Vol. 6, No. 12 December 2015 ISSN 2079-8407
Journal of Emerging Trends in Computing and Information Sciences

Autonomous Issue Queue Distribution Control for Simultaneous Multi-Threading CPUs

Yilin Zhang[1] **Wei-Ming Lin**[2]
[1]Advanced Micro Devices, Inc., Austin, TX 78735, USA.
[2]Department of Electrical and Computer Engineering
The University of Texas at San Antonio
San Antonio, TX 78249-0669, USA.

ABSTRACT

—Simultaneous Multi-Threading (SMT) is a technique that improves overall system performance by allowing concurrent execution of multiple independent threads with sharing of key datapath components and better utilization of the resources. Congested shared resources due to slower threads can easily lead to heavily under-used resources and thus an undesirable performance outcome. Most of existing resource allocation and distribution techniques adjust the resource allocation in real time based on certain pre-determined allocation parameter settings attempting to lead to better performance. Such an improved performance is contingent on the assumptions that the system environment parameters and workload characteristic remain unchanged. Once either is changed the same settings may no longer lead to the same performance gain. In this paper, we propose an adaptive technique to allow for a complete autonomous adjustment process to control the distribution of a critical shared resource – Issue Queue (IQ) – in an SMT system. The proposed process adjusts in real time the resource distribution based on the impact to performance caused by the previous adjustment, with a simple algorithmic guideline in constantly aiming to improve performance from one adjustment to the next. No *a priori* information in system environment parameters or workload characteristics is needed for this process to acquire beforehand for the autonomous adjustment. Our simulation results show that our proposed technique is able to improve the system's overall IPC by an average of 7.9% for a 4-threaded workload and 12.5% for an 8-threaded workload, as opposed other known adaptive techniques achieving similar performance under one condition but degrading poorly under another.

Keywords: *Simultaneous Multi-Threading; Superscalar; Performance Feedback; Autonomous Control*

1 Introduction

Noting the resource utilization deficiencies in the traditional superscalar processors, Simultaneous Multi-Threading (SMT) offers an improved mechanism to enhance overall system performance without having to invest a proportional amount of extra hardware. In a conventional superscalar processor, not only the functional units and other resources 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 further diminishes CPU utilization. Simple multi-threaded processors remove the necessity of task switching by OS but still do not fully exploit the capacity of the functional units and other shared resources since a single thread does not usually have sufficient instruction-level parallelism. A coarse-grained multi-threaded processor does not interleave instructions from different threads in processing, while a fine-grained one allows for a cycle-to-cycle interleaving from different threads' instructions, but neither permits instructions from different threads to be issued in the same clock cycle. SMT takes this one step further by allowing instructions from different threads to be issued in the same clock cycle in order to exploit the full potential of the shared resources. The most common characteristic of SMT processors is the sharing of key datapath components among multiple independent threads concurrently running at the same time in order to better utilize the resources. 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

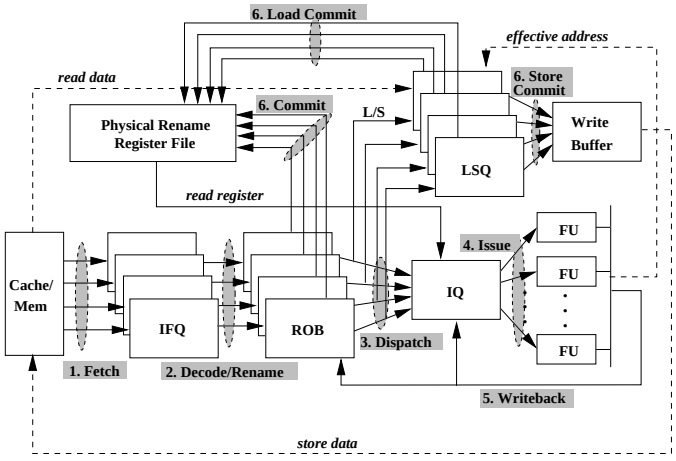

Fig. 1. Pipeline Stages in a 4-Threaded SMT System

Fetch Queue (IFQ). After the stages of *decode* and register-*rename* they are allocated into their respective Re-Order Buffer (ROB) and through the *dispatch* stage into the shared Issue Queue (IQ). Load/Store instructions have their operations dispatched into individual Load Store Queues (LSQ) with address calculation operations also sent into IQ. When the issuing conditions (all operands ready and the required functional unit available) are met, the operations are then *issued* to corresponding functional units and have their results *writeback* to their target locations or forwarded to where the results are needed in IQ. Load/Store instructions, once their addresses are calculated, will initiate their memory operation. Finally all instructions are *committed* from ROB (synchronized with

Load/Store instructions in LSQ) in order.

Essentially SMT improves the overall performance by exploiting Thread-Level Parallelism (TLP) among threads to overcome the limitation of Instruction-Level Parallelism (ILP) presented in a single thread [1], [2]. Subsequently, due to the sharing of resources, the amount of hardware required in an SMT system is significantly less than employing multiple copies of superscalar processors while achieving a similar performance.

