Rage Against the Context Switch

Tony Yu

April 2024

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Undergraduate Thesis Committee:
Dimitrios Skarlatos, Chair

Submitted in partial fulfillment of the requirements
for the SCS Honors Undergraduate Research Thesis.

Keywords: Operating Systems, Hardware-Software Co-Design

For my parents, who have been wonderfully supportive at every step of my journey here at CMU

Abstract

Modern computers must have the capability to handle hundreds of processes. A core component of this capability is the context switch, wherein the state of a process is saved so that execution can be paused and resumed at a later point, freeing the CPU for use by another process. Context switches allow processes to share CPUs and help computers to hide stalls from blocked processes. However, context switches are computationally intensive, and typically have a negative impact on system performance. Despite this, context switches are extremely prevalent in all computer systems, as they are necessary to handle I/O interrupts, system calls, VM-exits, core scheduling and preemption, and other multiprocessing operations.

A key design paradigm is that core scheduling and preemption is a responsibility delegated to the operating system. Using user-space instrumentation, we quantify the consequent context switching and kernel thread preemption overheads and show that they are significant in common multiprocessing settings. We also present Nemo, a novel CPU simulator for the Structural Simulation Toolkit. We also propose means by which such overheads can be eliminated from modern computing systems.

Acknowledgments

Thank you to my advisor Dimitrios Skarlatos for his seemingly limitless patience, and to Kevin Xue and Kaiyang Zhao of the CMU CAOS group for their technical support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed systems are the foundation on which many of the services we rely on today are built on. Search engines, cloud computing, and databases are just a few examples of services that would be impossible without efficient distributed computing. In all of these applications, performance is critical not just for user satisfaction, but for basic functionality. Historically, a reliance on Moore's law to compensate for ever-increasing demands on high performance computing services has largely paid off in the favor of software engineers. However, with the death of Moore's law, our overreliance on hardware improvements to solve performance woes has become a significant liability.

This hunt for performance gains in software has inspired a recent interest in hyper-optimized scheduling algorithms in distributed computing. Some improvements, such as Caladan [3] and Shinjuku [6], are implemented at the operating system (OS) level. Although such systems are highly effective at improving preemption latencies and performance in distributed systems, they can be technically difficult to deploy. Moreover, these schedulers may require additional fine-tuning based on the application, which can lead to service downtimes and lost productivity. On the other hand, user-space modifications to the scheduler core, such as ghOSt [5], are easier to deploy and tune, but perform slightly worse than kernel-level custom schedulers.

1.1 Thesis Contributions

Recently, there has been a paradigm shift towards moving functionality out of the operating system and into hardware. Hardware accelerators such as Intel's Infrastructure Processing Unit have been shown to bring large performance benefits and are also economically viable. In the spirit of this shift, we investigate hardware-level solutions to scheduling inefficiencies. We believe that the slow pace of hardware performance improvements has biased research away from investigating hardware-level optimizations to task scheduling in modern multiprocessing systems. However, a key obstacle is that there is no recent study on the performance of the Linux kernel under heavy loads. Such a study is crucial for motivating future hardware design.

In this thesis, we demonstrate the overhead of scheduling and interrupts under four different webs-server workloads using user-space instrumentation of the Linux kernel. We use both

QEMU full system emulation and real system benchmarking to demonstrate these overheads. We also present progress on Nemo, an out-of-order CPU simulator that we hope to apply to future work in this direction. We also use the results of our benchmarking to propose methods by which context switch latency can be decreased.

Chapter 2

Background

We now provide some background on threading, scheduling, and their associated overheads.

2.1 Threading in Modern Multiprocessor Systems

Modern processors have the ability to work on multiple programs at once. A program is merely a set of instructions that the user wants the computer to execute. When a program is run, it spawns a *process*, which is an instance of a program. Within each process, there exists one or more *threads*, which are the smallest unit of sequential execution that can be managed by the OS scheduler. These threads are known as software threads, since they simply represent a stream of code execution.

While programs and processes are simply abstractions that assist programmers in reasoning about what happens "under the hood," threads are firmly grounded in the hardware implementation of processor units. Each CPU supplies at least one execution context on which a software thread can run; this execution context is often referred to as a hardware thread. Figure **??** illustrates a hardware execution context. Modern CPUs can provide over 32 hardware threads in a single unit [], which are organized and grouped to form cores. Despite this low number of hardware threads, even single-processor systems can support hundreds or thousands of software threads at once despite having a much smaller amount of physical cores. The ability to multiplex software threads onto hardware threads is the linchpin of multiprocessing.

To facilitate the multiplexing of software threads onto individual hardware threads, it must be possible for the execution of any software thread to be paused and the corresponding architectural state to be saved and loaded. The process of saving one thread's state and loading another is known as a context switch. In current state-of-the-art OS and hardware implementations, context switches represent an unavoidable source of performance degradation, the magnitude of which depends on the underlying instruction set architecture and threading implementation. In general, context switches are considered expensive events, due to both the direct processing demands and the architectural side-effects of context switching.

2.2 Scheduling and its Pitfalls

Scheduling is the process by resources are partitioned to facilitate processing of different tasks. It is well-known that CPU scheduling is an NP-hard problem [9], therefore, any optimal solution requires non-polynomial runtime. Consequently, scheduler software cannot make optimal decisions in real time. Typically, schedulers adhere to a *scheduling discipline*, a set of rules or heuristics that are applied to approximate the optimal solution to the scheduling problem. Common disciplines include first in, first out; priority scheduling; and fair scheduling.

Scheduler software is usually implemented in the kernel and typically operates with a limited set of algorithms. There is also typically little flexibility for the user to customize or fine-tune the scheduling algorithm to a particular use case. In this thesis, we are primarily concerned with the Linux scheduler, as Linux is the operating system most commonly used in large datacenter applications. Since Linux kernel version 2.6 [1], the Completely Fair Scheduler (CFS) has been the default scheduler in Linux. The CFS tracks a virtual runtime for each thread and maintains a red-black tree that represents an ordering of future tasks to be run. Whenever a process reaches either a maximum runtime or is stopped by any means, the process is reinserted into the tree, and the process with the least virtual runtime is selected to run.

The CFS functions well in a the typical consumer use-case. However, for applications with heavy performance demands, the CFS performs poorly compared to more specialized algorithms, primarily due to a combination of poor preemption latencies and suboptimal resource partitioning. The default timeslice length used by the Linux scheduler is 100 milliseconds, therefore, if a particular load balancing decision is made poorly or otherwise negatively impacts performance, the issue cannot be fixed until the impacts of the scheduling decision are felt by the end-user.

