Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications

Svilen Kanev Kim Hazelwood[†] Gu-Yeon Wei David Brooks
Google, Inc.[†] Harvard University

***Abstract*—The growth in datacenter computing has increased the importance of energy-efficiency in servers. Techniques to reduce power have brought server designs close to achieving energy-proportional computing. However, they stress the inherent tradeoff between aggressive power management and quality of service (QoS) – the dominant metric of performance in datacenters.**

In this paper, we characterize this tradeoff for 15 benchmarks representing workloads from Google's datacenters. We show that 9 of these benchmarks often toggle their cores between short bursts of activity and sleep. In doing so, they stress sleep selection algorithms and can cause tail latency degradation or missed potential for power savings of up to 10% on important workloads like web search. However, improving sleep selection alone is not sufficient for large efficiency gains on current server hardware. To guide the direction needed for such large gains, we profile datacenter applications for susceptibility to dynamic voltage and frequency scaling (DVFS). We find the largest potential in DVFS which is cognizant of latency/power tradeoffs on a workload-per-workload basis.

I. Introduction

The explosion of cloud services in the last decade has led to computation moving away from the desktop into a datacenter environment. Large Internet services are run in a new type of datacenter, referred to as a warehouse-scale computer (WSC) [3]. While the major focus in designing WSC services is on predictable and scalable performance, the mere scale of such datacenters also makes them prime candidates for techniques that reduce server power – the aggregate energy savings are beneficial from both a cost and an environmental perspective. A typical WSC workload, such as web search, or ad serving, is comprised of a tree of individual services, each with their respective service-level agreement (SLA) for performance. Applying power saving techniques on the servers responsible for those services can adversely affect performance and lead to invalidation of SLAs. Prior work on web search from Microsoft [13] and Google [20] outlines this tradeoff between power efficiency and request latency.

Power efficiency of datacenters, and WSCs in particular, has been the target of a significant body of research [2], [5], [10], [26]. It is well-established that datacenters spend a large portion of time under-utilized. For example, Barroso and Hölzle show a typical CPU utilization of 5,000 Google servers in the 10-50% range over 6 months [3]. This utilization variance is caused by changing user demand and difficulties in inter-datacenter load-balancing. Figure 1 also demonstrates the varying user demand in a Google production cluster in North America running content ad matching. Notice that for extended periods of time incoming requests (measured in queries per second, QPS) spawn the very wide range of 15-105% of allocated capacity.

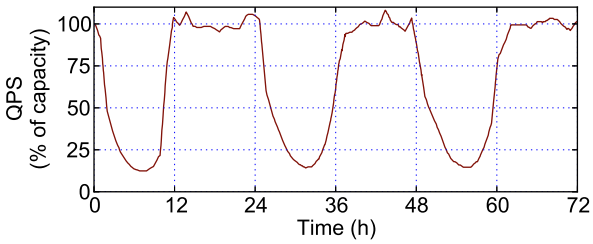


Figure 1: *Utilization of content ad matching in a production cluster, measured in incoming queries per second (QPS), normalized by the allocated cluster capacity.*

Handling such large swings in utilization in an energy-efficient manner is the main motivation behind energy-proportionality [2]. In an ideal energy-proportional system, server power consumption would perfectly track the arrival patterns of requests shown in Figure 1. Achieving proportionality requires that most power-hungry components be able to scale down their power consumption with usage.

Today's server systems are far from energy-proportional [3], [20]. Figure 3 verifies this claim on three contemporary server platforms running a websearch leaf. If these platforms were proportional, their power consumption would follow the dashed line which scales linearly from zero to maximum power. This is not the case for the platforms in question. At a utilization of 50% they consume ≈80% of the maximum system power, not 50%, creating an `energy proportionality gap` [31]. Note that, similar to observations by Wong and Annavaram [31], there is no drastic difference between the three platforms in terms of power scaling, except for the data point at 0% usage, where `Platform A` (the oldest of the three) consumes significantly more power.

Furthermore, the processor consumes the largest portion of system power in recent WSC servers [3], [13]. While DRAM power was considered as a challenger for this dominant position, recent advances in memory power efficiency (especially the broader adoption of the low-voltage DDR3L standard) have changed this trend. Figure 4 demonstrates that for a contemporary server platform: at full load the processors consume 78% of the system power.[1] Furthermore, just the

[1]Power distribution data is measured at sense resistors before component regulators on a 16-core, Intel SandyBridge platform with 256GB of DRAM.

978-1-4799-6454-3/14/$31.00 ©2014 IEEE

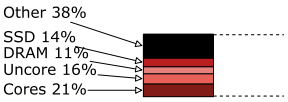


Figure 2: *Idle server power is evenly split between components.*

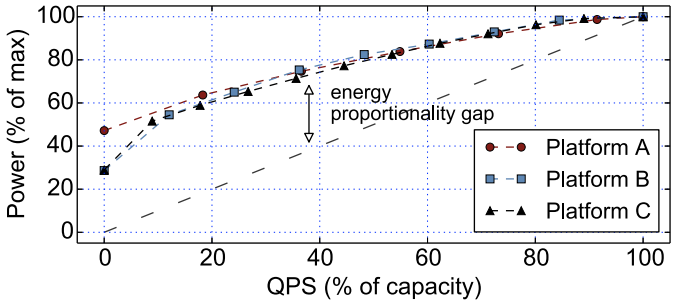


Figure 3: *Energy proportionality (power as a function of incoming load) across three successive x86 server platforms has remained virtually unchanged.*

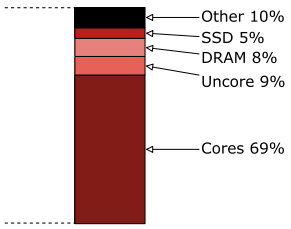


Figure 4: *Processor power dominates on a fully loaded machine.*

dynamic range of processor power between fully idle and fully loaded is 67% of the maximum system power (from Figures 2 and 4). This suggests that the processor has the largest potential to bridge the energy proportionality gap in Figure 3, especially at mid-range utilization. Therefore, in this paper, we focus on processor power management.