There have been numerous research efforts targeting in improving SMT performance by adopting scheduling algorithms for effective resource allocation. Some of them allocate the resources among threads based on the amount of resources that a thread has already hold, e.g. ICOUNT [3] assigns a higher fetching priority to a thread with fewer instructions in pre-issue stages. Cache and memory performance is also taken into consideration in some research works. For example, STALL and FLUSH [4] adopt a fetch policy to address issues from L2 cache misses; a dynamical fetch policy DCRA presented in [5] is a technique based on memory performance of each thread to exploit parallelism beyond stalled memory operations; PEEP [6] controls the fetch unit by exploiting the predictability of memory dependencies. APRA dynamically assigns resources to threads according to the changes of threads' behavior [7]. In [8], an allocation technique on write buffer is proposed for efficient resource occupation by limiting the maximal number of write buffer entries that a thread is allowed to have. Several researchers have focused their work on allocating resource based on threads' speculative situation. SAFE-T (Speculation-Aware Front-End Throttling) in [9] and a speculation control technique in [10] both give higher fetching priorities to threads with higher prediction accuracy. Speculative-Control in [11] improves threads' utilization of IQ by reducing the amount of flush-out instructions caused by miss-speculation. Hill-Climbing [12] is a learning-based algorithm that uses performance feedback to partition the shared hardware resources in the pipeline including IFQ (Instruction Fetch Queue), rename registers, ROB (Re-Order Buffer) and IQ (Issue Queue). All these techniques are either computationally expensive due to the excessive amount of target resources involved in allocation, or cannot reach the intended performance gain due to the myriads of possible combinations of system parameters and workload behaviors, which renders the techniques' adaptiveness very limited.

All the aforementioned techniques dynamically allocate resources (buffers, bandwidths, etc.) according to some set criteria on certain real-time parameters predetermined hoping to adapt to the changing behaviors of threads in real time. These real-time parameters can be, for example, during a time window the ratio between a thread's resource occupancy/allocation compared to the other's, the ratio between the number of instructions past the issue-stage and that before it, or the number of clock cycles memory instructions stall in a buffer, etc. Some criteria are then set on these parameters to trigger the adjustment of resource allocation. Amount of adjustment in the re-allocation process is another uncontrollable parameter highly correlated to all system settings in order for optimal performance. Note that all these criteria are usually heuristically determined, either arbitrarily or through an empirical process testing on a set of workload on a limited scope of system settings. Improvement in performance usually can be delivered from these approaches on the target scope of system settings with the target set of workload. Once the workload characteristics or the system settings are changed, performance delivered by these "adaptive" approaches may not show the desired improvement, if there is still any. It is obvious that the "adaptiveness" of these dynamic techniques is strictly contingent on the effectiveness of the heuristically set criteria. Since the criteria are not adaptive to the varying environment, the associated effectiveness cannot be in any way guaranteed. That is, one cannot simply assume that a complete set of criteria can be pre-determined covering all possible combinations of workload and system settings with each tailored to one combination.

Our goal in this research is to identify a critical shared resource component and develop an allocation algorithm which can dynamically allocate the resource without having to know any of the system settings or workload characteristics beforehand. Note that the common resources in an SMT system shared by threads include various machine bandwidths (e.g., inter-stage bandwidth, read/write ports for register files and memory, etc.), inter-stage buffers (e.g., Issue Queue (IQ)), functional units, write buffer, etc. Due to the significantly large size of each IQ entry and the control complexity involved in the IQ circuitry, 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 and easily becomes the bottleneck in the pipeline that dearly affects the overall system performance.

There have been a few research establishments focused on the *dispatch* stage, including limiting the number of instructions in IQ per thread [13] and recalling instructions of inactive threads from IQ [14]. Delivery of the desired performance gain from these techniques again relies on criteria obtained through empirical test runs, and is only guaranteed for selected workload under the target system settings. For example, the technique adopted by the IQ-capping technique in [13] employs a fixed cap value on the the maximal IQ slots that a thread is allowed to occupy. Although a very significant improvement in performance is obtained averaged among a set of testing benchmark mixes, very different optimal cap values are observed for different mixes and different cap values lead to very inconsistent performance gain (or even loss) when system parameters are varied. Thus, in order to relieve this environment dependency issue, this paper proposes an autonomous process allowing the cap value to self-adjust according to the observed performance fluctuation, hoping to "seek" the transient optimal cap value that constantly changes in real time. Our simulation results show that the proposed technique is able to improve the system's overall IPC under all different possible workloads on various system settings.

2 Simulation Environment

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

section.

2.1 Simulator

We use the M-Sim [15], a multi-threaded micro-architectural simulation environment model, to estimate performance of the proposed scheme. The M-Sim includes accurate models of the pipeline structures such as explicit register renaming, concurrent execution of multiple threads, separate ROB, Load-Store Queue (LSQ) which are necessary for an SMT model. The IQ, functional units and write buffer are shared among threads while register files and branch predictor is exclusive to each thread. The detailed configuration is shown in Table I.

Parameter	Configuration
Machine Width	8 wide fetch/dispatch/issue/commit
L/S Queue size	48-entry Load/Store queue
ROB & IQ size	128-entry ROB, 32-entry IQ
Write buffer size	96-entry write buffer
Function Units & Latency (total/issue)	4 Int Add (1/1) 1 Int Mult (3/1) / Div (20/19) 2 Load/Store (1/1), 4 FP Add (2/1) 1 FP Mult (4/1) / Div (12/12) / Sqrt (24/24)
Physical registers	integer and floating point 256 (for 2/4-thread) / 512 (for 8-thread)
L1 I-cache	64KB, 2-way set-associative 64-byte line
L1 D-cache	64KB, 4-way set-associative 64-byte line write-back, 1 cycle access latency
L2 Cache unified	512KB, 16-way set-associative 64-byte line write-back, 10 cycles access latency
BTB	512 entry, 4-way set-associative
Branch Predictor	bimod: 2K entry
Pipeline Structure	5-stage front-end (fetch-dispatch) scheduling (for register file access: 2 stages, execution, write back, commit)
Memory	32-bit wide, 300 cycles access latency

TABLE I
CONFIGURATION OF THE SIMULATED PROCESSOR

2.2 Workloads

Simulation runs for multi-threaded workloads in this paper all use the mixed SPEC CPU2000 benchmark suite [16] with mixtures of various levels of ILP for diversified representation of workloads. ILP classification of each benchmark is obtained by initializing it in accordance with the procedure mentioned in Simpoints tool and simulated individually in a simplescalar environment. Three types of ILPs, low ILP (memory bound), medium ILP and high ILP (execution bound), are so identified. A number of multi-threaded workloads are used in our simulation with threads of various mixtures of ILP types, as shown in Table II, Table III and Table IV for 2-threaded, 4-threaded and 8-threaded workloads, respectively.

