SMT QoS: Hardware Prototyping of Thread-level Performance Differentiation Mechanisms

Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten and Martin Dixon
Intel Corporation, Hillsboro Oregon
Contact Author: Andrew.J.Herdrich@Intel.com

ABSTRACT
Absolute throughput often fails to scale linearly with core count in chip multiprocessors (CMPs) due to contention in shared platform resources, including cache, memory bandwidth and busses. This nonlinear scaling is exacerbated by the addition of simultaneous multithreading (SMT) to CMPs by introducing resource contention at the pipeline resource level, and increasing the number of active threads in the system which further increases contention in shared resources, leading to a loss in performance stability and fairness.

This work introduces and evaluates a new form of source-based execution rate control at the processor pipeline level, based on instruction fetch, instruction queue and reservation station partitioning between SMT threads. The efficacy of these controls is demonstrated through experiments with SPEC workloads on a modified test version of an Intel® codename *Nehalem* microprocessor. This new SMT rate control is presented as a critical building block to restoring the fairness and determinism in performance once inherent in simpler uniprocessors utilizing time-slicing schedulers, and is proposed for inclusion in future microprocessors supporting SMT.

Categories and Subject Descriptors
C.1 [**Computer Systems Organization**]: Processor Architectures; C.4 [**Computer Systems Organization**]: Performance of Systems – *Measurement techniques*, *Performance attributes*; B.8.0 [**Hardware**]: Performance and Reliability – *general*.

General Terms
Management, Measurement, Performance, Design.

Keywords
Intel Nehalem (NHM), Simultaneous Multithreading (SMT), Chip-Multiprocessors (CMP), Performance, Power, Quality-of-Service (QoS), Cache, Memory, Rate Control, P-States, Q-States, Performance Differentiation, ACPI.

1. INTRODUCTION
In this paper we examine and directly address shared pipeline resource contention in SMT platforms through the use of a new form of rate control mechanism at the microarchitectural pipeline level between SMT threads on a processor core. The first component of this rate control acts by re-steering the frontend using both biased instruction fetching and decoding. The second mechanism acts through the redistribution of reservation station entries between threads, which effectively changes the size of the out-of-order window that a given thread executes within. When combined with ACPI T-States (commonly used for thermal runaway control) in a novel series of heavily multithreaded tests performance gains of nearly 20% are realized, as demonstrated in Section 4.3. We further show that in the case of a heavily loaded system that this technique can be extended to increase total system throughput.

The remaining sections are organized as follows: Section 2 details the need for QoS technologies at a thread level with a brief overview of the thread contention problem; Section 3 gives an overview of the proposed new SMT rate control mechanisms; Section 4 provides a detailed analysis of the effectiveness of the proposed SMT QoS mechanisms, including highly threaded cases.

2. THE NEED FOR QoS TECHNOLOGIES
With the introduction and proliferation of CMPs in the last decade, resources that were once dedicated to a particular hardware thread such as caches and interconnects are now shared, and with the addition of SMT now internal microprocessor resources such as reservation stations, ROB entries and execution units are shared as well. Traditional OS-level time-slicing thread prioritization methods do not suffice in light of these new architectures since a scheduler will simply observe that cores are available and schedule threads on them whether thread contention is an issue or not.

2.1 CMP and SMT Scaling
One classic method to illustrate the shared resource contention inherent in a system is to run an increasing number of identical workload threads and measure total system throughput as a function of the number of threads [5, 11]. Ideally throughput scales linearly, and in the case of compute-intense workloads that require little cache space such as Eon from the SPEC CPU2000 suite (*Figure 1*) performance scales as expected on the quad-core test processor at 3.2GHz with 2-way SMT.

A decrease in slope is evident moving from four to five threads and beyond as SMT contention is introduced, as is a slight decrease at eight threads as OS threads are forced to share compute time with the eighth instance of the workload and multi-level cache contention increases.

In the case of highly memory-intense applications such as CPU2000 Swim [10] performance trails off after just a few copies of the workload are instantiated as last-level cache space and memory bandwidth become the primary performance-limiting factors (locking and synchronization issues are nonexistent since the workload threads are independent). Some cache-intense applications such as CPU2000 Art (*Figure 1*) decrease in system throughput as more threads are added since this application becomes highly memory-bandwidth sensitive as the cache fills quickly to its maximum occupancy. These simple test cases

illustrate that sharing of platform resources on a CMP platform can artificially constrain throughput, a point that will become even more important as core counts continue to scale upward.

