Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads using Simultaneous Multithreading

Xi Yang[†] Stephen M. Blackburn[†] Kathryn S. McKinley[‡]
[†]*Australian National University* [‡]*Microsoft Research*

Abstract

Web services from search to games to stock trading impose strict Service Level Objectives (SLOs) on tail latency. Meeting these objectives is challenging because the computational demand of each request is highly variable and load is bursty. Consequently, many servers run at low utilization (10 to 45%); turn off simultaneous multithreading (SMT); and execute only a single service — wasting hardware, energy, and money. Although co-running batch jobs with latency critical *requests* to utilize multiple SMT hardware contexts (lanes) is appealing, unmitigated sharing of core resources induces non-linear effects on tail latency and SLO violations.

We introduce *principled borrowing* to control SMT hardware execution in which batch threads borrow core resources. A batch thread executes in a reserved batch SMT lane when no latency-critical thread is executing in the partner request lane. We instrument batch threads to quickly detect execution in the request lane, step out of the way, and promptly return the borrowed resources. We introduce the `nanonap` system call to stop the batch thread's execution without yielding its lane to the OS scheduler, ensuring that requests have exclusive use of the core's resources. We evaluate our approach for co-locating batch workloads with latency-critical requests using the Apache Lucene search engine. A conservative policy that executes batch threads only when request lane is idle improves utilization between 90% and 25% on one core depending on load, without compromising request SLOs. Our approach is straightforward, robust, and unobtrusive, opening the way to substantially improved resource utilization in datacenters running latency-critical workloads.

1 Introduction

Latency-critical web services, such as search, trading, games, and social media, must consistently deliver low-latency responses to attract and satisfy users. This requirement translates into Service Level Objectives (SLOs) governing latency. For example, an SLO may include an average latency constraint and a *tail constraint*, such as that 99% of requests must complete within 100 ms [6, 7, 13, 34]. Many such services, such as Google Search and Twitter [6, 8, 18], systematically underutilize the available hardware to meet SLOs. Furthermore, servers often execute only one service to ensure that latency-critical requests are free from interference. The result is that server utilizations are as low as 10 to 45%. Since these services are widely deployed in large numbers of datacenters, their poor utilization incurs enormous commensurate capital and operating costs. Even small improvements substantially improve profitability.

Meeting SLOs in these highly engineered systems is challenging because: (1) requests often have variable computational demands and (2) load is unpredictable and bursty. Since computation demands of requests may differ by factors of ten or more and load bursts induce queuing delay, overloading a server results in highly non-linear increases in tail-latency. The conservative solution providers often take is to significantly over-provision.

Interference arises in chip multiprocessors (CMPs) and in simultaneous multithreading (SMT) cores when contending for shared resources. A spate of recent research explores how to predict and model interference between different workloads executing on distinct CMP cores [8, 23, 25, 28], but these approaches target and exploit large scale diurnal patterns of utilization, e.g., co-locating batch workloads at night when load is low. Lo et al. explicitly rule out SMT because of the highly unpredictable and non-linear impact on tail latency (which we confirm) and the inadequacy of high-latency OS scheduling [23]. Zhang et al. do not attempt to reduce SMT-induced overheads, but rather they accommodate them using a model of interference for co-running workloads [35]. Their approach requires ahead-of-time profiling of all co-located workloads and over-provisioning. Prior work lacks dynamic mechanisms to monitor and control batch workloads on SMT with low latency.

This research exploits SMT resources to increase utilization without compromising SLOs. We introduce *principled borrowing*, which dynamically identifies idle cycles and borrows these resources. We implement borrowing in the ELFEN[1] scheduler, which co-runs batch threads and latency-critical requests, and meets request SLOs. Our work is complementary to managing shared cache and memory resources. We first show that latency-critical workloads impose many idle periods and they are short. This result confirms that scheduling at OS granu-

[1]In the Grimm fairy tale, *Die Wichtelmänner*, elves borrow a cobbler's tools while he sleeps, making him beautiful shoes.

larities is inadequate, motivating fine-grain mechanisms.

ELFEN introduces mechanisms to control thread execution and a runtime that monitors and schedules threads on distinct hardware contexts (*lanes*) of an SMT core. ELFEN pins latency-critical requests to one SMT lane and batch threads to $N-1$ partner SMT lanes on a N-way SMT core. A *batch* thread monitors a paired lane reserved for executing latency-critical requests. (We use '*requests*' for concision.) The simplest *borrow idle* policy in ELFEN ensures mutual exclusion — requests execute without interference. Batch threads monitor the request lane and when the request lane is occupied, they release their resources. When the request lane is idle, batch threads execute. We introduce `nanonap`, a new system call, that disables preemption and invokes `mwait` to release hardware resources quickly — within ~3 000 cycles. This mechanism provides semantics that neither yielding, busy-waiting, nor `futex` offer. After calling `nanonap`, the batch thread stays in this new kernel state without consuming microarchitecture resources until the next interrupt arrives or the request lane becomes idle. These semantics ensure that requests incur no interference from batch threads and pose no new security risks. Since the batch thread is never out of the control of the OS, the OS may preempt it as needed. The shared system state that ELFEN exploits is already available to applications on the same core, and ELFEN reveals no additional information about co-runners to each other.

We inject scheduling and profiling mechanisms into batch applications at compile-time. A binary re-writer could also implement this functionality. The instrumentation frequently checks for a request running on the paired SMT lane by examining a shared memory location. When the batch thread observes a request, it immediately invokes `nanonap` to release hardware resources. This policy ensures that the core is always busy, but it only utilizes one SMT lane on a two-way SMT core at a time.

More aggressive borrowing policies use both lanes at once by giving batch threads a budget that limits overheads imposed on requests, ensuring that SLOs are met. The budget is shaped by the SLO, the batch workload's impact on the latency-critical workload, and the length of the request queue. These policies monitor the request in various ways, via lightweight fine-grain profiling [32].

We implement ELFEN in the Linux kernel and in compile-time instrumentation that self-schedules batch workloads, using both C and Java applications, demonstrating generality. For our latency-sensitive workload, we use the widely deployed Apache Lucene open-source search engine [3]. Prior work shows Lucene has performance characteristics and request demand distributions similar to the Bing search engine [10]. We evaluate ELFEN co-executing a range of large complex batch workloads on two-way SMT hardware. On one core, ELFEN's borrow idle policy achieves peak utilization with essentially no impact on Lucene's 99th percentile latency SLO. ELFEN improves core utilization by 90% at low load and 25% at high loads compared to a core dedicated only to Lucene requests. It consistently achieves close to 100% core utilization, the peak for this policy — one of the two hardware contexts always busy. On an eight core CMP, the borrow idle policy usually has no impact or slightly improves 99th percentile latency because cores never go to sleep. Occasional high overheads at high load may require additional interference detecting techniques. Improvements in CMP utilization are more substantial than for one core because at low load, many cores may be idle. ELFEN consistently achieves close to 100% of the no-SMT peak, which is also the borrow idle policy's peak utilization.

Choosing a policy depends on provider workloads, capacity, and server economics, including penalties for missed SLOs and costs for provisioning servers. Providers currently provide excess capacity for load bursts and SLOs slack for each request. Our approach handles both. Our most conservative borrow idle policy steps out of the way during load bursts and suits a setting where the penalties for missed SLOs are very high. Our more agressive policies can soak up slack and handle load bursts. They offer as much as two times better utilization but at the cost of higher median latencies and higher probability of SLO violations. For server providers with latency-critical and batch workloads, the main benefit of our work is to substantially reduce the required number of servers for batch workloads.

