Software Engagement with Sleeping CPUs

Qi Zhu[*‡] Meng Zhu[†] Bo Wu[§] Xipeng Shen[*] Kai Shen[†] Zhiying Wang[‡]
[*] *North Carolina State University, USA*
[†] *University of Rochester, USA*
[§] *Colorado School of Mines, USA*
[‡] *National University of Defense Technology, China*

Abstract

Idle CPUs may enter power-saving hardware sleeps by, for instance, lowering the operating voltage and flushing the caches. However, wakeup delays that reach one hundred μSecs or more are disrupting the operations of fast devices like solid-state disks and tightly integrated accelerators. On the other hand, maximal power savings on modern multicores are only realized through continuous, simultaneous CPU sleeps. We argue that strong software engagement (at the OS and applications) is needed to maximize the power saving while maintaining the desired performance. Specifically, we present anticipatory CPU wakeups for latency-sensitive operations on fast devices. We also explore power-saving sleep shaping opportunities through non-work-conserving scheduling on smartphones and staged bursts on servers.

1 Motivation and Approach

An idle CPU can enter hardware sleep states that power down various resources to conserve energy. The wakeup of a sleeping CPU incurs a delay due to the restoration of the hardware operating condition and state. For instance, an Intel E5-2620 v3 Haswell socket can save a few dozen Watts of power by sleeping at the ACPI C6 state (cycles halted; clocks shut off; L1 / L2 cache flushed; core voltage removed). On the other hand, its wakeup causes 106 μSecs of extra delay for a simple network ping-pong service.

Emerging hardware and workload trends have highlighted the challenges of the CPU sleep management. From the performance perspective, wakeup delays of dozens or hundreds of μSecs could cause excessive disruptions. Commodity solid-state disks are already operating at 100 μSec-level latencies (particularly for reads). Emerging integrated accelerators like GPUs (exemplified by Intel IvyBridge and Haswell, AMD Fusion APUs, and NVIDIA Denver) allow fast CPU / accelerator interactions and facilitate efficient acceleration of fine-grained tasks. A slow CPU wakeup produces significant slowdown for these device operations. In a further example, latency-sensitive network services (e.g., a memory-cached hash table) inside a data center may suffer multi-fold slowdown if a wakeup from a CPU deep sleep is involved.

From the power perspective, the benefit of CPU sleeping is highly influenced by the sleep pattern. First, it is desirable for a CPU to sleep continuously to minimize the energy costs during active / sleeping state transitions. Second, due to aggressive resource sharing, modern multicores realize maximal power saving only when all CPU cores/threads sleep simultaneously. Unfortunately, continuous, simultaneous CPU sleeps are rare under the conventional work-conserving CPU scheduling when fine-grained units of work may activate one or a few CPUs in intermittent fashions.

This paper argues for strong software engagement (at both OS and application levels) of CPU sleep management to maintain high performance when needed and maximize the energy saving when possible. Specifically in low-latency operating conditions, the CPU should start waking up early (before the work-triggering interrupt) such that the CPU is immediately ready for work when needed. Such anticipatory wakeups are best requested by the software layers with knowledge of or ability to model the time of future work resumption. The OS should aggregate anticipatory wakeup requests and maintain a CPU sleep plan with high efficiency and (if desirable) proper energy accounting.

On the other hand, many system and application contexts manifest a high degree of slacks in their quality-of-service requirements. For example, due to the long operation time of the wide area network and other peripheral devices in smartphones, some tasks can be slowed substantially without compromising the user experience. Also, requests in a web server application may be delayed as long as they are all completed within a specified latency threshold. Such quality-of-service flexibility presents opportunities for shaping the CPU idleness patterns (through delaying, staging, and consolidating work) toward continuous, simultaneous CPU sleeps with high power saving.

Much of prior attention on energy-efficient CPU management has targeted the processor frequency control [13, 22, 23]. Among the most seminal, Weiser et al. [23] proposed fine-grained frequency adjustment and workload scheduling to conserve energy. Recent work by Le Sueur and Heiser [20] experimentally demonstrated the large energy effects of CPU hardware sleeps, but without proposing an approach for systematic management. The network community has explored the use of "sleep" proxies to keep machines continuously shutdown [5, 17]. These works target the ACPI S state management where a "sleeping" machine may take seconds to wakeup, which is very different from the much finer-grained CPU hardware sleeps that we study.

