dependent α is easily understandable since the more concurrently running threads are in the system, the smaller the maximal number of IQ entries each thread is allowed to use should be. This is to accommodate the scenario that, when there are more threads, it is more likely the whole IQ may be completely overwhelmed by inactive threads if the maximal cap value is not reduced. The base cap value α is preset with the following formula:

$$\alpha = (1 - x \cdot \sqrt{T}) \cdot q \tag{5.1}$$

where T is the number of threads and x is an adjustment factor to control how drastic this upper cap value should vary according to the number of threads. This formula leads to a rather desirable trend of setting α relative to the change of thread count. This formula is established with the mind-

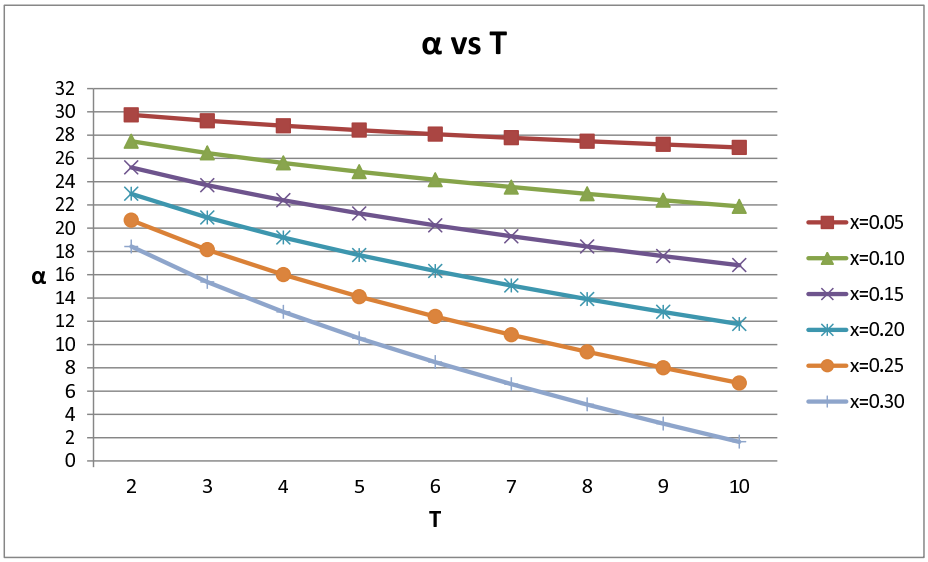

Figure 5.1: Variation of α versus T for Different Value of x under (R, q) = (128, 32)

set of multi-threading application, thus the case of $T = 1$ which should not have required any capping at all is left out for the sake of simplicity. Figure 5.1 shows the variation of α according to the number of thread varying from 2 to 10 with 6 different values of x applied. The result shows that when the number of threads increases, the base cap value decreases, and a system with a larger x value leads to a more aggressive capping with a smaller base cap value to start with.

5.2 Issuing-based Activeness Capping

As aforementioned, both of our designs are based on a 4-state state machine to simplify the transition control logic. The base state is with the base cap value (α) and each of the other three states corresponds to a fixed portion of the base cap value, $\frac{3}{4}\alpha$, $\frac{2}{4}\alpha$, $\frac{1}{4}\alpha$, respectively, again for the sake of feasibility.

How activeness of a thread is classified distinguishes between the two designs. In the first design, activeness is based solely on the issuing activity of a thread. Figure 5.2 shows the state machine. Such an Issuing-based Activeness Capping (denoted as IAC) switches between two adjacent states according the current activeness indicator of the thread. If there are any instructions issued in the current clock cycle, state transition continues to move toward higher cap values, and

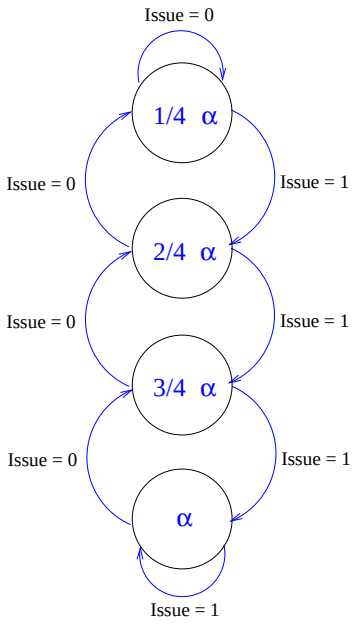

Figure 5.2: State Machine for Issuing-based Activeness Capping (IAC)

reverses if otherwise.