2.3 Communication in Multiprocessing Systems

A critical component of multiprocessing systems is the ability for processes and cores to communicate with one another. In Linux, this is typically performed through UNIX signals or interprocessor interrupts (IPIs). In this thesis, we focus on IPIs, since these interrupts are responsible for expensive events such as TLB flushes and task preemption. Linux IPIs come in three flavors:

1. `RESCHEDULE_VECTOR` - instructs another CPU to make a call to `schedule()`, thereby preempting whatever task is currently running on the CPU.
2. `CALL_FUNCTION_VECTOR` - instructs another CPU to make a call to a function, and clears any pending IPI function callbacks.
3. `CALL_FUNCTION_SINGLE_VECTOR` - as of Linux 6.2.0-rc7, this interrupt is handled the same way as `CALL_FUNCTION_VECTOR`.

Chapter 3

The Nemo CPU Simulator

Nemo is an out-of-order CPU simulator that is compatible with the Structural Simulation Toolkit (SST). Nemo is comprised of a frontend using the QEMU Tiny Code Generator and a backend core that is connected to SST. The data flow of the Nemo simulator is illustrated in 3.1. The goal

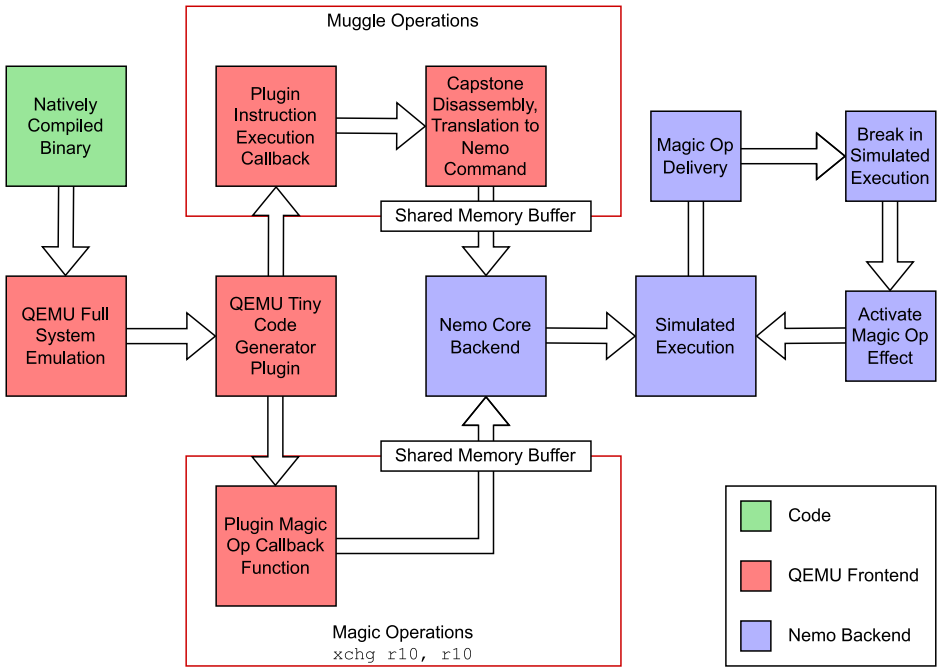


Figure 3.1: Nemo simulation dataflow

of the Nemo simulator is to provide an easy interface on which binaries can be directly instrumented in simulation without needing conversion to a trace. This presents a significant advantage in comparison to other well-established CPU simulators, such as gem5, as trace generation can be a slow, expensive process that detracts from software engineer and researcher productivity.

3.1 The QEMU TCG frontend

The frontend of the Nemo simulator is an instance of the QEMU full-system emulator with a custom TCG plugin. For more information on QEMU TCG plugins, we refer the reader to []. The user can run an application of their choosing in the QEMU emulator. The TCG plugin installs callback functions to instruction execution events and memory access events. In the case of an instruction execution event, the raw bytecode is sent to the Capstone disassembly engine [], where the original `x86` assembly instruction is extracted. These instructions are then translated to `NemoCommands`, which hold information on how the instruction should be executed. In the case of a memory access event, the QEMU plugin intercepts the salient details of the memory access (such as the physical and virtual address of the accessed memory) and packages those in a `NemoCommand`. Finally, the `NemoCommands` are sent through a shared memory buffer to the Nemo core backend, where simulated execution occurs. This mechanism is similar to micro-operation generation on modern processor units.

Nemo is designed with ease of instrumentation in mind. A common way to model ISA extensions and instrumentation is through special non-functional no-ops, or "magic operations" [7]. Nemo implements these in an extensible way. Our QEMU TCG plugin considers `xchg` instructions on the infrequently used `r12` register to be magic operations. This instruction is selected due to its rarity in compiler-optimized code. When the plugin intercepts `xchg r12, r12`, it discards this instruction and instead reads the value of the `r12` register. It then uses this value to select a magic operation, and forwards the message to the simulator backend. The magic operation is then handled in a simulator-defined method.

3.2 The Nemo Core backend

The Nemo core simulator models an out-of-order CPU, and is loosely modeled off of the gem5 O3CPU model [2] and the Ariel CPU simulator [8]. Using SST allows us to parameterize our CPU model with a user-supplied `python` configuration file, which allows the user to specify fetch width, decode width, instruction queue size, load/store queue size, and reorder buffer size, among other CPU characteristics. Figure 3.2 shows an example configuration for the Nemo CPU.

3.2.1 Pipeline Stages

The Nemo Core models six pipeline stages: fetch, decode, rename and issue, execute, writeback, and commit:

1. Fetch - In the fetch stage, `NemoCommands` are received from the TCG plugin and added to the fetch queue (similar to a prefetch queue). Magic operations are directly handled in the fetch stage, which allows for maximum flexibility in magic operation usage. This stage also handles branch prediction.
2. Decode - In the decode stage, `NemoCommands` are read off of the fetch queue and translated into decode entries. Read and write events are also spawned in the decode stage.

```
nemo = sst.Component("N0", "nemo.nemo")
nemo.addParams({
    "branchMispredPenalty" : "5",
    "clock" : "2GHz",
    "max_insts" : "2000000000",
    "max_cycles" : "0",
    "verbose" : "0",
    "fetch_width" : "5",
    "decode_wdith" : "5",
    "maxcorequeue" : "512",
    "maxtranscore" : "256",
    "iq_size" : "160",
    "rob_size" : "352",
    "maxissuepercycle" : "4",
    "lq_size" : "128",
    "sq_size" : "72",
    "int_ports" : " 2",
    "int_mul_ports" : "1",
    "int_div_ports" : "1",
    "fp_mul_ports" : "2",
    "fp_div_ports" : "1",
    "load_ports" : "6",
    "store_ports" : "6",
    "pipetimeout" : "0",
    "nemomode" : "1",
    "defaultlevel" : "0",
    "shmem_file_name": "simulator_interface_tonyy_0"
})
```

Figure 3.2: An example configuration of the Nemo core

3. Issue - In the issue stage, decode entries are converted into reorder buffer (ROB) entries and instruction or load/store queue entries. Register renaming also occurs in this stage.
4. Execute - In the execute stage, instruction and load/store queue entries are dispatched to their corresponding functional units. No actual instruction execution is done; instructions have fixed latency depending on functional units except for memory operations.
5. Writeback - In the writeback stage, completed instructions propagate their results to any registers which might be waiting on their results. Also, read events are directly executed in this phase.
6. Commit - In the commit stage, architectural updates made by instructions are committed in-order. Also, write events are executed once committed.