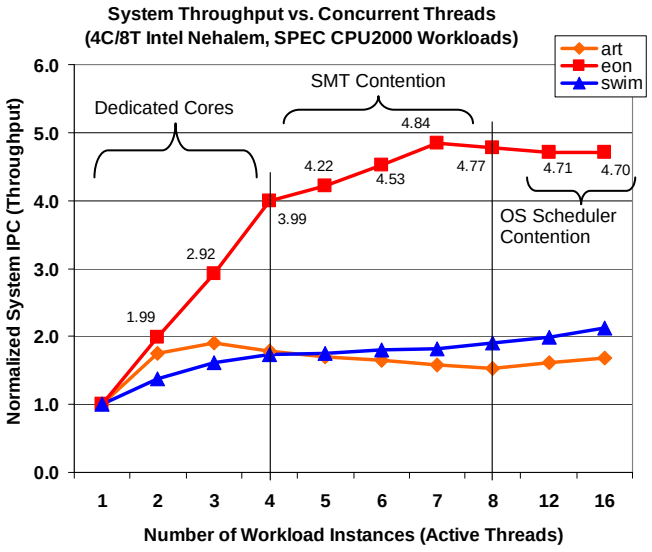

Figure 1. System throughput for three SPEC CPU2000 workloads vs. active thread count.

Total system throughput is only one aspect of the platform behavior, however—the primary concern of this work is the individual performance of a workload running on a system with multiple workloads executing simultaneously and with SMT contention.

Though SMT implementations in industry have been shown to potentially improve the throughput of a system by 30% or more [1,2] and provide compelling power/performance benefits [2], they currently lack mechanisms to provide performance differentiation between threads.

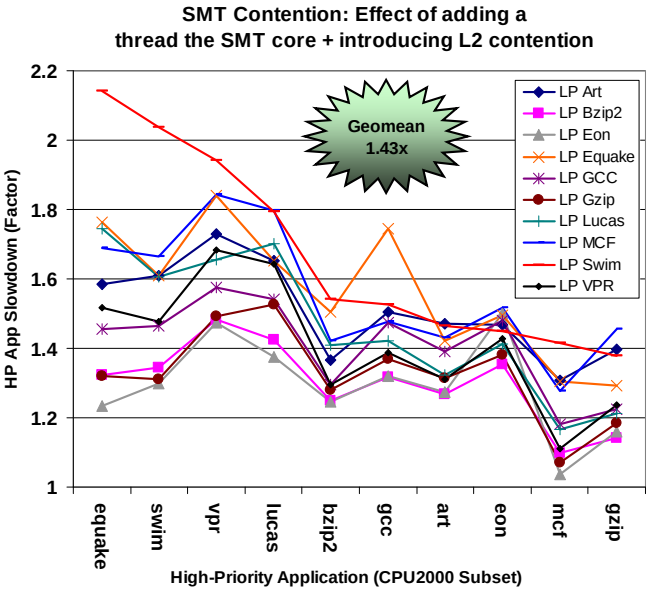

Figure 2. Impact of shared platform resources (Pairwise SMT contention, No QoS, triple-channel DDR3-1067)

This point is illustrated in *Figure 2*, showing the slowdown of a high-priority (HP) application due to SMT contention. In this case two applications share a physical core (bound to OS logical CPUs) but the system is otherwise idle. The workloads selected in this example are all from a subset of the SPEC CPU2000 suite, which despite its age includes several applications that still manage to saturate the memory bandwidth of a modern three-channel DDR3 system [6]. The decision of which applications to choose when subsetting the SPEC suite is based on the work presented in [8,9,10] to provide a mix of compute, memory and cache-sensitive apps.

If an OS scheduler places a high priority workload on OS CPU1 (core 1, thread 0) and a different low priority workload on OS CPU5 (core1, thread 1), SMT and L1 and L2 cache contention are introduced. In situations where an OS scheduler creates such a condition as the non-ideal pairwise Equake-Swim case (resulting in the 2.14x Equake slowdown in *Figure 2*) it becomes very desirable to provide priority enforcement mechanisms to control the rate of each executing thread both dynamically and deterministically; the new SMT rate control mechanisms presented in later sections are key to reaching this goal.

