Experimental Calibration and Validation of a Speed Scaling Simulator

Arsham Skrenes Carey Williamson
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
{arsham.skrenes, carey}@ucalgary.ca

***Abstract*—In this paper, we use experimental measurements to calibrate and validate a discrete-event simulator for dynamic speed scaling systems. The experimental implementation work is carried out in an Ubuntu Linux environment using a quad-core 2.3 GHz Intel i7 processor with the Ivy Bridge micro-architecture. Our implementation provides fine-grain user-level control of process execution, and uses the Running-Average Power Limit (RAPL) Machine Specific Registers (MSRs) to track energy usage. Through careful micro-benchmarking experiments, we determine the power consumption for each of the 12 discrete speeds supported by the processor, while also quantifying the costs of context switches and CPU speed changes. Finally, we use our suitably-parameterized speed scaling simulator to evaluate three different CPU speed scaling algorithms from the literature on simple batch workloads. To the best of our knowledge, our paper provides the first direct comparison of these speed scaling strategies with realistic system costs.**

***Index Terms*—Speed scaling; energy consumption; experimental implementation; measurement; simulation validation**

I. Introduction

In dynamic CPU speed scaling systems, the speed at which the processor executes is adjusted over time based on the ambient workload demand. If no jobs are present, the processor can enter a rest state (e.g., "sleep", "idle", or "gated off") to reduce power consumption. In the presence of one job, the processor can run at a modest baseline speed. If the number of active jobs increases, the processor speed can be increased, perhaps to some maximum rate, to dissipate the backlog quickly.

Speed scaling strategies produce interesting tradeoffs between response time, fairness, and energy consumption. In essence, the design goal is to run the system just fast enough to complete all of the work in a timely fashion, but no faster. By doing so, the energy consumption for completing the workload is as low as possible. Such energy-efficient speed scaling strategies are relevant in at least two different contexts, namely hand-held mobile systems where battery power is a scarce resource, and commercial datacenters where the energy consumption of thousands of processors can affect operating costs [15], as well as the carbon footprint.

Modern processors provide rich functionality to support dynamic speed scaling. For example, the Intel i7 processor used in our work has 12 distinct execution speeds, 4 different sleep modes, 6 idle modes, and an API for selecting amongst 5 different speed scaling governors. Furthermore, the Intel Sandy Bridge and newer microarchitectures have specific hardware support for measuring energy consumption via the Running Average Power Limit (RAPL) Machine Specific Registers (MSR) [11]. However, most operating systems, including Linux, still use relatively simple algorithms for speed scaling (e.g., threshold approaches based on CPU utilization, with only a few distinct coarse-grain levels).

In the published literature, there is a dichotomy between the speed scaling results for the systems and theory research communities. The theoretical work tends to provide elegant results on the optimality and efficiency of speed scaling algorithms [1], [2], [4], [5], [6], [8], [25], albeit under many assumptions (e.g., weighted cost functions for delay and energy consumption; known job sizes; energy-proportional operation; job-count-based speed scaling; continuous and unbounded speeds; zero cost for context switches, speed changes, or return from sleep states). Simulation is sometimes used to augment the evaluation of speed scaling systems, but the simulators often have similar assumptions as the analytical work. In the systems community, research tends to focus on Dynamic Voltage and Frequency Scaling (DVFS). In this context, practical issues such as processor utilization, heat dissipation, and job size variability are primary considerations [14], [16], [17], [21], [24], while optimality is not.

Our work attempts to bridge between theory and systems, by evaluating several speed scaling strategies under realistic settings, using a calibrated and validated speed scaling simulator. Our objective is to provide an accurate comparison and evaluation of speed scaling algorithms, when deployed on modern processors. We do so by building an experimental framework to collect fine-grain power measurements, which are used to calibrate and validate a discrete-event simulator for dynamic speed scaling systems. The simulator is then used to study the response time and energy consumption of three speed scaling policies from the literature.

There are three main contributions in this paper. First, we describe a proof-of-concept implementation that supports precise user-level control of speed scaling using the features of the Intel i7 processor. Second, we use micro-benchmarking to measure the actual system costs for context switches, speed changes, and different operating modes on the i7 processor. Third, we provide both experimental and simulation evaluation of three different speed scaling strategies from the literature. To the best of our knowledge, our results provide the first

"apples to apples" comparison of these speed scaling strategies under realistic system settings.

The remainder of this paper is organized as follows. Section II provides background context for our work and discusses relevant prior work from the literature. Section III provides an overview of our experimental framework for evaluating speed scaling strategies, while Section IV presents implementation details. Section V describes our experimental measurement methodology and results, while Section VI presents our simulation results. Finally, Section VII concludes the paper.

II. Related Work

In speed scaling systems, there are many tradeoffs between service rate, response time, and energy consumption. These issues have fostered research efforts in both the systems community and the theory community.

One of the earliest systems papers on speed scaling was by Weiser *et al.* [24]. In their work, they considered a diverse mix of processes in a Unix system, and attempted to determine the energy savings if the jobs were executed using different system speeds. A subset of the same authors later contributed to one of the seminal theoretical papers on speed scaling [26]. Their paper proposed an optimal offline algorithm for speed scaling (now known as YDS, based on the names of the authors), with the objective of minimizing power consumption. They also proposed a heuristic online algorithm for the same problem. Both algorithms are deadline-based, and require knowledge of job sizes and deadlines.

In the systems community, CPU speed scaling is often referred to as Dynamic Voltage and Frequency Scaling (DVFS), in which either the voltage or the clock frequency (or both) of the processor are adjusted. Power consumption is usually a quadratic (or cubic) function of the frequency used. In systems work, many practical concerns arise regarding granularity of control, the set of discrete speeds available, non-linear power consumption effects, and unknown job characteristics [19], [20], [21], [23], [24]. Practical speed scaling policies include threshold-based control, Rush-to-Idle, gated on-off, and Intel's Turbo Boost technology [12].

In the theory community, speed scaling typically assumes a continuous and unbounded range of available speeds, with the choice of speed determined either by job deadlines [26] or system occupancy [3], [6]. Albers *et al.* have done extensive work on energy-efficient algorithms [1], [2]. Some of this work optimizes the tradeoff between energy consumption and mean response time [1]. Several studies on this metric suggest that energy-proportional speed scaling is near optimal [3], [6]. An alternative approach has focused on minimizing the response time in systems, given a fixed energy budget [4], [5].