We characterize two complementary mechanisms for power management: idle and active. In the first one, idle power states (C-states), decrease core power when no threads have claimed a core and it is executing the operating system idle loop. Processors expose different C-states to the OS that trade sleep/wake-up latency for power savings by powering down different parts of the core and its corresponding caches. In a distributed system, the longer sleep/wake-up latencies of aggressive sleep states can fall on the critical path of incoming requests, and subsequently increase request latency. Thus, a WSC has conflicting requirements between aggressive power savings and aggravated request latency (and missing SLAs). In this light, we address the following questions:

- What are the sleep patterns of current WSC application? Are applications' idle periods short enough to be affected by the choice of a particular sleep state?
- How much does C-state selection influence request latency and system power on the macro level? In other words, by how much can proper selection improve latency, and what are the maximal power savings from using idle periods?

We evaluate 15 benchmarks based on Google production workloads. We show that for a fraction of them, sleep activity is sufficiently coarse-grained that their latency response is not affected by the choice of a particular sleep state. However, fine-grained sleep activity does exist for certain applications. For them, choosing inappropriate sleep states can result in a latency or system power cost of up to 10%.

Aggressive idle power management, which is already included in contemporary processors, is however not sufficient to achieve an energy-proportional system. Thus, we turn our focus on active power management, which slows down cores while they are busy with execution. We evaluate the potential benefits of WSC applications from reducing voltage and frequency during memory-bound phases of execution.

We first show that the same set of WSC benchmarks is highly memory-bound on average, suggesting that active management through dynamic voltage and frequency scaling (DVFS) can be efficient. We identify a wish list of characteristics that a practical DVFS solution in the datacenter should abide by. We then show that it is unlikely to satisfy everything on this wish list simultaneously by implementing a prototype system. Finally, we identify that the directions holding the largest promise are workload-specialized and ultra-fine-grained DVFS, which warrant further study.

II. Idle Management and Latency

We briefly describe the mechanisms for idle power management, which is largely responsible for current systems' power savings at low load. While idle management saves significant energy, we show that it can also cause latency degradation, given workloads that are bursty enough.

The mechanism for processor idle power management involves shutting down cores (or whole sockets) without work to do. Figure 5 illustrates the process of entering a core idle state (C-state): after there is no work to be scheduled in userspace, the kernel executes a specific instruction (`mwait` on x86), with a parameter indicating the requested C-state. In Linux terminology, the logic to select the appropriate C-state is called a *governor*.

The trade-off made by C-state governors is between power savings and wake-up latency. Deeper C-states save more power by power gating larger portions of the chip, but require a longer wake-up time (and potentially more energy) to restore state [19]. The minimum idle period for a specific C-state to be profitable energy-wise is referred to as *target residency*. For example, Intel's SandyBridge microarchitecture exposes 5 core C-states [29]:

state	residency	wake-up latency
C0	*(active)*	*(active)*
C1	1 μs	1 μs
C3	106 μs	80 μs
C6	345 μs	104 μs
C7	345 μs	109 μs

The kernel governor is not the only system in charge of idle power management. Recent architectures include a shared power control unit (PCU), whose purpose is to orchestrate power management for the processor. The PCU can ignore

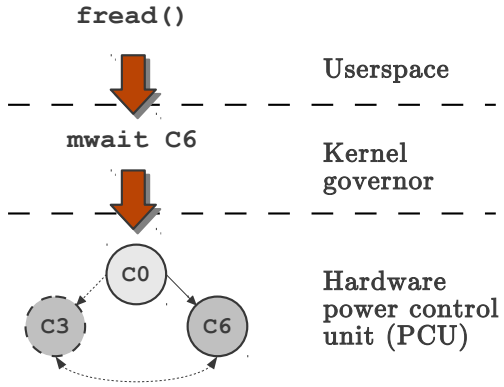


Figure 5: *The different layers of the hardware/software stack involved in managing idle power states. Dashed lines indicate state transitions only possible in hardware.*

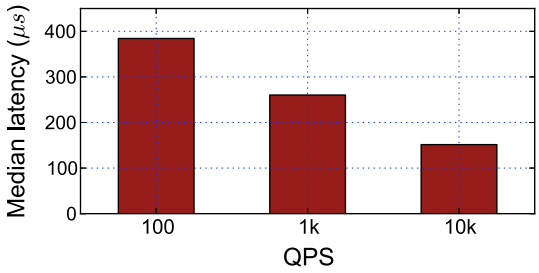


Figure 6: *Round-trip latency degradation for a RPC transport layer for varying queries per second (QPS). Too aggressive sleep states (at low QPS) significantly degrade request latency.*

software requests for a specific C-state, choosing to enter a shallower one, if it estimates that the residency requirement of the deeper state will not be met. This behavior is called *C-state demotion* and is controlled by a proprietary algorithm set by processor vendors. Furthermore, the PCU can choose to transition a core between different sleep states without waking it up – a knob not available to software.

Thus, idle power can be independently managed both in hardware and software. Both approaches have their benefits. The PCU has a closer and more fine-grained view of different cores' power consumption, as well as finer-grained control knobs. On the other hand, software has a global, non-processor-centric view of the full system, and can predict future events, such as incoming disk interrupts, and react accordingly. Combining management on the two layers such that they co-operate and do not get involved in "power struggles" [25] requires a detailed understanding of both of them in isolation. In this paper, we focus predominantly on the software layer.

[Latency cost] Idle power states can save a significant amount of system power [1], [16], [20]. They are largely responsible for the power scaling of current platforms demonstrated in Figure 3 (for example, the sharp "knee" of the curve for `Platform C` near 12% QPS is the result of a whole socket being able to go idle at the same time). However, selecting the optimal sleep state requires accurate prediction of sleep length. Predicting an idle period too short may cause missed opportunities for power savings, if a deeper sleep state is available. Similarly, predicting it too long may cause a premature wake-up, adding the state wake-up latency to the already time-critical interrupt processing.

We refer to the second effect as the *latency cost of sleep*. Figure 6 illustrates how significant that cost can be. It shows the median round-trip latency of a remote procedure call (RPC) layer microbenchmark at different loads. [2] The resulting workload is ideal for investigating the latency effects of idle states – CPU utilization is low, so cores sleep often, and request processing times are smaller than $100\mu s$, resulting in very fine-grained sleep behavior.

Under low load (100 QPS), the cores on the critical path of computation are idle for a significant fraction of time and do enter deep sleep states. This results in an overall $2.4\times$ increase in request latency compared to the high-load (10,000 QPS) case. Such latency degradation is unintuitive – in the canonical distributed system governed by queuing effects increasing incoming request rates leads to higher latencies, not lower.