In summary, contributions of this paper include:

- analysis of why latency-critical workloads systematically underutilize hardware and the opportunities afforded by idle periods;
- `nanonap`, a system call for fine-grain thread control;
- ELFEN, a scheduler that borrows idle cycles from underutilized SMT cores for batch workloads without interfering with latency-critical requests;
- a range of scheduling policies;
- an evaluation that shows ELFEN can substantially increase processor utilization by co-executing batch threads, yet still meet request SLOs; and
- an open-source implementation on github [33].

Our approach requires only a modest change to the kernel and no changes to application source code, making it easy to adopt in diverse systems and workloads.

2 Background and Motivation

We motivate our work with workload characteristics of latency-critical services; the non-linear effects on latency from uncontrolled interference with SMT; the opportunity to improve utilization availed by idle resources; and

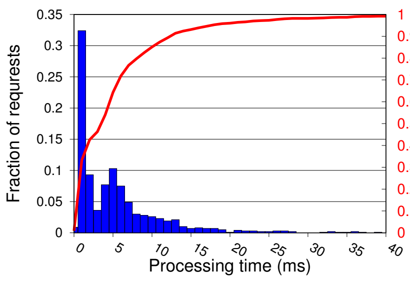


Figure 1. Highly variable demand is typical for latency-critical workloads. Lucene demand distribution with request processing time on x-axis in 1 ms buckets, fraction of total on left y-axis, and cumulative distribution red line on right y-axis.

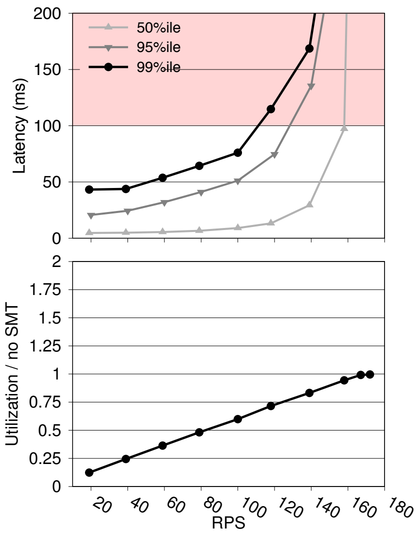


Figure 2. Overloading causes non-linear increases in latency. Lucene percentile latencies and utilization on one core. Highly variable demand induces queuing delay, which results in non-linear increases in latency.

requirements on responsiveness dictated by idle periods. Section 4 describes our evaluation methodologies.

Processing demand The popular industrial-strength Apache Lucene enterprise search engine is our representative service [3]. Prior work shows that services such as Bing search, Google search, financial transactions, and personal assistants have similar computational characteristics [6, 8, 10, 12, 29]. Figure 1 plots the distribution of request processing times for Lucene executing in isolation on a single hardware context. The bars (left y-axis) show that most requests are short, but a few are very long. This high variance of one to two orders of magnitude is common in such systems.

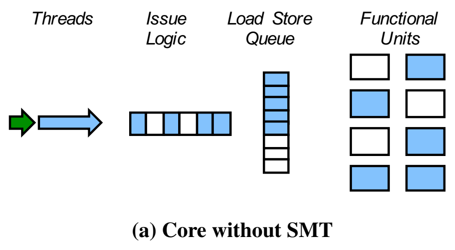
Load sensitivity This experiment shows that high load induces non-linear increases on latency. We assume a 100 ms service level objective (SLO) on 99th percentile latency for requests. A front end on separate hardware issues search requests at random intervals following an exponential distribution around the prescribed requests per second (RPS) mean rate. As soon as Lucene completes a request, it processes the next request in the queue. If no requests are pending, it idles. We show results for a single Lucene worker thread running on one core.

Figure 2 shows Lucene percentile latencies and utilization as a function of RPS only on one lane of a two-way SMT core using one Lucene task. The two graphs share the same x-axis. The top graph shows median, 95th, and 99th percentile latency for requests, the bottom graph shows CPU utilization which is the sum of the fraction of time the lanes are busy normalized to the theoretical peak for a system with SMT disabled. The maximum utilization is 2.0, but the utilization in Figure 2 never exceeds 1.0 because only one thread handles requests, so only one lane is used. As RPS increases, median, 95th, and 99th percentile latencies first climb slowly and then quickly accelerate. The 99th percentile hits a wall when RPS rise above 120 RPS, while the request lane utilization is only 70% at 120 RPS, leaving the two-way SMT core substantially underutilized when operating at a *practical* load for a 100 ms 99th percentile tail latency target.

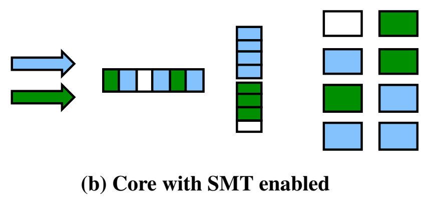
Random request arrival times and the high variability of processing times combine to produce high variability in queuing times and non-linear increases in latencies at high RPS. As we show next, adding a co-runner on the same core using SMT has the effect of throttling the latency-critical workload, effectively moving to the right in Figure 2. Movements to the right lead to increasingly unpredictable latencies, and likely violations of the SLO.

Simultaneous Multithreading (SMT) This section gives SMT background and shows that simultaneously executing requests on one lane of a 2-way SMT core and a batch thread on the other lane degrades request latencies. This result confirms prior work [8, 23, 35] and explains why service providers often disable SMT. We measure core idle cycles to show that the opportunity for improvement is large, if the system can exploit short idle periods.

We illustrate the design and motivation of SMT in Figure 3. Figure 3(a) shows that when only one thread executes on a core at a time, hardware resources such as the issue queue and functional units are underutilized (white). Figure 3(b) shows two threads sharing an SMT-enabled core. The hardware implements different sharing policies for various resources. For example, instruction issue may be performed round-robin unless one thread is unable to issue, and the load-store queue partitioned in half, statically, while other functional units are shared fully dynamically. It is important to note that such policies



(a) Core without SMT



(b) Core with SMT enabled

Figure 3. Simultaneous Multithreading (SMT) A single thread often underutilizes core resources. SMT dynamically shares the resources among threads.

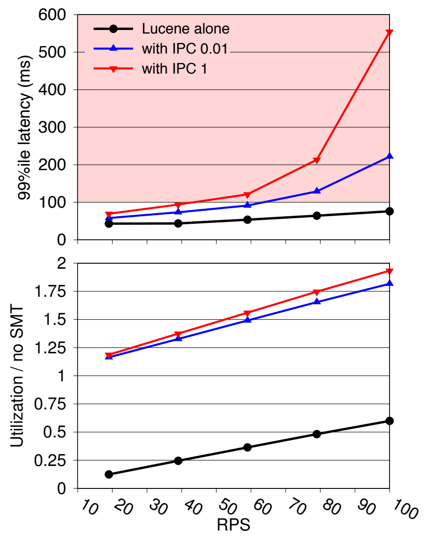


Figure 4. Unfettered SMT sharing substantially degrades tail latency. Lucene 99th percentile latency and lane utilization with IPC 1 and IPC 0.01 batch workloads.

mean that a co-running thread consumes considerable core resources *even when that thread has low IPC.*

To measure lower bounds on SMT interference, we consider two microbenchmarks as batch workloads executing on an Intel 2-way SMT core. The first uses a non-temporal store and memory fence to continuously block on memory, giving an IPC of 0.01. For instance, the Intel PAUSE instruction has a similar IPC. The other performs a tight loop doing nothing (IPC=1) when running alone. Neither consume cache or memory resources. Figure 4 shows the impact of co-running batch workloads on the 99% percentile latency of requests and lane utilization. Utilization improves over no co-runner significantly since the batch thread keeps the batch lane busy, but request latency degrades substantially, even when the batch thread has very low resource requirements (IPC = 0.01). For instance, at 100 RPS without a co-runner, 99th percentile latency is 76 ms. RPS must fall to around 40 RPS to meet the same 99th percentile latency with a low IPC co-runner.