Our CPU model does not explicitly implement reservation stations or similar hardware constructs for Tomasulo's algorithm, similar to gem5. Instructions are directly updated and marked ready in the instruction queue.

Chapter 4

Methodology

4.1 Kernel Modifications

All experiments were executed on a modified version of Linux kernel version `6.2.0-rc7`. Typically, application instrumentation is concerned with hardware performance metrics which can be obtained by reading from performance counters. In our study, we are primarily focused on the performance overhead measured in time. Therefore, kernel-level time APIs like the `ktime` API are almost completely sufficient for our instrumentation. We address some limitations of this instrumentation below.

To implement the instrumentation, we wrap the code associated with the Linux CFS scheduler core and Linux SMP IPI send and handle functions with calls to `ktime_get_ns()` and take the difference between the measured start time and the measured end time as the runtime of the corresponding function. This instrumentation carries minimal overhead as compared to an unmodified kernel. Moreover, we only measure the time elapsed after initialization of the instrumentation and before the post-processing necessary to generate data, further minimizing the performance impact of our kernel modifications. With this in mind, we reasonably anticipate that the instrumentation is faithful to real-system performance.

We also provide an interface through which a user can easily toggle if instrumentation is desired and retrieve results. This is done through an extension to `procfs`. Users can `write()` to virtual files located in `/proc` to start instrumenting relevant sections of the code. The corresponding `read()` stops the instrumentation, and any text editor or the `cat` utility can be used to retrieve the data.

There are some limitations to the instrumentation as implemented. First, `ktime_get_ns()` is not always strictly monotonic. Therefore, some data points are discarded since the measured start time is actually later than the measured end time. Experimentation in QEMU shows that this occurs extremely infrequently relative to the total number of data points; therefore, we do not anticipate this issue significantly impacting our results. Secondly, our instrumentation measures *all* context switches and interrupts in a given execution interval. This means that our results are prone to being influenced by system noise. Finally, this instrumentation does not capture the overhead of transitions between kernel and user modes.

4.2 Adaptation of LEBench

We also evaluate context switching latencies using the LEBench context switch microbenchmark [], and adapt this microbenchmark into a simple toy benchmark to demonstrate the degradation of context switch performance when complex states are involved. The basic LEBench context switch benchmark is described in Algorithm 1. The key to this microbenchmark is that the

Algorithm 1 LEBench Context Switch Instrumentation

```
Process 1:
    Open pipe 1
    Open pipe 2
    read from pipe 1
    write to pipe 2
    read from pipe 1
    ...
Process 2:
    Open pipe 1
    Open pipe 2
    write to pipe 1
    read from pipe 2
    write to pipe 1
    ...
```

`read()` and `write()` are blocking when used on pipe file descriptors. Thus, stalling on these system calls forces the processes to yield to each other, invoking a context switch. The advantage of this microbenchmark is that it captures the cost of transitions between CPU privilege levels. However, it does not accurately capture context switching latency in real applications, since there is no meaningful state being manipulated between context switches.

In order to better test context switching under complex states, we slightly change the code of this microbenchmark, as shown in Algorithm 2 (only relevant code is shown). Process 1 performs blocked matrix multiplication, with the timer signals forcing context switches to process 2 by the same mechanism as in LEBench. Process 2 then immediately returns control to process 1. The matrix multiplication function heavily utilizes registers, including the `xmm` vector/float registers. The register usage means that a complex architectural state may arise between context switches, which better models the behavior of real applications. We discuss the results of both LEBench and our modified microbenchmark on the entropy server in Section 5.1.

4.3 Selected Workloads and Experimental Conditions

For our experiments, we selected four common webserver applications as workloads: `nginx`, `httpd`, `memcached`, and `redis`. We benchmarked the performance of `nginx` and `httpd` with `siege`[4]. We benchmarked the performance of `redis` with `redis-benchmark` and `memcached` using `memtier-benchmark`. Of our workloads, only `httpd` was run and installed as a daemon, all other applications were run as programs compiled from source.

Algorithm 2 Matrix Multiplication Context Switch Instrumentation

```
Process 1:
void handler1(int signo) {
    struct timespec startTime, endTime;
    char w = 'a', r;
    clock_gettime(CLOCK_MONOTONIC, &startTime);
    write(fds1[1], &w, 1);
    read(fds2[0], &r, 1);
    clock_gettime(CLOCK_MONOTONIC, &endTime);
    add_diff_to_sum(&sum, endTime, startTime);
    itrs++;
    alarm(INTERVAL);
}
...
// Open pipe 1
// Open pipe 2
struct sigaction act = { 0 };
act.sa_handler = &handler1;
retval = sigaction(SIGALRM, &act, NULL);
// Initialize random matrix of floats and do multiplication
alarm(INTERVAL)

Process 2:
// Open pipe 1
// Open pipe 2
while(true) {
    read(fds1[0], &r, 1);
    write(fds2[1], &w, 1);
}
```

4.4 QEMU Experiment Methodology

For experiments done in QEMU, the `qemu-system-x86_64` emulator was used with Linux Kernel-based Virtual Machine (KVM) acceleration. The CPU fall-through option was used to avoid ISA compatibility issues. For all experiments, an SMP system with two single-core CPUs was simulated by QEMU. The workloads and benchmarking programs were run together on the same instance of QEMU.

4.5 Real System Experiment Methodology

For experiments on real systems, the custom kernel was installed on Parallel Data Laboratory's entropy server. The specs of the entropy server are listed in Table 4.1. The applications and benchmarks were then run together on the server.

CPU	Intel Xeon Gold 5320 (2 threads/core, 26 cores/socket, 2 sockets)
L1 dcache	48 KB per core, 12-way associativity
L1 icache	32 KB per core, 8-way associativity
L2 cache	1.28 MB per core, 20-way associativity
L3 cache	39 MB per socket, 12-way associativity
DRAM	509 GiB total memory
OS	Ubuntu 22.04 (Linux kernel 6.2.0-rc7)

Table 4.1: Specification of the entropy server

Chapter 5

Results

We now discuss the results of our experiments.

5.1 Analysis of LEBench Results

In our expeirments using LEBench, we ran the same matrix multiplication workload with increasing intervals between forced context switches. The results are summarized in Figure 5.1. The data show that as the interval between forced switches increases, the latency of a context switch increases as well. This implies that the more complex the state of the thread is, the longer a context switch takes, down to some minimum. We conclude from this that there is some baseline cost of a context switch that cannot be avoided, and points towards *predictive* hardware-implemented scheduling as the solution to the issue of scheduling-related overheads. Furthermore, this indicates that further optimization would require completely removing the need for context switching through hardware means.