2 Latency-Sensitive Anticipatory Wakeup

In a latency-sensitive environment, the wakeup delay of a sleeping CPU can be a substantial performance concern. If the future work triggering event can be anticipated with a predictable elapsed time, the CPU should start waking up in advance so that it is immediately ready for work when needed. Errors of such prediction can be tolerated by earlier CPU wakeups at the cost of additional power consumption.

Anticipatory CPU wakeups are particularly beneficial for operations on fast devices. For example, the I/O completion time on a Flash-based solid state disk can be modeled on the operation type (read / write) and I/O size. The operating system device layer can maintain such a model and initiate an anticipatory CPU wakeup for each synchronous I/O operation so that the waiting application receives a prompt response. This support can be realized with full application transparency.

On the other hand, it is generally impossible to predict the run time of computational kernels on Turing-complete accelerators like GPUs. Fortunately, many GPU applications iterate over similar computational kernels in their execution (e.g., in an iterative linear system solver or an iterative machine learning refinement like K-means clustering). Such an iterative pattern enables an application to make history-based runtime prediction and request for anticipatory CPU wakeups accordingly.

On a network server, the CPU wakeup delay also prolongs the response time to client requests. However, it is difficult for the server to anticipate future client requests, especially if the request arrival follows a memoryless Poisson process (when requests come from a large number of independent clients). In such cases, anticipatory wakeups may only be possible if they are requested in advance by remote clients before they issues requests for real work.

On a multicore machine with a large number of CPUs, the anticipatory wakeup should only activate the CPU(s) necessary for latency-sensitive operation to conserve the power consumption. On the other hand, it must ensure the readiness for all tasks on the notification path of the wakeup event. This includes the blocking user process (e.g., the one blocked on the `read` system call to the SSD) as well as the device interrupt handler. Waking up the interrupt handling CPU is straightforward if the interrupts are handled by a fixed CPU. It requires more coordination if the interrupt handling is load-balanced across a number of CPUs.

It is easy to foresee a system environment where multiple fast devices and network services are serving latency-sensitive tasks. Therefore multiple simultaneous anticipatory wakeups may be requested in the system. The operating system should aggregate such requests and maintain a CPU sleep plan with high efficiency. Such multi-use environments also raise the question on fine-grained energy resource accounting and attribution between concurrent tasks [19]. In particular, the energy use of a resource principal should include that of its anticipatory CPU wakeups even if the system idles for much of the wakeup durations.

In terms of resolving the power / latency conflict, our anticipatory CPU wakeup is related to the Linux Wakelocks mechanism (developed primarily for Android). However, the Wakelocks mechanism is not anticipatory and it currently manages the system suspension but not CPU sleep states. Our idea is also reminiscent of the anticipatory I/O work [11] that targets a very different problem context—anticipating future high-locality I/O accesses to reduce the seek time on mechanical disks.

SSD Case Study We made a preliminary implementation of the proposed anticipatory CPU wakeup mechanism in the Linux 3.12.13 kernel. A wakeup request on a CPU is made with two parameters—the starting time and duration of the wakeup. The initial wakeup is implemented using a kernel high-resolution timer. In the wakeup duration, the CPU idle management is prevented from entering any of the hardware sleep states. The CPU may return to the normal sleep if the anticipated event does not occur by the end of the wakeup duration. To enable fast responses of SSD I/O accesses, we augment the device layer to initiate an anticipatory CPU wakeup for each synchronous I/O operation.

We perform an experimental study on a dual-socket (6-core, 12-hyperthread per socket) Intel E5-2620 v3 Haswell machine with a Samsung 850 PRO SSD. A device-level 4 KB read to the SSD takes about 100 μSecs. When a 4 KB read is dispatched to the device, the OS lets the CPU to enter the C1-HSW sleep state. Given approximately 20 μSecs of sleep exit latency, we make an anticipatory request to wake up the CPU in 80 μSecs for the duration of 20 μSecs.