Performance of the IAC is shown in Figure 5.3 from tests on the 6 4-threaded mixes (prescribed in Chapter 3) with (R, q) = (128, 32) using various values of the adjustment factor x. The original IPC value from default configuration is also provided for comparison. In this 4-threaded simulation result, all mixes see their maximal improvement at $x = 0.1$ although the difference is not significant, and all obtain considerable performance improvement over the default system. The improvement percentage averaged over all 6 mixes is shown in Figure 5.4. The average IPC improvement reaches its maximal value of about 5.5% at $x = 0.1$. This improvement is a considerable jump from the gain obtained using the fixed capping method (3.7% as shown in Figure 4.3).

A similar comparison is performed on an 8-threaded set-up with the IPC results shown in Figure 5.5. Average improvement percentage over the default method is then shown in Figure 5.6. Maximal improvement value of 24% is observed at $x = 0.25$, a shift from $x = 0.1$ for the 4-threaded case indicating that a tighter base cap value is required for a system with more threads competing for the shared resources.

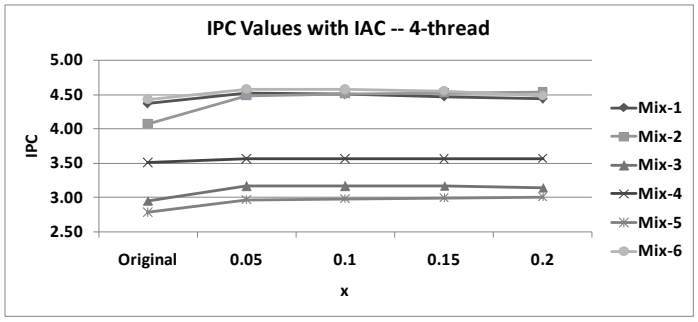

Figure 5.3: IPC Performance Comparison Using IAC on a 4-threaded SMT under (R, q) = (128, 32)

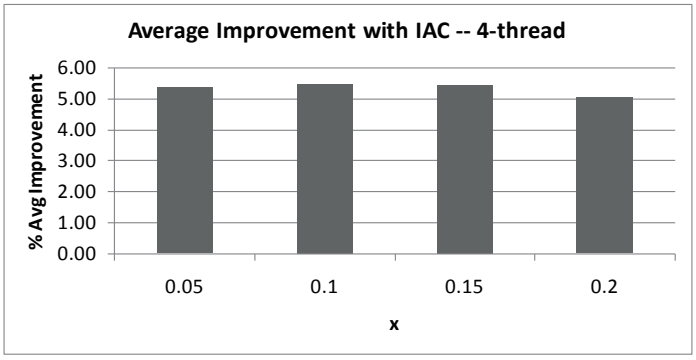

Figure 5.4: Average IPC Improvement by Using IAC on a 4-threaded SMT under (R, q) = (128, 32)

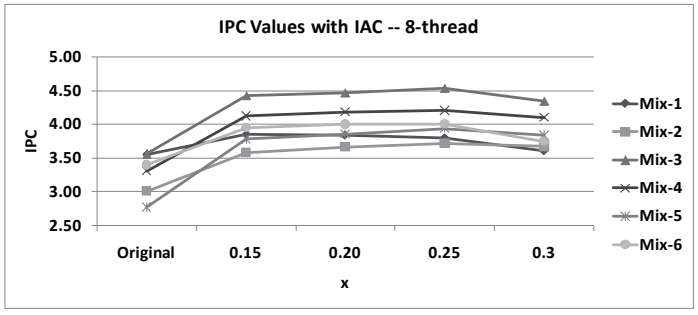

Figure 5.5: IPC Performance Comparison Using IAC on an 8-threaded SMT under (R, q) = (128, 32)