2.3 Metrics

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

$$\text{Overall\_IPC} = \sum_{i}^{n} \text{IPC}_i \qquad (1)$$

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	swim, locus	0	0	2
Mix 2	galgel, twolf	0	0	2
Mix 3	equake, vpr	0	2	0
Mix 4	mcf, gap	0	2	0
Mix 5	bzip2, apsi	2	0	0
Mix 6	crafty, mgrid	2	0	0
Mix 7	swim, mesa	0	1	1
Mix 8	applu, equake	0	1	1
Mix 9	applu, mgrid	1	0	1
Mix 10	twolf, crafty	1	0	1
Mix 11	ammp, bzip2	1	1	0
Mix 12	mesa, apsi	1	1	0

TABLE II
SIMULATED 2-THREADED WORKLOAD

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	swim, locus, galgel, twolf	0	0	4
Mix 2	locus, galgel, twolf, applu	0	0	4
Mix 3	equake, vpr, mesa, ammp	0	4	0
Mix 4	vpr, mesa, ammp, gap	0	4	0
Mix 5	gcc, perlbmk, crafty, mgrid	4	0	0
Mix 6	crafty, mgrid, apsi, bzip2	4	0	0
Mix 7	crafty, mgrid, lucas, galgel	2	0	2
Mix 8	gcc, perlbmk, twolf, applu	2	0	2
Mix 9	apsi, bzip2, equake, vpr	2	2	0
Mix 10	mgrid, apsi, ammp, gap	2	2	0
Mix 11	swim, lucas, vpr, mesa	0	2	2
Mix 12	twolf, applu, ammp, gap	0	2	2

TABLE III
SIMULATED 4-THREADED WORKLOAD

Mix	Benchmarks	Classification (ILP)		
		Low	Med	High
Mix 1	mesa, vpr, equake, applu twolf galgel, lucas, swim	0	0	8
Mix 2	gcc, mcf, gap, ammp, mesa, vpr, equake, applu	0	8	0
Mix 3	bzip2, apsi, mgrid, crafty, perlbmk, gcc, mcf, gap	8	0	0
Mix 4	applu, twolf, galgel, lucas, mcf, gap, ammp, mesa	0	4	4
Mix 5	twolf, galgel, lucas, swim, ammp, mesa, vpr, equake	0	4	4
Mix 6	mcf, gap, ammp, mesa, bzip2, apsi, mgrid, crafty	4	4	0
Mix 7	ammp, mesa, gcc, equake, mgrid, crafty, perlbmk, gcc	4	4	0
Mix 8	applu, twolf, galgel. lucas, bzip2, apsi, mgrid, crafty	4	0	4
Mix 9	twolf, galgel, lucas, swim, mgrid, crafty, perlbmk, gcc	4	0	4
Mix 10	bzip2, apsi, gap, ammp, mesa, twolf, galgel, lucas	2	3	3
Mix 11	bzip2, apsi, mgrid, mcf, gap, applu, twolf, galgel	3	2	3
Mix 12	crafty, perlbmk, gcc, mesa, vpr, equake, lucas, swim	3	3	2

TABLE IV
SIMULATED 8-THREADED WORKLOAD

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,

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

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

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

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

3 Instruction Dispatch

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 (see Figure 1), 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. Traditional simple round-robin dispatching scheme simply starts dispatching all dispatchable instructions from a thread according to 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. As demonstrated in [13] and [14], imbalanced usage of IQ entries among threads and blockage of a significant proportion of them by a stagnant thread is very prevalent. That is, due to the limited size of IQ and the fluctuating behavior of the workloads, most of IQ slots are easily tied up by one thread, and instructions from inactive threads tend to get stuck in IQ for a long time.

4 IQ Capping

IQ-capping [13] is a technique to partially solve the aforementioned problem. The complete dispatch algorithm employing this technique is described in Figure 2. By setting a cap value that limits the maximal number of IQ slots which a thread is allowed to occupy, IQ-capping efficiently prevents over-use or even blocking by any thread; however the process comes with a risk in leading to under-use when the cap value is set too small. There simply does not exist an "optimal" cap value for all arrangements. For example, a system with a fairly large IQ size may not need any capping at all, while, on the other hand, a small IQ size demands a relatively tighter cap value to prevent IQ-starvation from happening. Even with