Andrew, Lin, and Wierman [3] formally consider the tradeoffs between response time, fairness, and energy consumption. Their paper identifies algorithms that can optimize up to two of these metrics, but not all three. For example, SRPT (Shortest Remaining Processing Time) is optimal for response time [18], but can be unfair, while PS (Processor Sharing) is always fair, but suboptimal for both response time and energy [3], [8].

Decoupled speed scaling divorces speed selection from system occupancy [8]. This violates the definition of "natural" speed scaling in [3], since it can require speed changes at arbitrary points in job execution, even if occupancy remains the same. While decoupled speed scaling provides an elegant theoretical model, it has not been implemented or evaluated in a practical system. Our paper provides the first experimental results for decoupled speed scaling in such a system.

III. System Overview

This section describes the design and implementation of the `Profilo` software tool that we developed. It provides user-level control over process execution for micro-benchmarking and collecting energy profile measurements.

A. Workload Specification

The workload input for the `Profilo` environment is specified using an external file. A user can specify a list of jobs to be executed on the CPU using a workload file like this:

Job	Work	Speed
P1	3	1
P2	7	5
P3	1	2
P2	8	3
P4	10	1

In this example, there are 4 different jobs. The first column identifies the job (based on a process ID number), while the second column identifies how much work the job needs to perform, and the third column indicates the (arbitrary) speed at which the CPU should run when doing that work. The file is implicitly in timestamp order, and user-defined processes are numbered consecutively from P1 to Pn, for some n. In this particular example, job P1 runs first, and has 3 units of work to complete at speed 1. The next job to begin execution is job P2, which runs at speed 5 to complete 7 units of work. It then yields the CPU to job P3, which does 1 unit of work at speed 2. Then job P2 regains the CPU, and runs at speed 3 to finish 8 more units of work. Finally, job P4 does 10 units of work at speed 1.

There are four subtle but important points in this example. First, the order of job execution is completely specified in the file, as are the job sizes and system speeds. There are no scheduling decisions or speed scaling decisions for `Profilo` to make; it simply needs to read in the file and execute the jobs accordingly. Second, it is the *user* that constructs the file and provides it to `Profilo`. As such, the user can create files with arbitrary scheduling policies (e.g., FIFO, PS, SRPT, FSP [10]), and arbitrary speed scaling strategies (e.g., static, coupled, decoupled, random). In practice, these workload files can be generated either by hand or using simulation tools. Third, the system speeds are *relative*, not absolute. Specifically, we treat the speed as an index that is normalized and mapped to one of the actual speeds available on a given processor. To understand this point, consider a trace file that contains only a single job P1. It is ambiguous whether this job should be run at the

lowest available speed on the system (to minimize energy consumption), or the highest available speed (to minimize response time). Both interpretations are possible, and equally valid. By inserting a dummy job P0 into the trace file twice, once with the minimum speed and once with the maximum speed, we can disambiguate this scenario, and force P1 into any desired choice of the available system speeds. If P0 runs for only the minimum possible time slice in our system (e.g., 1 ms), its effect on the overall execution cost is negligible. Finally, it is important to note that "work" is expressed in normalized "work units", and not in time, since the execution of 10 units of work could take different amounts of time, depending on the speed used. (This is also the underlying reason why there are no job arrival timestamps in the workload file.) One of the parameters provided to `Profilo` specifies the normalized work unit for any given run of the system.

In our system, we use a simple primality-testing algorithm as the underlying compute task for work units. There are several reasons for this choice. First, primality-testing is a CPU-bound computation that is fully contained within the processor package (e.g., core, cache). This feature means that the Running Average Power Limit (RAPL) counters can be used to accurately profile its energy consumption [12]. Second, primality-testing is easily implemented in kernel space without the need for complicated mathematical operations and/or floating point units. Third, it suitably utilizes the superscalar and pipelined integer architecture, while reasonably disrupting branch prediction [22]. Finally, primality-testing is easily parameterized to generate jobs that range in duration from microseconds to minutes.

This workload configuration framework is a key part of our experimental platform for evaluating speed scaling strategies. While this framework is simple, it is also very powerful and flexible. The most difficult part is making the operating system do exactly what `Profilo` wants. We discuss this issue next.

B. Software Prototype

The initial version of `Profilo` was written as a multi-process user-space application to measure hardware context switches. The simplicity of the initial version allowed the application to be created quickly, and the different schedulers and speed scaling algorithms to be tested to see if there were statistically significant differences in timing and energy usage.

One of the challenges with a fully-user space application, however, is that higher-priority processes and/or system interrupts can preempt the user-defined processes at any time. Furthermore, user processes can occupy the CPU for at most a maximum duration, commonly called a *time slice*, quantum, or jiffy. On most platforms, this duration is 10 ms. This short time limit is problematic since some scheduling policies (e.g., FCFS) may require uninterrupted execution for the entire lifetime of a process.

To address these issues, kernel code is needed. An open-source Linux operating system (Ubuntu 14.04 LTS) was chosen because it has the necessary hardware support, including an x86 Machine Specific Register (MSR) module.

Kernel space in Linux can be accessed either by modifying the kernel or through the use of a kernel module. Modifying the kernel requires gaining control to perform the profiling operations on synthetic workloads. The modified kernel would still need to interact with the user, run the specified workload, and log the timing and energy consumption information. This would be complex, and would require a rebuild and reboot of the system every time the kernel is modified. Portability could also become an issue, if the kernel was incompatible with other Linux community patches.

In most cases, the kernel module approach is preferred, since it is portable across different distributions and kernel versions. Modules are also loadable and unloadable, without the need to recompile the entire kernel or reboot the system, which makes development significantly easier. Kernel code can be exposed to a user-space API by using the `sysfs` virtual file system. Additional "interrupt disable" code is also required to avoid unwanted context switches, since contemporary Linux kernels provide multicore, pre-emptive scheduling.

The final version of `Profilo` was implemented as a hybrid of the above. It uses a kernel module to perform uninterrupted work, busy waiting, and sleeping, while performing high-resolution timing and energy profiling. It exposes these features through `sysfs` files, allowing a user-space application to read the workload file, process jobs, collect data, and generate output. Retaining as much as possible in user space makes programming and debugging substantially easier.

IV. Implementation Details

This section provides details on the implementation of `Profilo`, as well as the specific hardware platform used.

