2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)

Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores

Artemiy Margaritov
University of Edinburgh

Siddharth Gupta*
EPFL

Rekai Gonzalez-Alberquilla
Arm Ltd.

Boris Grot
University of Edinburgh

***Abstract*—In a drive to maximize resource utilization, today's datacenters are moving to colocation of latency-sensitive and batch workloads on the same server. State-of-the-art deployments, such as those at Google, colocate such diverse workloads even on a single SMT core. This form of aggressive colocation is afforded by virtue of the fact that a latency-sensitive service operating below its peak load has significant slack in its response latency with respect to the QoS target. The slack affords a degradation in single-thread performance, which is inevitable under SMT colocation, without compromising QoS targets.**

This work makes the observation that many batch applications can greatly benefit from a large instruction window to uncover ILP and MLP. Under SMT colocation, conventional wisdom holds that individual hardware threads should be limited in their ability to acquire and hold a disproportionately large share of microarchitectural resources so as not to compromise the performance of a co-running thread. We show that the performance slack inherent in latency-sensitive workloads operating at low to moderate load makes it safe to shift microarchitectural resources to a co-running batch thread without compromising QoS targets. Based on this insight, we introduce Stretch, a simple ROB partitioning scheme that is invoked by system software to provide one hardware thread with a much larger ROB partition at the expense of another thread. When Stretch is enabled for latency-sensitive workloads operating below their peak load on an SMT core, co-running batch applications gain 13% of performance on average (30% max) over a baseline SMT colocation and without compromising QoS constraints.

***Keywords*-quality of service; datacenter; simultaneous multithreading; latency-sensitive applications; microarchitecture**

I. Introduction

Today's datacenters strive to maximize performance per Watt and per TCO dollar. To that end, the industry is moving toward aggressive colocation of latency-sensitive and batch workloads, as evidenced in both public clouds like Amazon EC2 and private infrastructures including Google's [1]. In the most aggressive deployments, colocation of latency-sensitive and batch workloads happens not just on the same CMP but even within a single SMT core [1], [2].

Prior research has shown that when a latency-sensitive service, such as web search, is highly loaded, the loss of single-thread performance stemming from SMT colocation is deleterious from the quality-of-service (QoS) perspective [3], [4], [5]. To counteract this interference, researchers and practitioners have proposed a number of proactive and reactive scheduling policies targeting SMT colocations [1], [3], [5].

We corroborate earlier findings regarding high SMT inter-thread interference at high load rates. However, we also observe that the absence of persistent queueing at lower loads means that there is significant *slack* between the actual latency and the QoS target, even at the tail (e.g., at 99th percentile latency). This slack naturally affords a fair degree of performance loss, making SMT colocation of latency-sensitive services feasible even with resource-hungry co-runners. Quantitatively, we find that our evaluated quartet of varied latency-sensitive services can afford to lose as much as 90% in single-thread performance while still meeting stringent QoS targets at low to moderate load.

We also observe that, in terms of performance degradation, the batch co-runners are often victimized by colocation much more than the latency-sensitive workloads. On average, we find that whereas latency-sensitive workloads lose 14% of single-thread performance across a range of batch co-runners, the batch workloads lose 24%, and up to 46%, when colocated with latency-sensitive co-runners.

We study the reasons for such different behavior under colocation and find that the two types of workloads have radically different sensitivity to ROB capacity. Latency-sensitive workloads show little benefit from large ROB capacities in modern server processors, corroborating prior studies showing that lean server cores are sufficient [6], [7] because frequent cache misses and data-dependent computation limit both instruction and memory-level parallelisms (ILP and MLP) [2], [8]. In contrast, many batch workloads benefit from a large ROB that helps unlock higher ILP and MLP. In processors such as Intel's, where the ROB is statically partitioned between the two hardware threads, batch workloads stand to lose an average of 19% (31% max) of their performance versus having the entire ROB to themselves. Dynamically sharing the ROB is similarly detrimental, as frequent cache misses by a latency-sensitive thread clog the shared ROB and prevent the co-runner from acquiring the resources it needs.

In response to these observations, we introduce Stretch – a simple mechanism to boost the performance of batch

*This work was done while the author was at University of Edinburgh.

2378-203X/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCA.2019.00024

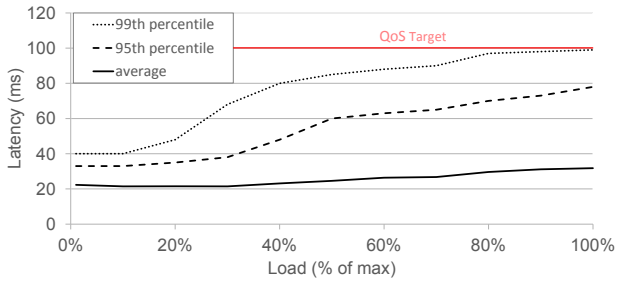

Figure 1: Web Search latencies as a function of load.

workloads co-running with latency-sensitive services. Stretch takes advantage of the performance slack inherent in latency-sensitive workloads operating below their peak load to shift ROB capacity to the co-running batch applications. To do so, Stretch employs one or more asymmetric ROB partitioning configurations that can trade single-thread performance of one hardware thread for higher performance of the other thread.

The asymmetric configuration(s) are chosen at design time and carry a negligible hardware cost and runtime switch overhead. When system software detects a sufficiently low load on a latency-sensitive workload executing on one of the hardware threads, it can trigger a pre-configured ROB partitioning setting that leaves the latency-sensitive thread with a fraction of its original capacity, buying the batch co-runner higher performance via greater ROB capacity.

Using a diverse set of latency-sensitive and batch workloads executing on real hardware and in a detailed simulator, we make the following contributions:

- Latency-sensitive workloads are highly sensitive to single-thread performance only at peak load rates. At lower loads, 70-90% of single-thread performance can be sacrificed without impacting QoS targets.
- Latency-sensitive workloads place modest demands on shared microarchitectural resources, making them good candidates for SMT colocation at lower load rates. In contrast, many batch workloads are highly sensitive to ROB capacity, losing up to 46% of single-thread performance under colocation.
- We propose Stretch, a technique that shifts ROB capacity from one hardware thread to another based on software control. By provisioning as few as two fixed ROB partitioning configurations, which carry negligible microarchitectural cost, Stretch enables one SMT thread to attain higher performance at the expense of another thread.
- We demonstrate that when a latency-sensitive workload is operating at low to moderate load, Stretch affords 13% higher performance on average (30% max) for a batch co-runner sharing a dual-threaded SMT core. Stretch can also be used to boost the performance of a latency-sensitive workload at high load, providing a best-case 18% improvement in single-thread performance over a baseline SMT core.

II. Aggressive Workload Colocation

Colocating heterogeneous workloads on a server is an effective way to maximize throughput per Watt and per TCO dollar. For instance, Google aggressively colocates latency-sensitive and batch workloads from its vast application portfolio on the same machine [1], [2]. One concern with aggressive colocation is meeting QoS targets for latency-sensitive applications. Recent work from Google has indicated that, despite aggressive workload colocation on commodity servers, significant QoS degradation is infrequent [1]. The finding may appear surprising, but can be attributed to a confluence of two factors.

First, a given service running on a server is rarely operating at its peak QoS-compliant load. Demand on individual services is generally cyclical, with significant periods of low to moderate demand [1], [9]. Moreover, the number of servers dedicated to latency-sensitive services tends to be over-provisioned to maintain QoS targets in the face of load spikes. With client requests load balanced across a pool of servers, peak load periods account for only a fraction of the total service uptime.

Secondly, when the load is below the sustainable limit, the tail latency tends to stay considerably below the QoS target. The reason is that queueing delays, and not the processing time, dominate the latency at high load [5], [10], [11]. When a request has to wait for a set of previously-enqueued requests to finish, its effective service time increases by the combined service time of these older requests. Because queuing can occur even at low average loads due to bursty request arrival, latency targets are typically set at a multiple of the expected per-request service time.