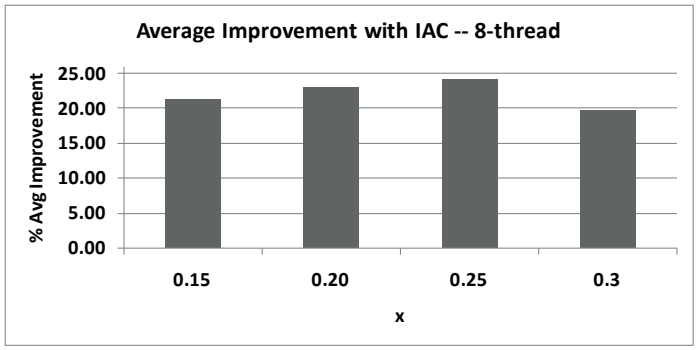

Figure 5.6: Average IPC Improvement by Using IAC on an 8-threaded SMT under (R, q) = (128, 32)

5.3 Dispatching/Issuing-based Activeness Capping

The IAC technique is based on a very simple definition of activeness relying solely on the issuing activities. Our second proposed technique expands the definition to incorporate the dispatching activities into state machine decision-making criteria with a minimal amount of extra control complexity introduced. This Dispatching/Issuing-based Activeness Capping (to be referred to as DIAC) should be able to address additional situations that the IAC cannot. Figure 5.7 shows the design of this approach. The cap values for the four states in DIAC are exactly the same as those in the IAC, with the transitions moving toward tighter cap values triggered by a combination of issuing and dispatching activities or inactivities. Note that, in this DIAC approach, there are three levels of "activeness" thus classified based on the said combination:

1. Instruction(s) issued (issue $= 1$)

2. No instruction(s) issued or dispatched (issue $= 0$, dispatch $= 0$)

3. No instruction(s) issued, but instruction(s) dispatched (issue $= 0$, dispatch $= 1$)

For a thread displaying a level-1 activeness, transitions between states behave exactly the same as those in IAC by moving toward a looser cap setting. A level-2 activeness status indicates a situation where a thread does not issue instruction(s) from IQ nor does it continue to dispatch instruction(s) into IQ, which triggers a change of cap value gradually toward the tighter direction.

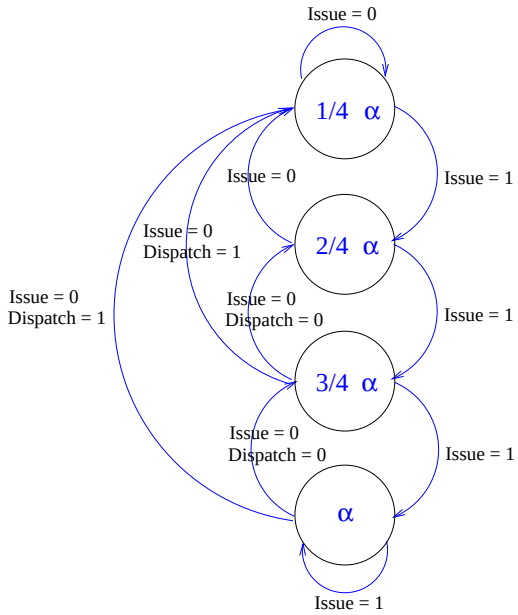

Figure 5.7: State Machine for Dispatching/Issuing-based Activeness Capping (DIAC)

A more urgent situation happens in the level-3 activeness in which a non-issuing thread continues to dispatch instruction(s) into IQ further occupying more IQ entries. Instead of gradually tightening it cap value, DIAC triggers a more aggressive transition changing the cap value to the tightest one ($1/4\ \alpha$) no matter which current state it is. Justification for this action is to ensure more IQ slots remain available for other more active threads.

A similar performance comparison for DIAC is undertaken. Figure 5.8 show the IPC for each individual mix on a 4-threaded SMT system. Average IPC improvement percentages among the 6 mixes are shown in Figure 5.9. These results show that the average IPC improvement has its maximal value of 5.76% when the x value is 0.1, which is superior to the result from IAC.

Simulation results on an 8-threaded system are also shown in Figure 5.10 and Figure 5.11.

5.4 Comparison and Analysis

Figure 5.12 shows the IPC improvement comparison between the two proposed techniques, IAC and DIAC, on a 4-threaded SMT system. It clearly shows that the additional investment made in DIAC over the IAC does pay off to an extent, consistently leading to a small margin of performance