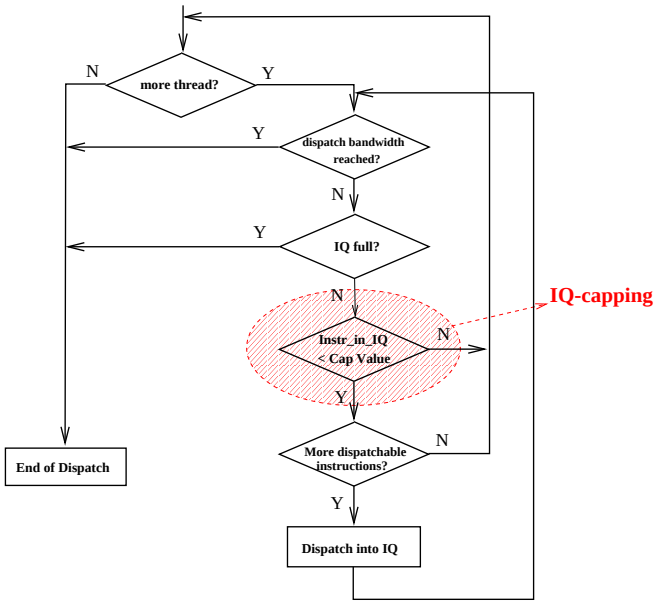

Fig. 2. Flowchart of IQ-Capping Algorithm

the same IQ buffer size, the number of concurrent threads running in an SMT system very much dictates how capping should be employed – the more the threads compete for the buffer, the more the IQ-capping needs to be involved, and thus the "optimal" cap value would drift from one time frame to another. From other system parameters' point of view, one cap value cannot be universally applied to all combinations of these parameters. Any of these system settings, e.g. inter-stage pipeline bandwidth, size of various buffers (e.g. IFQ, write buffer, etc.) or their sharing characteristics, numbers and mixes of various functional units, etc., can have a drastic effect on what the optimal cap value should be. In addition, for those workloads with varying behavior phases, the "optimal" cap value again may change throughout their lifespan as well.

The rest of this section is devoted to the illustration of how all these potential variations do affect the selection of the "optimal" cap values, illuminating the limitation in using such an approach. Throughout the rest of this paper, all techniques are tested in an SMT system as presented in Section 2.1 with the workload mixes described in Section 2.2. Note that, in order for a more "normalized" viewpoint on the cap value when a different IQ size is used, we adopt a variable, so-called "normalized-cap-value" (denoted as d throughout the rest of this paper), as a normalized fraction of the overall IQ size when the IQ is considered to be of 16 equal partitions. Therefore, the actual cap value is $d \times (Q/16)$, where d is an integer with $1 \leq d \leq 16$ and Q denotes the IQ size. For example, when $Q = 32$, a d value of 5 corresponds to an actual cap value of 10. Such a normalized setting is used throughout the rest of our simulation in order to present an unbiased indication of where the cap value should be set as a fraction of the IQ size.

4.1 Variety in Workloads

In practical SMT systems, the number of threads supported varies from system to system and on any given system this

number can actually fluctuate from time to time. This number can easily affect the best cap value to adopt. We first look into the performance of some workload mixes with different numbers of threads in an SMT system with the simple IQ-capping process [13]. Figure 3 depicts the average IPC value

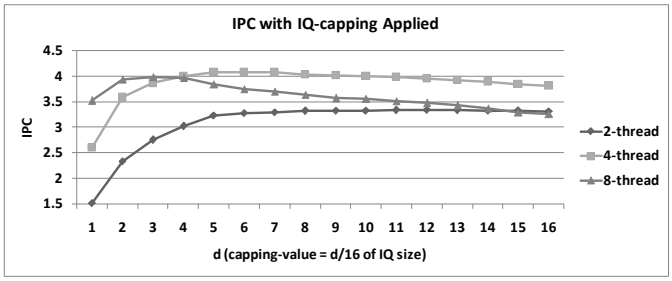

Fig. 3. IPC of Different Number of Threads with IQ-capping ($Q = 32$)

among 12 mixes of different workload mixes with the number of threads set to 2, 4 or 8. As Figure 3 clearly shows, with different number of threads running in the system, the corresponding optimal cap value varies. For example, a 2-threaded workload requires an optimal normalized cap value at $d = 12$, while 4-threaded and 8-threaded workloads see their maximal IPC at $d = 6$ and $d = 3$, respectively.

Even in a system with a fixed number of threads, the optimal cap value varies when different mixes run in the system. Figure 4 shows the IPC of each mix with IQ-capping

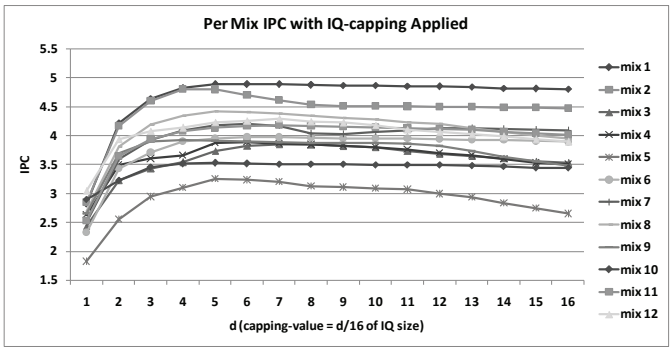

Fig. 4. Per-Mix IPC in a 4-threaded System with IQ-capping

applied in a 4-threaded system, and Figure 5 shows the optimal

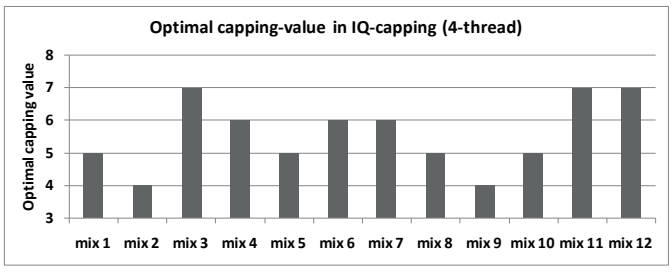

Fig. 5. Optimal Normalized Cap Value for Each Mix

normalized cap value for each mix. As described in these two figures, the optimal normalized cap value varies from 4 to 7 when mixes with various behaviors run in the system, and this variation does not even take into consideration the potential that each individual mix may call for different optimal cap values at different points of time "during" its lifespan when transient behaviors of various benchmark programs intertwine in real time.