2.2 Shared Platform Resources

The slowdown discussed in previous sections can be attributed entirely to shared resources within the platform. In the case of *Figure 2* only one physical core is active so the two active threads share memory bandwidth, cache space at all levels and internal processor busses and pipeline resources such as reservation stations and functional units. The thread contention discussed herein is common across many other similar CMP architectures and is not limited to the current CPU architecture under test.

Table 1. Details of the microarchitecture and the system under test.

System Configuration Parameters	
Core Clock Speed	3.2GHz
QPI Rate	5.87GT/s
Uncore Clock	2.1GHz
Memory	DDR3-1067 / 3CH
Reservation Stations	36
Instruction Queue	18 Entries
LD/ST Buffs / Reorder buffers	48/32 / 128
L1i + L1d Caches	Split 32KB+32KB
L2 Cache	256KB / 8 Way
L3 Cache	8MB / 16 Way

3. QoS METHODS AS A POTENTIAL SOLUTION

One method to restore a deterministic performance differentiation between SMT threads is to make use of Quality-of-Service (QoS) methods to provide mechanisms for the hardware to control the execution rate of a given SMT hardware thread relative to another on the same core. Thus, an operating system scheduler or other entity with system-level performance information could redistribute pipeline resources from a lower priority application to a higher-priority application, providing the fundamental functionality required to restore deterministic performance differentiation between threads on SMT-enabled CMP platforms.

3.1 Resource Control vs. Rate Control

Previous attempts to control thread contention and restore guarantees of execution speed and QoS broadly fall into two categories: (1) those focusing on controlling resource partitioning between threads, such as cache space, and (2) those focusing on controlling execution speed of individual threads to limit resource request generation and thus thread contention (rate control). Previous work has largely concluded that resource control requires careful control of all resource levels in the hierarchy (L2, L3, etc.) to prevent priority inversion, and rate control is generally preferable [6]. In this paper a new form of rate control is presented, which has the significant advantage that it provides variable balance between threads within an SMT core, potentially allowing a scheduler to make intelligent biased resource decisions and enforce them.

3.2 New SMT Rate Control Mechanisms

Several new hardware SMT rate control mechanisms are introduced in this paper, some of which have been studied in simulation in the past. Previous research has relied heavily on simulators such as SMTSIM and the use of Instruction Fetch and Instruction Queue throttling to control the redistribution of pipeline resources between threads, and many of these papers have then gone on to study various instruction fetch or resource distribution algorithms to provide performance differentiation such as IFETCH [5], or they extend fetch policies with cache profiling and prediction (such as DCache Warn, [12]). The type of Instruction Fetch (IF) and Instruction Queue (IQ) throttling provided on the Intel test system is similar to that used in past simulation-based work and is also somewhat similar to the type implemented in the IBM POWER5 Architecture [3,4], but the Reservation-Station (RS) based rate throttling approach advocated in this paper and has only been briefly studied previously (in simulated environments such as in [7, 14, 15]) and not in hardware. This feature is not commonly available on Nehalem-based products, and is only present on specialized test processors. To our knowledge the RS-based throttling approach has never been tested on real hardware before. The following sections present an overview of the rate control mechanisms, followed by a proposed unified standards-based architecture and supporting experimental results.

3.2.1 Instruction Fetch Throttling

Modern instruction fetch (IF) unit implementations such as that used in the Intel frontend (*Figure 3*) monitor threads and fetch instructions for the next ready thread, avoiding fetching instructions for a thread which is stalled waiting on long-latency memory requests or slow pipeline execution units. Though efficient, this method provides no guarantee of the number of instructions fetched for a particular thread, and thus no notion of thread priorities.

The instruction fetch algorithm described above can be augmented to enable QoS priority levels by simply weighting the unit to fetch favoring one thread over another with a programmable N:M ratio, still favoring a ready thread over a stalled one. The Nehalem IF unit includes such a feature, which allows programmable ratios with 7 levels of granularity, from 1:7 to 7:1 in fractional 0.125 linear increments (unlike the exponential knobs provided in the POWER5 [3]). Note that the middle state, 1:1, is the default state of the processor as it boots. Henceforth for simplicity we denote the ratios as fractions expanded in decimal form, such as 0.125 to define a ratio of 1/8.

