DreamWeaver: Architectural Support for Deep Sleep

David Meisner
meisner@umich.edu

Thomas F. Wenisch
twenisch@umich.edu

Advanced Computer Architecture Lab
University of Michigan

Abstract

Numerous data center services exhibit low average utilization leading to poor energy efficiency. Although CPU voltage and frequency scaling historically has been an effective means to scale down power with utilization, transistor scaling trends are limiting its effectiveness and the CPU is accounting for a shrinking fraction of system power. Recent research advocates the use of full-system idle low-power modes to combat energy losses, as such modes provide the deepest power savings with bounded response time impact. However, the trend towards increasing cores per die is undermining the effectiveness of these sleep modes, particularly for request-parallel data center applications, because the independent idle periods across individual cores are unlikely to align by happenstance.

We propose DreamWeaver, *architectural support to facilitate deep sleep for request-parallel applications on multicore servers. DreamWeaver comprises two elements:* Weave Scheduling, *a scheduling policy to coalesce idle and busy periods across cores to create opportunities for system-wide deep sleep; and the* Dream Processor, *a light-weight co-processor that monitors incoming network traffic and suspended work during sleep to determine when the system must wake. DreamWeaver is based on two key concepts: (1) stall execution and sleep anytime any core is unoccupied, but (2) constrain the maximum time any request may be stalled. Unlike prior scheduling approaches, DreamWeaver will preempt execution to sleep, maximizing time spent at the systems' most efficient operating point. We demonstrate that DreamWeaver can smoothly trade-off bounded, predictable increases in 99th-percentile response time for increasing power savings, and strictly dominates the savings available with voltage and frequency scaling and timeout-based request batching schemes.*

Categories and Subject Descriptors C.5.5 [*Computer System Implementation*]: Servers

General Terms Design, Measurement

Keywords power management, servers

1. Introduction

Modern data centers suffer from low energy efficiency due to endemic under-utilization [8]. The gap between average and peak load, performance isolation concerns, and redundancy all lead to low average utilization even in carefully designed data centers; conservative over-provisioning and improper sizing frequently result in even lower utilization. Low utilization leads to poor energy efficiency because current servers lack *energy proportionality*—that is, their power requirements do not scale down proportionally with utilization. Architects are seeking to improve server energy proportionality through low-power modes that conserve energy without compromising response time when load is low.

Unfortunately, the confluence of technology and software scaling trends is undermining the continued effectiveness of these low-power modes, particularly for interactive data center applications. On the one hand, device scaling trends are compromising the effectiveness of *voltage and frequency scaling* (VFS) [27, 30, 32, 51] due to the shrinking gap between nominal and threshold voltages [19], limiting both the range and leverage of voltage scaling. Recent research shows that, beyond the 45nm node, circuit delay grows disproportionately as voltage is scaled [15]. Figure 1 illustrates how the power-performance trade-off of VFS grows worse each generation. On the other hand, the prevalence of request-level parallelism in server software combined with the trend towards increasing cores per die is blunting the effectiveness of *idle low-power modes*, which place components in sleep states during periods of inactivity [3, 16, 21, 35, 37, 40, 41, 46]. In uniprocessors, the deep sleep possible with full-system idle low-power modes (e.g., PowerNap [40, 41]) can achieve energy-proportionality if mode transitions are sufficiently fast. However, for a request-parallel server application, full-system idleness rapidly vanishes as the number of cores grows—the busy and idle periods of individual cores (each serving independent requests) hardly ever align, precluding full-system sleep. Figure 2 illustrates the poor scalability of PowerNap for a Web serving workload when CPU utilization is fixed at 30% (i.e., load is scaled with the number of cores to maintain constant utilization; see Section 5.1 for methodology details).

In this paper, we propose *DreamWeaver*, architectural support to facilitate deep sleep for request-parallel applications on multicore servers. DreamWeaver comprises two elements: the *Dream Processor*, a light-weight co-processor that monitors incoming network traffic and suspended work during sleep to determine when the system must wake; and *Weave Scheduling*, a scheduling policy to coalesce idle and busy periods across cores to create opportunities for system-wide deep sleep while bounding the maximum latency increase observed by any request.