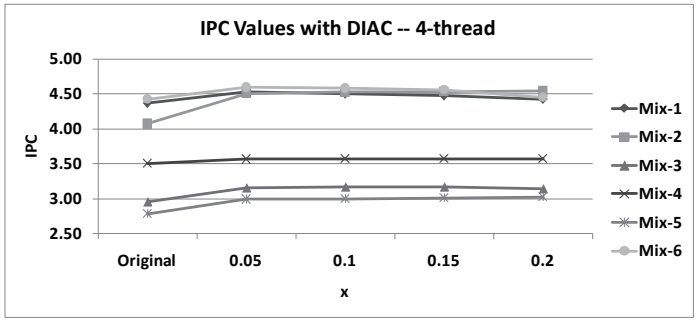

Figure 5.8: IPC Performance Comparison Using DIAC on a 4-threaded SMT under (R, q) = (128, 32)

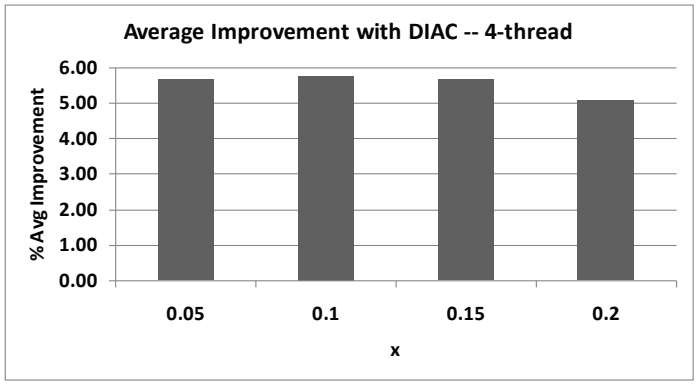

Figure 5.9: Average IPC Improvement by Using DIAC on a 4-threaded SMT under (R, q) = (128, 32)

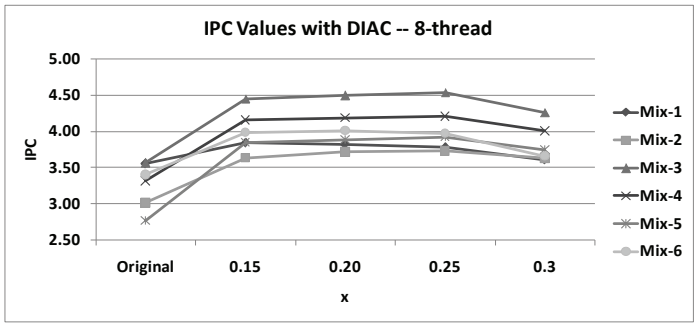

Figure 5.10: IPC Performance Comparison Using DIAC on an 8-threaded SMT under (R, q) = (128, 32)

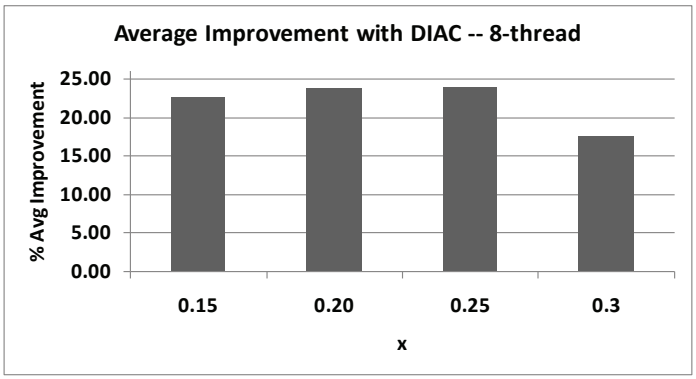

Figure 5.11: Average IPC Improvement by Using DIAC on a 8-threaded SMT under (R, q) = (128, 32)

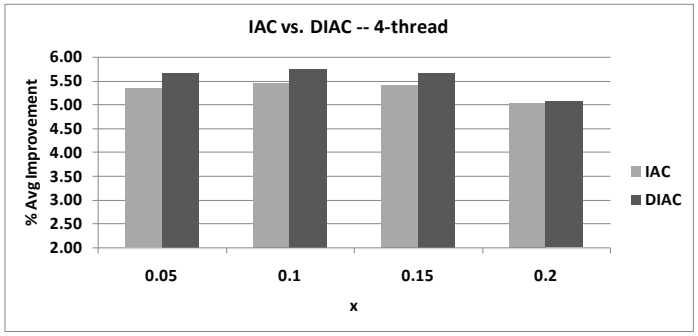

Figure 5.12: IPC Improvement Comparison between IAC and DIAC on a 4-threaded SMT under (R, q) = (128, 32)

gain, regardless what the base cap value is set at.