A simplified view of the pipeline is shown in *Figure 3* (based on [13]) to illustrate how the instruction fetch (IF) and other QoS mechanisms are implemented.

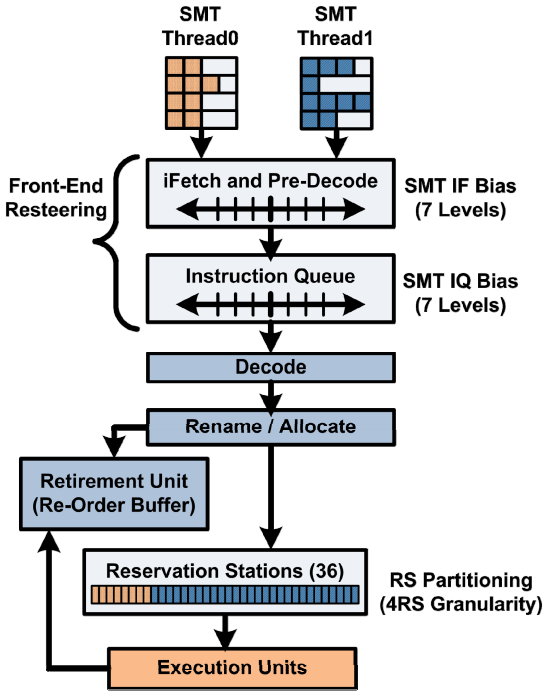

Figure 3. Basic pipeline structure, including the functional units in which Instruction Fetch (IF), Instruction Queue (IQ) and Reservation Station (RS) SMT QoS are implemented. (Adapted from [13])

3.2.2 Instruction Queue Throttling

Another useful place to implement SMT QoS rate control mechanisms is in the Instruction Queue (IQ), where x86 instructions wait to be decoded into the internal format used within modern x86 processors. The location of the IQ unit in the pipeline is shown in *Figure3*. The specific implementation simply favors picking instructions out of the queue for one thread or another in a programmable M:N ratio in precisely the same way as the previously discussed IF throttling mechanism. Since IF and IQ throttling are most effective when used together, generally we combined them and discuss their additive impact together.

3.2.3 Reservation Station Partitioning

One of the most critical microarchitectural pipeline resources are the Reservation Stations (RS), the decoupled storage array used to hold operands just before issuing them to the execution units. The number of reservation stations allocated to a given thread plays a large role in determining its effective instruction window, which directly relates to how much out-of-order benefit a thread can achieve. Some threads will also stall in the RS if they issue too many concurrent memory requests, which is one of the most compelling benefits of QoS partitioning of reservation stations.

In the case of the test processor each of the 36 reservation stations can either be assigned to Thread0, Thread1, or marked as shared. A set of mask registers controls the allocation, with a granularity of two RS.

Note that in later plots a notation of "X-Y-Z" is adopted to denote the number of reservation stations devoted to Thread0 (X), Thread1 (Y) or marked as shared (Z). The default allocation case as the machine boots guarantees 8 entries for Thread0 and 28 shared entries, denoted as 8-0-28 in this abbreviated notation.

4. SMT QoS EVALUATION

An evaluation of the instruction fetch (IF), instruction queue (IQ) and reservation station (RS) SMT QoS mechanisms provided in the test processor follows. The benchmark applications selected are from the SPEC CPU2000 suite and were run entirely to completion. Variation between runs is low—typically less than 3%. The next section details combined IF and IQ QoS results. An examination of RS QoS follows, then all three QoS mechanisms are combined additively.

4.1 Combined Instruction Fetch and Instruction Queue Results

Combining IF and IQ QoS into a single mechanism where settings for both are moved in lockstep re-steers the entire front-end; individual IF and IQ sensitivity studies are omitted in the interest of brevity. As shown in *Figure 3*, it is possible to realize a 20% performance gain in Eon while reducing the performance of the low-priority thread by roughly 40% through the use of IF/IQ SMT QoS.