Like prior work on scheduling for sleep, DreamWeaver rests on the fundamental observation that system-wide idle periods will

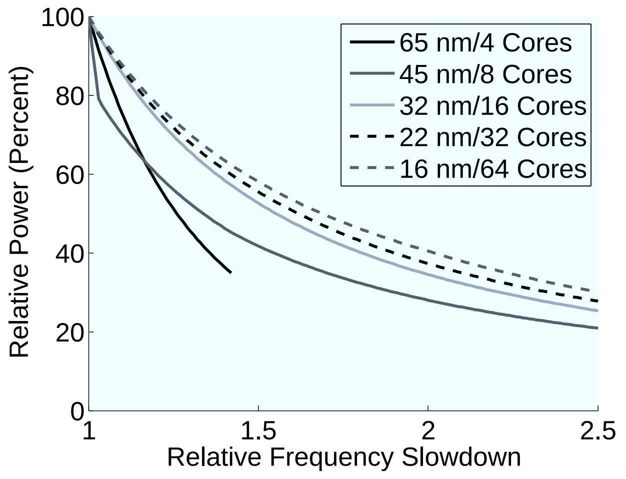


Figure 1: Voltage and frequency scaling. Future technology nodes require a disproportionate reduction in clock frequency for a given voltage reduction, breaking the classic assumption that dynamic power scales down cubically with frequency. Hence, VFS is becoming less effective: a 16nm processor requires a 2x slowdown for 50% power savings compared to 1.25x at 65nm. Data from [15].

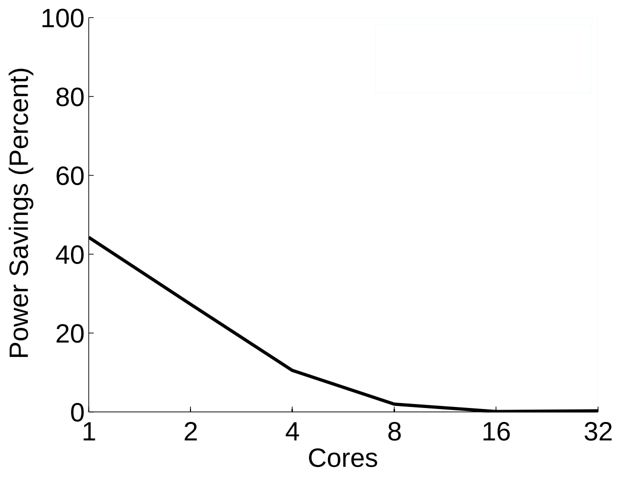


Figure 2: Full-system idle low-power mode. Power savings for a Web server at 30% utilization using a full-system idle low-power mode (e.g., PowerNap [41]). System-level idleness disappears with multicore integration, rendering coarse-grain power savings techniques ineffective.

not arise naturally in request-parallel systems; rather, per-core idle periods must be coalesced by selectively delaying and aligning requests. Prior work has proposed batching requests, using simple timeouts to control performance impact, to reduce the overhead of transitioning to/from sleep modes [5, 21, 46]. However, the fundamental flaw of timeout-based batching approaches is that they only align the *start* of a batch of requests. Since requests tend to have highly-variable long-tailed service times [25], there is nearly always a straggling request that persists past the rest of the batch, destroying the opportunity to sleep. A recent case study of request batching for Google's Web Search reveals an unappealing power-performance trade-off—even allowing a 5x increase in 95th-percentile Web search response time provides only $\sim$15% power savings for a 16-core system [42]. Naïve batching is not effective because it either (1) incurs too large an impact on response time if the batching timeout is too large, or (2) fails to align idle and busy times if the timeout is too small.