Figure 1 illustrates the performance and power of a synthetic I/O workload with varying inter-I/O idle time. Results show that the I/O latency of the original Linux

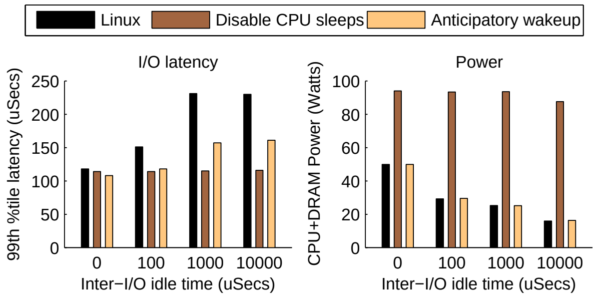

Figure 1: Performance (99th percentile I/O latency) and power of an SSD-based workload that repeatedly reads 4 KB of data at a random file location with varying inter-I/O idle time.

system doubles when there is 1 mSec or more inter-I/O idle time due to the CPU wakeup delay. While disabling all CPU sleeps can achieve the optimal I/O latency, it would increase the CPU+DRAM power by 2–5×. In comparison, our anticipatory CPU wakeup mechanism incurs no visible power cost while it significantly brings down the I/O latency slowdown (from 101% to 37% at 1 mSec inter-I/O idle time).

The predictability of the SSD I/O time is enabled by its strong linear correlation with the I/O size. On our Samsung 850 PRO SSD, the Pearson's correlation coefficient[1] between read latency and I/O size is very close to 1.00. The correlation coefficient between write latency and I/O size is also quite high at 0.91. However, if the OS dispatches multiple I/O operations to the SSD (which is beneficial to exploit SSD I/O parallelism [15, 18]), the I/O time prediction would be more difficult.

GPU Case Study We also test the idea of anticipatory CPU wakeup on GPU program executions. A GPU kernel is usually launched by a CPU control thread. CPU sleeping can delay the control thread from finding out the completion of a GPU kernel computation.

The problem is particularly pronounced on CPU-GPU integrated processors, where the tight integration of CPU and GPU allows the use of GPU to accelerate short requests—the lengths of which may be comparable to a sleeping CPU's wakeup delay. For instance, the GPU kernel lengths in most benchmarks in AMD SDK v2.9 [1] range from 100 μSecs to 1000 μSecs on an AMD A10-6800K processor. A 100 μSecs wakeup delay of CPU can degrade the responsiveness of the request processing substantially. Meanwhile, because such requests are often repeatedly issued in the programs, they can together weigh substantially in the overall execution time of the program. Wakeup delays hence can also significantly slow down an entire program.

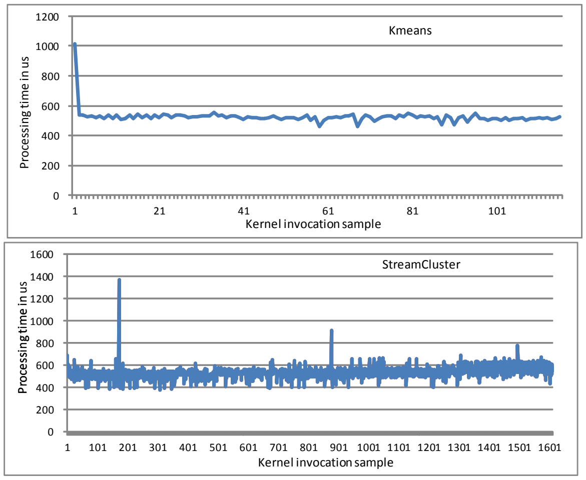

Figure 2: The processing time for the sequence of GPU kernel requests invoked by K-means and StreamCluster.

A main concern for utilizing the anticipatory CPU wakeup in GPU applications is the challenge of predicting a GPU kernel's runtime. Fortunately, many GPU applications iterate over similar computational kernels in their execution, which makes history-based runtime prediction promising. We experiment with two data clustering programs, K-means and StreamCluster [3]. In both, clustering happens through an iterative process. In each iteration, a GPU kernel is invoked to compute the distances between all data points and all potential cluster centers; after that, the potential cluster centers are updated based on the distances, the kernel is invoked again, and the process continues until centers stop changing.

Figure 2 reports the processing time for the sequence of GPU kernel requests invoked in the two benchmarks. On K-means, the kernel lengths are around 540 μSecs with less than ± 25 μSecs fluctuations (except for the first invocation). On StreamCluster, the lengths of most kernel invocations are around 500 μSecs with about ± 50 μSecs except for a few outliers. The stability suggests good predictability of the kernel running time. Our experiments show that for the two benchmarks, the kernel length can be predicted with about 87% accuracy. Anticipatory wakeup helps K-means and StreamCluster speed up about 10% over the default interrupt-based method. Meanwhile, it helps them save about 56% power, compared to busy-waiting, in which the CPU control thread keeps polling for the status of GPU.