Figure 1 shows the average, 95th and 99th percentile latency for the Web Search engine versus its load. The study is performed on an Intel i7-2600K system running at 3.4GHz. Consistent with prior work, we set Web Search 99th percentile latency target to be 100ms [3], [8], [12]. Thus, QoS constraints are satisfied only if the 99th percentile latency is below 100ms.

As the figure shows, the average latency climbs slowly with the load, increasing by 43% from the lowest to the highest load points. In contrast, the 99th percentile latency grows by over 2.5x as a larger fraction of requests queues for an extended period of time.

The reason why this trend is important for colocated workloads is that, when the service load is below its sustainable peak – which is often the case, per earlier point – there is significant *slack* [4] available in the per-request processing time. So, while microarchitectural contention arising from workload colocation can degrade single-thread performance and thus increase per-request processing time, in the absence of queuing, this degradation can generally be absorbed by the in-built slack in latency targets.

We characterize the amount of slack available in per-request processing time as a function of server load for Data

Name	Description	QoS Targets
Data Serving	Cassandra 2.1.12 [13], 15 threads, 1M ops/sec	20ms [14] 99th Percentile
Web Serving	Elgg Networking Engine [13], 10 clients, MySQL v5.5	1 sec [13] 95th Percentile
Web Search	Nutch 1.2, Lucene 3.0.1, 100 clients, 5 GB dataset	100ms [3] 99th Percentile
Media Streaming	Nginx Streaming Server [13], 500 clients, high bitrates	2 sec [13] timeout

Table I: Workloads and their parameters used to measure slack.

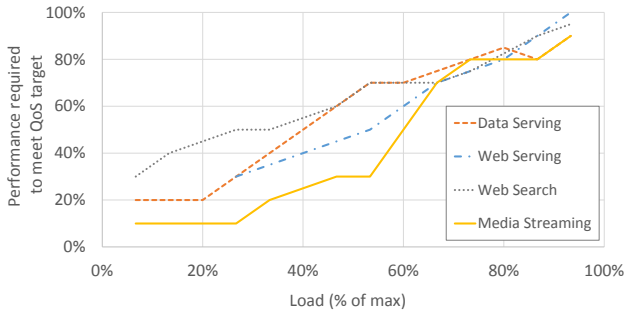

Figure 2: Slack in per request processing time for latency-sensitive workloads.

Serving, Web Serving, Web Search and Media Streaming workloads using the same Intel-based hardware as in the previous study. For each workload, we control the server load by adjusting the number of clients generating requests. We identify the maximum load that meets the respective latency target for each workload, which we then take as the peak sustainable load. QoS target for each workload is shown in Table I.

We measure the required performance as a function of load, varying the load in steps of 10% with respect to peak load. Slack is then defined as the lowest performance point, as a fraction of full core performance, that meets the QoS target. For instance, at 30% of peak load, Web Search can afford to lose 50% of its single-thread performance while continuing to meet its 100ms 99th percentile latency target.

To precisely modulate core performance across the full range, we control the fraction of time that the latency-sensitive workload runs on the core. We do this through a mechanism inspired by Elfen scheduling [3], whereby we interleave at a fine-grain, a non-contentious preemptive co-runner. When the co-runner runs, the latency-sensitive thread does not, and vice-versa, thus time-sharing the same core. The interleaving happens at a sub-millisecond granularity, which is orders of magnitude below the tail latency target for all four latency-sensitive workloads.

Figure 2 demonstrates that for the evaluated workloads, a significant amount of slack exists at low to moderate load rates. For instance, at a load of 20% relative to each service's peak load, 55-90% of single-thread performance can be sacrificed without violating QoS. This fraction decreases to 30-70% at a load of 50%. As the load approaches the

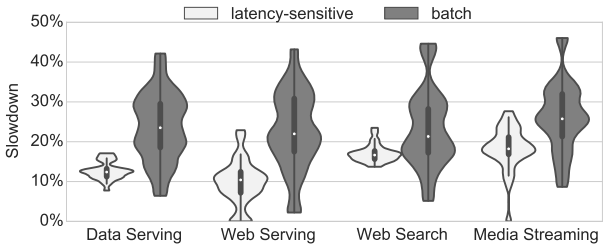

Figure 3: Slowdown incurred by colocating latency-sensitive and batch applications on an Intel-like SMT core. Data normalized to performance of a full core.

peak, the performance slack rapidly diminishes; at 80% load, at least 80% of full single-thread performance is required to meet QoS targets. The bottom line is that the high degrees of performance slack available at sub-peak load rates afford aggressive colocation by providing tolerance to contention-induced performance loss for the latency-sensitive workloads.

III. Colocation on SMT Cores

The previous section demonstrates the existence of considerable performance slack in response latencies of latency-sensitive workloads. The slack naturally motivates colocating latency-sensitive workloads with other applications not just on the same server, but also on the same SMT core, as contention-induced slowdown on the latency-sensitive thread will not cause a violation of QoS targets at all but the highest load rates. Indeed, Google's infrastructure is powered by SMT-capable servers, with SMT always enabled [2], and – despite sharing both core and uncore resources across a range of colocated applications – Google reports relatively infrequent QoS violations [1].

A. Extent of Contention

To precisely characterize the extent and sources of microarchitectural contention in the context of SMT, we use a detailed cycle-accurate simulation. We model a dual-thread SMT core roughly based on Intel's recent core microarchitecture. Details of the modeled processor, workloads and the simulation methodology can be found in Section V. We study configurations where one thread is latency-sensitive and the other is batch. To ensure high diversity in the set of batch workloads, we use all 29 SPEC'06 benchmarks as batch co-runners and colocate each latency-sensitive workload with each of the 29 batch workloads in turn.

Figure 3 shows the slowdown (IPC degradation) incurred by both latency-sensitive workloads and batch ones when colocated on an SMT core. For each latency-sensitive workload, the figure shows (i) the distribution of slowdown across all colocations with the 29 batch co-runners, and (ii) the distribution of slowdown experienced by different co-runners as a result of being colocated with the given latency-sensitive thread. The distribution is represented by width of

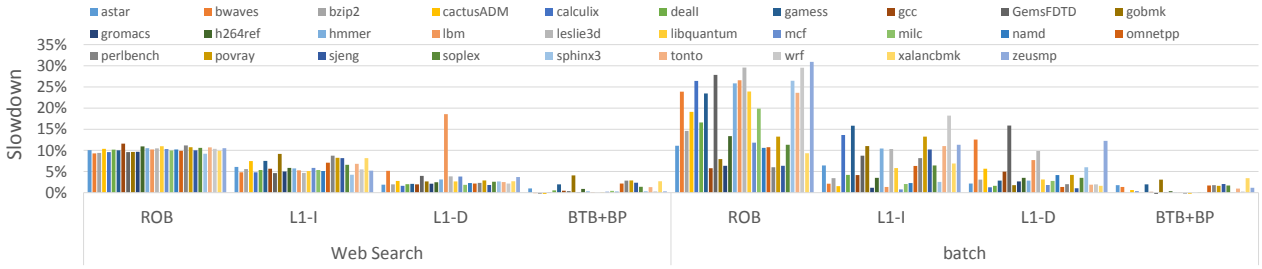

Figure 4: Slowdown for Web Search (left) and a batch application (right) when they share different core uarch resources. Bars represent different batch applications. Data normalized to stand-alone execution on a full core. Higher bars correspond to a larger performance drop.