For the 8-threaded system, the result is not as consistent as in the 4-threaded one, as displayed in Figure 5.13 When the base cap value is set too tight (e.g. $x = 0.25$ and $x = 0.3$), the aggressiveness in the DIAC to expedite the cap-tightening process may actually backfire, leading to a cap value too small and thus under-utilizing the IQ.

To have a better understanding on the true effect of the proposed techniques, an analysis is performed to find out the cap value distribution for each thread of the 6 mixes when the DIAC machine is applied in a 4-threaded system with $x = 0.1$. Figure 5.14 shows the results for all 6 mixes. An obvious trend for all mixes is that the distribution of cap values are mostly aggregated toward the two end values (α and $1/4\ \alpha$), with the two intermediate values serving as transient states. This trend can be easily explained by the behavior of a typical thread which usually is in either

an “active” or “inactive” status for a relatively long period of time, instead of constantly changing from one to the other. Another observation leads us to be more ascertained of effectiveness of this technique – different threads in a mix have very different distribution of their cap values, a good indication that the per-thread adjustment mechanism actually may work to our desire. This is very much confirmed by another analysis to determine the correlation between the individual per-thread IPC values in a mix and the respective per-thread average cap values. Figure 5.15 displays the correlation for all 6 mixes. First of all, for all 6 mixes, each composing thread’s IPC is increased compared to the default case, a clear indication that the proposed capping technique does not sacrifice one thread’s performance for another. With this, the tight correlation demonstrated in this result between the thread’s IPC and its respective average cap values imposed by our technique clearly validates the efficacy of the proposed per-cap-adjusting mechanism. That is, a thread with a higher IPC is considered in general a more active thread and thus should be allocated with more resources (IQ entries in this case), which should be in turn associated with a higher average cap value. Throughout all the 6 mixes as indicated in the result, a thread with a higher IPC is always provided with a higher average cap value. Coupled with the first observation in the congruent trend between the new IPC and the old IPC values, we can conclude that the proposed per-thread capping technique indeed adjusts the cap value for a thread correctly according to its demand (relative to demands from other threads), and increases per-thread IPC universally. This comparison further ascertains the validity of the proposed algorithms in assigning more resources to the more “productive” threads under various combinations of system load and parameter settings.

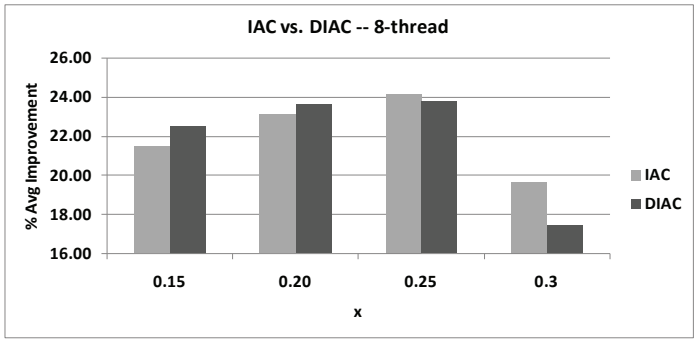

Figure 5.13: IPC Improvement Comparison between IAC and DIAC on a 8-threaded SMT under $(R, q) = (128, 32)$