3 Energy-Conserving Sleep Shaping

Other than causing latency concerns, CPU sleeps on modern multicores produce disproportionate power usage (due to hardware resource sharing on a multicore socket) and thus present new opportunities for saving energy. Figure 3(A) shows that on a dual-socket (6-core,

[1] The covariance of two variables divided by the product of their standard deviations. Nearing 1.0 indicates a strong linear correlation.

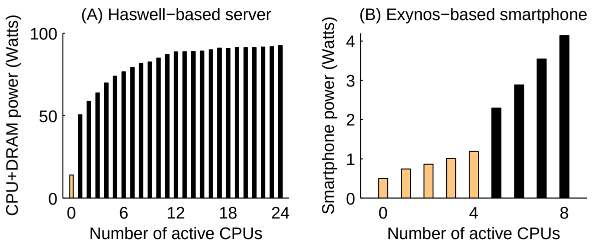

Figure 3: Disproportionate multicore power on the number of active CPUs. The CPU package and DRAM power on the Haswell server (A) was reported by the Intel processor's power MSRs. For the Exynos-based smartphone (B), we removed its battery, connected it with an external power source, and intercepted the power line to measure the phone power using an Agilent U1272A digital multimeter. The phone display was dimmed to the minimum brightness during our power measurement.

12-hyperthread per socket) Intel E5-2620 v3 Haswell machine, activating the first CPU consumes 37 Watts of power beyond idling while activating each additional CPU consumes 8 Watts or less. In another example, Figure 3(B) shows the power of an Exynos 5422-based Samsung Galaxy S5 smartphone. The Exynos system-on-chip adopts the ARM big.LITTLE architecture [9] with two heterogenous clusters of four cores each—the *small* cluster contains four Cortex-A7 cores that run up to 1.3 GHz, and the *big* cluster contains four Cortex-A15 cores that run up to 1.9 GHz. We observe a disproportionate power jump when activating the big cluster (starting at the fifth core).

Through two case examinations, we argue that flexible quality-of-service requirements in a broad range of system / application scenarios allow for the shaping of CPU sleep patterns toward energy conservation. Our work is related to earlier efforts of shaping a device's sleep / idle pattern for energy saving (e.g., page allocation that consolidates the memory uses on specific DRAM chips [12] and prefetching / caching to enable bursty disk accesses [14]), but we face new challenges due to different problem contexts.

Slack Enabled Quiescence Today's smartphones are equipped with powerful multicore processors. However, our experiments found that typical mobile applications do not exhibit sufficient parallelism to fully utilize them. This provides opportunities to apply aggressive energy saving techniques. On the other hand, high-quality user interactions discourage any resource-saving mechanisms that would negatively affect the user experience. We argue that strong software management of CPU sleep states can achieve energy savings on smartphones without compromising the quality-of-service.

Putting CPUs to sleep with no performance penalty is possible if all CPU tasks over a time period are not on the critical path leading to the user response. Such slack of non-critical CPU tasks are most common during synchronous network or I/O operations that block the user response. The slack allows a period of quiescence—simultaneous sleeps of all CPUs or CPUs on the high-power cluster of a heterogeneous platform. We note that not every I/O operation presents opportunities for CPU sleeps without performance penalty. In particular, during a background I/O operation (e.g., GPS beacon processing), concurrent CPU tasks may still be on the user response critical path and therefore present no slack for CPU quiescence. Delaying or slowing them may lead to longer user responses.

We discuss how to identify I/O operations on the critical user response path. A user interaction is generally triggered by a touch event and ends with a screen update. The OS may track events that signify causal dependencies [16, 25] between execution segments in a user interaction. They include kernel events such as process / thread forking and inter-process signals, pipes, and sockets, and application activities like Android Binder messages and looper thread work queue operations. The online dependency information can tell whether a synchronous I/O operation is causally dependent on the user input event. However, analysis of the entire critical path generally requires the offline construction of the full causal dependency graph leading to the final user response. We can leverage the unified UI framework of the modern mobile operating systems, in which interaction with a UI component generally triggers a deterministic series of operations that follow the same causal dependencies. Thus we can memorize the offline behavior characterization for each UI component to help identify the performance-critical execution path online.