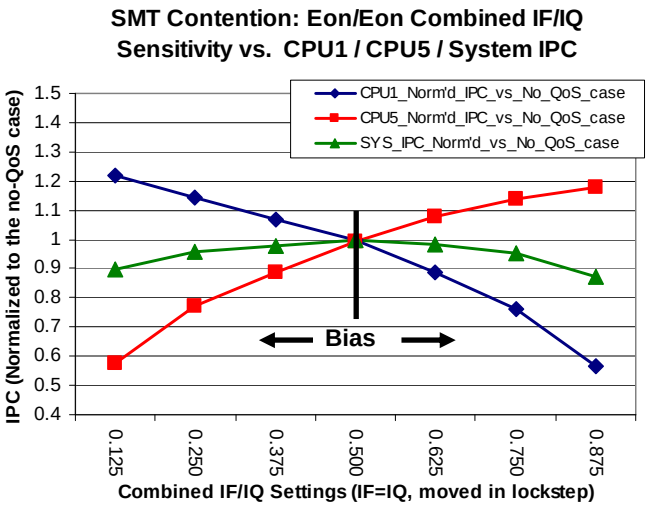

Figure 4. CPU2000 *Eon*, a compute-bound workload, shows moderate sensitivity to combined IF and IQ SMT QoS.

Note that memory-intense Swim (*Figure 5*) shows little sensitivity to either mechanism since the front-end is still sufficiently wide (4 or more instructions nearly all the way from fetch to the execution engines) that even throttling heavily so that it only receives 1/8 bias (such that it only receives an instruction in the decode stage every other cycle) is not sufficient to stop it from clogging the memory subsystem and L2 cache.

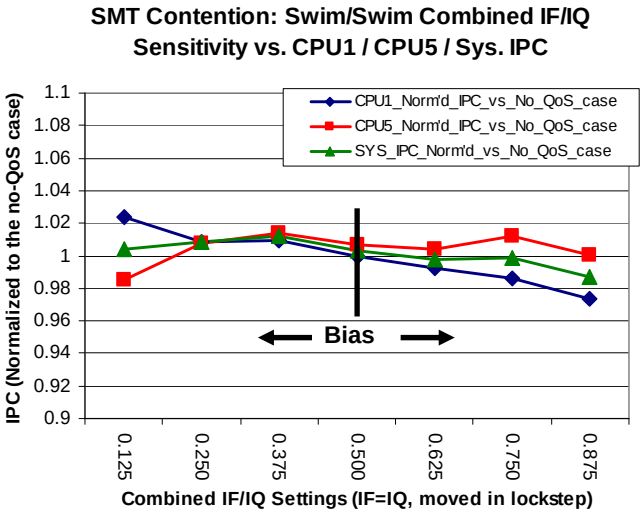

Figure 5. CPU2000 *Swim* shows little sensitivity to combined IF/IQ SMT QoS. (Note: Fine axis scale)

Examining combined IF and IQ results with two threads of a more common workload (GCC) in SMT contention yields similar results to Eon, where a performance gain of 15% is possible for the high-priority thread, while the low-priority thread suffers a 20% performance loss.

4.2 Reservation Station Scaling

As the system boots, a total of 8 RS entries are reserved for thread0 and 28 are shared (denoted 8-0-28 using the notation defined in Section 3.2.3).

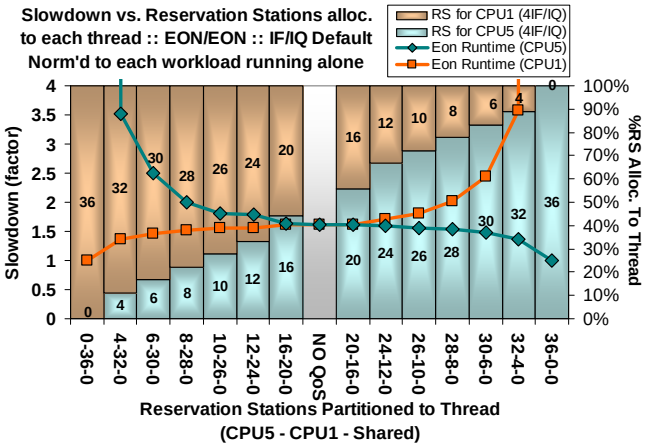

Figure 6. Reservation stations allocated to each *Eon* thread vs. runtime. Note that the bars in the background represent the static RS partitioning count for easy visualization, and the cases on the end where the workload runs alone represent the baseline cases.