We want to verify whether such sleep effects manifest themselves on macro-scale benchmarks. In order for them to be comparable to more typical queuing effects, request processing latencies, or query interarrival rates, must be comparable to deep C-state residencies – on the order of hundreds of μs. In other words, the workload must be "bursty". In examining C-states, LeSueur and Heiser [16] notice that an Apache web server exhibits such bursts with lengths smaller than 1ms. Furthermore, Meisner et al. [21] show that a web search cluster can be modelled with an average query interarrival rate of $300\mu s$. This motivates a more detailed characterization of WSC benchmarks, with the aim to determine whether they exhibit sub-millisecond sleep periods, and are therefore susceptible to the latency cost of sleep.

III. Examining Sleep Patterns

In this section, we look into the idleness patterns of current Google applications. This is a necessary first step in determining whether datacenter workloads are potentially vulnerable to the latency cost of deep sleep. Since our ultimate goal is to better understand power vs. tail latency trade-offs, we select a subset of our benchmarks, which satisfy the constraints of: being latency-sensitive; and requesting idle states on a sub-ms scale. We find that popular applications, like search, or ad serving, fall into this category. For them, we measure the maximum impact that C-state selection can have on tail latency and system power – up to 10-15%.

A. Experimental setup

[Hardware configuration] We perform all our experiments on a 2-socket, 16-core Intel SandyBridge-based server, which has a total of 32 thread execution contexts. Its idle power states are as described in Section II. We fix all cores' frequency at

[2] Specifically, one server sends a small (several bytes) payload over the network and waits for a response, measuring the round-trip latency. The receiver dedicates several cores to handling network interrupts and to forwarding the payload to a single core on the same chip, which immediately sends the payload back to the sender.

Name	Description	Relevant metric
latency-insensitive		
`saw`	String parsing in the Sawzall domain-specific language [24]. Test counts words in production logs.	QPS
`openssl`	Encryption test. Several standard encryption algorithms.	QPS
`flight-search`	Flight search and pricing engine.	QPS
`books`	Book scanning perspective correction.	QPS
`page-ranking`	Signaling search relevance by analyzing the hyperlink structure of web pages [22].	QPS
`ml1`	Machine learning framework.	QPS
`ml2`	Alternative machine learning framework for large dataset analysis.	QPS
latency-sensitive, IO-centric		
`sstable`	Immutable, key-value, string-based storage for BigTable data [6].	latency
`bigtable-single`	Scalable, distributed storage [6]. Local single-machine tests.	QPS; latency
`disk`	Low-level distributed storage driver. Test replays access traces from various production services.	QPS; latency
`bigtable`	Multi-machine BigTable test. More closely representative of real usage.	QPS; latency
latency-sensitive, CPU-intensive		
`search1`	Leaf node in a search cluster [20].	latency
`search2`	Alternative search leaf node.	latency
`ml3`	Machine learning framework to group text in meaningful clusters.	QPS; latency
`ads`	Content ad targeting – matches ads with web pages based on page contents.	latency

Table I: *Benchmark names and descriptions.*

2.6 GHz, disabling frequency up-scaling (referred to as Intel TurboBoost). This is done because the additional frequency headroom is heavily dependent on idle power management, and controlled by an unknown algorithm in the processor's PCU. Enabling TurboBoost can bring significant additional variance to our measurements.

We also disable *C-state demotion* for similar reasons. Some prior work has shown that demotion can improve performance on some workloads [29]. However, in our experiments, allowing hardware to override software C-state decisions resulted in significant run-to-run variance, large enough to hide any correlation between changing software policies and overall performance results.

[Software and workloads] We characterize the sleep behavior of a variety of applications. To that end, we capture timing information for every C-state transition as requested by the kernel; as well as for the state's corresponding wake-up transition to the active state C0. This allows us to measure the time each logical core spends at a certain C-state. We also measure total system power at the power supply unit, and average it over the whole benchmark execution.

The requested C-state transitions are captured using `ktrace` [4] in the Linux kernel and collected with `perf` [7] during the regions of interest for the different benchmarks, after a necessary warm-up period. We capture every transition (as opposed to sampling), because we require two consecutive transitions to determine the residency in a given C-state. Since transitions rarely occur more often than once every $10\mu s$, and collection is appropriately buffered, the characterization process does not incur a significant overhead.

We use a variety of workloads that represent stages of large-scale Internet services. While not completely representative of any particular WSC, these applications cover different classes of workloads from large datacenters. The application names and short descriptions are provided in Table I. We split them into three groups based on their latency tolerance, and the amount of IO operations that they perform. The last column in the table lists the performance metrics used to define the quality of service (QoS) for the particular service.

While in a real-world scenario these services are deployed on a large number of machines, for practicality of our experiments we constrain them to a single server, plus appropriate load generation. The resulting load tests retain idiosyncrasies of the live services they model (e.g. burtsy and changing incoming traffic). Since single-server tests are prone to run-to-run variance, we repeat all runs at least three times, often more, based on applications' individual characteristics.

B. The typical sleep duration varies

In order to characterize idle behavior of our benchmarks, we measure the time each logical core spends at every available C-state by recording state transitions in the kernel. We later use this information to filter the benchmarks that are likely to be affected by deep sleep. Since the main purpose of this experiment is to capture the sleep patterns of applications, and not of the hardware platform, we only distinguish between "sleep" and "active" states.

Figure 7 shows the results of this characterization. The figures are in histogram format, with the x-axis bins representing the amount of time between transitioning in and out of sleep, and the y-axis showing the fraction of total execution time spent in sleep or active mode. The sum of `Active` bars adds up to the processor utilization of the particular service. In these plots, bursty applications tend to spend a larger fraction of time transitioning between states with shorter residency and show up on the left side of those histograms, while those with very long periods of activity/idleness cluster towards the right.