We assess the feasibility of this idea on the Exynos 5422 based Samsung Galaxy S5 smartphone. Its CPU power characteristics were presented in Figure 3. We use Bbench [10], an automatic browser test script that repeatedly loads locally cached websites. We emulate the wide area network delay by injecting a 100 mSecs wait when the browser loads a web page. During this period, our system delays all the tasks on the four CPUs of the big cluster and lets them enter deep sleeps. Our experiments show that such CPU quiescence produces no observable performance impact while it reduces the total energy consumption by 6%.

Such slack enabled quiescence can be more effective in energy saving if there are substantial co-run background tasks. For example, the user may browse the web while the phone is performing software updates. The system can delay non-critical background tasks (software updates) while synchronous I/O operations (e.g., fetching web pages) are blocking the user response. Their executions are piggybacked while the high-power CPU cluster is activated for critical operations leading to the user response. Such co-execution does not impact performance as long as the critical tasks do not fully utilize the multicore CPUs.

Staged Bursts The response time to client requests is an important quality-of-service metric for web applications. In many situations, rather than completing every request as soon as possible, a request may be delayed as long as it is responded within a certain time threshold (e.g., the latency of pleasant human perception). At the same time, we recognize the potential of high parallelism in server applications—many independent requests, while running simultaneously, produce high energy efficiency on multicores. These motivate a non-work-conserving workload management approach, we call *staged bursts*, where the server operation alternates between two phases—a staging phase that buffers requests without running them, and a burst phase that runs the buffered requests at a high degree of parallelism.

We use a simple experiment to demonstrate the potential effectiveness of server staged bursts. We set the response time threshold at 0.5 second. On our 24-CPU Haswell machine running Linux 3.12.13, we set up the Apache Lucene / Solr [2] search engine as an Apache Tomcat Servlet container. We constructed a search workload of 1,414,444 indexed documents from the Wikipedia data dumps [4]. Queries in our test workload are generated by randomly selecting and sequencing article titles in the Wikipedia data dump.

When serving the workload at 100 requests/sec, the machine's processor package and DRAM consumes about 68 Watts of power under the conventional system setup while the 999th permille response time is 284 mSecs. We then add a staging proxy (running on another machine) which buffers requests and releases all buffered requests once every 250 mSecs. Under such staged bursts, the processor package and DRAM consumes only 53 Watts of power (22% reduction) while the 999th permille response time is still below 500 mSecs.

A robust support of staged burst processing faces important challenges that require strong engagement from all system layers. Specifically, the maintenance of tail latency service objectives [8] requires application-level performance feedbacks [13] as well as kernel-level resource control. For instance, resource containers [6]-style mechanisms can encapsulate activities belonging to individual client requests, and dynamically trigger burst processing when some staged requests are in danger of missing their latency deadlines.

During the burst phase, energy efficiency favors simultaneous utilization of all CPUs. However, a conventional OS sometimes hesitates in migrating tasks for CPU load balancing, due to the concern of losing the cache locality (and DRAM locality in the case of NUMA). In the above experiment, to achieve more simultaneous burst processing, we removed Linux's restriction that tasks are not migrated to idle CPUs with short idle periods in the recent past. Balancing the energy efficiency and cache locality goals may require the modeling of power [19] and locality effects [7, 21] from collectible processor hardware event counters and statistics.

To enforce simultaneous CPU sleeps, it is important to idle all CPUs on the main socket during the staging phase. Therefore the request receipt and buffering must be done elsewhere which still incurs an energy cost. To minimize such costs, one idea is to run the staging proxies for a large pool of servers on a single staging machine. Another is to use a companion low-power processor (such as the KnightShift architecture [24]) to stage and buffer requests. Our experience suggests that the latter idea can be accomplished by today's smartphone processors (using only one Watt of power while adding a few mSecs of additional latency).

4 Conclusion

This paper recognizes the significant power and performance implications of hardware sleeps on modern multicores. We thus argue for strong software engagement to realize the performance and energy saving potentials. Specifically, in latency-sensitive environments, anticipatory wakeups can maintain high performance despite the significant sleep wakeup delays. On the other hand, if the system quality-of-service requirements allow certain flexibility of delayed work, workload staging and parallel burst processing can enable high energy efficiency on multicores.