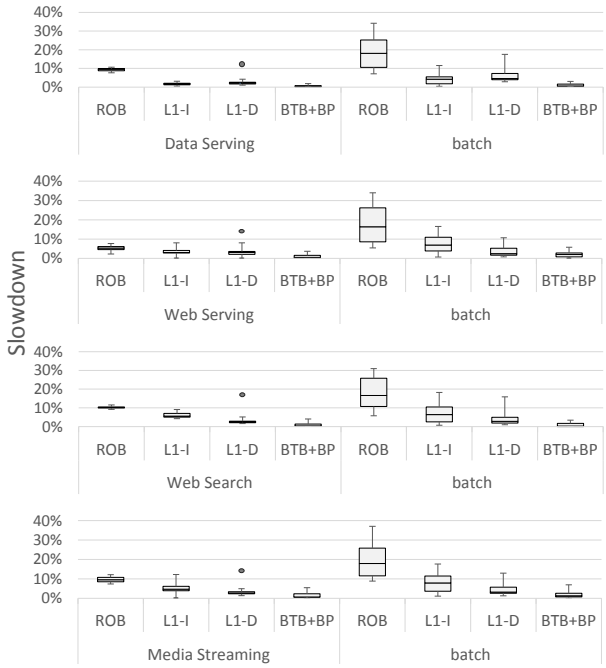

Figure 5: Slowdown caused by sharing core microarchitectural resources. Each chart represents average performance degradation among batch applications (right) colocated with different latency-sensitive applications (left). Performance normalized to stand-alone execution on a full core. Higher bars indicate a larger performance drop. When sharing L1-D, a batch outlier causing the high slowdown of latency-sensitive workloads is `lbm`.

a violin in the graph. A violin is annotated with the median and the interquartile range which is shown by a black box.

As the figure shows, latency-sensitive applications experience a mostly modest IPC degradation, with an average of 14% and a maximum of 28%. This result corroborates a prior datacenter-scale characterization effort [1] showing that slowdown stemming from colocation is rarely severe when it comes to latency-sensitive services. In contrast, we observe that slowdown of batch applications is considerably higher, with an average performance drop of 24% and a maximum of 46%. This result also corroborates a previous characterization study of SPEC applications colocated with CloudSuite workloads [5].

B. Sources of Contention

In order to understand the difference in performance sensitivity across the two types of applications, we study how they are affected by sharing-induced contention in individual microarchitectural structures inside the core. To do so, for each colocation, we simulate each hardware thread with completely private microarchitectural structures for everything *except* the resource under study. For instance, to understand the extent of interference in the L1-I cache, we model a shared L1-I with an otherwise private core for each of the two threads.

We focus on four types of resources as potential sources of contention: L1-I, L1-D, branch prediction structures (BTB and direction predictor), and the reorder buffer (ROB). We use the ROB as a proxy for other structures that make up the instruction window (including physical registers and the LSQ), since the pressure on these other structures is proportional to the utilization of the ROB. Complete details of the methodology can be found in Section V.

Figure 4 shows the results of the study for the Web Search engine. The Y-axis shows the performance drop, with respect to stand-alone execution, stemming from sharing a particular resource between Web Search and each particular co-runner. The left side of the graph shows the performance drop of Web Search workload caused by the co-runner, while the right side shows the performance drop of the co-runner caused by Web Search.

We observe two important trends. First, sharing any given resource has a modest effect on the performance of Web Search, with slowdown generally within 12%, except in colocation with `lbm`, where contention for L1-D capacity causes a higher performance loss. Secondly, a number of the batch co-runners experience a significant performance loss, primarily in the shared ROB, where the loss exceeds 15% for 15 out of 29 applications, reaching 31% in the worst case.

Extending the study to the three other latency-sensitive workloads and the same set of co-runners, we find that the trends hold. These are presented in Figure 5, which shows the average performance degradation attributed to individual resources for the various colocations. No single resource, when shared, is responsible for a significant performance drop across all of the latency-sensitive workloads, the exception

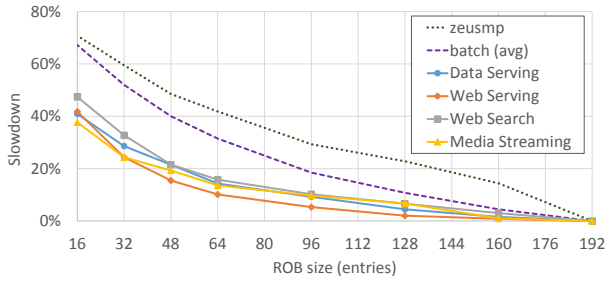

Figure 6: Applications' sensitivity to ROB capacity. Data normalized to performance of a core with 192 ROB entries.

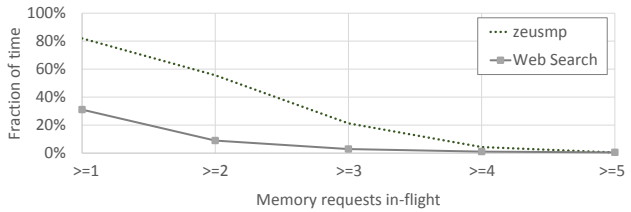

Figure 7: Fraction of time Web Search and `zeusmp` exhibit MLP.

again being the colocation with `lbm`, where the performance drop ranges from 12% to 19%. Meanwhile, for batch applications, the ROB stands out as a consistent source of performance degradation, accounting for 19% on average and 31% in the worst case.

C. ROB in Focus

To understand why the ROB plays a much larger role in batch workloads' performance as compared to that of latency-sensitive ones, we study each workload's sensitivity to ROB capacity. For this study, we consider each individual workload executing in isolation on a core whose ROB capacity is varied from 32 to 192 entries. At its maximum 192-entry capacity, the ROB – and therefore the entire core – matches the one used for the studies in the previous section. To keep the graph legible, we plot Data Serving, Web Serving, Web Search and Media Streaming as representative latency-sensitive workloads, along with the average of batch workloads. For comparison, we also plot `zeusmp`, which is a batch application with high ROB sensitivity.

Figure 6 presents the plots. First, we note that all of the evaluated latency-sensitive workloads are remarkably similar in their sensitivity to ROB capacity. None of the four benefit from a large ROB, achieving 90-95% of peak performance with just a half of the maximum ROB capacity (i.e. 96 entries). For a ROB size as small as 48 entries, the performance drop of latency-sensitive workloads is within 23% of performance attained with a full 192-entry ROB.

The situation is very different for batch workloads, which exhibit much higher sensitivity to ROB capacity. At 96 entries, the average performance drop is 19%, reaching 31% in the worst case. The slowdown reduces to just 4% on average with a ROB size of 160 entries.

A key reason for the difference in ROB sensitivity between latency-sensitive and batch workloads is their memory-level parallelism, or MLP. For applications that have high degrees of MLP, a large ROB facilitates uncovering of independent memory accesses that can be launched concurrently, thus hiding some of the memory latency. This is the case for many batch applications in our evaluated suite. Meanwhile, as shown in prior research, scale-out server workloads tend to have low MLP due to data-dependent access patterns [2], [8], which reduces the utility of a large ROB.

To highlight the difference in MLP between the two classes of workloads, we compare the MLP of Web Search and `zeusmp`. Figure 7 plots the cumulative concurrent memory accesses in-flight for the two applications. MLP is exhibited only when the number of concurrent accesses exceeds 1. Because the hardware coalesces accesses to the same cache block, for the purpose of measuring MLP, we only consider concurrent accesses to different cache blocks.

As seen in the figure, Web Search exhibits MLP (i.e., has two or more concurrent in-flight memory requests) only 9% of the time. In contrast, `zeusmp` exhibits MLP for 55% of its execution time. Moreover, `zeusmp` frequently has higher degrees of MLP, with three or more concurrent in-flight requests for 21% of its execution time. Web Search, on the other hand, achieves the same degree of MLP for only 3% of time. Because of the difference in MLP, a larger ROB is more beneficial for `zeusmp` than for Web Search.

D. Summary

Latency-sensitive applications place modest demands on core resources, which makes them good target for SMT colocation as they lose just 14% on average of single-thread performance in the presence of a co-runner. This range of performance loss can be comfortably absorbed without violating latency targets for all but the highest service loads. In contrast, many batch workloads experience a much larger performance drop of 24% on average, and up to 46%, in the presence of a co-runner. Our analysis reveals that limited ROB capacity due to colocation is largely responsible for the drop. Making more ROB capacity available to the batch applications can restore much of their performance.

IV. Stretch

A. Overview

Building on the observations above, we introduce Stretch – a light-weight mechanism for shifting ROB capacity from one SMT thread to another[1]. Whereas prior SMT resource management schemes have sought to achieve fairness in resource usage [15], [16] or maximize total throughput across threads [17], [18] (refer to Section VII), Stretch aims to boost

[1]While the ROB is the primary management target, we also manage the LSQ in proportion to the ROB. For simplicity, we only refer to the ROB in the rest of the paper.