A. Hardware Platform

Our experiments were conducted on a mid-2012 Apple MacBook Pro Retina running Ubuntu Linux. The laptop was equipped with a 2.3 GHz quad-core Intel Core i7-3615QM Ivy Bridge processor with 32 KB of L1 instruction cache per core, 32 KB of L1 data cache per core, 256 KB of L2 cache per core, 6 MB of on-chip L3 shared cache, and 8 GB of 1600 MHz DDR3 RAM.

This processor has 12 discrete frequencies that range from 1200 MHz to 2300 MHz, in increments of 100 MHz. In addition to these 12 speeds, there is a special speed setting of 2301 MHz configurable on the processor. This speed enables Intel's patented "Turbo Boost" technology. This feature allows the processor to clock one or more of its cores above its top-rated frequency, if the power, current, and thermal limit are within a specific range. For this particular processor, the maximum Turbo Boost frequency is 3.3 GHz for a single active core, 3.2 GHz for two active cores, and 3.1 GHz for three or four active cores. While all of our micro-benchmarking results include the "2301 MHz" Turbo Boost mode, that mode is avoided in our speed scaling experiments because it is not directly controllable, even in Ring 0 of the hierarchical protection domains.

On the Intel i7, there are a total of four Running Average Power Limit (RAPL) domains. However, there are only three available on any given CPU [12]. Enterprise-class (server) machines have PKG, PP0, and DRAM domains, while consumer-class (client) machines have PKG, PP0, and PP1. The Intel Core i7-3615QM CPU falls under the latter category. For compatibility with both product categories, `Profilo` only makes use of two domains, namely PP0 (processor cores) and PKG (entire CPU). While this captures most of the dynamics in CPU-bound activities, the hardware platform has more than just the CPU. For example, the package has an integrated GPU (which we disable in order to get accurate power measurements).

B. Kernel Space Implementation Details

`Profilo` is composed of two parts: the kernel module, and the user-space application. This subsection focuses on the kernel-space features.

The kernel module makes use of the `sysfs` virtual file system provided by the Linux kernel [7]. This virtual file system is intended to expose kernel subsystems, device drivers, and two-way communication between kernel functionality and user space. Despite the limited documentation for `sysfs`, this approach is superior to the use of `procfs`, which is intended only for process-related system information [13].

The `Profilo` kernel module uses four `sysfs` files:

- `work_unit`: this file defines a normalized unit of work for a given run of `Profilo`. In our current implementation, the file indicates the number of consecutive primes to find (starting from 2), using the primality testing algorithm. This value determines the minimum granularity for time slices in our experiments. Writing an integer value to this file sets the number of primes, while reading it returns the current value.
- `do_work`: this file configures the actual workload. To use it, simply write the integer number of loops (replications) of `work_unit` to the file. Reading from the file only displays the kernel module name and version.
- `sleep_busy`: writing an integer value (in microseconds) to this file busy-loops the processor without context-switches until that duration has elapsed. Subsequently reading the file reveals the formerly written value as well as the actual time (in microseconds) that the processor busy-waited (including the time it took for the RAPL MSR's to roll over).
- `sleep_deep`: writing an integer value (in microseconds) to this file sleeps the processor for that duration. Similar to sleep_busy, subsequently reading the file reveals the formerly written value as well as the actual time that the processor slept, including any busy-wait period waiting for the RAPL counters to roll over.

Both `sleep_busy` and `sleep_deep` make use of the high-resolution timers supported in 64-bit Linux distributions. Specifically, they use `ktime_t`, which is a 64-bit integer expressed in nanoseconds. The `sysfs` files output their respective time values (in microseconds), with three decimal places to provide nanosecond precision.

There is one more requirement in order to support uninterrupted process execution. Specifically, the hard lockup detector, which is implemented with a non-maskable interrupt (NMI), needs to be disabled. The Linux kernel has both soft and hard lockup detection. Both base their timeouts on a `sysfs` configurable value in `/proc/sys/kernel/watchdog_thresh`, which is typically 10 seconds.

C. User Space Application

The user-space part of `Profilo` is a command-line utility that begins by reading, translating, and sanity-checking the input arguments and trace file provided to it. It then modifies the operating system environment, and uses a compact version of the trace file to control job execution on the CPU via the `sysfs` interface. When execution is complete, it restores the former operating system settings, and then reports the results.

`Profilo` always verifies that the input trace file is properly formatted, and semantically correct, before running any experiment. The trace file is a CSV file with a header row and three columns: Process Name, Work, and Speed. The name uniquely identifies a process, which may appear more than once in the job schedule. Work is a positive integer specifying how many contiguous "work units" the process is to perform during the run. Speed is a positive integer that is mapped to a specific processor frequency before performing the work.

`Profilo` scans the entire file to determine the number of processes, their sizes, and the set of speeds to be used. This information is stored in two internal data structures, with one for processes, and one for speeds. The process structure records the name and index of each process in the trace, as well as the remaining work for each (to determine when the process leaves the system), and two variables that store the start time and end time for each process. The time values are post-processed to compute response times.

The speeds s_i from the trace file are mapped to a CPU frequency f_i by the following formula:

$$f_i = f_{min} + (f_{max} - f_{min}) \frac{(s_i - s_{min})}{(s_{max} - s_{min})}$$

where f_{min} is the processor's slowest frequency, f_{max} is the fastest frequency, and s_{min} and s_{max} are the smallest and largest speed values (respectively) in the entire trace file. This equation normalizes the speeds in the trace file, as well as those of the processor, and provides a mapping between them. If the calculated frequency falls between two available discrete frequencies, it is rounded up.

The MSR module is a kernel module that provides an interface to x86 processors through the virtual file `/dev/cpu/0/msr`. Reading and writing to the file requires elevated (root) privileges, and is done in 8-byte (64-bit) chunks. `Profilo` uses the low-level `pread` function with the MSR file.

Profilo sets its system priority to be as high as possible. Linux, like most operating systems, supports process priorities. These priorities, called *nice* values, range from -20 to +19. From a scheduling perspective, a nice value of -20 is the highest priority, while +19 is the lowest priority. The default priority for processes is a nice value of zero. In most Linux distributions, setting a high-priority nice value requires root privileges. Profilo invokes the setpriority system call to set its nice value to -20.

Next, Profilo changes the current governor so that it can control the processor's frequency. The Linux kernel implements dynamic speed scaling through a mechanism called cpufreq. It provides a generic governor interface that allows software-defined speed scaling policies to be implemented when the processor is busy. The Linux kernel uses a separate cpuidle subsystem when no work remains for the processor.