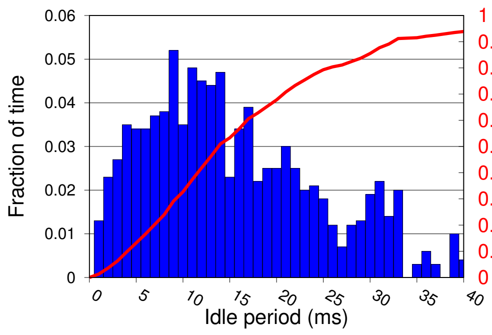


Figure 5. Lucene Inter-request idle times are highly variable and are frequently short. Histogram (left y-axis) shows the distribution of request idle times. The line (right y-axis) shows the cumulative time between completing one request and arrival of the next request at 0.71 lane utilization on one core at RPS = 120.

Co-running moves latencies to the right on RPS curves, into the steep exponential, with devastating effect on SLOs. Because SMT hardware shares resources such as issue logic, the load store queue (LSQ), and store buffers, tail latency suffers even when the batch workload has an IPC as low as 0.01. If a request is short, a co-runner may substantially slow it down without breaching SLOs. Unfortunately request demand is not known a priori. Moreover, request demand is hard to predict [15, 17, 19, 24] and the prediction is never free or perfect, thus we do not consider request prediction further.

To increase utilization without imposing any degradations on latency-critical requests cannot use multiple SMT lanes simultaneously. The strategies we explore are thus (1) to enforce mutual exclusion, executing a batch thread only when the partner lane is idle (*borrow idle*), and (2) to give the batch thread a budget for how much it may overlap execution with requests. These strategies require observing requests, detecting idle periods, and controlling batch threads.

Idle cycle opportunities Now we explore the frequency and length of idle periods to understand the requirements on the granularity of observing requests and controlling batch threads. Figure 5 shows the fraction of all idle time (y-axis) due to periods of a given length (x-axis). The histogram (blue) indicates the fraction of all idle time due to idle times of a specific period, while the line (red) shows a cumulative distribution function. For example, this shows that 2.3% of idle time is contributed by idle times of 15 ms in length (blue), and 53% of total idle time is due to idle times of 15 ms or less (red). **Highly variable and short idle periods dictate low-latency observation and control mechanisms.**

3 ELFEN Design and Implementation

This section describes the design and implementation of ELFEN, in which latency-critical requests and batch threads execute in distinct SMT hardware contexts (*lanes*) on the same core to improve server utilization. Given an N-way SMT, $N-1$ SMT lanes execute batch threads, *batch lanes*, and one SMT lane executes latency-critical requests, the *request lane*. We restrict our exposition below to 2-way SMT for simplicity and because our evaluation Intel hardware is 2-way SMT.

As Figure 4 shows, unfettered interference on SMT hardware quickly leads to SLO violations. ELFEN controls batch threads to limit their impact on tail latency. We explore borrowing policies applying the principle of either eliminating interference or limiting it based on some budget. The simplest policy enforces mutual exclusion by forcing batch threads to relinquish their lane resources whenever the request lane is executing a request. More aggressive borrowing policies add overlaping the execution of batch threads and requests, governed by a budget.

The ELFEN design uses two key ideas: (1) high-frequency, low-overhead monitoring to identify opportunities, and (2) low-latency scheduling to exploit these opportunities. The implementation instruments batch workloads at compile time with code that performs both monitoring and self-scheduling. The simple borrow-idle policy requires no change to the latency-critical workload. More aggressive policies require the latency-critical framework to expose the request queue length and a current request identifier via shared memory. Batch threads use `nanonap` to release hardware resources rapidly without relinquishing their SMT hardware context.

Our current design assumes an environment consisting of a single latency-critical workload, and any number of instrumented batch workloads. (Scheduling two or more distinct latency-critical services simultaneously on one server is a different and interesting problem that is beyond our scope.) Our instrumentation binds threads to cores with `setaffinity()` to force all request threads onto the identifiable request lane and batch threads onto partner batch lane(s). The underlying OS is free to schedule batch threads on batch lanes. Each batch thread will then fall into a monitoring and self-scheduling rhythm.

3.1 Nanonap

This section introduces the system call `nanonap` to monitor and schedule threads at a fine granularity. The key semantics `nanonap` delivers is to put the hardware context to sleep *without* releasing the hardware to the OS scheduler. We first explain why existing mechanisms, such as `mwait`, `WRLOS`, and `hotplug` do not directly deliver the necessary semantics.

The `mwait` instruction releases the resources of a hardware context with low latency. This instruction is available in user-space on SPARC and is privileged on x86. The IBM PowerEN user-level `WRLOS` instruction has similar semantics [26]. Calls to `mwait` are normally paired with a `monitor` instruction that specifies a memory location that `mwait` monitors. The OS or another thread wakes up the sleeping thread by writing to the monitored location or sending it an interrupt. The Linux scheduler uses `mwait` to save energy. It assigns each core a privileged idle task when there are no ready tasks. Idle tasks call `mwait` to release resources, putting the hardware in to a low-power state. Unfortunately, simply building upon any of these mechanisms in *user space* is insufficient because the OS may, and is likely to, schedule other ready threads to the released hardware context. In contrast, because it disables preemption, `nanonap` ensures that no other thread runs on the lane, releasing all hardware resources to its partner lane.

Another mechanism that seems appealing, but does not work, is `hotplug`, which prohibits any task from executing in specified SMT lanes. The OS first disables interrupts, moves all threads in the lane(s), including the thread that invoked `hotplug`, to other cores, and switches the lane(s) to the idle task which then calls the `mwait` instruction. While `hotplug` moves threads off a lane to other cores, user-space calls such as `futex` yield the lane, so other threads may execute in it. Therefore, neither the `hotplug` interface nor user-space locking nor calls to `mwait` are designed to release and acquire SMT lanes to and from each other because a thread does not *retain exclusive ownership of a lane while it pauses*.

We design a new system call, `nanonap`, to control the SMT microarchitecture hardware resources directly. Any application that wants to release a lane invokes `nanonap`, which enters the kernel, disables preemption, and sleeps on a per-CPU `nanonap` flag. From the kernel's perspective, `nanonap` is a normal system call and it accounts for the thread as if the thread is still executing. Because `nanonap` does not disable interrupts and the kernel does not preempt the thread that invoked the `nanonap`, the SMT lane stays in a low-power sleep state until the OS wakes the thread up with an interrupt or the ELFEN scheduler sets the `nanonap` flag. After the SMT lane wakes up, it enables preemption and returns from the system call. Figure 6 shows the pseudocode of `nanonap`, which we implement as a wrapper that invokes a virtual device using the Linux OS's `ioctl` interface for devices.