We conclude with a series of open questions. Anticipatory CPU wakeups require predicting the time of future work triggering events that can be challenging, particularly for network servers responding to external client requests. Identifying quality-of-service slacks and opportunities for simultaneous CPU sleeps requires online tracking of execution dependencies and identification of performance-critical task segments. Finally, work consolidation for energy-efficient parallel multicores execution must balance the need for cache affinity and reconcile with the limitation of application / system scalability.

Acknowledgments This work was supported in part by the U.S. National Science Foundation grants CNS-1217372, CNS-1239423, CCF-1255729, CNS-1319353, CNS-1319417, CNS-1320796, CCF-1464216, and a Career Award, and by a Department of Energy Early Career Award, a Google Research Award, and the Semiconductor Research Corporation Contract No. 2013-HJ-2405.

References

[1] AMD APP SDK 2.9. developer.amd.com/tools-and-sdks/.

[2] Apache Solr search server. lucene.apache.org/solr/.

[3] Rodinia benchmark suite. www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page.

[4] Wikipedia data dumps. dumps.wikimedia.org/enwiki/.

[5] Y. Agarwal, S. Savage, and R. Gupta. SleepServer: A software-only approach for reducing the energy consumption of PCs within enterprise environments. In *USENIX Annual Technical Conf.*, San Deigo, CA, June 2009.

[6] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Third USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 45–58, New Orleans, LA, Feb. 1999.

[7] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for NUMA-aware contention management on multicore systems. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

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

[9] P. Greenhalgh. Big. little processing with arm cortex-a15 & cortex-a7. *ARM White paper*, 2011.

[10] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver. Full-System Analysis and Characterization of Interactive Smartphone Applications. In *2011 IEEE Int'l Symp. on Workload Characterization (IISWC)*, pages 81–90, Austin, TX, USA, 2011.

[11] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *18th ACM Symp. on Operating Systems Principles (SOSP)*, pages 117–130, Banff, Canada, Oct. 2001.

[12] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *9th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–116, Cambridge, MA, Nov. 2000.

[13] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *41st Int'l Symp. on Computer Architecture (ISCA)*, pages 301–312, Minneapolis, MN, June 2014.

[14] A. E. Papathanasiou and M. L. Scott. Energy efficient prefetching and caching. In *USENIX Annual Technical Conf.*, Boston, MA, June 2004.

[15] S. Park and K. Shen. FIOS: A fair, efficient Flash I/O scheduler. In *10th USENIX Conf. on File and Storage Technologies (FAST)*, San Jose, CA, Feb. 2012.

[16] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. AppInsight: Mobile app performance monitoring in the wild. In *10th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 107–120, 2012.

[17] J. Reich, M. Goraczko, A. Kansal, and J. Padhye. Sleepless in Seattle no longer. In *USENIX Annual Technical Conf.*, Boston, MA, June 2012.

[18] K. Shen and S. Park. FlashFQ: A fair queueing I/O scheduler for Flash-based SSDs. In *USENIX Annual Technical Conf.*, San Jose, CA, June 2013.

[19] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen. Power containers: An OS facility for fine-grained power and energy management on multicore servers. In *18th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[20] E. L. Sueur and G. Heiser. Slow down or sleep, that is the question. In *USENIX Annual Technical Conf.*, Portland, OR, June 2011.

[21] D. Tam, R. Azimi, and M. Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Second EuroSys Conf.*, pages 47–58, Lisbon, Portugal, Mar. 2007.

[22] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The TURBO diaries: Application-controlled frequency scaling explained. In *USENIX Annual Technical Conf.*, Philadelphia, PA, June 2014.

[23] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *First USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, pages 13–23, Monterey, CA, Nov. 1994.

[24] D. Wong and M. Annavaram. KnightShift: Scaling the energy proportionality wall through server-level heterogeneity. In *45th Int'l Symp. on Microarchitecture (MICRO)*, pages 119–130, Vancouver, Canada, Dec. 2012.

[25] L. Zhang, D. R. Bild, R. P. Dick, Z. M. Mao, and P. Dinda. Panappticon: event-based tracing to measure mobile application and platform performance. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2013 International Conference on*, pages 1–10. IEEE, 2013.