Most x86-based Linux distributions have five governors from which to choose. These rely on sysfs files located in /sys/devices/system/cpu/cpun/cpufreq/ (for the nth logical processor). The existing governors in Ubuntu Linux are: powersave (run at the lowest available frequency); performance (run at the highest available frequency); ondemand (dynamically change the processor's frequency based on system load, with aggressive increases and step-wise decreases); conservative (similar to ondemand, but with step-wise increases); and userspace.

The userspace governor is the one that we use. It changes the frequency of the processor based on input from scaling_setspeed. This gives full control to user-space processes (with root privileges) to set the frequency. We use this mechanism in our user-defined speed scaling experiments.

V. Experimental Methodology and Results

This section describes the methodology used to run Profilo, and how it collects and reports measurements.

A. Running Profilo

There are several steps before energy profiling takes place. First, the sysfs file time_unit for the Profilo kernel module needs to be set, based on the value that was provided as a command-line argument to Profilo. Once this is done, the do_work file can be written as well. The final thing to do is to read the energy and timing counters. The startRAPLmetric function is invoked. This reads MSR_PP0_ENERGY_STATUS and MSR_PKG_ENERGY_STATUS into a temporary variable, and then continuously rereads the MSRs until both values roll over. This is an unpredictable amount of time that can take up to a millisecond [11], [23]. When it returns, the clock_gettime function is invoked, with nanosecond resolution. The result is stored in the first of two locally scoped time variables. The second variable stores the end time for the profiling.

The profiling stage of Profilo involves a for-loop that systematically traverses the list of tasks from the trace file. The number of loop iterations is equal to the number of lines in the trace file (minus the header). Each run begins by checking the isRunning array to see if the current process is already running. If the process is not running, then the startTime structure for the current process is initialized with the clock_gettime function, before activating the process.

To ensure repeatability of results, we disable all unnecessary functionality when running Profilo. In particular, we disable the WiFi network interface, the monitor backlight, and the GPU, while Profilo itself disables the three extra cores. All of our results are from 10 replications of each experiment, with the minimum result reported. With these approaches, all of our measurement results are repeatable within 1% (except in Turbo Boost mode).

B. Microbenchmarking Results

Table I and Figure 1 provide a summary of our power measurement results for the 12 different speeds on the i7 processor, as well as the Turbo Boost mode and several of the sleep/idle modes. In general, there is a strong linear relationship between power consumption and processor frequency, though there is some evidence of non-linearities beyond 2200 MHz. In particular, the power consumption in the Turbo Boost mode ("2301 MHz", not shown on the graph) is extremely high, approximately double that for 2300 MHz. The extended C1 mode (C1E) for the processor has power consumption similar to that when the main core (CPU0) is active (C0) at 1700 MHz, while the other cores are idle (C7). There are also several low-power idle states that are frequency-independent. For example, C7/C7 consumes only 10% of the PP0 power used at 1200 MHz for a single active core.

TABLE I
Microbenchmarking Results for Selected Intel i7 States

Core 0	Core 1-3	Frequency	PP0 (W)	PKG (W)
C0	C7	2301 MHz	11.32	15.28
C0	C7	2300 MHz	5.26	9.14
C0	C7	2200 MHz	4.85	8.70
C0	C7	2100 MHz	4.66	8.50
C0	C7	2000 MHz	4.47	8.31
C0	C7	1900 MHz	4.29	8.11
C0	C7	1800 MHz	4.09	7.90
C0	C7	1700 MHz	3.91	7.72
C0	C7	1600 MHz	3.73	7.52
C0	C7	1500 MHz	3.57	7.35
C0	C7	1400 MHz	3.39	7.17
C0	C7	1300 MHz	3.20	6.98
C0	C7	1200 MHz	3.02	6.78
C1E	C1E	–	3.90	7.63
C3	C3	–	1.83	4.67
C6	C6	–	0.47	3.29
C7	C7	–	0.31	3.07

In addition to the 12 speed settings mentioned earlier, the Intel i7 processor also has multiple sleep and idle states. Table II provides a summary of the sleep states, as well as the results from our measurements of their power consumption.

The sleep states were measured using our Profilo kernel API, and validated using an external power consumption meter. Specifically, we configured a long duration in each of the sleep modes, and used the MSR values to calculate

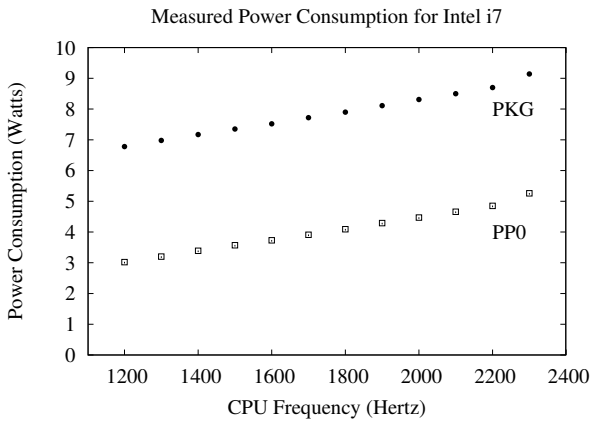


Fig. 1. Experimental Measurements for Power Consumption (Intel i7)

TABLE II
MICROBENCHMARKING RESULTS FOR SLEEP STATES ON INTEL I7

Mode	State	Description	Power
On	G0/S0	Normal operation	Varies
POS	G1/S1	Power on Suspend	0.8 W
Suspend	G1/S3	Suspend to RAM	0.7 W
Hibernation	G1/S4	Suspend to Disk	0.3 W
Soft Off	G2/S5	Soft off, no WiFi	0.3 W
Soft Off	G2/S5	Soft off, WiFi on	0.6 W
Off	G3	Mechanical off	0.0 W

power consumption. These values were corroborated against the (video recorded) readings from a Kill-A-Watt meter. The observed values from each were in very close agreement.

Last but not least, we have used our software framework to assess the costs of context switches and speed changes on the Intel i7 architecture. For example, we measured context switching costs by devising a simple PS schedule that alternated execution between two processes P1 and P2, with no speed changes on the processor. By varying the time quantum that PS was using, we were able to estimate the context-switching overhead. As another example, we measured the overhead of speed changes by running a job schedule with a single process P1, which alternates between two different speeds in each successive interval of execution.