No starvation or new security state The `nanonap` system call and monitoring of request lanes do not cause starvation or pose additional opportunities for security breaches. Starvation does not occur because `nanonap` does not disable interrupts. The scheduler may wake up any napping threads and schedule a different batch thread on the lane at the end of a time quanta, as usual. When a batch thread wakes up or a new one starts executing, it tests whether its request lane partner is occupied and if so, puts itself to sleep. Since the OS accrues CPU time to batch threads waiting due to a `nanonap`, user applications cannot perform a denial of service attack simply by continuously calling `nanonap`, since the OS will schedule a napping thread away after they exhaust their time slice.

The ELFEN instrumentation monitors system state to make decisions. It reads memory locations and performance counters that reveal if the core has multiple threads executing. All of this system state is *already* available to threads on the same core — ELFEN reveals no additional information about co-runners to each other.

3.2 Latency of Thread Control

This section presents an experiment that measures the latency of sleeping and waking up with `nanonap`, `mwait`, and `futex`. Measuring these latencies is challenging because detecting exactly when a lane releases hardware resources must be inferred, rather than measured directly.

When a batch thread executes `mwait` on our Intel hardware, the lane first enters the shallow sleeping C1 state immediately. If no other thread executes in the lane for a while, it then enters a deep sleep state and releases its hardware resources to the active request lane. We measure how long it takes the lane to enter the deep sleeping state indirectly as follows. The CPU executes a few μops to transition an SMT lane from the shallow to the deep sleep state. For measurement purposes, we thus configure the measurement thread to continuously record how many μops the measurement thread has retired and how many μops the whole core retires every 150 cycles. When the measurement thread notices that the sleeping SMT lane does not retire any μops for a while, then retires a few more μops, and then stops retiring μops, it infers that the SMT lane is in the deep sleep state.

Figure 7 shows a microbenchmark that measures the latencies of sleeping and waking up with `nanonap`, `mwait`, and `futex`. The microbenchmark has two threads: a measurement thread and another thread, T2. The measurement thread puts T2 to sleep and wakes it up. The two threads execute on the same core but different SMT lanes. The measurement thread sets a flag, forcing T2 to sleep (line 5). T2 then executes `sleep` which either calls `nanonap` or `futex` to put the SMT lane to sleep, according to which is being measured. The `wait_until_t2_goes_to_sleep()` (line 6) call performs the deep sleep detection process described in the above paragraph. We measure wake-up latency directly (lines 10 to 13). The measurement thread sets a flag (line 11) and then detects when T2 starts executing instructions (line 12).

```
/******************* USER ********************/
void nanonap() {
  ioctl(/dev/nanonap);
}
/******************* KERNEL *******************/
nanonap virtual device: /dev/nanonap;
per_cpu_variable: nap_flag;
ioctl(/dev/nanonap) {
  disable_preemption();
  my_nap_flag = this_cpu_flag(nap_flag);
  monitor(my_nap_flag);
  mwait();
  enable_preemption();
}
```

Figure 6. Pseudo code for `nanonap`.

```
 1 /***** MEASUREMENT THREAD ON ONE SMT LANE *****/
 2 void measure() {
 3   /* measure send-to-sleep latency */
 4   start_sleep_request = timestamp();
 5   ask_t2_sleep();
 6   wait_until_t2_goes_to_sleep();
 7   finish_sleep_request = timestamp();
 8
 9   /* measure wake-up latency */
10   start_wakeup_request = timestamp();
11   wakeup_t2();
12   wait_until_t2_wakes_up();
13   finish_wakeup_request = timestamp();
14
15   if (measuring_futex || measuring_nanonap) {
16     sleep_latency =
17       finish_sleep_request - start_sleep_request;
18     wakeup_latency =
19       finish_wakeup_request - start_wakeup_request;
20   }
21   if (measuring_mwait) {
22     sleep_latency = finish_sleep_request - mwait_start;
23     wakeup_latency = mwait_finish - start_wakeup_request;
24   }
25 }
26 /************ T2 ON OTHER SMT LANE ************/
27 void sleep() {
28   if (measuring_futex)
29     wait_on_futex();
30   else if (measuring_nanonap || measuring_mwait)
31     nanonap();
32 }
33 void nanonap() {
34   ...
35   mwait_start = timestamp();
36   monitor(flag);
37   mwait();
38   mwait_finish = timestamp();
39   ...
40 }
```

Figure 7. Microbenchmark that measures time to sleep with `nanonap`, `mwait`, and `futex`.

We execute each configuration 100 times. Figure 8 presents the time and the 95% confidence interval for using `nanonap`, `mwait`, and `futex` to sleep and wake-up a thread executing in a partner SMT lane. The time to put a lane to sleep for `mwait` is 2 443 cycles, is 3 285 cycles for `nanonap`, and is 11 518 cycles for `futex`, 3.5 times slower than `nanonap`. Waking up a lane directly with `mwait` takes 1 036 cycles — essentially the hardware latency of wake-up. The latency of `nanonap`'s wake-up is

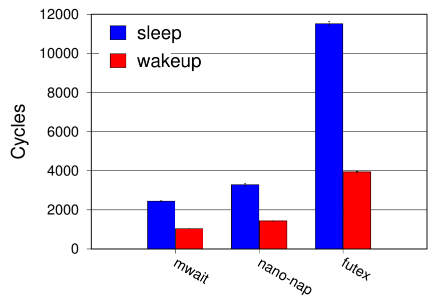


Figure 8. Time to sleep and wake-up SMT partner lanes.

similar to `mwait`'s at 1438 cycles. However, `futex` takes 3 968 cycles, which is 2.72 times slower than `nanonap`. Although `futex` is substantially slower, this latency is likely tolerable, since most idle periods are more than 1 million cycles on our 2 GHz machine (1 ms in Figure 5). However, as explained above, neither the semantics of locks nor user-space calls to `mwait` are adequate for our purposes.

3.3 Continuous Monitoring and Signaling

Sleeping and waking up fast is necessary but not sufficient. The scheduler has to know when to act. We further exploit the `nanonap` mechanism to improve over our SHIM [32] fine-grain profiling tool. SHIM views time-varying software and hardware events as continuous 'signals' (in the signal processing sense of the word). Rather than using interrupts to examine request threads, as many profiling tools do, we configure our batch threads to continuously reads signals from memory locations and hardware performance counters to *profile* request threads. In the simplest case, the profiling observes whether the request thread is executing. Our prior work shows that SHIM accurately observes events at granularities as fine as 100s to 1000s of cycles with overheads of a few percent when executing on another core. However, when threads share an SMT core, we saw similar overheads from SMT sharing as shown in Figure 4. In this paper, we use the `nanonap` mechanism to essentially eliminate this overhead.

Whereas SHIM observes signals from a *dedicated thread*, here we (1) use GCC `-pg` instrumentation [9] to insert checks at method prologues into *C batch workloads* and (2) piggyback on the default Java VM checks at every method entry/exit and loop backedge [22]. These mechanisms add minimal overhead as shown by Lin et al. [22] and, most importantly, remove the need for a third profiling thread to observe request threads.

At each check, the fast-path consists of a few instructions to check monitored signals. For efficiency, this fast path is inlined to the body of compiled methods. If the observed signal matches the condition (e.g., the scheduler sets the memory location that indicates the request lane is idle), the batch thread jumps to an out-of-line function to handle the task of putting itself to sleep.

3.4 ELFEN Scheduling

We design and implement four policies that borrow underutilized resources without compromising SLOs.

Borrowing Idle Cycles The simplest way to improve utilization is to run the batch workload *only* when the latency-critical workload is idle. Section 2 analyzed the maximum CPU utilization of Lucene while meeting a practical SLO at ~70% of one SMT lane, which corresponds with prior analysis of latency-critical workloads [6, 8, 10, 12, 29]. Therefore even when the latency-sensitive workload is executing at the maximum utilization at which it can meet SLOs, there is an opportunity to improve utilization by 30% if the batch workload can borrow this excess capacity. At lower loads, there is even more opportunity.