[Latency-insensitive applications] We first look into those

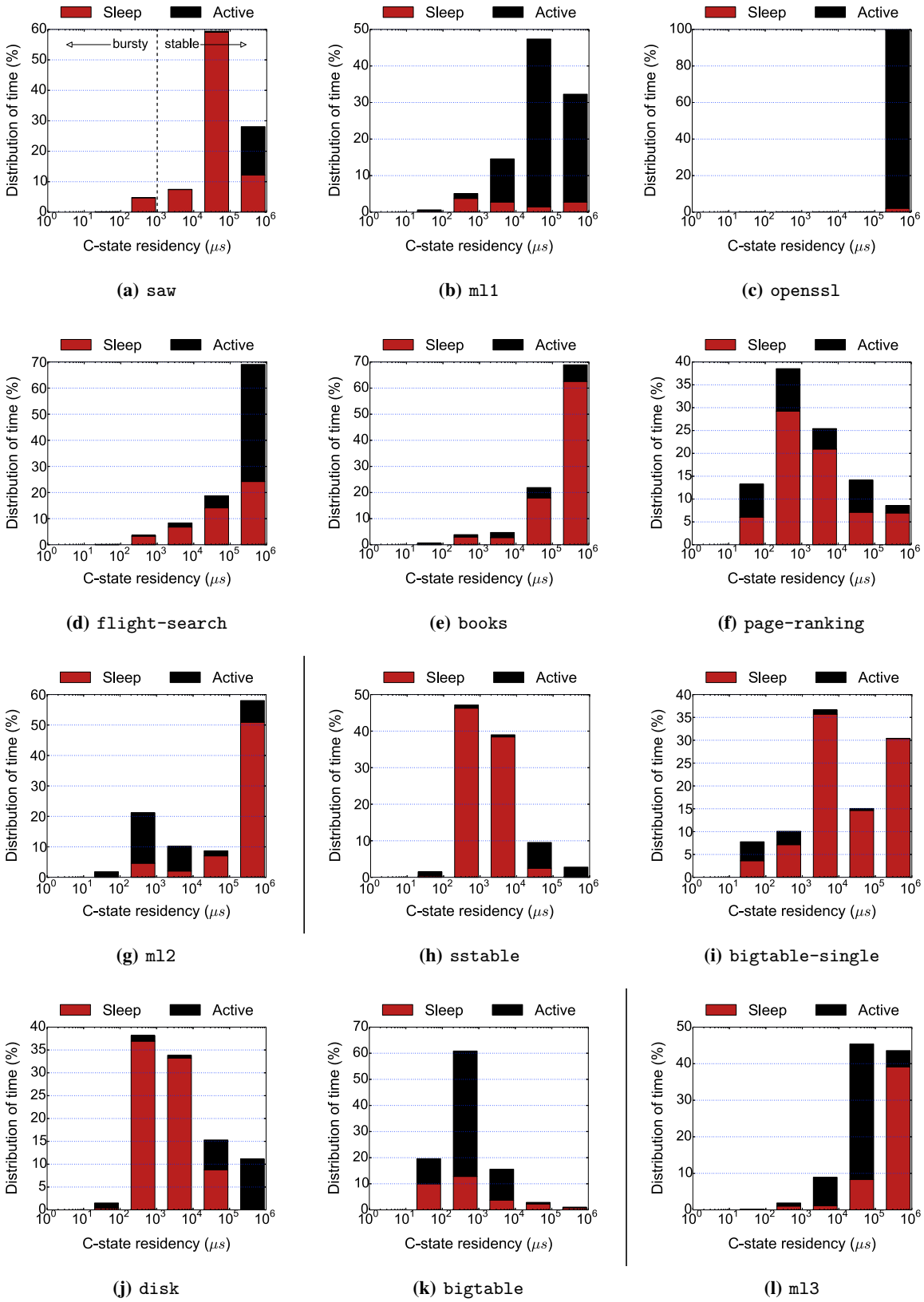
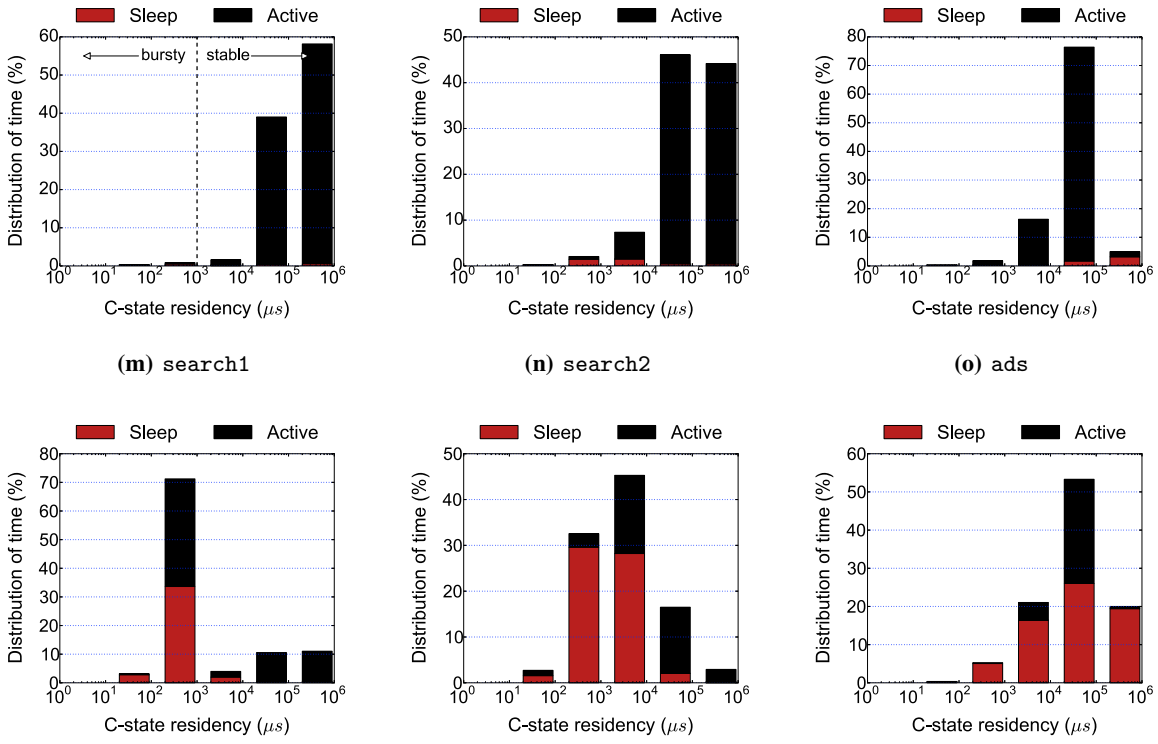


Figure 7: *Idle state distribution of WSC benchmarks. (a)-(g)* ***Latency-insensitive*** *benchmarks: the majority of sleep activity is coarse-grained, with sleep/computation lengths, larger than 1ms. (h)-(k)* ***Latency-sensitive, IO-centric*** *benchmarks: a significant amount of execution is spent requesting sleep for short periods of time (<1ms).*