Our measurement results show that mode switches, context switches, and speed switches all have different costs. Mode switches (user-to-kernel or kernel-to-user) are the least expensive, taking 64 to 123 nanoseconds, depending on the processor frequency, and only 45 nanoseconds in Turbo Boost mode. These have negligible energy costs, which are below 1 microJoule. Context switches between threads or between processes are more expensive. These take 1.6 to 3.2 microseconds, depending on processor frequency, and only 1.1 microseconds in Turbo mode. The energy cost is about an order of magnitude higher than for a mode switch. Changing the processor frequency has a time cost that is about 1-2 times that of a context switch, though it depends somewhat on the target frequency chosen. Furthermore, the energy cost of a speed switch is about four times that of a context switch.

VI. SIMULATION METHODOLOGY AND RESULTS

With the foregoing measurement results to calibrate our simulator, the next step is to compare and evaluate the three chosen speed scaling algorithms. For this purpose, we use a discrete-event simulation model of this system [9], with configurable scheduling policies and speed scaling strategies, and adequate instrumentation to record system occupancy, execution speeds, process schedules, and job response times.

A. Speed Scaling Strategies

For our simulation experiments, we use three different speed scaling strategies from the literature: PS, decoupled speed scaling, and YDS. These approaches are all fundamentally different, illustrating the generality and flexibility of our experimental framework.

We use Processor Sharing (PS) as the baseline for comparison of speed scaling strategies. By definition, PS is fair, since it provides an equal share of processing capacity to each job in the system. However, it is neither efficient nor optimal. That is, there are known approaches that outperform PS with respect to response time, energy consumption, or even both [8].

Our PS approach uses job-count-based speed scaling, in which the CPU speed is adjusted dynamically based on the instantaneous number of jobs in the system. On the i7, the maximum speed (2300 MHz) is only about twice that of the lowest speed (1200 MHz), and we map each occupancy to one of the 12 different speeds available. In theoretical models of speed scaling, the processor speeds are continuous and unbounded. In practical systems, there is a finite maximum speed available, and the number of discrete levels is finite, so a quantization step is needed to map from continuous to discrete speeds. For most purposes, mapping to the closest available speed suffices, though in deadline-based scheduling, rounding up to the next higher available speed is required.

The second speed scaling strategy that we evaluate is decoupled speed scaling [8], specifically FSP-PS. This algorithm runs the FSP scheduler, but uses the same CPU speeds as PS would be using at any point in time. With this approach, the energy consumption should be exactly the same as PS, though the response time would be much lower. Furthermore, fairness is preserved since FSP strictly dominates PS.

The third speed scaling strategy that we consider is YDS [26]. This algorithm is known to complete all of its jobs on time with the minimum possible energy consumption. To do so, a deadline needs to be specified for each job. For our purposes, we first use the PS execution schedule to determine the completion time of each job, and then provide those completion times to YDS as the deadlines for each job. With this approach, we again achieve the strict dominance property: no job completes later under YDS than it does under PS. This allows us to compare response time and energy consumption, without compromising fairness.

B. Workloads

To simplify the presentation and comparison of results, we use batch workload examples from prior work [9]. The model assumes a single server system, initially empty, to which a batch of 12 jobs arrive. This job count was chosen so that all 12 system speeds could be exercised.

We use three different batch workloads to examine the behavior of different speed scaling strategies. Workload 1 is a batch of 12 homogeneous jobs. Each of these jobs would need about 1-2 seconds of execution time on our system, depending on the speed used. Workload 2 is a batch of 12 jobs whose sizes differ additively in a simple arithmetic progression. The largest job represents about 6-12 seconds of execution time, depending on the speed used. Workload 3 is a batch of 12 jobs whose sizes differ by successive factors of 2. The largest job needs 50-100 seconds of execution time, depending on the speed used. These tests cover homogeneous jobs, as well as heterogeneous jobs with medium and high variability. In Workload 3, for example, the final job contributes about half of the total system work.

Under PS, it is known that jobs leave the system in the same order as under SRPT [10]. By definition, FSP schedules jobs based on their order of departure under PS. Therefore, in batch scheduling, FSP is equivalent to SJF (Shortest Job First) scheduling. Though it is algorithmically different, YDS also behaves similarly on these simple workloads, with respect to the order of job completion.

C. Simulation Calibration and Validation

Table III summarizes the `Profilo` experimental results for mean response time and energy consumption under the three speed scaling policies. The tables represent the homogeneous (Workload 1), medium variance (Workload 2), and high variance (Workload 3) job size distributions, respectively. For each workload, the four columns show the total elapsed wall-clock time for one run of the experiment, the mean response time ($E[T]$) for the 12 jobs, and mean energy consumption (in Joules) for PP0 and PKG, for each policy evaluated.

There are two observations that are immediately evident from the results in Table III. First, the total execution time for a given workload is similar, regardless of the speed scaling strategy used. This makes sense intuitively, since the volume of work is the same. However, there is a small execution time advantage evident for YDS, especially on the heavier workloads (Workload 2 and Workload 3). The latter is explainable by the average speed that YDS computes for each critical interval. Recall that this speed might need to be rounded up to the next discrete speed available on the processor, in order to guarantee that job deadlines are met. As a result, YDS tends to run its job schedule slightly faster than required by PS. Second, size-based scheduling provides dramatic improvements in mean response time compared to PS. On Workload 1, for example, the mean response times for FSP-PS and YDS are both about half that for PS. Again, this makes sense intuitively for this particular workload, since the jobs are all the same size. By sharing the CPU equally across the jobs, PS keeps all the jobs in the system until the very end[1], which hurts response time. By contrast, FSP-PS and YDS both provide exclusive service to one job at a time, resulting in more timely departures. This is a huge advantage on Workload 1, but less of an advantage in the other workloads when the final job accounts for most of the total execution time.

Several observations regarding energy consumption are also possible from Table III. First, the decoupled speed scaling approach FSP-PS should (by definition) have exactly the same energy consumption (PP0 and PKG) as PS. However, it turns out to perform slightly *better* than that, as is particulary evident in Workload 2 and Workload 3. The fundamental reason for this is fewer context switches than PS. Recall that PS will have a context switch between processes every time slice (10 ms) in its workload schedule, while FSP-PS executes each selected job to completion. This subtle advantage of FSP-PS over PS in terms of operating system overhead becomes noticable in our fine-grain measurement environment. Second, YDS has an energy advantage over both PS and FSP-PS. The primary reason is that YDS uses a "blended" average speed to process all jobs in each critical interval. As such, it avoids using the higher CPU speeds that PS and FSP-PS use, for which the energy cost is (at least) quadratic with the processor frequency. Supplementary reasons include fewer context switches, fewer speed changes, and slightly shorter execution times, as mentioned above, all of which are advantageous to YDS. These subtle effects emerge in our experimental framework.