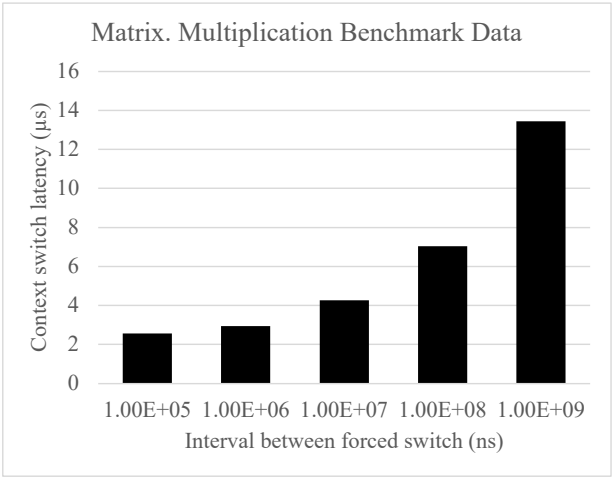


Figure 5.1: Context switching data from the modified LEBench benchmark

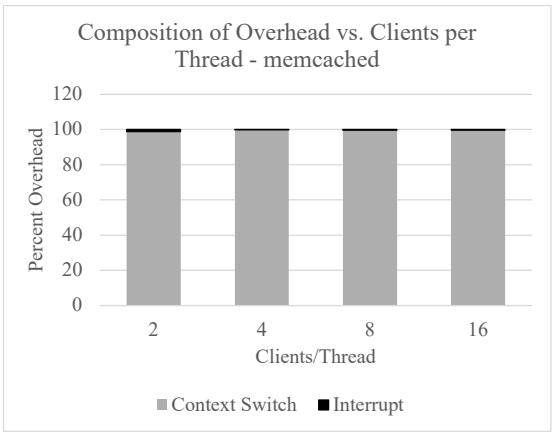
5.2 Analysis of QEMU Workload Results

Our experiments on QEMU served as both a proof of concept as well as demonstrating that context switching and interrupt delivery/handling represents a significant overhead in modern applications. The `memcached` workload demonstrated overheads of around 8% or more under all configurations tested. Roughly 99% of this overhead is attributable to context switching, as shown in Figures 5.2a and 5.2b. For the experiments in Figure 5.2c, 20 threads and 524288 requests per client were used across all conditions, and the number of clients per thread was the only configuration variable that was manipulated. For the experiments in Figure 5.2d, 20 clients per thread and 524288 requests per client were used across all conditions, and the number of threads was the only configuration variable that was manipulated.

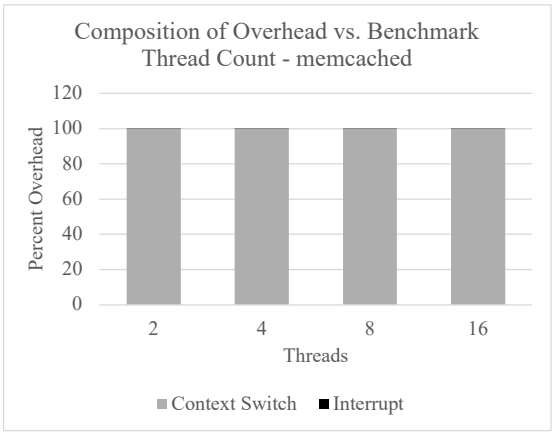
As can be seen from the data, the percent overhead does not necessarily strictly increase as the server load increases. However, as the thread count increases, the overhead incurred comprised a larger percentage of the total runtime (from around 9% to around 10%), and the percentage of the overhead attributable to interrupts decreases as well (from around 0.3% to around 0.2%). This confirms our expectation that the amount of context switches should increase if the thread count increases. However, we expected the increase to be more substantial. We also anticipated a substantial increase in the overall overhead. The data show that this is clearly not the case. This discrepancy is likely due to the single-threaded nature of the `memcached` application. The single `memcached` thread is unlikely to be ”booted” off of an active core, since it is always busy handing connection requests from the benchmarking application. The benchmarking application is similarly unlikely to be forced to yield to a different thread or process, since the only blocking call in the benchmark is socket connection. Thus, we expect low rates of preemption outside of regular scheduler preemption, and hence we don’t anticipate many interrupts relative to the total number of context switches.

Experiments on `nginx` showed very low overheads in comparison to `memcached`. The same pattern of overhead composition was observed, with an outlier point at 8 worker threads, as seen in Figures 5.3a and 5.3b. Neither the worker thread count nor the number of total requests received by the server seemed to affect the observed overhead nor the constitution of the observed overhead, as seen in 5.3. This is an unexpected result; we would anticipate extra worker threads for `nginx` to cause additional overheads. This is especially true when running multithreaded web applications on QEMU, since our QEMU emulator only simulated two-core SMP. The reasons for the lack of overhead is unclear. It is possible that `siege` did not generate enough requests to force contention between the threads, or that the number of concurrent threads used by `siege` was too much for the QEMU environment. It is also possible that the ”optimal” distribution of processor time (as decided by the Linux CFS) was simply to let each core run until its virtual timeslice was consumed or the currently running process was no long the most in need of runtime, and then to handle preemption at the scheduler tick instead of using an intercept. Overall, the behavior of the `nginx` in QEMU is not particularly easy to handle.

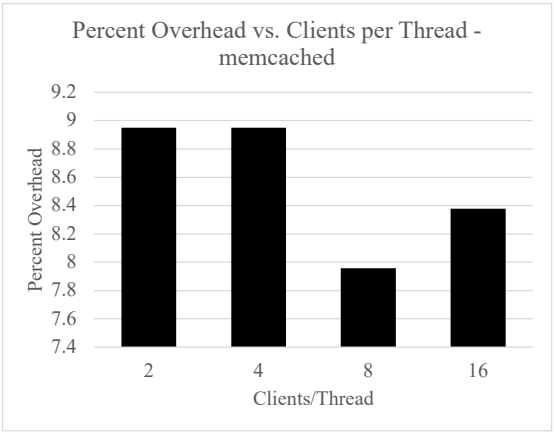
Experiments on `httpd` also showed very low overheads. Interestingly, interrupts composed more of the overheads (roughly 25%) associated with `httpd`, as shown in Figure 5.8a. This is most likely due to the application having four worker threads which are each doing priority



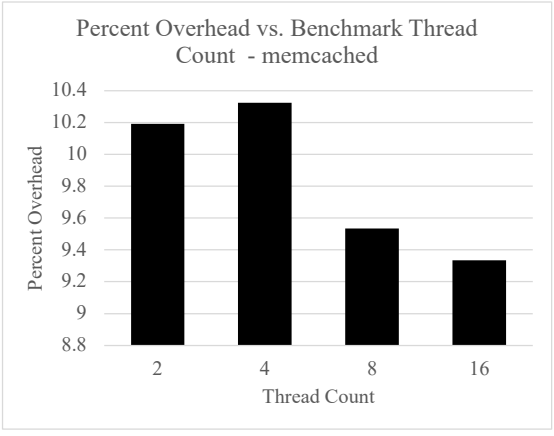
(a) Composition of overhead vs clients/thread