Shown in the background of each RS result plot are bars which represent the number of RS partitioned to each thread. In the case of the right and left-most bars only a single workload is run, so this represents the runtime of the application alone, which provides the normalization point for all data in the plots.

Examining the results in *Figure 6*, where two instances of the CPU2000 workload Eon (a ray-tracing workload) are run together with SMT contention as more reservation stations are allocated to the thread on CPU1 its runtime reduces significantly and in a rather predictable exponential fashion. When only 4 reservation stations are allocated to Eon it requires over 3x the runtime as when run alone, and when statically allocated a total of 32 RS entries it has nearly the same runtime as the case where it runs alone, despite the background eon instance being present.

4.3 Expanding to 4 Cores and 8 Threads

In the case of *Figure 7* four instances of Art contend with four instances of Swim. Through the use of SMT QoS it is possible to bias against the Swim instances and redistribute more shared platform resources to Art, which uses them more efficiently, leading to a total system throughput gain of up to 1.09x. Note that in the figure the test cases on the abscissa simply denote all of the SMT QoS combinations possible, sweeping through RS and IF/IQ settings, where IF/IQ are moved in lockstep, then the data is sorted by total system IPC and normalized to the no-QoS (full contention) case.

In analyzing the settings that produce a net positive system throughput improvement the average IF/IQ setting is 0.483, with an average of 20.7 RS allocated to the HP app, 10.3 RS allocated to the LP app, and 5.0 shared. Though not meaningful by themselves, these statistics indicate that RS repartitioning is largely responsible for the cases where a net throughput gain is observed. For cases that show a negative system throughput benefit the overarching trend is that these cases allocate more reservation stations to Swim instead of Art, so in a real scheduler implementation it would be critical to give more reservation stations to the application that uses them the most effectively.

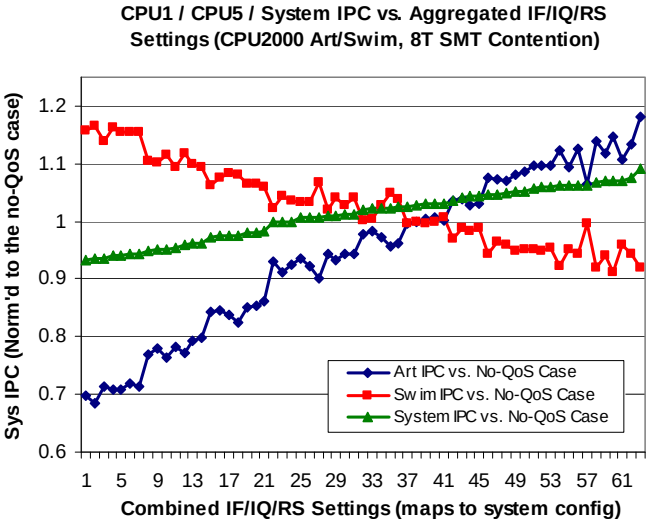

Figure 7. Four instances of *Art* contending with four instances of *Swim* with various SMT QoS settings. In cases where we allocate more reservation stations to *Art*, system throughput increases, by up to 9%, while the performance of the high-priority application improves by nearly 20%.

4.4 Experimental QoS Results Summary

IF and IQ throttling work well when grouped and moved in lockstep. The usefulness of combined IF/IQ throttling is limited since memory-intense workloads can still fill memory queues and starve compute-bound and other workloads. A more generally applicable (and novel) throttling mechanism is RS partitioning, which allows a high degree of granularity and affects both compute and memory-bound workloads in a similar manner. When combined IF/IQ/RS SMT QoS can be quite powerful, and can even provide a significant system IPC (throughput) boost when a system is heavily loaded. The addition of features such as ROB partitioning could further increase the utility of this feature, allowing the degree of out-of-order execution of a thread to be finely controlled. Note that these results vary somewhat from those presented in [3] by Boneti et. al. due to the substantial differences in microarchitectural features, layout and implementation between the Intel Nehalem test chip and the IBM POWER5.