4.2 Variety in System Settings

IQ-capping is not only sensitive to the workload running in the system, but also to the system settings as well. With a different buffer size adopted, the optimal cap value may also shift, as demonstrated in the following results. The average IPC value among 12 mixes under different IQ sizes is depicted in Figure 6 and the respective optimal normalized cap values

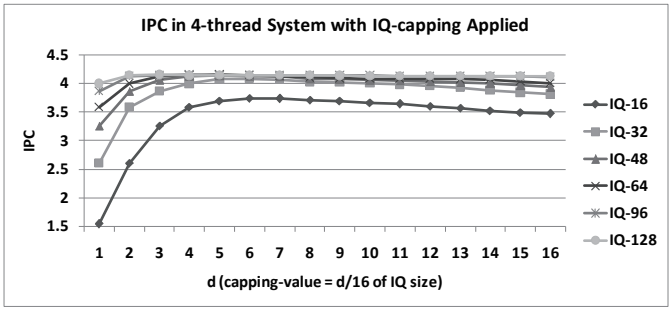

Fig. 6. IPC with IQ-capping with Different IQ Sizes

are shown in Figure 7. From the simulation results we can

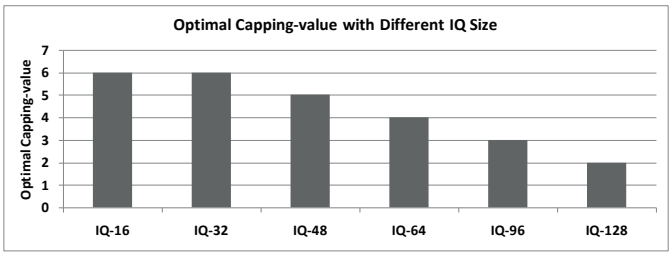

Fig. 7. Optimal Normalized Cap Value for Different IQ Sizes

clearly see that when IQ size increases from 16 to 128, the corresponding optimal normalized cap value gradually shifts from 6 to 2, again demonstrating the clear susceptibility of optimal cap value to any variation in the system.

In a nutshell, IQ-capping is a very effective technique that is capable of preventing any unjust occupation of resource by inactive threads in an SMT system. However, due to the uncertainty in workloads' behaviors and myriads of possible system settings, it is very unlikely to preset a cap value suitable, let alone optimal, for any given system. Setting a cap value under so many uncertainties may not see the intended improvement but can instead easily impede the process of the system. Thus, an enhancement of the IQ-capping method is required to adapt to all possible combinations of the transient behaviors of workloads and the system settings.

5 Proposed Method

The proposed method is an autonomous process to adjust the target setting of control (TSC), the "cap value" in this case, from one window (time frame) to the next based on

some performance indicator(s) periodically monitored. This process can be easily defined by the following algorithmic descriptions:

- monitor a specific system performance indicator (for example, throughput of a pipeline stage) in the "current" time frame (window);
- compare the monitored performance of the current window with that of the previous window;
- determine the direction of adjustment (up versus down) for the TSC (cap value):
  - maintain the same direction of adjustment if the performance comparison indicates an improvement;
  - reverse the direction of adjustment if otherwise.
- repeat the process in the next window.

Essentially this process would incur an automatic adjustment of the TSC periodically based on the performance change observed, aiming at approaching the "optimal" TSC (a transient value) that leads to the maximal performance output gradually and autonomously. A block diagram of this system is depicted in Figure 8, where the system performance is

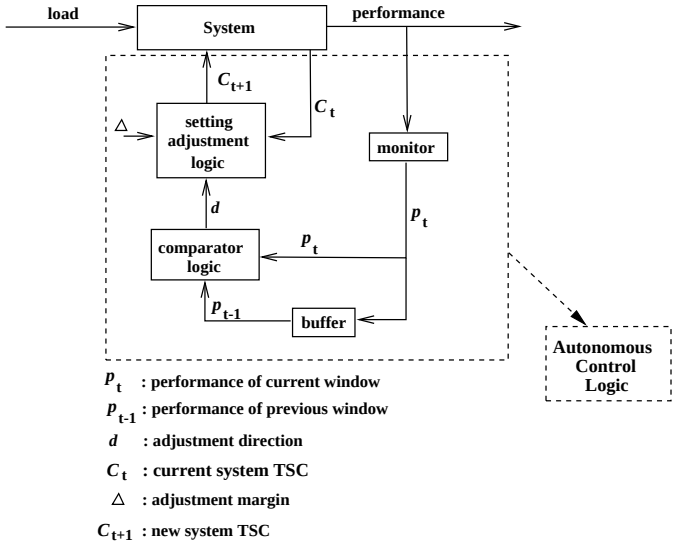

Fig. 8. An Autonomous Control System

monitored (measured) in the current window, denoted as p_t, which is then compared with the performance from the previous window (p_{t-1}) with a comparator logic to determine the next direction of adjustment to the system TSC for the next window. As indicated in the aforementioned algorithmic descriptions, the process will maintain the same direction of adjustment if an improvement is observed and reverse the direction if otherwise. In this research, the two adjustment directions correspond to "increasing" and "decreasing" the current cap value by a preset adjustment margin (denoted as Δ).

The cap value thus determined is then used as a global cap value for all threads in the next window. Figure 9 depicts the dispatching scheme of this autonomous capping process in a 4-threaded system. Once a thread's instruction count in IQ reaches the cap value, no more instructions from this thread

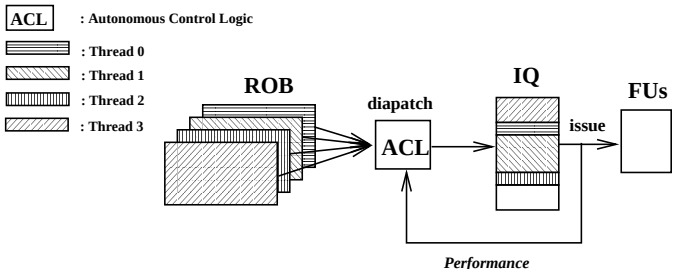

Fig. 9. Auto-capping in 4-threaded System

will be dispatched until the count falls below the cap value again.