(b) Composition of overhead vs thread count



(c) Percent overhead vs clients per thread

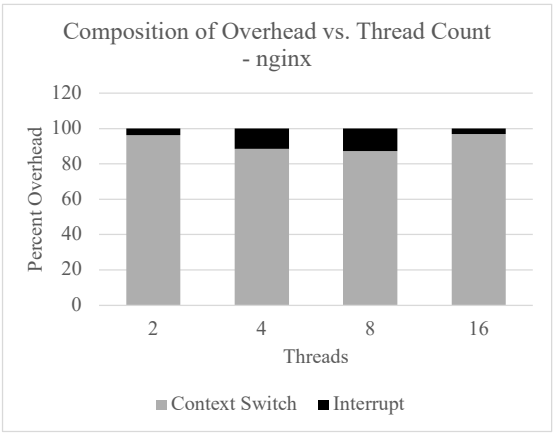


(d) Percent overhead vs threads

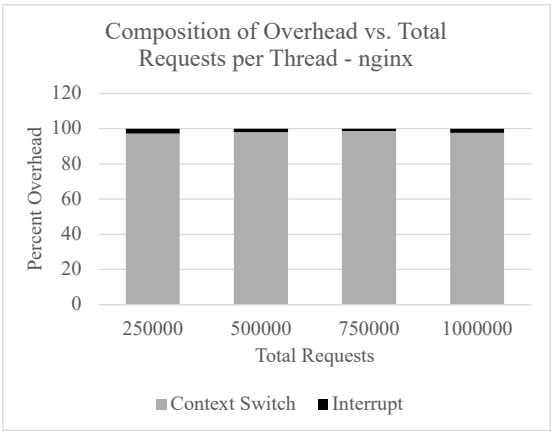
Figure 5.2: Overheads of varying configurations of `memcached` on QEMU.

tasks with potentially poor load balancing. This may necessitate extra communication between processors to force preemption and allow faster, more efficient tasks to execute first. Once again, the total number of requests had no influence on the percent overhead or on the composition of the overhead. One interesting feature of the `httpd` workload is that it was substantially slower than `nginx` even while using more threads than `nginx`.

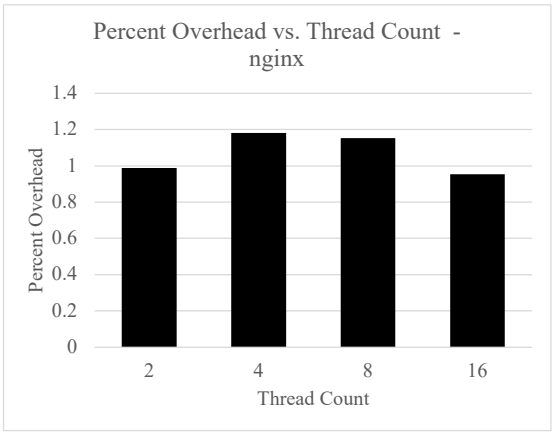
The inefficiency of `httpd` may have contributed to scheduling difficulties and therefore a higher demand for interrupts. It is possible that individual worker threads of `httpd` must be preempted in the middle of performing expensive computation, which would require an extra signal delivery and an expensive context switch event. It is possible that poor individual thread performance leads to poor load balancing outcomes in the CFS, which would also lead to excessive preemption. These hypotheses are supported by the fact that the overhead increases as the number of requests processes increases, as shown in Figure 5.4. This implies that as the number of requests increases, the ability of the CFS to make close-to-optimal scheduling choices that minimize the amount of preemption necessary is degraded.



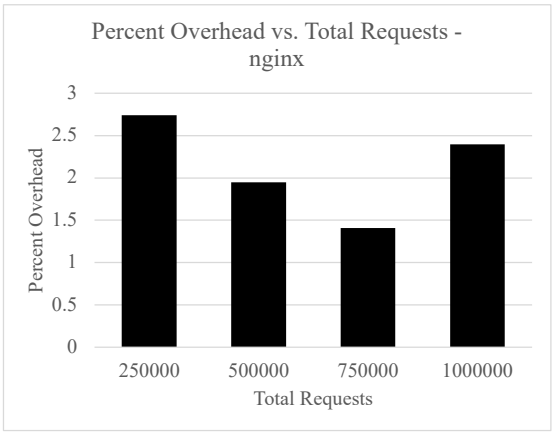
(a) Composition of overhead vs thread count



(b) Composition of overhead vs total requests



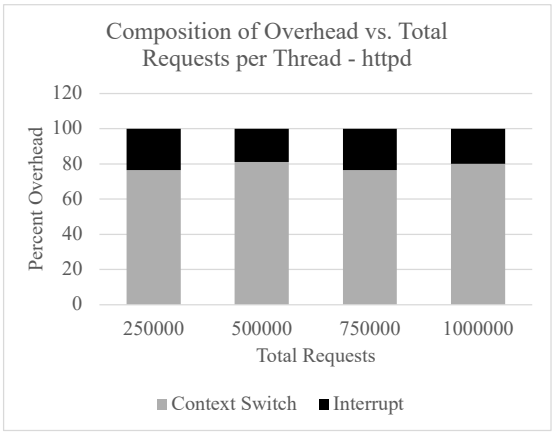
(c) Percent overhead vs thread count



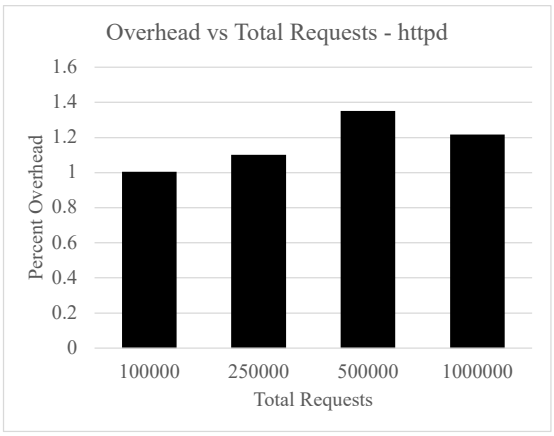
(d) Percent overhead vs total requests

Figure 5.3: Overheads of varying configurations of `nginx` on QEMU.

Experiments on `redis` showed moderate overheads on the order of roughly 3.5%. The data indicate that the benchmark thread count had little effect on the overhead, as shown in Figure 5.5. Of the four workloads, `redis` had the highest proportion of overhead accounted for by interrupts. This is most likely due to `redis` writing information to disk (i.e. self-interrupts) and not due to scheduling-related interrupts. We would not expect the `redis` to demand a particularly large amount of signals for preemption purposes, since it is a single-threaded workload. We would not expect the `redis` thread to be preempted frequently since it is always doing critical tasks such as accepting and servicing connection requests from the benchmarking utility. If the `redis` application was multithreaded, it would be unclear if the interrupt overhead was primarily due to excessive preemption of threads with complex state or purely due to I/O devices and self-interrupts. This is supported by the data in Figure **??**, which shows that the threads used in `redis-benchmark` does not affect the overhead at all. Since the primary side effect of increasing the thread count is increased contention for CPU compute resources, it is safe to assume that the effects of preemption, if there are any, are minimal.