This policy enforces mutual exclusion. Batch threads execute only when the request lane is empty. When a request starts executing, the batch thread immediately sleeps, relinquishing its hardware resources to the latency-critical request. When the request lane becomes idle, the batch thread wakes up and executes in the batch lane.

Figure 9(a) shows the simple modifications to the kernel and batch workloads required to implement this policy. We add an array called `cpu_task_map` that maps a lane identifier to the current running task. At every context switch, the OS updates the map, as shown in `task_switch_to()`. By observing this signal, the scheduler knows which threads are executing in the SMT lanes. At each check, the scheduler determines whether the `idle_task` is executing in the request lane. If the request lane is idle, the scheduler either continues executing the batch thread in its lane or starts a batch thread. If the request lane is occupied, the scheduler immediately forces the batch thread to sleep with `nanonap`.

Fixed Budget Borrowing idle cycles is simple and as we show, effective, but we can further exploit underutilized resources when requests may incur some overhead and still meet their SLO. In particular, short requests, which typically dominate, easily meet the SLO under moderate loads. We consider the maximum slowdown requests can incur under a certain load as a budget for the batch workload. As an example, consider an SLO latency of 100 ms for 99% of requests. If 99% of requests executing exclusively on the core complete in 53 ms at some RPS, then there exists headroom of $100 - 53 = 47$ ms. We thus could take a budget of 47 ms for executing batch tasks. (We leave more sophisticated policies that also incorporate load along the lines of Haque et al. [10] to future work.)

Given a budget, the fixed-budget scheduler will execute

```
 1 /******************* KERNEL *******************/
 2 /* maps lane IDs to the running task */
 3 exposed SHIM signal: cpu_task_map
 4
 5 task_switch(task T) { cpu_task_map[thiscpu] = T; }
 6 idle_task() { // wake up any waiting batch thread
 7   update_nap_flag_of_partner_lane();
 8   ......
 9   mwait();
10 }
11 /**************** BATCH TASKS *****************/
12 /* fast path check injected into method body */
13 check:
14 if (!request_lane_idle) slow_path();
15
16 slow_path() { nanonap(); }
```

(a) Borrow idle policy.

```
 1 /********** LATENCY CRITICAL WORKLOAD *********/
 2 exposed SHIM signal: queue_len
 3
 4 /**************** BATCH THREADS ***************/
 5 per_cpu_variable: lane_status = NORMAL;
 6 per_cpu_variable: start_stamp;
 7 check:
 8 if (request_lane_idle && queue_len == 0) {
 9   lane_status = NEW_PERIOD;
10 } else if (!request_lane_idle) {
11   slow_path();
12 }
13 slow_path() {
14   switch (lane_status) {
15   case NORMAL:
16     nanonap();
17     break;
18   case NEW_PERIOD:
19     lane_status = CO_RUNNING;
20     start_stamp = rdtsc();
21     break;
22   case CO_RUNNING:
23     now = rdtsc();
24     if (now - start_stamp >= budget)  // expired
25       lane_status = NORMAL;
26 } }
```

(b) Fixed budget policy.

```
 1 /********** LATENCY CRITICAL WORKLOAD *********/
 2 exposed SHIM signals: queue_len, running_request
 3
 4 /**************** BATCH THREADS ***************/
 5 /* Same as the fixed budget policy, except... */
 6 per_cpu_variable: last_request
 7 ...
 8  case NEW_PERIOD:
 9    ...
10    last_request = running_request;
11    ...
12  case CO_RUNNING:
13    if (running_request != last_request &&
14        queue_len == 0) {
15      last_request = running_request;
16      start_stamp = rdtsc();
17    }
18 ...
```

(c) Refresh budget policy.

```
 1 /**************** BATCH THREADS ***************/
 2 /* Same as the refresh budget policy, except... */
 3 ...
 4  case CO_RUNNING:
 5    ...
 6    /* calculate IPC of LC lane */
 7    ratio = ref_IPC / (ref_IPC - LC_IPC)
 8    real_budget = budget * ratio;
 9    if (now - start_stamp >= real_budget)
10      lane_status = NORMAL;
11 ...
```

(d) Dynamic refresh policy.

Figure 9. The pseudocode of four scheduling policies.

batch threads concurrently with requests in their respective SMT lanes when the scheduler is confident that the batch threads will not slow down any request longer than the given budget. Co-running with a request for *T* ms slows down the processing time of the request less than *T* ms. For requests that never wait in the queue, the processing time is the same as the request latency. So, it is safe to co-run with these requests for a budget period. Figure 9(b) shows the implementation of this policy. Line 7 detects when the request queue is empty and renews the budget period, such that the next request will co-execute with the batch thread for the fixed budget.

As we showed in Section 2, the request lane is frequently idle for short periods because after one request finishes there are no pending requests, and most requests are short. The fixed-budget scheduler only uses its budget when a new request that never waits in the queue starts executing in the request lane. When the scheduler detects that a new request starts executing and the `lane_status` is set to `NEW_PERIOD` because the request queue was empty before this requests started, it co-schedules the batch thread in its lane for the budget period. If the request is finished in the period and there are no waiting requests, the scheduler resets the budget and uses it for the next request. When the budget expires, the scheduler puts the batch thread to sleep. When another idle period begins because the request terminates, the request queue is empty, and no other request is executing, the scheduler restarts the batch thread and repeats this process. Note that if *N* requests execute in quick succession without idle gaps, this simple scheduler only co-executes the batch thread with the first request that begins after an idle period. This conservative strategy ensures that each request is only impacted for the budget period of its execution.

Refresh Budget The refresh budget policy extends the fixed budget policy based on the observation that once a request has completed *and* the queue is empty, the budget may be refreshed. The rationale is that the original budget was calculated based on avoiding a slowdown that could prevent the just-completed task from meeting the SLO. Once that task completes, then the budget may be recalculated with respect to the *new* task meeting the SLO. However, because the slowdown imposed by the batch workload is imparted not just on the running request, but on all requests behind it in the queue, we only refresh the budget if the task changes *and* the queue is empty. Figure 9(c) shows the code.

Dynamic Budget The dynamic budget policy is the most aggressive policy and builds upon the refresh budget policy. It uses a *dynamic budget* that is continuously adjusted according to the base budget and the IPC of the latency-critical request. This policy requires us first to profile the IPC with no interference and then to monitor the impact of co-running on request IPC. We implement the monitoring based on the sampling ideas in SHIM [32]. We read the IPC hardware performance counter of the request lane from the batch lane, at high frequency with low overhead. When the latency-critical request's IPC is high, it will be proportionately less affected by the batch workload, so we adjust the dynamic budget accordingly.

4 Methodology

Hardware & OS We use a 2.0 GHz Intel Xeon-D 1540 Broadwell [16] processor with eight two-way SMT cores, a 12 MB shared L3. Each core has a private 256 KB L2, a 32 KB L1D and a 32 KB L1I. The TurboBoost maximum frequency is 2.6 Ghz, TDP is 45 W. The machine has 16 GB of memory and two Gigabit Ethernet ports. We disable deep sleep and TurboBoost.

We use Ubuntu 15.04, Linux version 4.3.0, and the perf subsystem to access the hardware performance counters. We implement the `nanonap` mechanism as a virtual device as shown in Figure 6. We modify the idle task to wake up sleeping batch lanes as shown in Figure 9(a). We expose a memory buffer to user space to determine which tasks are running on which cores.