Although the whole autonomous process seems very straightforward, there are still a few essential components in it that require special attention for it to function properly, especially in the performance monitoring section. The rest of this section is devoted to the discussion of these components.

5.1 Performance Indicator

Selection of a performance indicator for the process to monitor (measure) is the first step and maybe the most critical one to the success of the process. In an SMT pipeline, there can be many performance indicators to adopt for this process. In order to have the performance measured properly reflect the effect from employing the new TSC (cap value) in the current window, the one selected has to satisfy to the most extent each of the following selection criteria: (note that, unless otherwise specified, the "process" is referred to the autonomous control process)

1) it is directly affected by the employed TSC;
2) its extraction point is located in the pipeline
   a) downstream to the stage where the TSC is imposed at;
   b) close to stage where the TSC is imposed at;
3) it is closely matched by the overall system performance.

Criterium 1 is mandatory since the performance indicator measured is used to judge the merit of newly imposed system TSC. Criterium 2(a) is needed to ensure that the effect is "direct" instead of through other adaptive (feedback) process(es). Criterium 2(b) is also critical since the effect from the adjustment needs to be observed as soon as possible; that is, the temporal lag should be as small as possible between the time when the adjustment is made to the time its effect is measured. If the lag is too large, then the performance measured in the current window does not truly reflect the effect from the current setting. Criterium 3 ensures that the measured performance optimized by the process is also to optimize the overall system performance, essentially the goal of the whole process.

As shown in Figure 9, we choose to adopt the *issue rate* as the performance indicator. Issue rate is defined as the number of instructions issued from IQ in a window. Compared to other potential performance indicators, such as , write-back rate, commit rate (IPC), or even fetch/rename rate, issue rate obviously is the one that satisfies the three criteria the most – it is directly affected by the IQ utilization (criterium 1) and is

measured at a stage downstream to the dispatch stage where the control is imposed (criterium 2(a)) and the closest to it (criterium 2(b)). Also, since the execution in functional units is an "unblocked" process, the issue rate can be considered a close match to the write-back rate which is again very similar to the commit rate, the overall system throughput (criterium 3).

5.2 Performance Sampling Window

Performance sampling window size, in processor clock cycles, is another important parameter because it affects the process' adaptivity dramatically. If the window size is set too large, then the adjustment may not adapt quickly enough to changes in the workloads' resource demands. On the other hand, if the window size is set too small, performance measured in a window may become too dynamic, rendering the performance comparison result between two adjacent windows arbitrary and unreliable. Thus, to ensure that the performance measured reflects closely the effect from the imposed setting adjustment without being polluted by a sudden change of a thread's behavior, window size needs to be sufficiently large to tune out these noises. Note that there are two main causes for a thread to suffer a sudden performance drop: an L2 cache miss and a miss speculation. An L2 cache miss requires accesses to the main memory, easily with a latency of more than a few hundred cycles in a typical system. A miss speculation leads to a flush-out of the wrong-way instructions from all system buffers including IFQ, ROB, IQ and functional units, which then would take some time for the thread to refill the pipeline buffers. The window size in the proposed mechanism is set to be $1,000$ clock cycles based on these two concerns to allow for a reliable performance measurement.

5.3 Qualified Performance Sampling

To further address the concerns aforementioned in the previous section regarding the cache miss and miss speculation, one may choose to screen out performance samples measured in some clock cycles when the said perturbation comes into play. Note that once a thread encounters either of the situations, the thread will normally stop dispatching instructions either right away (under miss speculation) or in the next few clock cycles (under cache miss). Thus, dispatching activity usually is a good indicator to identify between an active or stagnant (stalled) thread. In order to prevent such a misleading sampling situation when issue rate encounters a drastic decline due to all threads becoming stagnant (from either cache miss or miss speculation), only the issue counts in the clock cycles when at least one thread is dispatching are considered qualified samples. The formula for the qualified performance in a window is then

$$\text{Performance} = \frac{\sum_{Q\_cc} \text{num\_issue}}{\text{num\_}Q\_cc} \tag{4}$$

where Q_cc denotes each qualified clock cycle, num_issue is the number of instructions issued in the corresponding qualified cycle and num_Q_cc is the number of qualified cycles in a window. If, in some extreme cases, there is no qualified cycle in a window span, this window will then be simply ignored (as a "null window") without any adjustment made by the controller; that is, performance comparison is made only between two nearest non-null windows.

5.4 Comparison Threshold for Adjustment

To avoid randomly adjusting the TSC (cap value) under even a slight variation in performance between two windows, a minimum threshold is imposed to the difference in performance for initiating an adjustment. That is, the difference in performance between the two adjacent windows should be greater than the set threshold to be considered valid; otherwise, the system's performance is assumed to be static between the two windows and no adjustment is made. With this threshold the process should be able to eliminate random adjustments from slight fluctuation of performance. In our proposed technique, the threshold is set as 3% of performance from the previous window. That is, the change in performance between the two windows should be greater than 3% of the previous one to lead to an adjustment. Due to these potential non-adjustment windows, the 3% threshold is referred to the most recent window with a made adjustment.