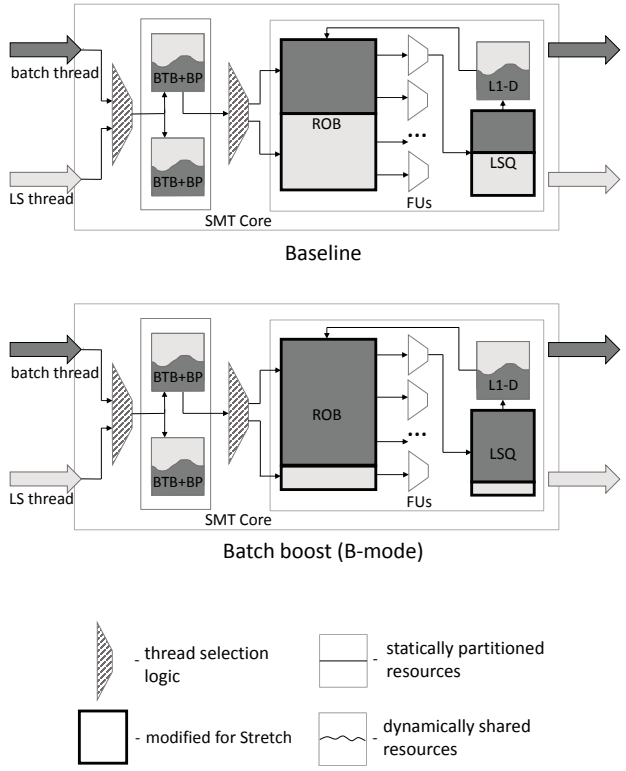

Figure 8: SMT core microarchitecture.

the throughput of one thread (batch) at the expense of the other in a controlled manner.

The key insight behind Stretch is that at low to moderate load, performance slack in latency-sensitive workloads affords a significant reduction in their ROB capacity without degrading QoS targets. The data in Figure 6 indicates that with as little as one-fourth of full ROB capacity (i.e., 48 out of 192 ROB entries), the slowdown of latency-sensitive workloads is limited to 15-21% when compared to an isolated execution on a full core. Placing this data in the context of Section II, we observe that this degree of performance degradation can be tolerated for all four of the evaluated latency-sensitive workloads up to 70% of the peak load without violating QoS targets.

To leverage the performance slack in latency-sensitive workloads, Stretch provides one or more *asymmetric* partitioned ROB configurations. When enabled through a software control, Stretch's asymmetric partitioning leaves the latency-sensitive thread with a fraction of the original ROB capacity, providing its co-runner with the bulk of ROB (and, by extension, instruction window) resources. Optionally, Stretch can also feature a "reverse" mode that provides high ROB capacity to the latency-sensitive thread as a way of boosting its single-thread performance to cope with high load rates.

Next, we detail the various aspects of Stretch.

B. Microarchitecture

Stretch requires minimal modifications over a baseline SMT core that partitions its instruction window into equal portions for private use by the co-running threads. Such equal partitioning approach is employed in Intel's processors [19].

To enable the QoS-throughput trade-off, Stretch provides at least one additional ROB configuration with asymmetrically-sized partitions. In total, a Stretch core has two or more different ROB configurations, as follows:

Baseline: Equal partitioning of the ROB between the two threads. This setting is used if Stretch has not been enabled by system software.

Batch boost mode (B-mode): This configuration provides a small ROB capacity to the latency-sensitive thread and is invoked when the load on the service is low to moderate. The bulk of the ROB capacity is given to the co-running thread to maximize its performance. Figure 8 depicts the Baseline and the B-mode configurations.

QoS boost mode (Q-mode): This configuration is used when the latency-sensitive thread is experiencing high load rates to maximize its QoS at the expense of the co-runner. The bulk of the ROB capacity is assigned to the latency-sensitive thread. This configuration is optional; if not present, the Baseline configuration is used at high load rates.

The asymmetric Stretch configuration(s) are provisioned at processor design time. Partitioning a structure (ROB or LSQ) across threads requires just two registers per thread along with minimal control logic. The first register is the *limit register*; it contains the maximum number of entries that can be occupied by the thread. The second register is the *usage register*; value of this register indicates the number of entries allocated by the thread in the given structure. On each cycle, the control logic checks if the value of the usage register is below that in the limit register; if the two are equal, issue is blocked for the given thread.

Both registers already exist in the baseline core that supports an equal partitioning. The only change required to enable asymmetric partitioning is making the *limit register* programmable. With such an extension, the maximal occupancy for a given structure (ROB, LSQ) is selected and loaded based on the selected configuration. To control the partitioning of both ROB and LSQ, Stretch requires two such pairs of registers – one pair per structure. As such, the actual hardware cost of Stretch is trivial as it does not require any new microarchitectural structures or complex control logic beyond what already exists in a baseline core that supports an equal partitioning of the ROB and LSQ.

C. Hardware-Software Interface

To take advantage of the asymmetric pipeline resource partitioning provided in Stretch, system software (likely with application guidance) maintains the following bits in an architecturally-exposed control register:

S-bit: If set, engages one of the Stretch modes based on the B/Q bits. When reset, the Baseline resource partitioning is engaged.

B/Q-bits: Indicates whether the B-mode or Q-mode configuration should be selected.

To decide which mode – Batch boost, QoS boost or Baseline – should be engaged, we extend the CPI2 [1] software framework. CPI2 is a software monitor deployed by Google to identify interference across workloads at the server level at runtime. In addition to existing CPI2 performance metrics, such as IPC, Stretch software monitor also tracks a QoS metric which reflects the amount of available performance slack in the system. When the software monitor detects performance slack (i.e. when latency-sensitive thread load is low), the software monitor enables B-mode.

In our work, we use tail latency as a representative and easily-available QoS metric for the amount of performance slack. An alternative strategy is to use queue length, which is an indirect metric of performance slack. For instance, recent work has used queue length to drive operating frequency and voltage settings, observing that when queue length is short, high single-thread performance is not necessary and the core can run at low voltage and frequency [11]. The same insight can be applied to invoke the B-mode when the queue length is small (i.e. there is no queueing) and Q-mode when queue length is large.

When the software monitor detects that QoS targets are violated, it first disengages B-mode by changing the ROB and LSQ configuration to equal partitioning or, if Q-mode is present, to a Q-mode configuration. After that, the software continues monitoring the QoS metric. If QoS violations persist, the software takes a corrective action in the same way as the baseline the CPI2 framework – that is, it throttles the co-runner for an interval of time.

Any mode change is accompanied by a pipeline flush in both threads. Periods of low and high service load are cyclical and long in duration (see Figure 14 in Section VI-D). As a result, a particular Stretch mode can stay engaged for a long time. Since mode changes generally occur only in response to the OS scheduler placing a new thread on the core or a swing in the load on the latency-sensitive thread, the associated pipeline flushes are highly infrequent in comparison to "routine" flushes triggered by branch mispredictions, exceptions, etc.

D. Discussion

Partitioning strategy: As noted above, our partitioning strategy only considers the ROB and LSQ. In principle, other instruction window resources (e.g., rename registers) could also be partitioned. This might be advantageous to, for instance, simplify the pipeline flush logic. We leave the design decisions as to which exact set of resources should be partitioned to microarchitects working on actual products.

Core type	ISA: SPARC V9, Freq.: 2.5 GHz
Front-end	
Fetch BW	6 instrs., up to 2 caches blocks, up to 1 branch
L1-I Cache	64KB, 64B line, 8-way set assoc., 2 banks, LRU
BP	Hybrid (16K gShare & 4K bimodal)
BTB	2K entries
Pipeline flush	12 cycles
Back-end	
ROB	192 entries total, 96 entries per thread
LSQ	64 entries total, 32 entries per thread
L1-D Cache	64KB, 64B line, 8-way set assoc., 2 banks, 10 MSHRs (5 per thread), LRU replacement, stride prefetcher tracking up to 32 load/store PCs
FUs	Int ALUs: 4 Add + 2 Mult, 3 FPU, 2 LSU Decode/Dispatch BW: 6 instrs. Commit BW: 6 instrs.
Uncore	
LLC	8MB NUCA, 16-way set associative Mesh NOC, 3-cycles per hop Average LLC access latency: 28 cycles
Memory	Access latency 75ns

Table II: Simulated processor parameters.