D. Simulation Results

In this subsection, we present simulation results using our calibrated speed scaling simulator. We start with a graphical overview of simulator dynamics, as a validation against the experimental measurements. Then we explore other scheduling and speed scaling alternatives.

Figure 2 presents simulation results for Workload 1 (homogeneous jobs). The top row of graphs is for the PS scheduling policy, while the second row is for FSP-PS (decoupled speed scaling). (For space reasons, the graphical results for YDS are excluded, but they are summarized in Table IV.) In each row, the leftmost graph illustrates the instantaneous system occupancy (number of active jobs), while the middle graph shows the amount of work remaining in the system (i.e., the sum of the remaining sizes of all active jobs), and the rightmost graph shows the CPU speed being used at any time.

For PS on Workload 1, Figure 2 shows that the processor runs at the maximum allowed speed, until all the jobs finish and depart at the same time. The mean job response time is 14.4 seconds, which closely matches the experimental result in Table III. For FSP-PS, the processor runs at the same speed as PS, but service is devoted to one job at a time, so the job

[1]In theory, this is true, but in a practical implementation, the PS jobs actually depart one-by-one in a staggered fashion, separated by at most one full time slice between jobs. In a speed scaling system, this departure process actually gets slightly elongated, since the processor speed decreases with each successive departure. However, this effect on overall performance is minor.

TABLE III
EXPERIMENTAL RESULTS FOR MEAN RESPONSE TIME $E[T]$ AND ENERGY CONSUMPTION (PP0 AND PKG) (12 JOBS, $\alpha = 1$)

Speed Scaling Policy	Workload 1				Workload 2				Workload 3			
	Time (s)	$E[T]$ (s)	PP0 (J)	PKG (J)	Time (s)	$E[T]$ (s)	PP0 (J)	PKG (J)	Time (s)	$E[T]$ (s)	PP0 (J)	PKG (J)
PS	14.57	14.49	76.80	131.50	46.23	30.10	199.99	372.98	166.15	38.05	562.47	1184.36
FSP-PS	14.57	7.89	76.77	131.60	46.21	16.33	199.41	372.36	166.08	25.43	560.35	1180.83
YDS	14.55	7.88	76.49	130.93	45.80	17.81	198.83	369.88	163.12	27.15	560.94	1170.05

departure points and system occupancy are different. The mean response time is 7.8 seconds. YDS behaves exactly the same as FSP-PS on this workload.

Figure 3 presents the corresponding simulation results for Workload 2 (additive jobs). For this workload, the job departure points under PS are all distinct, since the job sizes are different. Thus the system occupancy and the CPU speed both decrease over time, in a step-like fashion, until the system empties. The mean response time for jobs under PS is 29.9 seconds. FSP-PS has exactly the same CPU speed profile as PS (by definition). However, it has a response time advantage over PS because of its dedicated service to one job at a time, rather than time-slicing. The mean response time for jobs under FSP-PS is 16.3 seconds. YDS uses a (rounded up) fixed speed of 1900 MHz on this workload, resulting in slightly lower energy consumption than FSP-PS, but slightly higher response time. The mean response time is much lower than PS, though.

Figure 4 shows the simulation results for Workload 3 (multiplicative jobs). For this workload, about half of the execution time is spent on the largest job, and it finishes at the lowest speed, since it is the last remaining job in the system. The mean response time under PS is 38.4 seconds, while that for FSP-PS is 25.7 seconds. YDS selects a fixed speed of 1400 MHz (rounded up) for this workload, resulting in a response time of 27.4 seconds, but the lowest energy consumption (as expected) amongst the three policies being compared.

The foregoing results show that our speed scaling simulator is working correctly, and that decoupled speed scaling (FSP-PS) has a distinct response time advantage over PS with coupled speed scaling. FSP-PS also provides fairness, since no job finishes later than it did under PS. YDS minimizes energy consumption, while sacrificing response time slightly.

With the validated simulator, we can now evaluate the response time, energy, and fairness characteristics for other scheduling policies and speed scalers that might be difficult to implement in Linux. Table IV provides examples of such results for Workload 1, Workload 2, and Workload 3. Among these results, SRPT-PS consistently provides the best response time, though it does not have the fairness properties of FSP-PS. FCFS performs identically to SRPT since these are batch workloads, and the jobs are in non-decreasing order of size. As expected, LRPT has the worst response time amongst the policies evaluated. However, it does not always have the highest energy consumption, since in some cases it completes the entire batch workload up to 40% sooner than other policies, such as PS. The results in Table IV provide further evidence of the many tradeoffs between response time, fairness, and energy consumption in dynamic speed scaling systems.

VII. CONCLUSIONS

In this paper, we discussed the design and implementation of a novel experimental environment for speed scaling measurements. Our implementation, which uses a mix of kernel-space and user-space functionality, provides a very general and flexible platform to quantify performance tradeoffs between different scheduling and speed scaling strategies.

We used our experimental platform in two different ways. First, we did micro-benchmarking to measure system costs and power consumption for different operating modes on the i7 processor. We used these measurement results to calibrate and validate a discrete-event simulator for dynamic speed scaling systems. Second, we used our simulator and experimental platform to evaluate three different speed scaling strategies from the literature: PS, FSP-PS, and YDS. We believe that our results provide the first direct comparison of these speed scaling strategies using realistic system costs.

ACKNOWLEDGEMENTS

Financial support for this work was provided by Canada'a Natural Sciences and Engineering Research Council (NSERC). The authors thank the anonymous MASCOTS 2016 reviewers for their constructive suggestions on improving our paper.

REFERENCES

[1] S. Albers, F. Mueller, and S. Schmelzer, "Speed Scaling on Parallel Processors", *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures* (SPAA), pp. 289-298, 2007.

[2] S. Albers, "Energy-Efficient Algorithms", *Communications of the ACM*, Vol. 53, No. 5, pp. 86-96, May 2010.

[3] L. Andrew, M. Lin, and A. Wierman, "Optimality, Fairness, and Robustness in Speed Scaling Designs", *Proceedings of ACM SIGMETRICS*, pp. 37-48, June 2010.

[4] N. Bansal, T. Kimbrel, and K. Pruhs, "Speed Scaling to Manage Energy and Temperature", *Journal of the ACM*, Vol. 54, 2007.

[5] N. Bansal, K. Pruhs, and C. Stein, "Speed Scaling for Weighted Flow Time", *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2007.