(m) `search1` **(n)** `search2` **(o)** `ads`

(p) `search1` at 15% of max QPS **(q)** `search2` at 30% of max QPS **(r)** `ads` at 45% of max QPS

Figure 7: *Idle state distribution of WSC benchmarks. (l)-(o)* ***Latency-sensitive, CPU-intensive*** *benchmarks: the programs completely occupy the processor (or a subset of cores) for long periods, leaving no room for fine-grained power management. (p)-(r)* ***Latency-sensitive, CPU-intensive (low QPS)*** *benchmarks: when emulating low-activity periods, short sleeps begin to emerge.*

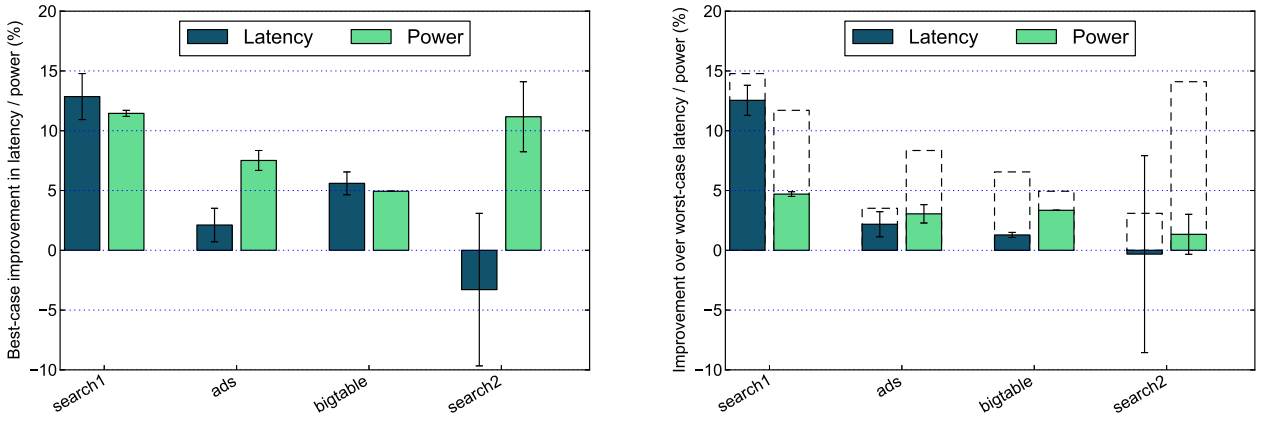
throughput-oriented benchmarks for which latency is not a relevant metric. For all of them, latency is not important because either they are implemented in a throughput-centric model (such as MapReduce); or they are off the critical path of major services (`openssl`); or the quanta of work over which latency can be defined are too large from an architecture standpoint (e.g. multi-second requests for `flight-search`). Figures 7(a)-(g) show the sleep length distribution for such applications. For most of them (except `page-ranking` and `ml2`), idle power management activity is very coarse-grained, with sleep and active periods well longer than 1ms. Since end performance is not sensitive to individual request latency in this case, the latency effects of deep sleep are irrelevant.

[Latency-sensitive IO-heavy applications] The second group of benchmarks has relatively low CPU usage and generates a large number of IO requests. These benchmarks are mostly different components of BigTable – the scalable distributed storage system at Google [6]. Figures 7(h)-(k) show the granularity of their sleep behavior. All benchmarks in this group show a significant fraction of sleeps and bursts of activity with sub-millisecond lengths. In the case that is closest to real usage, the multiple-machine BigTable test (`bigtable`, Figure 7k), short-length activity occurs for more than 80% of the execution time. This confirms the preconception of IO codes being bursty. It also implies that their latency might be significantly affected by C-state selection algorithms.

[Latency-sensitive CPU-heavy applications] Finally, Figures 7(l)-(o) show the group of CPU-bound benchmarks. They either occupy every core in the system (`search1`, `search2`, `ads`), or completely utilize just a subset of cores (the rest). Either case requires very minimal power management.

For the results so far, we assumed that services were fed with the maximum QPS sustainable by a single machine. If we restrict the rate of incoming requests, power management becomes relevant for these applications, too. In fact, this is a more realistic scenario since datacenter server CPU utilization is typically far below 100%, as seen in Section I. Figures 7(p)-(r) show the sleep length distribution for the active logical cores of `search1`, `search2` and `ads` when incoming QPS is respectively 15, 30 and 45% of the maximum sustainable by a single server. These particular numbers are chosen to represent CPU utilization in the 30-60% range. Notice that in this case, a significant part of sleep and activity periods is short in length (<1ms). For example, `search1` has sub-millisecond sleep activity for more than 70% of the execution time.

Based on these workload observations, we can select a subset of benchmarks that are likely to be affected by sleep policies, and continue the analysis with them. We choose the major services that show bursty sleep behavior – `bigtable` and the low-QPS variants of `search1`, `search2` and `ads`.



(a) `ideal-improvement` **(b)** `menu`

Figure 8: *Maximum improvement in average power and tail latency achievable by C-state selection (a). Achieving the maximum improvement for both power and latency with a single policy is unlikely, as evidenced by the realistic* `menu` *governor (b).*

C. Selecting the proper C-state matters

After identifying macro-level benchmarks that can be affected by C-state management, we can quantify the maximum effect that different governors could have on the benchmark execution. The metrics that we look at are tail request latency and average power. Tail latency here refers to 95-th percentile latency for `search1` and `ads`, and 99-th percentile for `bigtable` and `search2`[3].

In order to find the maximum improvement in both metrics, we use two trivial C-state governors – `deep` and `shallow`. As their names imply, `deep` always selects the deepest sleep state (C7) whenever a thread is idle, saving a maximal amount of power, but at the same time having the worst effect on latency due to its long residency requirement; while `shallow` always selects the least aggressive state (C1), saving little power, but also increasing latency minimally due to its fast wake-up time.[4]

Thus, the room that a C-state governor has for power savings is at most the difference between `shallow` power and `deep` power. Similarly, the room for latency improvement is given by the difference in `deep` and `shallow` latency. Because of the trade-off between power savings, and wake-up latency, a specific C-state selection algorithm would not realize these maximum gains in request latency and power simultaneously.

Figure 8a shows the amount of potential gains for the four applications. The bars labelled `Latency` measure latency reduction of `shallow` over `deep`; ones labelled `Power` measure power savings of `deep` over `shallow`. As expected, the potential power reduction from using deeper C-states are significant, averaging 8.6% across applications. Notice that this is aggregate power consumed by the server, not only by its processors. On the other hand, potential tail latency reduction varies more with the choice of workload.