More broadly, our work is not meant to be prescriptive in the exact configuration to be used for B-mode and Q-mode execution points. Rather, the goal is to highlight the opportunity and potential benefits of asymmetric partitioning under colocated workloads. The exact configurations will be microarchitecture specific and may even cater to the demands of individual high-volume customers.

Number of configurations: For both B-mode and Q-mode points, multiple configurations may be provisioned that differ in the fractions of ROB capacity assigned to the two hardware threads. These would enable finer-grain control over per-thread performance but would necessitate more sophisticated software control to choose the appropriate configuration as a function of load.

Facilitating scheduling: To facilitate scheduling, Stretch does not require a particular type of a software thread (e.g., a latency-sensitive thread) to be run on a dedicated hardware thread. Thus, either B- or Q-mode can be invoked on either hardware thread. This is trivial to support, since invoking a mode requires simply loading appropriate settings into the *limit registers* of the partitioned resources (ROB and LSQ in this work).

Colocation options: While this work has focused on colocating a latency-sensitive thread with a batch thread, our insights can be applied to a colocation of two latency-sensitive threads. In particular, if the two threads belong to different applications with one at high load and the other at low load, the skewed configuration provided by Stretch would be beneficial to preserve QoS of the thread experiencing high load rates. On the other hand, if both applications are either at low load or high load, an asymmetric Stretch configuration would not be useful and the baseline equal partitioning should be applied.

V. Methodology

A. Processor Model

We model a 16-core processor with a 2-level cache hierarchy. Table II lists parameters of the simulated processor.
Baseline core: We simulate a dual-threaded, 6-wide out-of-order core. Every cycle thread selection logic determines which thread should be fetched, decoded and dispatched using ICOUNT [17], which selects a thread with the lowest number of in-flight instructions. If the selected thread cannot fill the width of the core in full, then the core switches to the other thread. The private L1-I and L1-D caches are shared dynamically between threads; i.e., any thread can allocate a block in any entry in a cache. Both L1 caches are address-interleaved: consecutive cache blocks are allocated to different banks. A cache bank can supply one cache block per cycle.

Similarly to L1 caches, the capacities of the BTB and branch predictor are also dynamically shared. However, each thread has a private branch predictor global history register and a return address stack.

The simulated core has a 192-entry ROB. Similar to existing Intel cores, this capacity is equally partitioned between threads, yielding 96 entries per thread. Selection logic determines which thread should retire using Round Robin policy [17], and the oldest instructions from the ROB partition of the selected thread are committed. If the number of committed instructions is less than the core width, the other thread commits instructions.
Uncore: We model an 8MB NUCA LLC with a mesh-based interconnect at the CMP level. To avoid performance loss due to LLC contention, we model a partitioned configuration to preserve each application's working set in the LLC. Technology to enable the simplistic partitioning assumed in this work exists for commercial processors; e.g., Intel Cache Allocation Technology [20]. More sophisticated schemes, such as Ubik [21] could be employed to make better usage of available capacity.

B. Workloads

Latency-sensitive workloads: We use a set of 4 representative open-source data center workloads from CloudSuite [22]. These applications are listed in Table III. The workloads are configured to provide maximum throughput while ensuring that QoS requirements are not violated. For Web Search and Web Serving, we simulate clients that send requests following a Zipfian distribution. For Data Serving, we use a 95:5 read-to-write request ratio. For Media Streaming, we monitor the AvgDelay metric reported by Darwin Streaming Server; a negative value of this metric indicates that the feed is being successfully delivered.
Batch workloads: Batch workloads are represented by benchmarks from SPEC'06. We evaluated each latency-sensitive workload in colocation with all 29 benchmarks from SPEC'06.

Name	Description
Data Serving	*Apache Cassandra 0.7.3,* 150 clients, 8000 operations per second
Web Serving	*Nginx web server 1.0 (front-end),* MySQL v5.5 database as a back-end
Web Search	*Nutch 1.2 / Lucene 3.0.1,* 92 clients, 1.4 GB index, 15 GB data segment
Media Streaming	*Darwin Streaming Server 6.0.3,* 200 clients, 60 GB dataset, high bitrates

Table III: Latency-sensitive workloads used for evaluation.

C. Simulation Methodology

We use a full-system multiprocessor simulator, Flexus [23], based on Simics which implements the SPARC v9 instruction set architecture and runs the Solaris 10 operating system.

For our evaluation, we use the sampling methodology proposed in [23]. We generate 320 samples over 4s of each workload's execution. At simulation time, we warm up the caches and branch predictor structures using functional simulation. Then, for each sample, we run cycle-accurate simulation for 150K instructions. The first 100K instructions are used to warm up the core structures. We collect measurements over the following 50K instructions. As figure of merit for evaluating performance, we use the number of application instructions executed per cycle (UIPC) [23].

For each latency-sensitive workload, we evaluate it being colocated with each of the 29 batch applications. We use the same set of sampling points across all colocations to ensure that the results are consistent across simulations.

VI. Evaluation

A. Stretch-ing the Performance Range

In this section, we quantify the performance benefits attained by applying Stretch asymmetric resource partitioning. We study several Stretch configurations. One is the baseline, which partitions the ROB in half. The other configurations employ asymmetrical ROB partitioning specified as N-M, where N entries are assigned to a latency-sensitive thread and M entries to a batch thread. We explored different degrees of asymmetry of configurations and measured performance for both batch and latency-sensitive applications. We evaluate performance change for both latency-sensitive and batch applications on asymmetric Stretch configurations and compare it to the baseline with an equally-partitioned ROB. The results of this study are depicted in Figure 9. We discuss the results for B-modes and Q-modes in Sections VI-A1 and VI-A2, respectively.

1) B-mode: We evaluate B-mode configurations, in which ROB capacity of batch applications is varied in range of 128 to 160 ROB entries with a step of 8. Recall that the full ROB capacity is 192 entries. The remaining capacity of ROB is given to the latency-sensitive thread. The results, normalized to the performance of a core with equally-portioned ROB, are shown in Figure 9 (left). For each Stretch configuration,

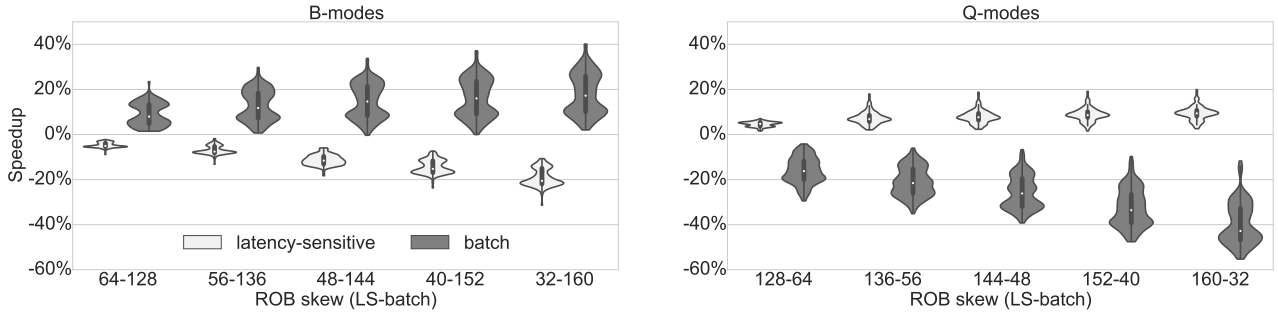

Figure 9: Performance change provided by different Stretch configurations. Data normalized to performance of a core with equally-partitioned ROB.