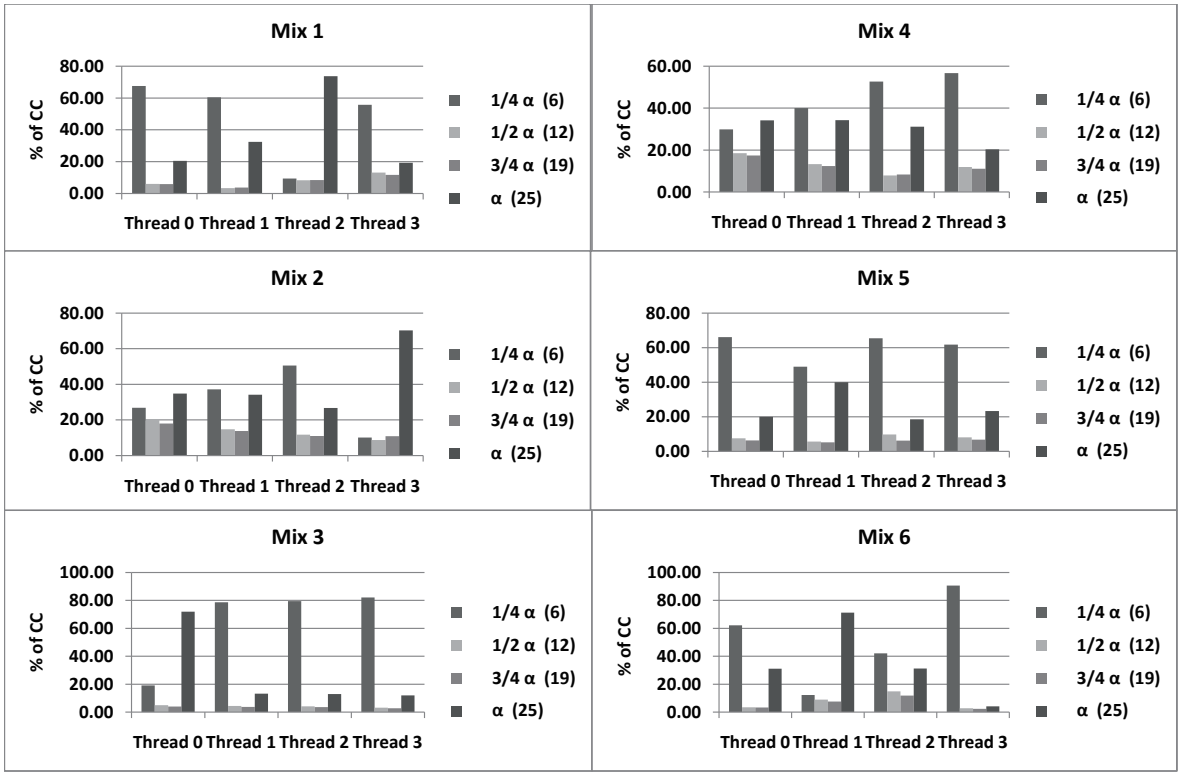

Figure 5.14: Cap Value Distribution for 6 Mixes

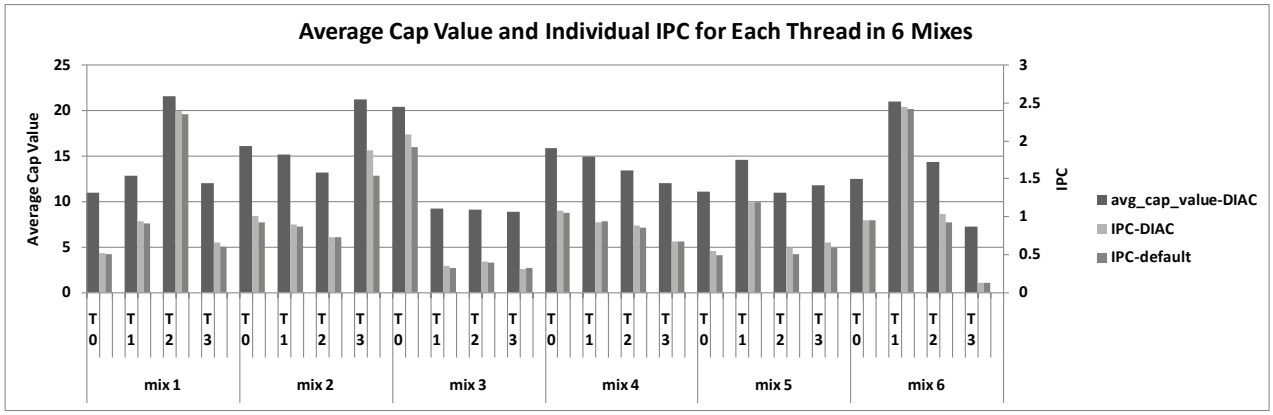

Figure 5.15: Average Cap Value and Individual IPC for Each Thread in 6 Mixes

Chapter 6: CONCLUSION

This research clearly demonstrates that utilization of resources shared among the threads in an SMT system can significantly affect the overall performance. With a small extra amount of hardware investment, the proposed techniques can easily improve the utilization by tailoring the allocation according to the per-thread real-time demands. We anticipate an even larger improvement if the same concept in per-thread allocation adjustment is further applied to other pipeline stages that also involve shared resources among threads. Additional performance gain can also be expected if additional intelligence is incorporated into the decisionmaking logic for the real-time adjustments, including having more states in the state machine.