The central innovation that allows Weave Scheduling to solve the problems of batching is *preemptive sleep*; that is, DreamWeaver will interrupt and suspend in-progress work to enter deep sleep. Weave Scheduling is based on two simple policies: (1) stall execution and sleep any time that *any* core is unoccupied, but (2) constrain the maximum amount of time any request may be stalled. DreamWeaver will preempt execution to sleep when even a single core becomes idle (i.e., a request completes), provided that no active request has exhausted its allowable stall time. Thus, DreamWeaver tries to operate a server only when all cores are utilized—its most efficient operating point.

The Dream Processor is a simple microcontroller that tracks accumulated stall time for suspended requests and receives, enqueues, and counts incoming network packets during sleep. When enough packets arrive to occupy all idle cores, or when the allowable stall time for any request is exhausted, the Dream Processor wakes the system to resume execution. The Dream Processor bears similarities to the hardware support for Barely-alive Servers [6] and Somniloquy [3], but is simpler because it need not run a full TCP/IP stack.

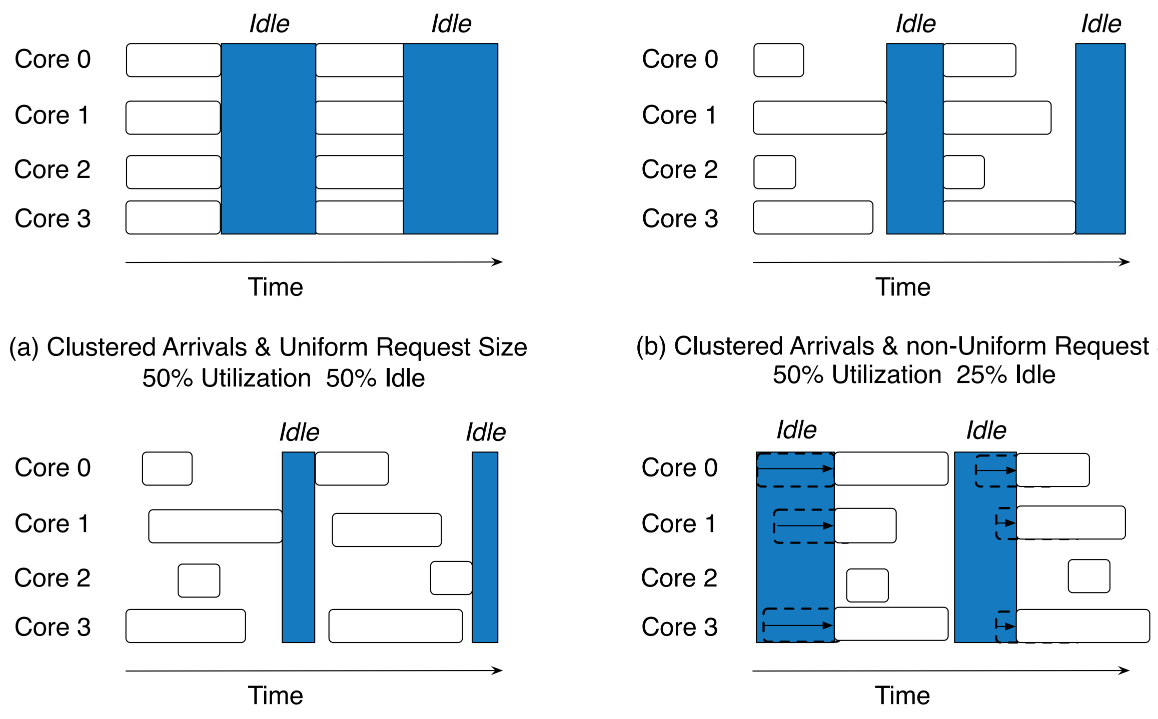
We present a two-part evaluation of DreamWeaver. First, we analyze the performance impact of Weave Scheduling using a software prototype that emulates the Dream Processor on the system's primary CPU. Through a case study of the popular open-source Solr Web search system, we show that Weave Scheduling allows an 8-core system to sleep 40% of the time when allowed a 1.5x slack on 99th-percentile response time. We also use our prototype to validate the performance predictions of our simulation model. Second, we evaluate the power savings potential of DreamWeaver, examine its scalability, and contrast it with other power management approaches using *Stochastic Queuing Simulation (SQS)* [44], a validated methodology for rapidly simulating the power-performance behavior of data center workloads. Our simulation study demonstrates that DreamWeaver dominates the power-performance trade-offs available from either VFS or batch scheduling on systems with up to 32 cores on four data center workloads, including Google Web search.