For both `search1` and `bigtable` the relative improvement in tail latency is significant, and close in value to the potential for system power savings. This implies that realistic C-state selection algorithms can trade off optimizing for any of the two metrics. In the case of `ads`, the potential for improving tail latency is substantially lower. This is consistent with Figure 7r, which shows that `ads` has the smallest fraction of bursty sleeps among the four applications examined here.

Interestingly, in the case of `search2`, the average 99-th percentile latency increases when using `shallow` sleep, relative to `deep`, although the run-to-run variance is so high that it is hard to draw any major conclusions. `search2` is known to contain many application-specific optimizations for tail latencies (e.g. keeping busy-spin threads), and some of those customizations might be the reason for degraded performance when sleep is cheap (`shallow`). This effect illustrates the problem of trying to simultaneously optimize a single metric in multiple layers of the hardware/software stack.

Finally, Figure 8b shows that current prediction algorithms can realize a large fraction of the ideal gains demonstrated in Figure 8a. The solid bars show the real improvements in power/latency of the `menu` governor [23] in the Linux kernel, version 3.7, [5] compared to the best-case gains in Figure 8b (dashed lines).

While it is unlikely that the ideal gains in both latency and power can be realized simultaneously, there is still room for improvement in C-state selection. To show that, we tested a simple extension to the `menu` algorithm, wrapping it in a feedback loop that curbs too aggressive sleep (measured by the frequency of pre-mature wakeups). That simple change alone provides 5 percentage points (pp) decrease in `bigtable` tail latency, while keeping power within $\pm$1pp of `menu` results over all benchmarks.

While similar changes in C-state selection can help offset worst-case performance degradation (like the one seen in Fig-

[3]For purely practical reasons – different benchmarks report different percentiles.

[4]We found that keeping the core active in C0 by busy-spinning in the kernel has worse latency effects than `shallow`. This was counter-intuitive, but consistent. Therefore, we use `shallow` as the policy with the least power savings.

[5]In short, the `menu` algorithm estimates the expected sleep time (by fitting a simple regression model) and selects the appropriate C-state for that time based on an estimated system latency tolerance.

ure 6), even the ideal additional power gains in Figure 8a are insufficient to bridge the energy proportionality gap identified in the beginning of this paper. One obvious direction for that is deeper C-states, which save more power, but do not take longer to resume from. However, such improvements are by no means WSC-specific, and hardware designers have likely already optimized their designs for such win-win opportunities. Thus, we turn our attention to another opportunity for power savings – active management through frequency scaling.

IV. Frequency Scaling

Exploiting periods of inactivity is not the only way to save power. While an application is not sleeping – not given up a core and with instructions in flight – there are still opportunities for power reduction. These come from exploiting memory and last-level cache (LLC) stalls. Dynamic voltage and frequency scaling (DVFS) has been widely studied as a mechanism for reducing power while a core is stalled. We will investigate the potential of DVFS which exploits workload phase behavior, ideally without performance overheads and without specific workload adaptation.

While there have been many (mostly academic) proposals to exploit stall periods with active power modes [11], [12], [14], [26], [32], [34], they have not made their way to conventional operating systems. For example, the Linux kernel's support for frequency scaling is based on a different premise. Power-saving frequency governors (`ondemand`, `powersave`) use OS-reported processor utilization as a proxy for the system's latency sensitivity, and scale frequency down when processor utilization is low. While such a heuristic might be useful for the desktop and mobile domains, WSC requirements differ – services can be very sensitive to latency regardless of how high processor utilization is (e.g. `disk` in Figure 7j).

Instead, our exploration focuses on memory-bound phases of execution, during which slowing down cores does not affect end performance. It is motivated by the fact that WSC applications appear highly memory-bound on aggregate. Figure 9 illustrates that – it measures average stall cycles, that is, cycles when the cores in the system are not sleeping, but also not committing instructions. The majority of WSC applications have stall ratios comparable to those of the most memory-bound SPECint applications (`429.mcf` and `471.omnetpp`), which have also shown largest benefits from DVFS [30]. Some open-source datacenter workloads also show comparable average memory-boundness [9].

This high degree of memory-boundness suggests that WSC applications may benefit significantly from DVFS. In the rest of this section, we postulate a wishlist of requirements that a practical datacenter-ready DVFS scheme would need. We then follow up with a simple prototype implementation, showing that large gains are not realistic without either workload-specific tuning, or fine-grain hardware support.

A. Ideal requirements

An ideal active power management scheme for a WSC environment would have the following characteristics:

[Workload-agnostic] It is well-known that different workloads have varying performance headrooms for power savings through DVFS [30]. Some of the factors influencing this headroom are easy to quantify with only global machine knowledge (e.g. memory-boundness through hardware performance counters). Others are more workload-specific. Consider the `search1` example in the bottom plot of Figure 10 (for now, focus on any set of circles, which represent average request latency). At low arrival rates, the average latency to handle a request is far from the targeted SLA and aggressive power management is desirable. However, when load is high latencies get dangerously close to the SLA and performance should be prioritized.

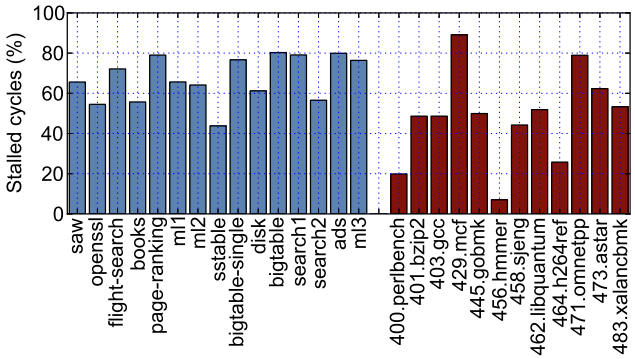


Figure 9: *WSC benchmarks are more likely to keep cores stalled than traditional SPECint applications.*

A DVFS policy that exploits such workload-specific knowledge can certainly lead to larger power savings than a generic one. But, in the WSC case, it also requires detailed understanding of (and a common software interface to) the performance properties of a potentially large number of workloads. Such complexity is significantly exacerbated in a shared cluster – with many jobs, often co-scheduled together, and having overprovisioned performance requirements [28]. A *workload-agnostic* policy, on the other hand, only uses global machine- and system-level information, without having to address such complexities. We investigate whether such a policy can also lead to power savings.