BIBLIOGRAPHY

[1] S. J. Eggers D. M. Tullsen and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *the Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, May 1995.

[2] J. S. Emer H. M .Levy J. L. Lo D. M. Tullsen, S. J. Eggers and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *the Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[3] J. Sharkey and D. Ponomarev. Efficient instruction schedulers for smt processors. In *the Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, February 2006.

[4] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *the Proceedings of the 15th edition of ASPLOS on Architectural support for programming languages and operating systems*, March 2010.

[5] D. Ponomarev D. Balkan, J. Sharkey and K. GhoseSPARTAN. Speculative avoidance of register allocations to transient values for performance and energy efficiency. In *the Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[6] K. Ghose J. Sharkey, D. Ponomarev and O. Ergin. Instruction packing: Toward fast and energy-efficient instruction scheduling. *Transactions on Architecture and Code Optimization (TACO)*, 3(2), June 2006.

[7] Y. Zhang and W.-M. Lin. Write buffer sharing control in smt processors. In *the Proceedings of the 19th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13)*, July 2013.

[8] H. Vandierendonck and A. SeznecManaging. Smt resource usage through speculative instruction window weighting. *Transactions on Architecture and Code Optimization (TACO)*, 8(3), November 2008.

[9] J. Loew J. Sharkey and D. Ponomarev. Reducing register pressure in smt processors through l2-miss-driven early register release. *Transactions on Architecture and Code Optimization (TACO)*, 5(3), November 2008.

[10] F. Fernndez O. Garnica J. Daz, J. .Hidalgo and S. Lpez. Improving smt performance: an application of genetic algorithms to configure resizable caches. In *the Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference*, July 2009.

[11] I. Koren H. Wang and C. Krishna. An adaptive resource partitioning algorithm for smt processors. In *the Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[12] S. Eyerman and L. Eeckhout. Memory-level parallelism aware fetch policies for simultaneous multithreading processors. *Transactions on Architecture and Code Optimization (TACO)*, 6(1), March 2009.

[13] W. Lin T. Nagaraju, C. Douglas and E. John. Effective dispatching for simultaneous multithreading (smt) processors by capping perthread resource utilization. *Journal of Computing Science and Technology*, 1(2):5–14, December 2011.

[14] C. Douglas Y. Zhang and W.-M. Lin. On maximizing resource utilization for simultaneous multi-threading (smt) processors by instruction recalling. In *the Proceedings of the 18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, July 2012.

[15] Y. Zhang and W.-M. Lin. Capping speculative traces to improve performance in simultaneous multi-threading cpus. In *the Proceedings of the 18th International Conference on Parallel and*

Distributed Processing Techniques and Applications (IPDPS'13) Workshop on Multithreaded Architectures and Applications, May 2013.

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

[17] G. Hamerly B. Calder T. Sherwood, E. Perelman. Automatically characterizing large scale program behavior. In *the Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[18] T. Austin D. Burger. *The SimpleScalar tool set: Version 2.0.* Tech. Report, Dept. of CS, Univ. of Wisconsin-Madison, June 1997 and documentation for all Simplescalar releases.

[19] J. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Transactions of IEEE Computer*, 33(7):28–35, July 2000.

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

[21] J. Sharkey and D. Ponomarev. Exploiting operand availability for efficient simultaneous multithreading. *IEEE Transactions on Computers*, 56(2), February 2007.

[22] D. Balkan J. Sharkey and D.Ponomarev. Adaptive reorder buffers for smt processors. In *the Proceedings of the 15th international Conference on Parallel Architectures and Compilation Techniques*, September 2006.

[23] R. Sangireddy H. Wang. Optimizing instruction scheduling through combined in-order and o-o-o execution in smt processors. *IEEE Transactions On Parallel And Distributed Systems*, 20(3):389–403, March 2009.

[24] B. Lee M. Debnath and W.-M. Lin. Prioritized out-of-order instruction dispatching techniques for simultaneous multi-threading (smt) processors. *WIP session of 30th IEEE Real-Time Systems Symposium (RTTS)*, December 2009.

VITA

Amin Sahba is from Karaj, Iran. He earned both Bachelor's and Master's degrees in Computer Engineering respectively from Sheikh Bahaee University and Science and Research University, Iran. Besides, he received a Master's degree in Computer engineering from The University of Texas at San Antonio. His future plans include attending a Ph.D. program.