Latency-Critical Workload We use the industrial-strength open-source Lucene framework to model behavior similar to the commercial Bing web search engine [10] and other documented latency-critical services [6, 8, 12, 29]. Load variation results from both the number of documents that match a request and from ranking calculations. We considered using `memcached`, a key-value store application, because it is an important latency-critical workload for Facebook [11, 27] and a popular choice in the OS and architecture communities. However, each request offers the same uniformly very low demand (<10 K instructions) [11], which means requests saturate the network before they saturate the CPU resources on many servers. Recent work offers OS and hardware solutions to these network scalability problems [4, 21], which we believe if combined with our work would be complementary. We leave such investigations to future work.

We execute Lucene (svn r1718233) in the Open JDK 64 bit server VM (build 25.45-b02, mixed mode). We use the Lucene performance regression framework to build indexes of the first 10 M files from Wikipedia's English XML export [31] and use 1141 term requests from `wikimedium.10M.nostopwords.tasks` as the search load. The indexes are small enough to be cached in memory on our machine. We warm up the server before running any experiments.

We send Lucene requests from another machine that has the same specifications as the server. The two machines are connected via an isolated Gigabit switch. For each experiment, we perform 20 invocations. For each invocation, the client loads 1141 requests, shuffles the requests, and sends requests 5 times. The client issues search requests at random intervals following an exponential distribution around the prescribed RPS mean rate. We report the median result of the 20 invocations. The 95% confidence interval is consistently lower than ± 0.02.

Batch Workloads We use 10 real-world Java benchmarks from the DaCapo 2006 release [5] and three micro C benchmarks, `Loop`, `Matrix`, and `Flush`. The DaCapo benchmarks are popular open-source applications with non-trivial memory loads that have active developers and users. Using DaCapo as batch workloads represents a real world setting. The C micro benchmarks demonstrate the generality of our approach and give us control over the interference pattern. `Loop` calls an empty function and has an IPC of 1. It consumes front-end pipeline resources. `Matrix` calls a function that multiplies a 5×5 matrix, a computationally intensive high IPC workload. It consumes both front-end pipeline and functional-unit resources. `Flush` calls a function that zeros a 32 KB buffer, a disruptive co-runner that flushes the L1D cache.

We run Java benchmarks with JikesRVM [1], release 3.1.4 + git commit fd68163, a Java-in-Java high performance Virtual Machine, using a large 200 MB heap. The JIT compiler in JikesRVM already inserts checkpoints for thread control and garbage collection into function prologues, epilogues and loop back-edges. We add to these a check for co-runner state, as shown in Figure 9. For C micro benchmarks, we use GCC's `-pg` instrumentation option [9] to add checks to method prologues.

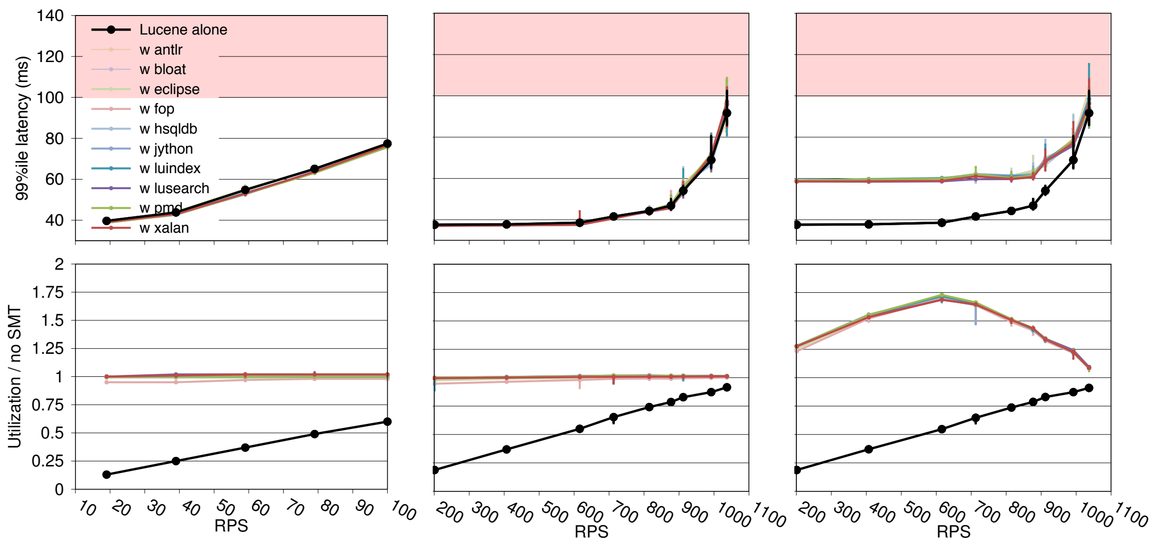
Measurements We use a target, 100 ms request latency for 99% of requests, as our SLO in all of our experiments, which is a practical SLO target for the search engine.

5 Evaluation

This section evaluates the ability of ELFEN to improve server utilization while meeting Service Level Objectives (SLOs) and ELFEN overheads.

Borrow idle We first present ELFEN configurations that use the *borrow idle* policy with DaCapo as the batch workload. This policy minimizes the impact on request latencies. Figure 10(a) plots latency (top) and utilization (bottom) versus requests per second (RPS) on the x-axis for Lucene without (black) and with each of the ten DaCapo batch workloads (colors) executing on one two-way SMT core of the eight-core Broadwell CPU. Figure 10(b) presents these same configurations executing seven instances of each DaCapo benchmark on seven cores. The eighth core manages network communication (receiving requests and returning results), queuing, and starting requests for the latency-sensitive workload. We plot median latency; error bars indicate 10th and 90th percentiles.

The results in Figure 10(a) and 10(b) show that *executing these batch workloads in idle cycles imposes very little impact on Lucene's SLO on a single core or a CMP.*



(a) Borrow idle, one core **(b)** Borrow idle, seven cores **(c)** Dynamic refresh budget, seven cores

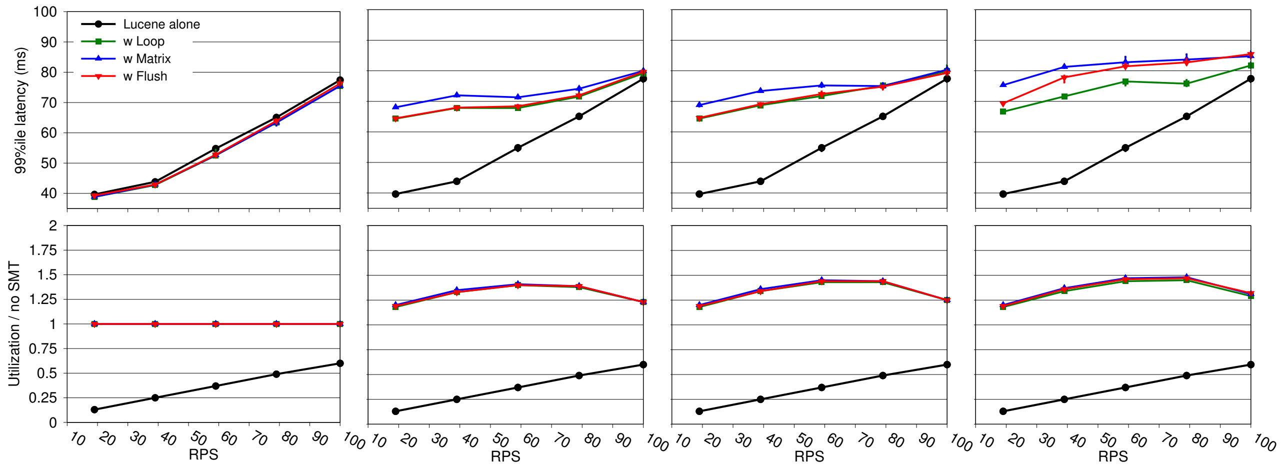
Figure 10. 99th percentile latency (top) and utilization (bottom) for Lucene co-running with DaCapo batch workloads.