5. CONCLUSIONS

While the performance of multithreaded applications theoretically scales linearly with core count, experimentally this is not the case due to contention within the shared resources of a platform, including cache space and memory bandwidth. Adding SMT to a processor, while beneficial in terms of power-to-performance ratios, can increase the number of active threads on a system, further exacerbating the shared resource issue. Traditional solutions such as over-provisioning memory bandwidth and cache space are insufficient to remedy the problem since mobile and consumer segments will not tolerate the added cost, die area and power consumption. This necessitates the introduction of QoS-inspired solutions, namely instruction fetch, instruction queue and reservation station based methods, which enable practical thread-level priorities. Under heavily loaded system conditions an improvement in system throughput of up to 9% is possible using a combination of IF/IQ/RS SMT QoS and performance improvements of over 20% are possible for the highest-priority threads. The proposed architecture was validated and demonstrated with experiments on a test version of an Intel Nehalem microprocessor. Further work remains in improving and expanding current scheduling algorithms to make use of these new capabilities and "closing the loop" by providing detailed performance feedback information to a system or user agent to consistently improve the quality of SMT QoS enforcement decisions passed down to the hardware.

6. ACKNOWLEDGMENTS

The authors would like to thank the Intel Oregon CPU Architecture Team (ORCA) and Nehalem implementation teams, especially Santhosh Srinath, Morris Marden, John Holm, Glenn Hinton and Matt Merten (also an author), who conceived of and implemented the hardware mechanisms, as well as helped provide test hardware.

7. REFERENCES

[1] Intel Corporation. "First the Tick, Now the Tock: Next Generation Intel Microarchitecture (Nehalem)," http://www.intel.com/pressroom/archive/reference/whitepaper_nehalem.pdf

[2] D. Marr et. al., " Hyper-Threading Technology Architecture and Microarchitecture," in the Intel Technology Journal, Vol. 6, Issue 1, http://developer.intel.com/technology/itj/index.htm

[3] C. Boneti, A. Buyuktosunoglu, F. J. Cazorla, C. Cher, R. Gioiosa, and M. Valero. “Software-Controlled Priority Characterization of POWER5 Processor”, (ISCA), 2008.

[4] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. “POWER5 System Microarchitecture”. IBM Journal of Research and Development, 2005.

[5] 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”, ISCA, 1996.

[6] A. Herdrich et. al., “Rate-Based QoS Techniques for Cache/Memory in CMP Platforms”, ICS'09.

[7] S. E. Raasch and S. K. Reinhardt. “The Impact of Resource Partitioning on SMT Processors”, 12th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003.

[8] A. Jaleel. “Memory Characterization of Workloads Using Instruction-Driven Simulation”. Intel Corporation, Web Copy: http://www.glue.umd.edu/~ajaleel/workload

[9] D. Chandra, et al, “Predicting inter-thread cache contention on a chip multiprocessor architecture”, 11th Int’l Symp. on High Performance Computer Architecture (HPCA), Feb, 2005

[10] Pointers to all SPEC CPU2000 material and results: http://www.spec.org/cpu/

[11] H. Tsao, “IBM @eServer p5 570 Server Consolidation Using POWER5”, White Paper, IBM Corporation

[12] F. J. Cazorla, E. Fernández, A. Ramirez, and M. Valero. “DCache Warn: An I-Fetch Policy to Increase SMT Efficiency”, International Parallel and Distributed Processing Symposium (IPDPS), 2004.

[13] R. Singhal, “Inside Intel Next Generations Nehalem Microarchitecture”, presented at the Intel Developer Forum in Shanghai (IDF), Spring 2008.

[14] S. Choi and D. Yeung. "Learning-Based SMT Processor Resource Distribution via Hill-Climbing", International Symposium on Computer Architecture (ISCA), 2006.

[15] F. J. Cazorla, A. Ramirez, M. Valero and E. Fernández. "Dynamically Controlled Resource Allocation in SMT Processors", MICRO-37, 2004.

[16] M. Merten, S. Srinath, M. Marden, J. Holm and G. Hinton. “Providing quality of service via thread priority in a hyper-threaded microprocessor”, *United States Patent Number 8,095,932*, issued 2007.