Assume that p_t represents the performance measured in the current window, while $p_{t'}$ denotes that of the most recent window with an adjustment, with τ being the threshold. In essence, the autonomous controller will take the respective action as in the following:

1) continue to adjust the TSC in the current direction if $p_t - p_{t'} \geq \tau \times p_{t'}$
2) reverse the direction of adjustment and adjust the TSC if $p_{t'} - p_t \geq \tau \times p_{t'}$
3) make no change to the direction nor any adjustment if $|p_t - p_{t'}| < \tau \times p_{t'}$

6 Simulation Results

Based on the simulation environment and the workloads described in Section 2, our proposed technique is tested compared to the default dispatching system. Figure 10 shows

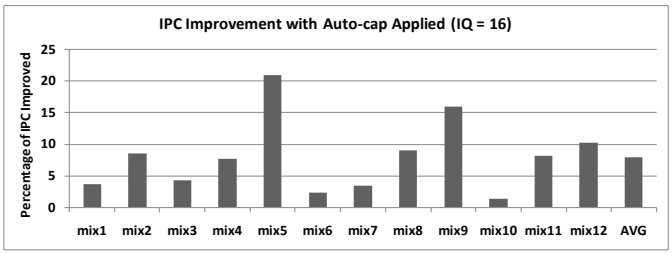

Fig. 10. Per-mix IPC Improvement with Auto-capping ($Q = 16$)

the percentage of IPC improvement for each mix with the proposed technique applied in a 4-threaded system with a 16-entry IQ. The simulation result clearly indicates that the proposed autonomous capping technique (denoted as auto-capping) improves the overall performance in all mixes, with a maximal IPC improvement of 20.8% for mix5 and an average improvement of 7.9% among all mixes.

Effectiveness of the auto-capping process is further tested in systems with various numbers of threads and/or with different system parameters. Figure 11 describes the IPC improvement in the systems with a 2-threaded, 4-threaded and 8-threaded

workload and systems with IQ size varying from 16 to 128. Under different numbers of threads with IQ size fixed at 32, the

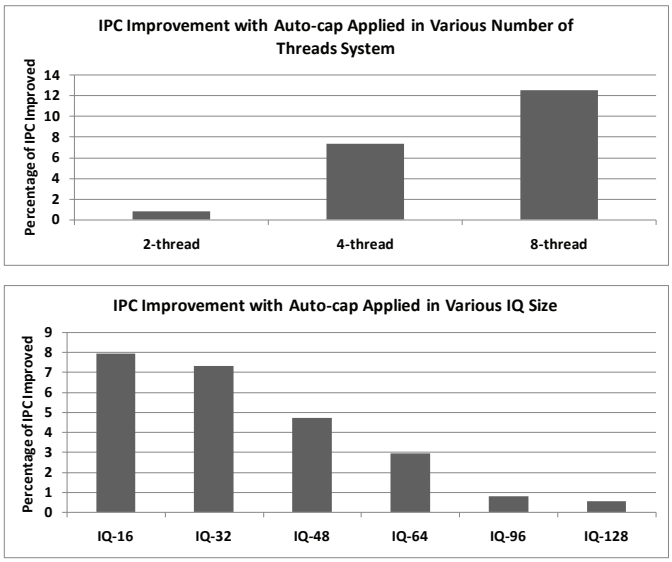

Fig. 11. IPC Improvement with Auto-capping in Various Systems

proposed auto-capping method achieves positive improvement in all cases, with a 12.5% increase in IPC for an 8-threaded system. The technique is more effective when more threads are in the system competing for the fixed amount of resources. When the size of IQ is varied with 4 threads running, again the technique achieves all positive improvements, with up to 8% when $Q = 16$. The effectiveness of the technique is clearly more prominent when the IQ size is smaller, which again can be explained by the higher degree of competition. These results clearly indicate that the auto-capping is capable of improving system's overall performance under any combination of workloads and system settings without having to know any of these varying parameters beforehand.

Further looking into these two results, one can easily deduce that when there is little competition for the resources (IQ entries) such as when there few threads running and/or there are sufficiently large amount of resources, arbitrarily capping the use of IQ may easily backfire and limit the potential system throughput, which is the reason why a fixed capping approach should not be considered a viable solution. A comprehensive comparison between the proposed auto-capping approach and the fixed capping one is presented in Figure 12. Two different performance average values from the fixed capping approach are used for comparison. First, an average of performance using different fixed normalized cap values ranging from $d = 1$ to $d = 16$ (that is, the cap value ranging from $\frac{1}{16}Q$ to $\frac{16}{16}Q$) is obtained. Secondly, to allow the fixed capping approach a more fair competition, results from any obvious under-use of IQ due to "over-capping" should be discarded. Over-capping may happen when the fixed cap value is set below Q/n where n is the number of threads in the system, in which case a portion of IQ entries are always left unused. Thus, in the second result, an average of performance from $d = \frac{16}{n}$ to $d = 16$ (that is, the cap value ranging from $\frac{1}{n}Q$ to $\frac{16}{16}Q$) is used. In the figure, these two fixed-capping results are denoted as "fix-cap