[Zero-tolerant] Once a large number of machines are involved in a tiered, high-fanout WSC service tree, individual machine performance variability is significantly amplified at the overall service level. For example, Dean and Barroso illustrate a case where the 99$^{\text{th}}$-percentile latency increases by more than $10\times$ between one random leaf node in such a tree and its parent node (which has to wait for all children to return an answer)[8]. This conventional wisdom leads datacenter operators to the conclusion that performance variability is unacceptable if it affects latency metrics. Then, unless a DVFS policy is able to monitor such latency metrics (which would require workload specificity[6]), it has to be very conservative, and only cause a minimal performance degradation, if any. We call this property *zero-tolerance*.

[Thread-granular] An ideal DVFS policy for WSC applications is also able to adjust performance states on a per-thread basis. This requirement is obviously beneficial in the shared

[6]... and possibly significant complexity. One example is the work by Raghavendra et al. [25], which uses a multi-layer coordinated control-theoretic framework that bounds the distribution of request latency.

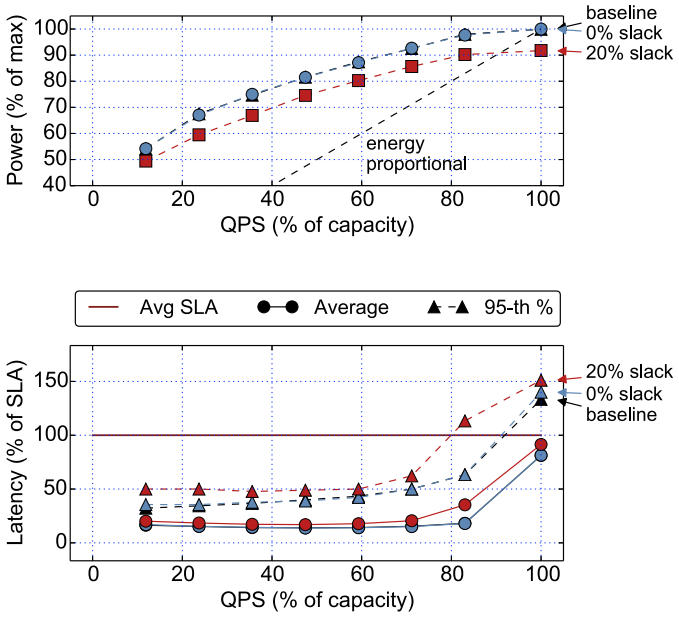


Figure 10: *DFS on* `search1` *becomes effective only after exploiting SLA slack (20% slack lines), not in the zero-tolerance case (0% slack lines).*

cluster case, where co-scheduling multiple tasks that do not allocate all cores in a machine is the norm [28]. Dedicated clusters would also benefit significantly from per-thread DVFS. They typically run query-driven workloads, where different threads execute different incoming queries, whose execution phases do not necessarily overlap.

Note that implementing thread-granular DVFS on contemporary x86 server processors is challenging. This is mostly due to the limited number of frequency and voltage domains, which only allow scaling voltage and frequency for all cores together [17]. This could change in the future when per-core voltage regulation [15] amplifies the potential power savings.

[Fine-grained] Finally, different incoming queries are uncorrelated not only among threads, but also within a single thread. This limits the duration of memory-bound phases exploited by DVFS to the latency of a single query. However, the majority of these queries are short – Meisner et al. [20] show an example of more than 90% of queries in a search leaf completing in 5ms.

Previous phase-based DVFS approaches looked into much coarser-grained intervals – e.g. 100M instructions [12] – which would contain many independent (i.e. uncorrelated) requests in a WSC application. Furthermore, in simulation, IPC variability (and, hence, potential DVFS gains) of SPEC benchmarks has been shown to decrease by more than 100× between intervals of 300 and 10K instructions [27]. This high sensitivity to granularity implies that an efficient WSC policy would benefit significantly from fine-grained control.

B. Prototyping zero-tolerance DFS

We implemented a simple DVFS prototype to estimate the importance of each of the above requirements. We use the prototype to show that it is impossible to satisfy all of them simultaneously on contemporary server hardware, and some need to be relaxed in order to realize significant power savings.

The mechanism we used for active power management is clock duty cycle modulation [33]. It allows adjusting per-core effective clock frequencies. Note that it does not adjust voltage (current server processors only have a single core voltage plane), so at best it can achieve power savings linear with frequency. We adjust frequencies within the TurboBoost range (2.6-3.3GHz at ≈200MHz increments for the system described in Section III-A).

The prototype implements a simple algorithm which empirically estimates the sensitivity of performance to frequency, similar to the one proposed by Hsu and Feng [11]. Sensitivity, or CPU-boundness, is defined by a simple linear model of normalized instructions per second (IPS) versus frequency: $Sensitivity = \frac{\Delta IPS}{IPS} \times \frac{f}{\Delta f}$, and used to predict performance at different frequencies. This is done on a per-core basis, with special attention to accounting for hyperthreads.[7]

The sensitivity model is used to predict the IPS decrease from lowering a core's frequency at every time step. If the predicted performance degradation is lower than a target "slack" parameter, the core asks for a lower frequency. A slack value of 0% represents the zero-tolerance policy from the previous section which only exploits highly memory-bound phases during which performance is insensitive to core frequency. Both more complex phase detection mechanisms [12] and frequency selection policies [32] are certainly possible (and have been summarized previously [14]), but are not the purpose of this paper. Our prototype implementation is the minimum realistic case that matches the four requirements in Section IV-A.

Figure 10 shows the results of applying this implementation to `search1` at different incoming QPS rates. The top plot displays power consumption at different values of the slack parameter, while the bottom one illustrates the effects of power savings on average and 95th-percentile latency. Most importantly, zero-tolerance DVFS does not find periods of complete memory-boundness at the 1ms granularity, and does not save power. This is evident from the virtually identical lines labelled "baseline" and "0% slack".