ELFEN achieves essentially the same 99th percentile latency at the same requests per second (RPS) with or without batch execution. In fact on one core, ELFEN sometimes delivers slightly lower latencies for Lucene when executing each of the batch workloads in the other lane during idle periods. This results occur because running the batch thread in the other lane causes the core never to enter any of its sleep states. When a new request arrives, the core is guaranteed not to be sleeping, its request lane is empty, and thus the core will service requests slightly faster. With the borrow idle policy, the peak utilization of the core is 100% out of 200% since each core has 2 hardware contexts, but by design, only one is active at a time. Because ELFEN keeps the core busy, executing requests as they arrive in one lane and batch threads with mutual exclusion in the other, it often achieves its peak potential of 100% utilization, but when the utilization of the batch workload is low, the total utilization may be less than 100%.

The chip multiprocessor (CMP) results in Figure 10(b) show better throughput scaling than just a factor of seven. For example, at 60 ms, the single core system can sustain about 70 RPS, while the seven-core system can sustain as much as 1000 RPS. Remember that most requests are short, and long requests contribute most to tail latencies. CMPs better tolerate long request latencies than a single core by executing multiple other short requests on other cores, so fewer short requests incur queuing delay when a long request monopolizes a core. At moderate loads, we again see some improvements to request latency when co-running with batch workloads because the cores never sleep, whereas cores are sometimes idle long enough without co-runners to sleep. However, continuously and fully utilizing all seven cores on the chip incurs more interference, and thus we see some notable degradations in the 99%ile latency at high load. There are two sources of increased latency. First, the effects of managing the queue and request assignment, which shows some non-scalable results. For example, even small amounts of contention for the request queue impacts tail latency independent of ELFEN. ELFEN slightly exacerbates this problem. Second, as prior work has noted and addressed [14, 23, 25], requests and batch threads can contend for shared chip-level resources on CMPs, such as memory and bandwidth. Adding such techniques to ELFEN should further improve its effectiveness.

Increasing Utilization on a Budget Figure 11 presents latency (top graphs) and utilization (bottom) for the four ELFEN scheduling policies described in Section 3: borrow idle, fixed budget, refresh budget, and dynamic refresh on one core. The budget-based policies all borrow idle cycles and trade latency for utilization, slowing the latency-critical requests to increase utilization. Comparing the top row in the figure shows that increasingly aggressive policies cause more degradations in the 99th percentile latency. In these RPS ranges, Lucene's requests meet the 100 ms SLO latency target, but are degraded.

Borrowing idle cycles and co-executing batch threads with requests increases utilization significantly. Comparing across the utilization figures reveals that the budget-based policies further improve utilization compared to borrowing idle cycles. Core utilization rises as load in-



(a) Borrow idle **(b)** Fixed budget **(c)** Refresh budget **(d)** Dynamic refresh

Figure 11. 99th percentile latency (top) and utilization (bottom) for Lucene co-running with C microbenchmarks under four ELFEN policies on a single two-way SMT core.

creases. At moderate loads, ELFEN achieves utilizations over 1.4 for the *fixed budget* policy and 1.5 for the *dynamic refresh* policy. All budget-based policies achieve utilizations of at least 1.2. When the system becomes highly loaded with requests, ELFEN adjusts by executing the co-runners less, and thus total utilization drops. While all of the ELFEN policies are effective at trading off utilization for SLOs, the most aggressive *dynamic refresh* policy consistently runs at higher utilization. The *dynamic refresh* policy is performing precise, fine grain monitoring of request IPC to more accurately and effectively manage this tradeoff. Although we study IPC, ELFEN may monitor and partition other resources, such as memory and cache. Although higher utilization is appealing, some service providers may not be willing to sacrifice throughput of latency-critical tasks, so for them the most practical policy may be to borrow idle cycles.

Figure 10(c) shows the latency and utilization results for the most aggressive dynamic refresh policy on our CMP with DaCapo as the batch workload. This policy degrades the 99th percentile latency by 20 ms before reaching a peak utilization of 1.75 at around 600 RPS. At larger RPS, ELFEN schedules the batch less, system utilization drops and the latency approaches to the same level of the borrow idle policy.

Overhead on Batch Workload Overhead on the batch workload comes from instrumentation, interference with the latency-sensitivity requests, and being frequently paused and restarted. As we pointed out above, Lin et al. [22] show the instrumentation overheads are low, at most a couple percent.

Figure 12 measures these other overheads. It presents the execution time, user time utilization, and user level IPC of each DaCapo benchmark co-running with Lucene normalized to its execution alone on one core. When co-running, we use the borrow idle policy and load the Lucene at 80 RPS, which leads to about 50% utilization for both Lucene and each DaCapo benchmark. The execution time of co-running each DaCapo benchmark increases by 49% on average as predicted by the 50% utilization. There are small variations in these slowdowns, but none of them are due to DaCapo programs executing more instructions when co-running — the number of retired instruction at user level is the same. Furthermore, DaCapo does not execute instructions less efficiency, because IPC decreases are only 1%.

Variation in execution times is due to variations in utilization already present in the DaCapo benchmarks. If the batch workload is idle for some other reason (e.g., waiting on I/O or a lock), then a request that forces it to stop executing will affect it less. The more idle periods the batch workload has, the less execution is degraded. This effect causes normalized execution time and utilization to be strongly correlated. For instance, the *pmd* benchmark incurs the largest slowdown in execution time, 59%, and the largest utilization reduction, 36%. The *fop* benchmark has the lowest native utilization in these benchmarks. Consequently, it has both the smallest slowdown and the smallest utilization reduction, 47% and 26%.

6 Related Work

Exploiting SMT Lo et al. [23] demonstrate that naively co-running batch workloads with latency-critical workloads violates Google's SLO, even under light load. They show that for many latency-critical workloads, uncon-

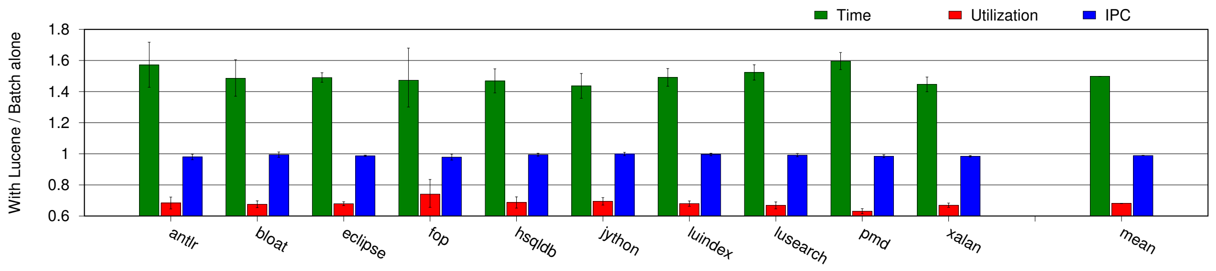


Figure 12. Normalized DaCapo benchmark execution time, user space CPU utilization, and IPC.

trolled interference due to SMT co-location is unacceptably high, and conclude that it is not practical to share cores with SMT. Our results contradict this conclusion.