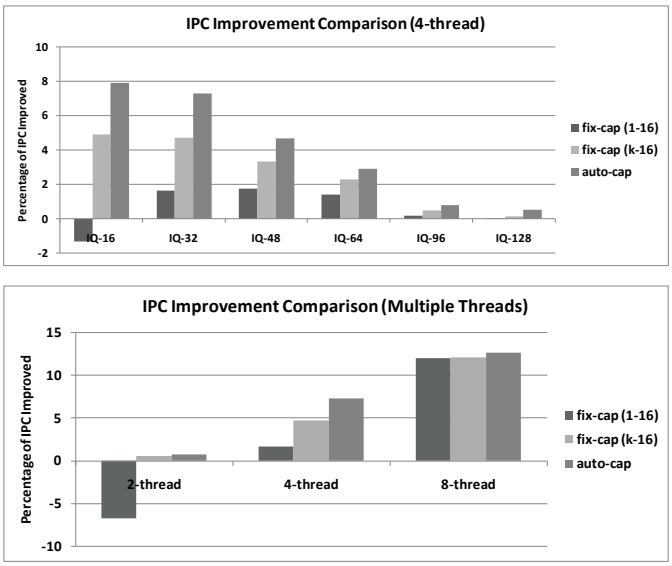

Fig. 12. IPC Comparison between Auto-capping and IQ-capping

(1-16)" and "fix-cap (k-16)", respectively, where $k = 16/n$. These comparison results clearly show that auto-capping is consistently superior to fixed capping, under all combinations of system parameters and workloads. The superiority is even more evident in the cases when there are relatively few threads in the system or when the size of IQ is relatively small.

In order to further verify how the auto-capping technique transpires during the simulation in its adjustment of the cap value so as to lead to the desired improvement, another analysis is carried out to show the distribution of cap values delivered by the process. One would imagine the process delivers a different spectrum of cap value distribution for a different set-up of system parameters. Figure 13 displays the

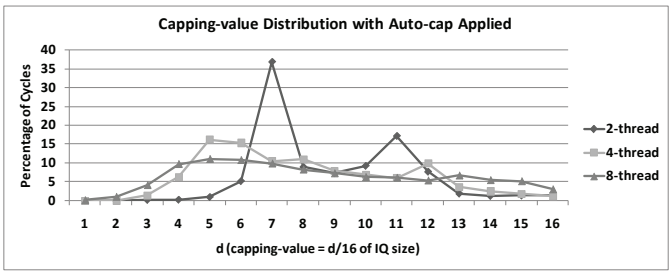

Fig. 13. Distribution of Cap Values by Auto-Capping in 2-threaded, 4-threaded and 8-threaded Systems

distribution of cap values in 2-threaded, 4-threaded and 8-threaded systems. We notice that, in a general trend, when the number of threads increases, auto-capping tends to adjust the cap value to a smaller range, which clearly demonstrates the true adaptivity of the process in adjusting itself to cope with different degrees of congestion. In all three distribution curves there exist multiple "peaks' of congregations, which can be explained by the transient behaviors of workloads that cause the process to tune itself toward certain congregate points at different times during the workloads' lifespan.

In order to reflect the degree of execution fairness among

the threads when the proposed technique is applied, Figure 14 shows the percentage of improvement in Harmonic IPC when

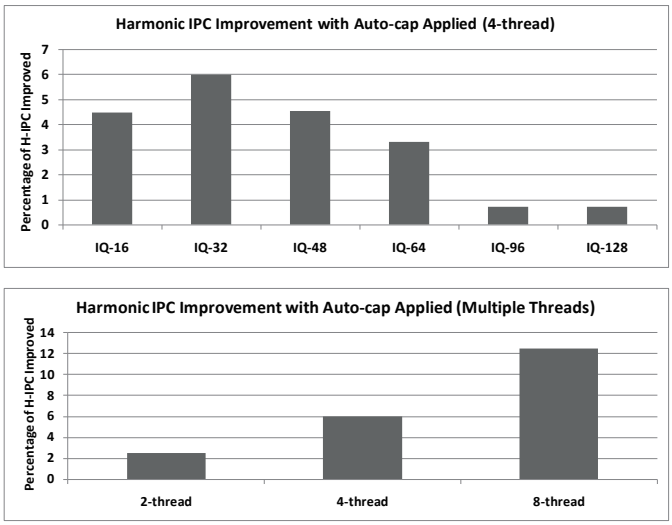

Fig. 14. Harmonic IPC Improvement with Auto-capping

auto-capping is applied to various systems. Similar to the simple "aggregate-IPC" performance result, improvement tends to decrease when the competition for the resources becomes less – with a larger IQ or with fewer threads. In a system with more threads, the effect in balancing resource among threads brought by the proposed technique is much more significant. For example, in a 2-threaded system, our algorithm gains a harmonic IPC improvement of 2.5%, while in a 8-threaded system, the improvement can be as high as 12.3%.

7 Conclusion

This paper presents an autonomous technique for optimizing resource distribution in an SMT system. We clearly show that the proposed technique is capable of adjusting itself to various system parameter settings and different workloads. This represents a very significant improvement over all the adaptive techniques in the literature which require certain system parameters known *a priori* for optimal performance. This autonomous process can be even applied to any other shared resource components in the system, for example, the write buffer and the inter-stage bandwidth. There are also potential improvements that can be made to this autonomous process in better qualifying performance sampling, in taking into consideration the lag between the time new setting is imposed to the time its effect can be correctly sampled, and several other aspects in window size selection or even with a dynamic window size. All these potentials are currently being explored for future dissemination.

Acknowledgment

This material is based in part upon work supported by the National Science Foundation under Grant Number HRD 0932339. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

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

[2] D. Tullsen, S. J. Eggers and H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", *In the Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 392-403, May 1995.

[3] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M .Levy, J. L. Lo 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*, pp. 191-202, May 1996.

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

[5] 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, December 2004.

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

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

[8] Y. Zhang and W.-M. Lin, "Write Buffer Sharing Control in SMT Processors", *2013 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'13)*, July 2013.

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

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

[11] Y. Zhang and W.-M. Lin, "Capping Speculative Traces to Improve Performance in Simultaneous Multi-Threading CPUs", *The 18th International Conference on Parallel and Distributed Processing Techniques and Applications (IPDPS'13) Workshop on Multithreaded Architectures and Applications*, May 2013.

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

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

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

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

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