[Workload-controlled DVFS] Relaxing the zero-tolerance constraint has the expected effect (9% full-system power savings on average at 20% IPS slack), but at a significant increase in tail latency (the line labelled "95-th %"). Relaxing even further, a hypothetical system that includes per-core voltage control can achieve 20% full-system power savings (assuming $P \propto f^{2.4}$ [20]). For some loads, when latency is significantly below the SLA (for example, QPS <80% on Figure 10), the increase in latency is completely tolerable – that is, there are no gains from aggressively beating the SLA. Recently, Lo et al. proposed a system that exploits this property for a websearch benchmark, adjusting DVFS aggressiveness based on the difference between observed latency and the latency agreement [18]. As discussed earlier, while such per-workload systems achieve impressive power savings for ubiquitous applications, deploying them across a wide range of different workloads could be challenging.

[7] Furthermore, if there is not enough frequency variability as a result of the control algorithm, the prototype occasionally perturbs cores to the minimum and maximum frequencies, so the sensitivity model does not get stale.

[Phase granularity] Another factor that could be limiting the power savings of the tested prototype is the granularity of program phases. For our software implementation, the minimal DVFS period that causes non-negligible performance loss is 1ms. We also ran experiments with $100\mu s$ intervals, and, despite the performance penalty of triggering decisions too often, power results were virtually identical to the ones in Figure 10. This implies that memory-bound phases in applications like `search1` either do not exist, or manifest themselves on a finer granularity than $\approx$100K instructions. The latter case is more likely – `search1` is highly stalled on average (Figure 9), and simulation studies [15], [27] (albeit on different workloads) have shown that shorter phases have orders of magnitude higher variability in CPU-boundness. Confirming the existence of such ultra-fine-grained phases in WSC applications would require at least a separate simulation study; exploiting them – hardware support which does not have to pay the overheads of switching to the kernel so often.

The analysis in this section suggests that, while initially appealing, a DVFS solution that is at the same time workload-independent, zero-performance-overhead, fine-grained and per-thread does not work on current server hardware. For significant power gains, one needs to either exploit workload characteristics or additional hardware which can track extremely short-lived memory-bound phases.

V. Conclusion

With the increasing popularity of online services, intelligently managing power for warehouse-scale machines is becoming ever more relevant. We have characterized datacenter workloads, focusing on opportunities to save power at all ranges of processor utilization. We have shown that such workloads are neither completely CPU- nor IO-bound. Instead, they mix bursts of computation with short periods of sleep, emphasizing the need for comprehensive sleep state selection algorithms. We have shown that power savings are possible while not sleeping, too, but only after careful and workload-specific frequency scaling policy.

Acknowledgements

We would like to thank the anonymous reviews for their feedback. We appreciate the help of various colleagues at Google, and specifically Rama Govindaraju, Artur Klauser and Luiz Barroso for their input on versions of this manuscript.

References

[1] H. Amur *et al.*, "Idlepower: Application-aware management of processor idle states," *MMCS, in conjunction with HPDC*, 2008.

[2] L.A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, 2007.

[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, 2013.

[4] M. Bligh *et al.*, "Linux kernel debugging on google-sized clusters," in *Linux Symposium*, 2007.

[5] P. Bohrer *et al.*, "The case for power management in web servers," *Power aware computing*, 2002.

[6] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, 2008.

[7] A.C. de Melo, "The new linux "perf" tools," in *Linux Kongress*, 2010.

[8] Jeffrey Dean and Luiz André Barroso, "The tail at scale," *Commun. ACM*, 2013.

[9] Michael Ferdman *et al.*, "Clearing the clouds," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[10] J. Hamilton, "Cooperative expendable micro-slice servers (CEMS): low cost, low power servers for internet-scale services," in *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[11] Chung-Hsing Hsu and Wu-chun Feng, "Effective dynamic voltage scaling through CPU-boundedness detection," in *Workshop on Power-Aware Computer Systems*, 2004.

[12] Canturk Isci *et al.*, "Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management," in *Microarchitecture (MICRO)*, 2006.

[13] V. Janapa Reddi *et al.*, "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *Computer Architecture (ISCA)*, 2010.

[14] Stefanos Kaxiras and Margaret Martonosi, "Computer Architecture Techniques for Power-Efficiency," *Synthesis Lectures on Computer Architecture*, 2008.

[15] W. Kim *et al.*, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *High Performance Computer Architecture (HPCA)*, 2008.

[16] E. Le Sueur and G. Heiser, "Slow down or sleep, that is the question," in *USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[17] Oded Lempel, "2nd generation intel core processor family: Intel core i7, i5 and i3," in *Hot Chips*, 2011.

[18] David Lo *et al.*, "Towards energy proportionality for large-scale latency-critical workloads," in *Computer Architecture (ISCA)*, 2014.

[19] Y.H. Lu *et al.*, "Quantitative comparison of power management algorithms," in *Design, Automation and Test in Europe (DATE)*, 2000.

[20] D. Meisner *et al.*, "Power management of online data-intensive services," in *Computer Architecture (ISCA)*, 2011.

[21] ——, "Bighouse: A simulation infrastructure for data center systems," in *Performance Analysis of Systems and Software (ISPASS)*, 2012.

[22] L. Page *et al.*, "The PageRank citation ranking: bringing order to the web." 1999.

[23] V. Pallipadi *et al.*, "cpuidle: Do nothing, efficiently," in *Linux Symposium*, 2007.

[24] R. Pike *et al.*, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, 2005.

[25] R. Raghavendra *et al.*, "No power struggles: Coordinated multi-level power management for the data center," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[26] K. Rajamani *et al.*, "Power management solutions for computer systems and datacenters," in *Low Power Electronics and Design (ISLPED)*, 2008.

[27] Krishna K Rangan *et al.*, "Thread motion: fine-grained power management for multi-core systems," in *Computer Architecture (ISCA)*, 2009.

[28] Charles Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale," in *Symposium on Cloud Computing (SoCC)*, 2012.

[29] E. Rotem *et al.*, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, 2012.

[30] Vasileios Spiliopoulos *et al.*, "Green governors: A framework for Continuously Adaptive DVFS," *International Green Computing Conference and Workshops*, 2011.

[31] Daniel Wong and Murali Annavaram, "Knightshift: Scaling the Energy Proportionality Wall through Server-level Heterogeneity," in *Microarchitecture (MICRO)*, 2012.

[32] Qiang Wu *et al.*, "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," in *Microarchitecture (MICRO)*, 2005.

[33] Xiao Zhang *et al.*, "Hardware Execution Throttling for Multi-core Resource Management," in *USENIX Annual Technical Conference (USENIX ATC)*, 2009.

[34] ——, "An evaluation of per-chip nonuniform frequency scaling on multicores," in *USENIX Annual Technical Conference (USENIX ATC)*, 2010.