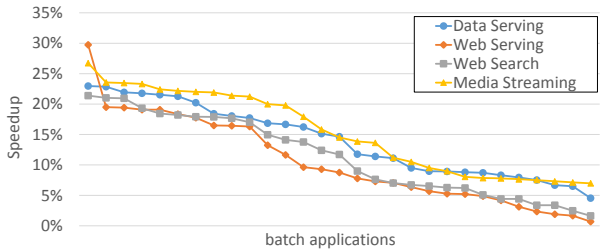

Figure 10: Speedup of batch applications provided by Stretch B-mode with ROB partitioning skew 56-136. Data normalized to performance of the baseline core with equally partitioned ROB.

the chart depicts two violin plots, for both batch and latency-sensitive applications. The width of each violin represents the distribution of performance change across all the colocations. In each violin, the height indicates the range of speedup observed across all of the colocations.

We observe that the B-mode configurations are effective at improving the performance of batch workloads while incurring only small performance drop for latency-sensitive applications. Such small performance drop can be tolerated if the latency-sensitive application is under sub-peak load. For instance, with a skewed ROB partitioning 56-136, a B-mode configuration improves performance of batch applications by 13% on average, and up to 30% in the best case.

To gain more insight into the gains achieved through B-mode configuration with a skew 56-136, Figure 10 plots the speedup for each of the 29 batch workloads when colocated with the four latency-sensitive workloads. For each latency-sensitive workload, the speedups among batch co-runners are sorted from largest to smallest. Because the sorted order of speedups differs for each latency-sensitive workload, the names of the benchmarks are not shown on the X axis.

As shown in the figure, for each latency-sensitive workload, there are at least 10 batch applications which enjoy a performance improvement of over 15%, while other 2 benefit by over 10%. Such big gains can be explained by the fact that these workloads have high sensitivity to the ROB capacity. The remaining workloads have diminishing ROB sensitivity, yet also register performance improvements of 2% to 9%.

While the batch applications significantly benefit from B-mode, latency-sensitive workloads lose little performance as their ROB capacity is diminished. Coming back to the left chart in Figure 9, we observe that the performance drop of latency-sensitive workloads averages just 7% (13% in the worst case) across all studied colocations on configuration with the skew 56-136. Placing this data in context of Section II, we find that we can maintain the QoS on this B-mode configuration for up to 85% of the peak load.

If a latency-sensitive workload can tolerate a large performance drop (e.g. when load is low), a B-mode configuration with a higher skew towards a batch application can be used to achieve even higher speedups for batch applications. For example, with B-mode configuration with a skew 32-160, batch applications' performance is increased by 18% on average over the baseline (40% max).

2) Q-mode: An evaluation of the Q-mode configurations is presented in Figure 9 (right). Analyzing Q-mode, we observe that it delivers a lower performance gain for the latency-sensitive workloads than what B-modes delivers for batch workloads. With the Q-mode configuration and ROB skew 136-56, average performance improvement is 7% on average, 18% in the best case. Such modest performance improvements can be explained by the lack of sensitivity in latency-sensitive workloads to large ROB configurations (see Section III-C).

As expected, when Q-mode is engaged, the performance of the batch co-runners drops due to diminished ROB capacity. With Stretch Q-mode and ROB skew 136-56, performance of batch workloads decreases by 21% on average, and up to 35% in the worst case. While the drop is considerable, the alternative is disallowing execution on one of the SMT threads at peak load periods [3], [4]. Compared to the complete loss of execution capability for the co-runner incurred by such a heavy-handed alternative, the degradation incurred in Stretch Q-mode is tolerable in that it maintains 79%, on average, of the co-runners performance.

B. Stretch versus Fetch Throttling

By managing the ROB, Stretch effectively controls resource allocation through the core back-end. An alternative

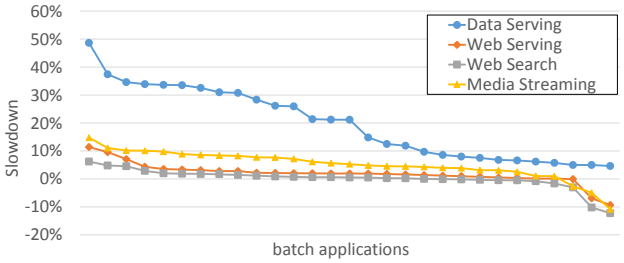

Figure 11: Slowdown of batch applications colocated with the indicated latency-sensitive workload using a dynamically shared ROB. Data normalized to performance with equally partitioned ROB.

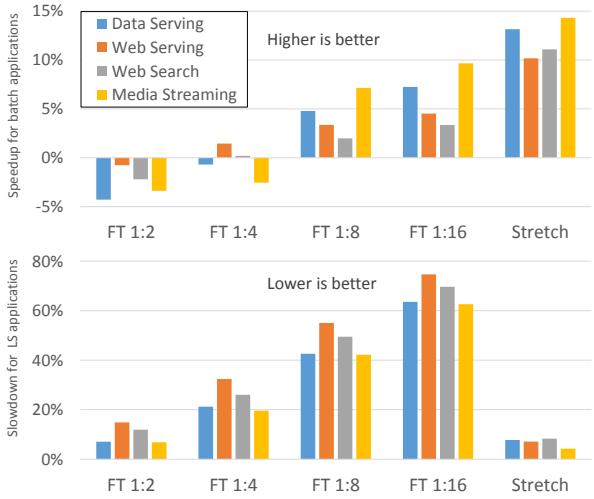

Figure 12: Average performance change provided by fetch throttling (FT) and Stretch B-mode. Data normalized to performance of the baseline core with equally partitioned ROB.

is front-end control using a fetch policy as suggested in a number of previous SMT resource management studies [17], [24], [25]. Indeed, IBM's POWER server processors provide a configurable fetch priority knob that allows one thread to gain a larger share of fetch bandwidth at the expense of another thread [26]. By controlling admission, such *fetch throttling* policies indirectly control the ROB occupancy.

In this section, we compare Stretch (a back-end resource management policy) to fetch throttling (front-end resource management). The first question we wish to answer is whether any sort of resource management (front or back) is necessary at all. To answer this question, we compare the baseline equal-partitioned ROB to a dynamically shared ROB, both with ICOUNT fetch policy.

Figure 11 presents the results of the study. The names of the benchmarks are not shown on the X axis due to the fact that the sorted orders differ among latency-sensitive workloads. We find that the vast majority of batch applications lose performance (8% average, 49% max) in colocations with latency-sensitive workloads under dynamic sharing as compared to equal ROB partitioning. Batch applications experience much higher slowdown in colocation with Data Serving than with others latency-sensitive workloads (20% and 3% on average, respectively). However, performance of latency-sensitive workloads improves slightly (4% average, 11% max – data not shown in the figure) under dynamic ROB sharing as compared to equal ROB partitioning.

The poor performance of dynamic sharing can be explained by the fact that in the absence of resource management, a latency-sensitive thread may occupy a large fraction of the ROB but not benefit from the capacity. Such monopolizing of ROB capacity by a latency-sensitive thread prevents a co-runner with high sensitivity to ROB capacity from acquiring the resources it needs, which causes an inevitable performance loss for the co-runner.

The next question we address is whether fetch throttling can prevent a latency-sensitive thread from monopolizing ROB capacity, thus providing more performance for the co-runner when the load on the latency-sensitive service is below peak. To answer this question, we allocate fetch bandwidth to the co-running threads via a ratio of 1:M, whereby for each cycle of fetch (on a per-port granularity) available to the latency-sensitive thread, the co-runner gets *M* cycles of fetch bandwidth. We experiment with several ratios varying M in powers of 2 in the range of 2 to 16. We compare performance of fetch throttling configurations against Stretch B-mode with a skew 56-136 for both batch and latency-sensitive applications. The results of this study normalized to performance of the baseline core with equally partitioned ROB are depicted in Figure 12. Note that the fetch throttling ratio 1:1 corresponds to the dynamically shared ROB configuration, discussed above and shown in Figure 11.

In comparison with equally partitioned ROB, batch applications experience a 3% loss and no loss with fetch throttling ratios of ratios 1:2 and 1:4, respectively. As noted above, with a fetch throttling ratio of 1:1 (i.e., dynamic ROB sharing), the performance loss for batch applications is 8% on average. Thus, increasing fetch throttling skew does benefit batch applications, though in a limited way.