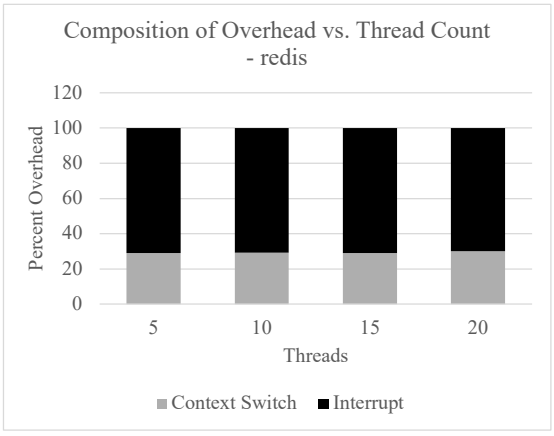


(a) Composition of overhead vs total requests

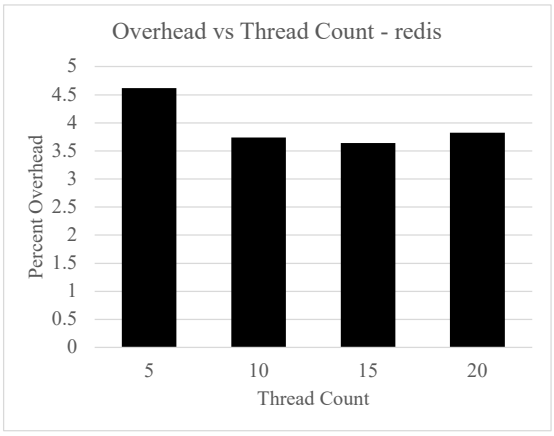


(b) Percent overhead vs total request count

Figure 5.4: Overheads of varying configurations of `httpd` on QEMU.



(a) Composition of overhead vs thread count



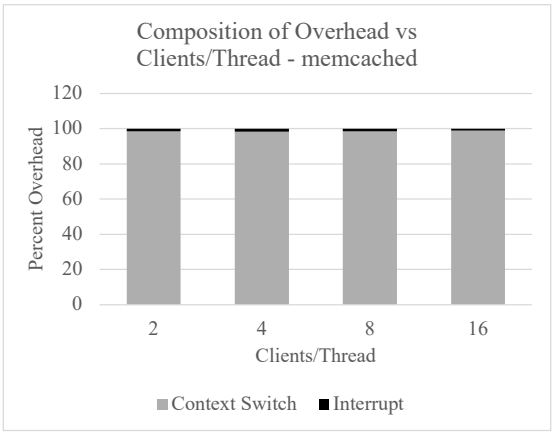
(b) Percent overhead vs thread count

Figure 5.5: Overheads of varying configurations of `redis` on QEMU.

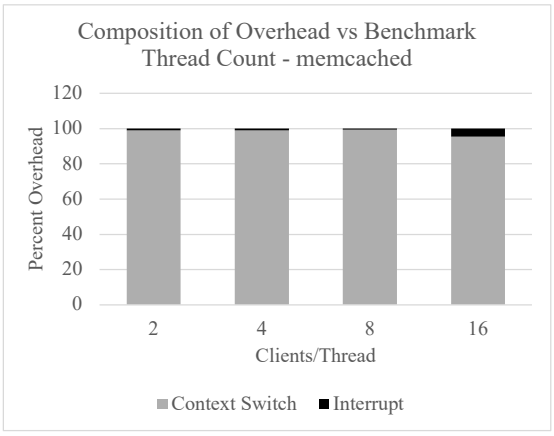
5.3 Analysis of Experiments on Real System

Our real system experiments were able to replicate our results from the controlled environment of QEMU. Experiments on the entropy servers introduced the potential issue of background noise from other process and opened more compute resources for threads, which substantially changed the nature of the overheads. No benchmark showed *less* overhead in real system than on QEMU, which partially validates our approach.

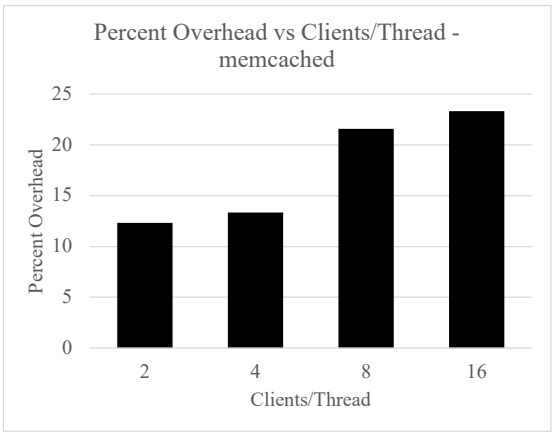
Experiments on `memcached` on the entropy servers demonstrated much higher overheads than in QEMU, but the overhead was still dominated by the cost of context switching. As the total work increased, the overhead percentage increased, as seen in Figures 5.6c and 5.6d. This is most likely associated with the increase in the number of available cores on the entropy servers, which inherently results in higher scheduling costs. This explains the increased overhead from context switching. This is supported by the fact that as thread count increased, the overhead increased,



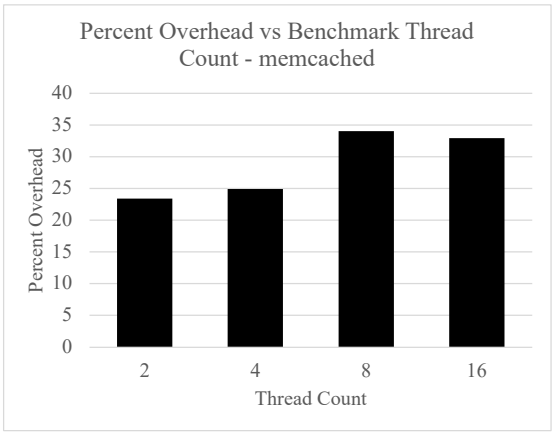
(a) Composition of overhead vs clients/thread



(b) Composition of overhead vs thread count



(c) Percent overhead vs clients per thread

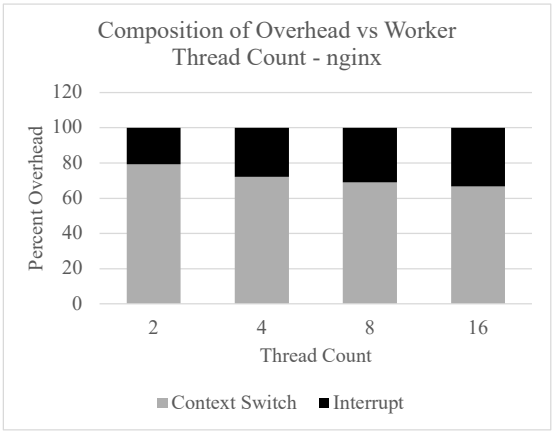


(d) Percent overhead vs threads

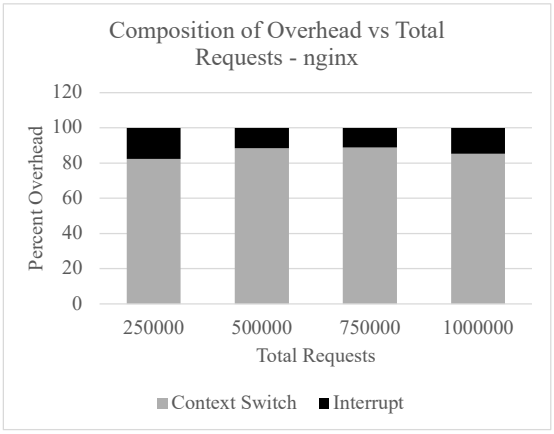
Figure 5.6: Overheads of varying configurations of `memcached` on a real system.