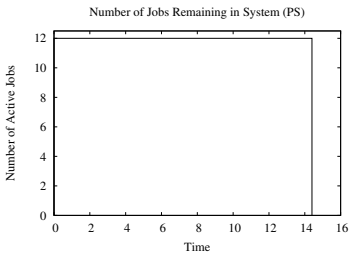
[6] N. Bansal, H. Chan, and K. Pruhs, "Speed Scaling with an Arbitrary Power Function", *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, 2009.

[7] D. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly, 2005.

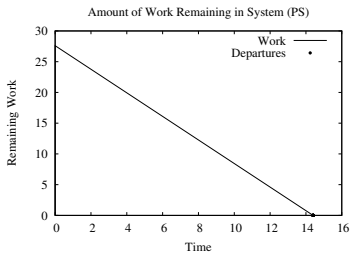
[8] M. Elahi, C. Williamson, and P. Woelfel, "Decoupled Speed Scaling: Analysis and Evaluation", *Performance Evaluation*, Vol. 73, No. 73C, pp. 3-17, March 2014.

[9] M. Elahi, C. Williamson, and P. Woelfel, "Turbocharged Speed Scaling: Analysis and Evaluation", *Proceedings of IEEE MASCOTS*, Paris, France, pp. 41-50, September 2014.

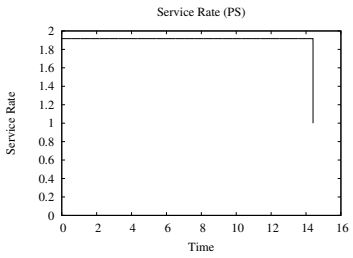
[10] E. Friedman and S. Henderson, "Fairness and Efficiency in Web Server Protocols", *Proceedings of ACM SIGMETRICS Conference*, San Diego, CA, pp. 229-237, June 2003.



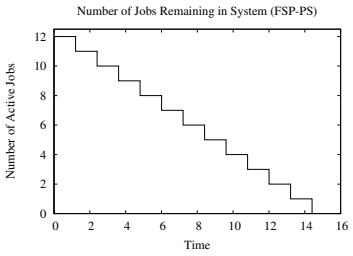
(a) Active Jobs (PS)



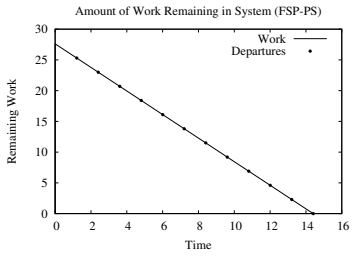
(b) Active Work (PS)



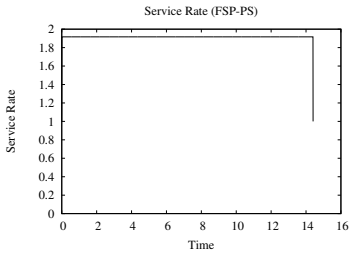
(c) CPU Speed (PS)



(d) Active Jobs (FSP-PS)

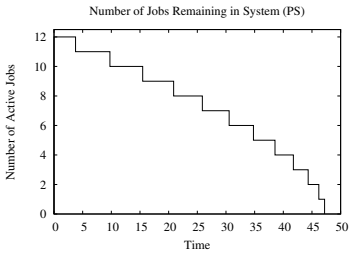


(e) Active Work (FSP-PS)

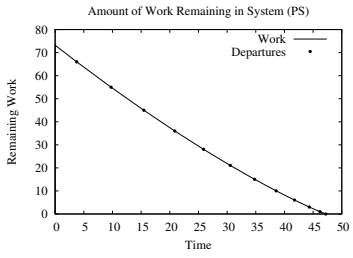


(f) CPU Speed (FSP-PS)

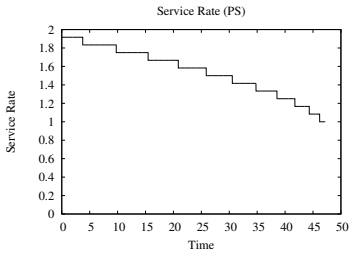
Fig. 2. Simulation Results for Dynamic Speed Scaling (Workload 1: Homogeneous jobs)



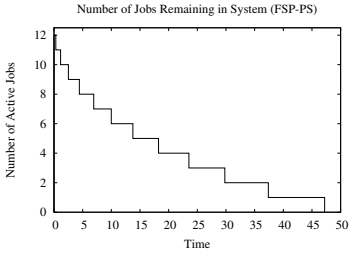
(a) Active Jobs (PS)



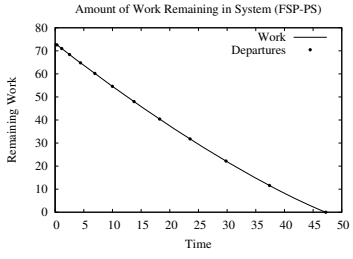
(b) Active Work (PS)



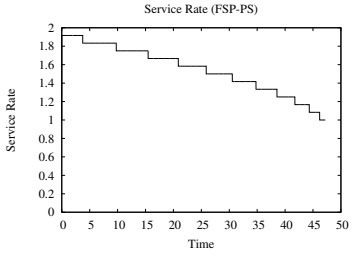
(c) CPU Speed (PS)



(d) Active Jobs (FSP-PS)

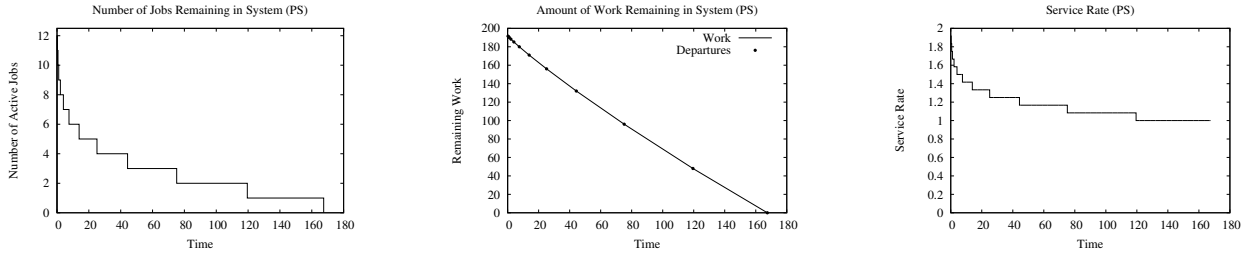


(e) Active Work (FSP-PS)