As Figure 12 shows, limiting fetch bandwidth for latency-sensitive workloads reduces their performance by 10% and 25%, on average, for fetch throttling ratios 1:2 and 1:4, respectively. Higher fetch throttling ratios (1:8 and 1:16) deliver little performance improvement for batch applications (4% and 6% on average) while hurting the performance of latency-sensitive workloads dramatically, by 48% and 68%, respectively.

The reason for the poor performance of fetch throttling is that fetch control does not guarantee high ROB occupancy. Even with diminished fetch priority, latency-sensitive threads can continue to clog the ROB in the presence of long-latency misses, preventing the batch co-runner from allocating instructions despite its higher fetch priority. While even higher fetch throttling ratio would likely improve batch performance,

it would severely penalize the latency-sensitive workloads that are already losing 68% in the 1:16 configuration as compared to equal ROB partitioning.

In contrast, we observe that Stretch B-mode with a skew 56-136 delivers an average performance gain of 13% for the batch co-runners while limiting the slowdown for latency-sensitive workloads to just 7%, on average, as compared to equal ROB partitioning. We thus conclude that controlling the back-end is more effective than controlling the front-end.

C. Stretch versus Software Scheduling

A number of software techniques have examined colocated workloads in the context of QoS, particularly focusing on detecting and mitigating QoS-degrading instances of contention. These include reactive software mitigation policies [1] and contention-aware scheduling policies [3], [4], [10], [27]. In general, software scheduling aims at minimizing contention on shared resources by identifying application pairs which don't lose performance when colocated together. SMiTe [5] is the state-of-the-art software scheduling technique targeting SMT-level interference. SMiTe predicts interference between a pair of applications using online profiling. Based on the prediction, SMiTe determines colocation-friendly mappings and avoids colocations where a latency-sensitive thread suffers significant interference from a co-runner.

In this section, we compare Stretch to an ideal software scheduling. We show that software scheduling delivers lower performance for batch applications than Stretch. Moreover, we demonstrate that software scheduling is complementary to Stretch, and could be used to avoid contentious application pairing at high load rates.

Software scheduling, such as SMiTe, only selects colocation-friendly application pairs and is unable to provision individual microarchitectural resources to threads. Given a sufficiently large set of potential colocation pairs, some will likely be found contentious and disallowed to share a core using a scheme like SMiTe. To understand the limits of software scheduling, we study an idealized case where *all* colocation pairs experience *no* contention in all of the dynamically shared structures in an SMT core; namely, L1-I, L1-D and branch predictor. We model this *ideal software scheduling* by simulating private L1-I, L1-D and branch predictor structures for each of the colocated threads. ROB (and similarly, LSQ) contention under software scheduling is avoided through static equal ROB partitioning, as in the baseline processor.

Figure 13 compares this idealized setup to Stretch B-mode with a ROB skew 56-136 *without* any idealization (i.e., fully shared L1-I, L1-D and branch prediction structures). Ideal software scheduling provides a moderate gain of 8%, on average, as compared to the baseline core. Remember that this result is an unrealistic upper bound achieved through complete contention elimination, which is not actually possible in software. Meanwhile, Stretch, which is a practical

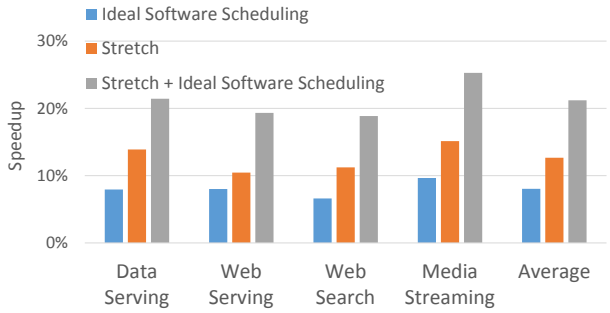

Figure 13: Speedup of batch applications colocated with the indicated latency-sensitive application with ideal software scheduling and Stretch. Data normalized to performance of the baseline core with equally partitioned ROB.

mechanism, improves the performance of batch co-runners by 13%, on average. The advantage of the Stretch can be explained by the fact that Stretch directly controls the ROB, which Section III-B showed to be the most critical resource for batch applications.

Finally, we note that Stretch and software scheduling are complementary and can be directly combined. We evaluate this option, which is labeled "Stretch + Ideal Software Scheduling" in the figure, and find that it improves the performance of batch applications by 21%, demonstrating that the benefits of the two techniques are additive as they target different sources of performance loss.

D. Impact Case Studies

In this section, we quantify benefits of Stretch for specific service deployments. Specifically, we aim to understand throughput gains for batch workloads that can be uncovered by enabling Stretch's B-mode when the load on a specific latency-sensitive service is low. Throughout this section, we consider a B-mode configuration with ROB skew 56-136.

Firstly, we consider a Web Search cluster. According to recent studies, a typical Web Search deployment is operating below 85% of its max load for about 11 hours per day (see the Figure 14(a)) [1], [9]. During this time, Stretch B-mode can be enabled to boost the throughput of batch jobs without sacrificing QoS guarantees for the Web Search workload. Using the Stretch B-mode configuration with ROB skew 56-136, batch workloads are able to gain 11% over the baseline SMT deployment. Extrapolating to 11 hours per day that this mode can be engaged, we find that Stretch can improve cluster throughput by an average of 5% in a 24-hour period.

Next, we consider a YouTube cluster. Gill et al. [28] show how the the interval 10am to 7pm concentrates most of the requests, peaking at 2pm (see the Figure 14(b)). During the other 17 hours of the day the amount of requests does not exceed 85% of peak. Similar to Web Search cluster case, Stretch B-mode configuration can be effectively applied during this time. In particular, applying the configuration B-mode with a skew 56-136 for 17 out of 24 hours improves

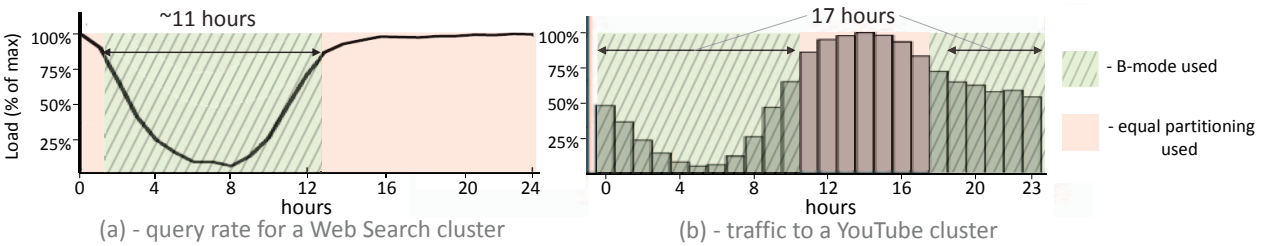

Figure 14: Diurnal pattern of load for different latency-sensitive services. Data is taken from [9] and [28].

the cluster throughput by 11% over a 24-hour period, without compromising QoS.

It is worth noting that both cases are doing a very coarse exploitation of the capabilities of Stretch. Finer grain management of the capabilities during the load times can lead to further improvements in cluster throughput.

VII. Related Work

There is a large amount of work addressing management of shared resources on SMT cores. Several techniques leverage fetch policies to maximize throughput [17], [18], [24], [25] without regard to quality-of-service of individual threads. Others have proposed mechanisms to improve throughput while maintaining fairness through dynamic distribution of shared microarchitectural resources [29], [15], [16], [30], [31]. For example, DCRA [29] tracks per-thread resource usage and partitions issue queue and register file entries dynamically. Sharkey et al. [30] follow up with using adaptive ROB partitioning to achieve the same goal. Choi et al. [31] present a mechanism that learns the best resource distribution via a hill-climbing framework.