indicating that contention for computing resources is the most likely origin of this overhead. The increased core count also is able to account for the low incidence of signals; with more cores free, it is more likely than not that each thread can get a core to itself, and that the active threads only need to yield briefly to the kernel or other background jobs, making expensive thread migrations unnecessary. This would correspond to a lower incidence of preemption and therefore a lower incidence of signals.

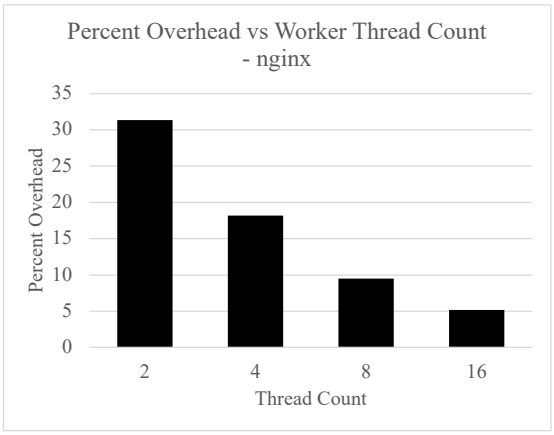
Experiments on `nginx` on the entropy servers demonstrated significantly higher overheads than on QEMU. Moreover, much more of the overhead was due to interrupts than on QEMU. There was no clear relation between the total number of requests and the overhead. Interestingly, as the thread count increased, the overhead due to scheduling and interrupts *decreased*. This is explained by the fact that we actually observed *slowdowns* from adding worker threads to `nginx`. Oddly, this was not observed in QEMU, where even less cores were available. This leads us to speculate that somehow the `nginx` load balancer failed to partition the work in a reasonable fashion, resulting in the kernel scheduler intervening via task preemption in an attempt to load-balance the cores. This would result in a higher proportion of signal overhead as



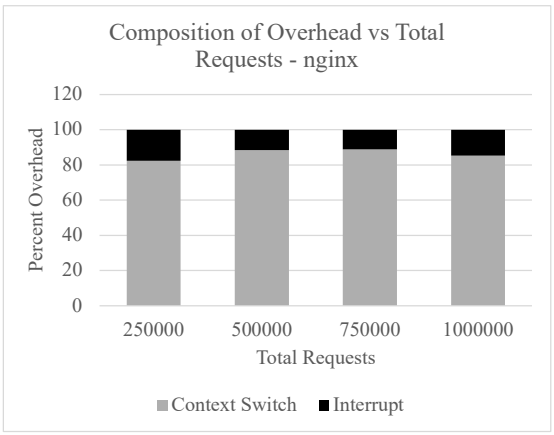
(a) Composition of overhead vs thread count



(b) Composition of overhead vs total requests



(c) Percent overhead vs thread count



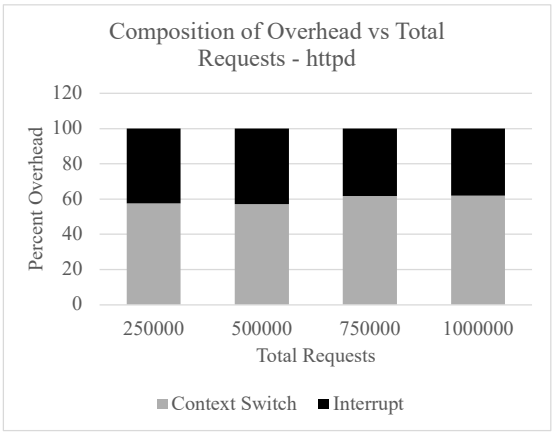
(d) Percent overhead vs total requests

Figure 5.7: Overheads of varying configurations of `nginx` on real system.

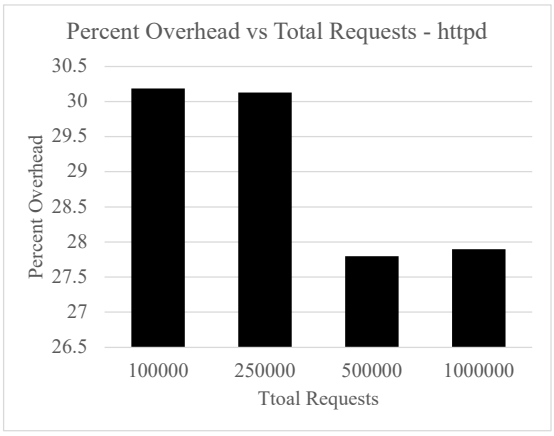
well as a degradation in performance with higher thread counts. Although the exact reasons for this phenomenon are unclear, this is clearly application-specific behavior that could be explored further using code tracing and execution analysis.

Experiments on `httpd` on the real system yielded similar results as on QEMU, but with higher overheads. The results are summarized in Figure 5.8. The overheads due to signals increased by about two-fold on the entropy servers. Moreover, the total requests sent seem to have no relation to the overhead observed. This is likely because all of the `httpd` worker threads were able to work simultaneously on the real machines due to the higher number of cores available, and `siege` was not able to generate enough requests at once to place the server under significant strain. This would also suggest that the increase in interrupt traffic is due to cross-thread communication on the real system, since in QEMU, at most one `httpd` worker thread was running at once due to the limitation on the number of cores available.