Ren et al. [29] exploit SMT for a server that exclusively handles latency-sensitive requests (no co-location) and thus requires over-provisioning to handle load spikes. Herdrich et al. [14] note that achieving latency objectives with current SMT hardware is challenging because the shared resources introduce contention that make it hard to reason about and meet SLOs. They propose SMT rate control as a building block to improve fairness and determinism in SMT, which dynamically partitions resources and implements biased scheduling. These mechanisms should help limit interference on requests and complement our approach. They do not evaluate latency-critical workloads, seek to borrow idle cycles, or offer a fine-grain thread-switching mechanism, as we do here.

Accommodating Overheads Zhang et al. [35] use offline profiling of batch workloads to precisely predict the overhead due to co-running with latency-critical requests on SMT. They then carefully provision resources to co-run batch workloads whilst maintaining SLOs for latency-critical workloads. Unlike our work, they do not attempt to minimize the overhead of co-running batch workloads. Rather, they predict and then accommodate it. They measured interference due to co-run batch workloads in the range of 30%-50%.

POSIX Real-Time Scheduling Leverich and Kozyrakis [20] propose using POSIX real-time scheduling disciplines to prioritize requests over co-run batch threads. When hardware contexts are scarce, this approach ensures that latency-critical requests have priority — batch threads will be the first to block. When given sufficient hardware contexts however, the approach does not control for interference due to co-running. Thus it does not address the problem we address here: avoiding interference due to co-running while utilizing SMT.

Exposing and Evaluating `mwait` Anastopoulos and Koziris [2] use `mwait` to release resources to another SMT thread when waiting on a lock. Wamhoff et al. [30] make `mwait` user-level visible and then use it to put cores into sleep states so as to provide power headroom for DVFS to boost performance on other cores which are executing threads on the program's critical path. They measure the latency of putting an entire core into a C1 sleep state on an Intel Haswell 4 770 and found that it was 4 655 cycles. This result is broadly consistent with our measurements, which are for a single hardware context on a more recent processor. With regard to semantics, Meneghin et al. [26] claim fine-grain thread communication requires user-level mechanisms, whereas we offer an intermediate point that involves the OS, but not the OS scheduler. None of this prior work has the same semantics as `nanonap` for hardware control, which we exploit for both fine-grain monitoring and scheduling.

7 Conclusion

This paper shows how to use SMT to execute latency-critical and batch workloads on the same server to increase utilization *without* degrading the SLOs of the latency-critical workloads. We show, given a budget, how to control latency degradations to further increase utilization while meeting SLOs. Our policies borrow idle cycles and control interference by reserving one lane for requests and one for batch threads. By reserving SMT lanes, ELFEN always immediately executes the next request when the previous one completes or a new one arrives. Using low-overhead monitoring and `nanonap`, ELFEN responds promptly to release core resources to requests or to control interference from batch threads. Our principled borrowing approach is extremely effective at increasing core utilization. Whereas current systems achieve utilizations of 5% to 35% of a 2-way core (by only using one lane at 10% to 70%) while meeting SLOs, ELFEN's *borrow idle* policy uses both lanes to improve utilization at low load by 90% and at high load by 25%, delivering consistent and full utilization of a core at the same SLO. On CMPs, ELFEN with the borrow idle policy is extremely effective as well, achieving its peak utilization without degrading SLOs for all but the highest loads. No prior work has managed this level of consistent server utilization without degrading SLOs.

References

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. J. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM Project: Building an open source research community. *IBM System Journal*, 44(2):399–418, 2005.

[2] N. Anastopoulos and N. Koziris. Facilitating efficient synchronization of asymmetric threads on hyper-threaded processors. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–8, April 2008.

[3] Apache Lucene. `http://lucene.apache.org/`, 2014.

[4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 49–65, 2014.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, Oct. 2006.

[6] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 205–220, 2007.

[8] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 127–144, 2014.

[9] GCC. Program instrumentation, 2016. URL `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`.

[10] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and K. S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 161–175, 2015.

[11] S. Hart, E. Frachtenberg, and M. Berezecki. Predicting memcached throughput using simulation and modeling. In *Symposium on Theory of Modeling and Simulation (IMS/DEVS)*, pages 40:1–8, 2012.

[12] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–238, 2015.

[13] Y. He, S. Elnikety, J. Larus, and C. Yan. Zeta: Scheduling interactive services with partial execution. In *ACM Symposium on Cloud Computing (SOCC)*, pages Article 12: 1–14, 2012.

[14] A. Herdrich, R. Illikkal, R. Iyer, R. Singhal, M. Merten, and M. Dixon. SMT QoS: Hardware prototyping of thread-level performance differentiation mechanisms. In *HotPar*, pages 219–230, 2013.

[15] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[16] Intel. Intel Xeon processo d-1540, 12m cache, 2.00 GHz, 2013. URL `http://ark.intel.com/products/87039/Intel-Xeon-Processor-D-1540-12M-Cache-2_00-GHz`.

[17] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. Speeding up distributed request-response workflows. In *ACM SIGCOMM*, pages 219–230, 2013.

[18] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, pages 158–169, 2015.

[19] S. Kim, Y. He, S.-W. Hwang, S. Elnikety, and S. Choi. Delayed-Dynamic-Selective (DDS) prediction for reducing extreme tail latency in web search. In *ACM International Conference on Web Search and Data Mining (WSDM)*, pages 7–16, 2015.

[20] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality of service. In *ACM European Conference on Computer Systems (Eurosys)*, pages Article 4:1–14, 2014.

[21] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[22] Y. Lin, K. Wang, S. M. Blackburn, M. Norrish, and A. L. Hosking. Stop and Go: Understanding yieldpoint behavior. In *ACM International Symposium on Memory Management (ISMM)*, pages 70–80, 2015.

[23] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, pages 450–462, 2015.

[24] J. R. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 50–61, 2001.

[25] 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 *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 248–259, 2011.

[26] M. Meneghin, D. Pasetto, H. Franke, F. Petrini, and J. Xenidis. Performance evaluation of interthread communicationmechanisms on multicore/multithreaded architectures. In *ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 131–132, 2012. URL `http://researcher.watson.ibm.com/researcher/files/ie-pasetto_davide/PerfLocksQueues.pdf`. Extended version at url, accessed May 2016.

[27] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[28] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 219–230, 2013.

[29] S. Ren, Y. He, S. Elnikety, and K. S. McKinley. Exploiting processor heterogeneity in interactive services. In *ACM International Conference on Autonomic Computing (ICAC)*, pages 45–58, 2013.

[30] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The TURBO diaries: Application-controlled frequency scaling explained. In *USENIX Annual Technical Conference (USENIX ATC)*, pages 193–204, 2014.

[31] Wikipedia. Wikipedia:database download, 2016. URL `https://en.wikipedia.org/wiki/Wikipedia:Database_download`. Accessed January 2016.

[32] X. Yang, S. M. Blackburn, and K. S. McKinley. Computer performance microscopy with Shim. In *ACM/IEEE International Conference on Computer Architecture (ISCA)*, pages 170–184, 2015.

[33] X. Yang, S. M. Blackburn, and K. S. McKinley. ELFEN sched-

uler open source implementation, June 2016. URL `https://github.com/elfenscheduler`.

[34] J. Yi, F. Maghoul, and J. Pedersen. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*, pages 257–266, 2008.

[35] 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 *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, pages 406–418, 2014.