Our work differs from these prior efforts in two important dimensions. First, Stretch intentionally sacrifices fairness to deliver more throughput for one thread at the expense of the other using the insight that a latency-sensitive application is not sensitive to core performance at sub-peak load rates. Secondly, Stretch uses two or three ROB configurations that are provisioned at processor design time and are engaged by application or system software based on readily-available QoS metrics. The design-time provisioning employed by Stretch avoids the tremendous complexity of finding preferred resource configurations required by adaptive/dynamic ROB management policies. Meanwhile, Stretch's software control relieves the hardware from maintaining application-level QoS metrics such as a request latency distribution.

Other researches have studied non-fair resource allocation across threads [32], [33], [34], [35], [36] with the aim of preserving the performance of a QoS-sensitive thread. In general, these techniques target strict QoS preservation (i.e., little to no performance drop), which means that the co-runner can suffer greatly depending on the dynamically-chosen configuration.

An important limitation of these works is their lack of analysis of when QoS targets should be enforced, how much performance loss is acceptable in practice, and when (if ever) QoS can be sacrificed for throughput. Stretch differs from these papers in observing that some QoS-sensitive workloads have performance slack that can be exploited to boost the performance of the co-runner at the expense of the QoS-sensitive thread, characterizes when such a trade-off is appropriate, and presents simple microarchitectural support for enabling it.

Core resource partitioning has been also studied in the context of reconfigurable CMP architectures. For example, The Sharing Architecture [37] distributes all core resources among small slices and form virtual cores from them on demand. Rather than introducing high complexity of forming virtual cores using a distributed ROB (along with other complex mechanisms), Stretch shows that a trivial static partitioning of just ROB and LSQ is sufficient, thus simplifying design and deployment.

VIII. Conclusion

With the slowdown in technology scaling, neither transistors not the energy to operate them is "free". This reality pushes processor design into a new regime of delivering higher performance without a commensurate complexity or energy cost.

In this work, we observe that latency-sensitive applications operating at a sub-peak load require only a fraction of performance afforded by today's out-of-order cores. We exploit this insight by shifting microarchitectural resources (namely, ROB and LSQ capacity) away from a latency-sensitive thread to its co-runner on an SMT core. By making minimal hardware modifications to an existing SMT core and without introducing any new hardware structures, the proposed Stretch design improves the performance of batch applications by 13% on average (30% max). This improvement comes without sacrificing service guarantees of latency-sensitive threads sharing the SMT core by exploiting existing software QoS monitoring and contention mitigation mechanisms. Stretch is one of the first instances of hardware support for improving core performance under QoS constraints.

ACKNOWLEDGEMENTS

The authors thank Priyank Faldu, Cheng-Chieh Huang, Rakesh Kumar, Amna Shahab, Dmitrii Ustiugov and the anonymous reviewers for their helpful comments. This work was supported by the EPSRC CDT in Pervasive Parallelism at the University of Edinburgh and Arm PhD Scholarship Program.

REFERENCES

[1] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI^2: CPU performance isolation for shared compute clusters," in *Proceedings of the European Conference on Computer Systems*, 2013, pp. 379–391.

[2] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the International Symposium on Computer Architecture*, 2015, pp. 158–169.

[3] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading," in *Proceedings of the USENIX Annual Technical Conference*, 2016, pp. 309–322.

[4] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the International Symposium on Computer Architecture*, 2015, pp. 450–462.

[5] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "SMiTe: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers," in *Proceedings of the International Symposium on Microarchitecture*, 2014, pp. 406–418.

[6] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *Proceedings of the International Symposium on Computer Architecture*, 2008, pp. 315–326.

[7] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: Quantifying and mitigating the price of efficiency," in *Proceedings of the International Symposium on Computer Architecture*, 2010, pp. 314–325.

[8] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 37–48.

[9] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceedings of the International Symposium on Computer Architecture*, 2011, pp. 319–330.

[10] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in *Proceedings of the European Conference on Computer Systems*, vol. 4, 2014, pp. 1–14.

[11] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proceedings of the International Symposium on Microarchitecture*, 2015, pp. 598–610.

[12] M. E. Haque, Y. h. Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 161–175.

[13] T. Palit, Y. Shen, and M. Ferdman, "Demystifying cloud benchmarking," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2016, pp. 122–132.

[14] A. Pahlevan, J. Picorel, A. P. Zarandi, D. Rossi, M. Zapater, A. Bartolini, P. G. Del Valle, D. Atienza, L. Benini, and B. Falsafi, "Towards near-threshold server processors," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 2016, pp. 7–12.

[15] J. Alastruey, T. Monreal, F. Cazorla, V. Viñals, and M. Valero, "Selection of the register file size and the resource allocation policy on SMT processors," in *Proceedings of the International Symposium on Computer Architecture and High Performance Computing*, 2008, pp. 63–70.

[16] H. Wang, I. Koren, and C. M. Krishna, "An adaptive resource partitioning algorithm for SMT processors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008, pp. 230–239.

[17] 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 *Proceedings of the International Symposium on Computer Architecture*, 1996, pp. 191–202.

[18] D. M. Tullsen and J. A. Brown, "Handling long-latency loads in a simultaneous multithreading processor," in *Proceedings of the International Symposium on Microarchitecture*, 2001, pp. 318–327.

[19] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, Intel Corporation.

[20] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer, "Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2016, pp. 657–668.

[21] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 49, no. 4, 2014, pp. 729–742.

[22] *CloudSuite: The Benchmark Suite of Cloud Services*, http://cloudsuite.ch/.

[23] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, pp. 18–31, 2006.

[24] F. J. Cazorla, E. Fernandez, A. Ramírez, and M. Valero, "Improving memory latency aware fetch policies for SMT processors," in *Proceedings of the International Symposium on High Performance Computing*, 2003, pp. 70–85.

[25] S. Eyerman and L. Eeckhout, "A Memory-Level Parallelism aware fetch policy for SMT processors," in *Proceedings of the International Symposium on High Performance Computer Architecture*, 2007, pp. 240–249.

[26] B. Hall, P. Bergner, A. S. Housfater, M. Kandasamy, T. Magno, A. Mericas, S. Munroe, M. Oliveira, B. Schmidt, W. Schmidt *et al.*, *Performance optimization and tuning techniques for IBM Power Systems processors including IBM POWER8*. IBM Redbooks, 2017.

[27] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the International Symposium on Microarchitecture*, 2011, pp. 248–259.

[28] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: A view from the edge," in *Proceedings of the Conference on Internet Measurement*, 2007, pp. 15–28.

[29] F. J. Cazorla, A. Ramírez, M. Valero, and E. Fernández, "Dynamically controlled resource allocation in SMT processors," in *Proceedings of the International Symposium on Microarchitecture*, 2004, pp. 171–182.

[30] J. Sharkey, D. Balkan, and D. Ponomarev, "Adaptive reorder buffers for SMT processors," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 244–253.

[31] S. Choi and D. Yeung, "Learning-based SMT processor resource distribution via hill-climbing," in *Proceedings of the International Symposium on Computer Architecture*, vol. 34, no. 2, 2006, pp. 239–251.

[32] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernández, A. Ramírez, and M. Valero, "Predictable performance in SMT processors: Synergy between the OS and SMTs," *IEEE Transactions on Computers*, vol. 55, no. 7, pp. 785–799, 2006.

[33] S. E. Raasch and S. K. Reinhardt, "Applications of thread prioritization in SMT processors," in *Proceedings of the Workshop on Multithreaded Execution And Compilation*, 1999.

[34] G. K. Dorai and D. Yeung, "Transparent threads: Resource sharing in SMT processors for high single-thread performance," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2002, pp. 30–41.

[35] C. Boneti, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, C.-Y. Cher, and M. Valero, "Software-controlled priority characterization of POWER5 processor," in *Proceedings of the International Symposium on Computer Architecture*, vol. 36, no. 3, 2008, pp. 415–426.

[36] A. Herdrich, R. Illikkal, R. Iyer, R. Singhal, M. Merten, and M. Dixon, "SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms," in *Proceedings of the USENIX Workshop on Hot Topics in Parallelism*, 2012.

[37] Y. Zhou and D. Wentzlaff, "The sharing architecture: sub-core configurability for IaaS clouds," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014, pp. 559–574.