Experiments on `redis` on the real system had roughly double the overhead and significantly less interrupt overhead compared to the experiments on QEMU. This is most likely due to a combina-

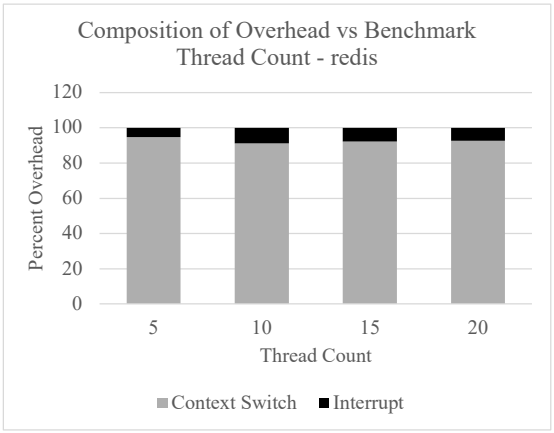


(a) Composition of overhead vs total requests

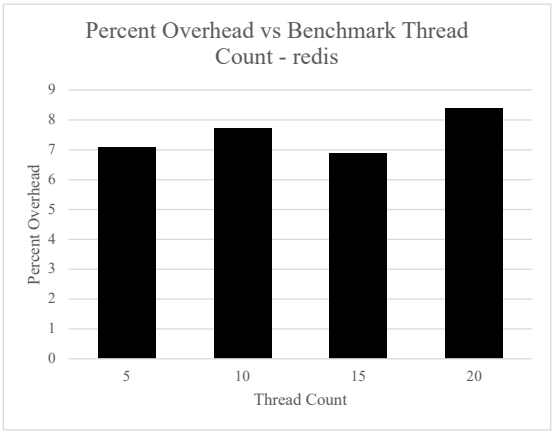


(b) Percent overhead vs total requests

Figure 5.8: Overheads of varying configurations of `httpd` on real system.



(a) Composition of overhead vs thread count



(b) Percent overhead vs benchmark thread count

Figure 5.9: Overheads of varying configurations of `redis` on real system.

tion of instrumentation noise and a higher efficiency in writing to the disk in real systems, since writing to disk in QEMU is much slower than writing to main memory on a real machine. This would explain why the interrupt overhead decreased almost 10-fold. The results are summarized in Figure 5.9.

A holistic view of the data indicates that scheduling and communication-associated overheads originate from a variety of sources, and that no one algorithm "fits the bill." This emphasizes the need for adaptive algorithms that respond dynamically not just to virtual runtimes and process timeslices, but also to architectural side-effects of process execution, such as cache interference and expected execution times of threads. We discuss potential approaches to this problem in our concluding remarks.

Chapter 6

Conclusion

Our data shows that overheads due to context switching and interrupts are significant in four common web applications. We also showed that the majority of these overheads are due to context switching and not due to scheduler-related interrupt handling. Moreover, the data show that these overheads and their causes are not predictable by simple heuristics, i.e. giving each thread or process a ”fair” slice of time is not sufficient to minimize these overheads. Therefore, specialized scheduling algorithms that take into account the architectural side effects of a task, as well as its expected runtime, are essential for achieving higher performance.

We propose that CPUs have on-board scheduler modules which prefetches instruction streams and estimates their runtime. Moreover, this scheduler module should be able to dynamically assign priorities to tasks either independently or following some kernel policy and grant resource guarantees to high-priority processes. This extension to modern CPU architectures would not be particularly expansive: instruction pre-fetchers already exist, and runtime estimation need not be more complicated than a series of opcode lookups in a latency table. The primary changes would be at the hardware thread level and ISA level, where hardware threads will require additional state registers to denote their status as either a high-priority or low-priority thread. This state must be preserved by context switching. One possible solution to this issue would be to reserve a rarely-used general-purpose register to mark process priority.

In order to decrease context switching latencies in such an architecture, it would be prudent for each core to have additional hardware threads that can hold idle high-priority threads. This can completely eliminate the need to context switch between a small number of threads with complex states, which can save a significant amount of overhead. Modern CPUs already provide multiple hardware threads on each core, typically for the purposes of simultaneous multithreading (SMT). With the multitiude of attacks exploiting insecurities in SMT, it may be time to do away with a security liability and use the extra hardware threads to find performance benefits elsewhere.

Although our data indicate that interrupt latencies are significant, they do not make up a meaningful proportion of the scheduling-related overheads we have instrumented. Therefore, while inter-core communication and handware interrupts are areas where significant optimizations can be realized, we do not consider them to be high-priority areas of study at the moment.

6.1 Future Work

6.1.1 Evaluation of the Nemo Core

We do not present results on our evaluation of the Nemo core because we deem it to still be out of acceptable performance ranges. In future work, we hope to bring the performance of the Nemo core within tolerance so that we can implement and evaluate a simulated hardware scheduler.

Over the course of the year, we have continuously benchmarked and refined the accuracy and functionality of the Nemo CPU simulator using the SPEC2017 benchmark suite. Benchmarks were compiled using `gcc`, `gfortran`, and `g++` using `-march=x86-64 -mtune=generic` flags. The configuration of the simulation exactly matches that of the entropy servers given in Table 4.1. Benchmarks were adapted for use in the simulator framework by the injection of start simulation and end simulation magic operations. Afterwards, benchmarks were run in simulation for 200 million instructions, and IPC data was recorded. As a baseline, real processor performance was measured by using the PAPI library to instrument program performance. The same SPEC2017 benchmarks (excluding the magic operations) used in simulation were compiled using the same `Makefile` configurations. The benchmarks were then run on the entropy servers. Each benchmark was pinned to a core, which was throttled to 2 GHz using the Linux userspace CPU frequency governor. Afterwards, the benchmarks were run for 200 million instructions and statistics collected. The benchmarks were terminated after the specified number of instructions were executed. The IPC was then computed and compared to the simulation IPC.

To our current understanding, there are a number of limitations of the Nemo core backend, which are intimately connected to the memory system and vectorized operations available in the x86 architecture. First, there is no support for an instruction cache There are a number of limitations of the Nemo core backend. Thus, all memory operations are directed to the data cache, which can cause inaccurate cache behavior as well as limiting the data cache memory bandwidth. Also, the register renaming system requires further verification, due to the fashion by which `NemoCommands` are generated and communicated to the backend. This affects the level of instruction-level parallelism, which also affects the simulation accuracy of the Nemo simulator. Finally, vectorized memory accesses, such as the `rep movs` instruction, are not properly optimized in the Nemo core, which leads to large performance degradation in simulation.

6.2 Limitations of Instrumentation and Results

Our instrumentation does not discriminate between processes of interest and background processes. This means that background noise or other processes running on the system could affect the measurement of the overhead associated with a given application. We argue that this is not a substantial downside, as real servers have hundreds of jobs running simultaneously which will all affect any given application one wishes to instrument.

Bibliography

[1] Cfs scheduler. https://docs.kernel.org/scheduler/sched-design-CFS.html. 2.2

[2] O3 cpu. https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU. 3.2

[3] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/fried. 1

[4] Jeffrey Fulmer. Siege. https://github.com/JoeDog/siege. 4.3

[5] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. Ghost: Fast & flexible user-space delegation of linux scheduling. SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450387095. doi: 10.1145/3477132.3483542. URL https://doi.org/10.1145/3477132.3483542. 1

[6] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/kaffes. 1

[7] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. 33(4):92–99, nov 2005. ISSN 0163-5964. doi: 10.1145/1105734.1105747. URL https://doi.org/10.1145/1105734.1105747. 3.1

[8] NTESS. sst-elements. https://github.com/sstsimulator/sst-elements. 3.2

[9] Vairavan and Demillo. On the computational complexity of a generalized scheduling problem. *IEEE Transactions on Computers*, C-25(11):1067–1073, 1976. doi: 10.1109/TC.1976.1674556. 2.2