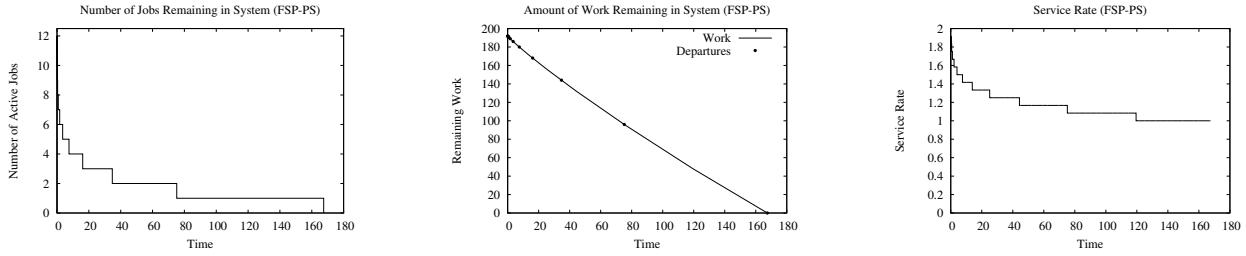


(f) CPU Speed (FSP-PS)

Fig. 3. Simulation Results for Dynamic Speed Scaling (Workload 2: Additive jobs)



(a) Active Jobs (PS) (b) Active Work (PS) (c) CPU Speed (PS)



(d) Active Jobs (FSP-PS) (e) Active Work (FSP-PS) (f) CPU Speed (FSP-PS)

Fig. 4. Simulation Results for Dynamic Speed Scaling (Workload 3: Multiplicative jobs)

TABLE IV
SIMULATION RESULTS FOR MEAN RESPONSE TIME $E[T]$ AND ENERGY CONSUMPTION (PP0 AND PKG) (12 JOBS, $\alpha = 1$)

Speed Scaling Policy	Workload 1				Workload 2				Workload 3			
	Time (s)	$E[T]$ (s)	PP0 (J)	PKG (J)	Time (s)	$E[T]$ (s)	PP0 (J)	PKG (J)	Time (s)	$E[T]$ (s)	PP0 (J)	PKG (J)
PS	14.4	14.4	75.5	132.4	47.2	29.9	205.1	387.3	167.5	38.4	564.8	1199.0
FSP-PS	14.4	7.8	75.5	132.3	47.2	16.3	205.0	387.3	167.5	25.7	564.8	1199.0
YDS	14.4	7.8	75.5	132.3	46.2	17.5	204.4	383.3	164.5	27.4	562.9	1186.8
FCFS	19.7	9.5	78.8	154.7	58.3	19.7	211.9	434.0	179.0	27.9	572.0	1247.2
SRPT	19.7	9.5	78.8	154.7	58.3	19.7	211.9	434.0	179.0	27.9	572.0	1247.2
LRPT	14.4	14.4	75.5	132.4	38.2	38.2	199.5	349.5	100.2	100.2	522.9	916.4
FCFS-PS	14.4	7.8	75.5	132.3	47.2	16.3	205.0	387.3	167.5	25.7	564.8	1199.0
SRPT-PS	14.4	7.8	75.5	132.3	47.2	16.3	205.0	387.3	167.5	25.7	564.8	1199.0
LRPT-PS	14.4	14.4	75.5	132.5	47.2	47.2	205.0	387.4	167.5	167.5	564.8	1199.1

[11] M. Hahnel, B. Dobel, M. Volp, and H. Hartig, "Measuring Energy Consumption for Short Code Paths Using RAPL", *Proceedings of ACM GreenMetrics*, London, UK, June 2012.

[12] Intel, "Intel 64 and IA-32 Architectures Software Developer Manuals", http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[13] R. Love, *Linux Kernel Development*, Pearson Education, 2010.

[14] D. Lu, H. Shen, and P. Dinda, "Size-based Scheduling Policies with Inaccurate Scheduling Information", *Proceedings of IEEE/ACM MASCOTS*, Volendam, Netherlands, pp. 31-38, October 2004.

[15] D. Meisner, B. Gold, and T. Wenisch, "PowerNap: Eliminating Server Idle Power", *Proceedings of ACM ASPLOS*, Washington, DC, pp. 205-216, March 2009.

[16] I. Rai, G. Urvoy-Keller, and E. Biersack, "Analysis of LAS Scheduling for Job Size Distributions with High Variance", *Proceedings of ACM SIGMETRICS Conference*, San Diego, CA, pp. 218-238, June 2003.

[17] T. Rauber and G. Runger, "Energy-Aware Execution of Fork-Join-based Task Parallelism", *Proc. IEEE MASCOTS*, Washington, DC, Aug. 2012.

[18] L. Schrage, "A Proof of the Optimality of the Shortest Remaining Processing Time Discipline", *Operations Research*, Vol. 16, pp. 678-690, 1968.

[19] D. Snowdon, S. Petters, and G. Heiser, "Accurate On-line Prediction of Processor and Memory Energy Usage under Voltage Scaling", *Proceedings of the 7th International Conference on Embedded Software*, pp. 84-93, Salzburg, Austria, 2007.

[20] D. Snowdon, E. Le Sueur, S. Petters, and G. Heiser, "Koala: A Platform for OS-level Power Management", *Proceedings of ACM EuroSys*, pp. 289-302, 2009.

[21] V. Spiliopoulos, A. Sembrant, and S. Kaxiras, "Power-Sleuth: A Tool for Investigating your Program's Power Behavior", *Proceedings of IEEE MASCOTS*, Washington, DC, August 2012.

[22] C. Stolte, R. Bosche, P. Hanrahan, and M. Rosenblum, "Visualizing Application Behavior on Superscalar Processors" *Proceedings of IEEE InfoVis*, pp. 10-17, 1999.

[23] V. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring Energy and Power with PAPI", *Proceedings of International Conference on Parallel Processing* pp. 262-268, September 2012.

[24] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for Reduced CPU Energy", *Proceedings of USENIX Operating System Design and Implementation* (OSDI), 1994.

[25] A. Wierman, L. Andrew, and A. Tang, "Power-Aware Speed Scaling in Processor Sharing Systems", *Proc. of IEEE INFOCOM*, April 2009.

[26] F. Yao, A. Demers, and S. Shenker, "A Scheduling Model for Reduced CPU Energy", *Proceedings of ACM Foundations of Computer Systems* (FOCS), pp. 374-382, 1995.