2. Background

We begin with a brief overview of the challenges that make power management for request-parallel data center workloads difficult. Then, we review related work on server power management.

2.1 Power Management Challenges

Power management for data center workloads is challenging because many of these workloads are latency-sensitive. Moreover, it is growing more challenging with multicore scaling [28]. Servers must meet strict *service level agreements* (SLAs), which prescribe per-request latency targets that must be met to prevent stringent penalties. SLAs are typically based on the 99th-percentile (or simi-



(a) Clustered Arrivals & Uniform Request Size
50% Utilization 50% Idle

(b) Clustered Arrivals & non-Uniform Request Size
50% Utilization 25% Idle

(c) Non-Clustered Arrivals & Non-Uniform Request Size
50% Utilization 10% Idle

(d) Batch Scheduling for Idleness
50% Utilization 30% Idle

Figure 3: Full-system idleness varies widely as a function of arrival and request size patterns. As seen in (a), a workload with clustered arrivals (high coefficient of variation, or "C_v") and uniform request sizes (low C_v) maximizes idleness. Notice that in this case, core-level idleness and full-system idleness are the same (50%). If requests are non-uniform in size, as in (b), full-system idleness decreases (25%) although core-level idleness does not (50%). Similarly in (c), with both non-clustered request arrivals (low C_v) and non-uniform request sizes (high C_v), full-system idleness is significantly decreased. One technique to mitigate these effects is batching, shown in (d), which increases request latency and creates artificial idle periods.

lar high percentile) latency, not the mean. Meeting this requirement is complicated by workloads with long-tailed and unpredictable service times [25]. The majority of existing literature (particularly works that have focused on power management) has concentrated on the average latency of server systems; we instead set targets for 99th-percentile latency, but our results generalize to other high quantiles.

Furthermore, data center workloads are often highly variable. For instance, for Web serving, the difference between the mean and 99th-percentile latency is over a factor of four. This constraint means designers must take care: a change that has a small impact on mean response time may have a large effect on the 99th percentile.

2.2 Related Work

Previous literature has demonstrated that reducing power at low utilization is critical to increasing server efficiency [8, 41]. System designers use numerous approaches to improve energy efficiency of under-utilized systems. These approaches fall into three broad classes: cluster-grain approaches, active low-power modes, and idle low-power modes. Though our study is focused on idle low-power modes, we briefly discuss the merits and challenges of each.

Cluster-grain approaches to energy-proportionality. The cause of poor efficiency in servers is rooted in their low utilization and lack of energy-proportional components. Techniques such as dynamic cluster resizing and load dispatching [4, 12–14, 26, 33, 47] or server consolidation and virtual machine migration [10, 49] seek to increase average server utilization, which improves efficiency on non-energy-proportional hardware. By moving the work of multiple server onto a single machine, fixed power and capital costs may be amortized.

Though this approach is effective for many workloads, there are several data center workload paradigms for which consolidation/migration is inapplicable. For many workloads of increasing importance (e.g., Web search, MapReduce), large data sets are distributed over many servers and the servers must remain powered to keep data accessible in main memory or on local disks [39, 42]. In the case of Web search, clusters are sized based on memory capacity and latency constraints rather than throughput—the entire cluster must remain active to serve even a single search request with acceptable latency [42]. Task migration typically operates over too coarse time scales (minutes) to respond rapidly to unanticipated load. In latency-sensitive interactive workloads, compacting multiple services onto the same machine may make service increasingly vulnerable to the effects of variance (e.g., traffic spikes). Low utilization is common for this exact reason; well-designed services are intentionally operated at 20-50% utilization to ensure performance robustness despite variable load [8].

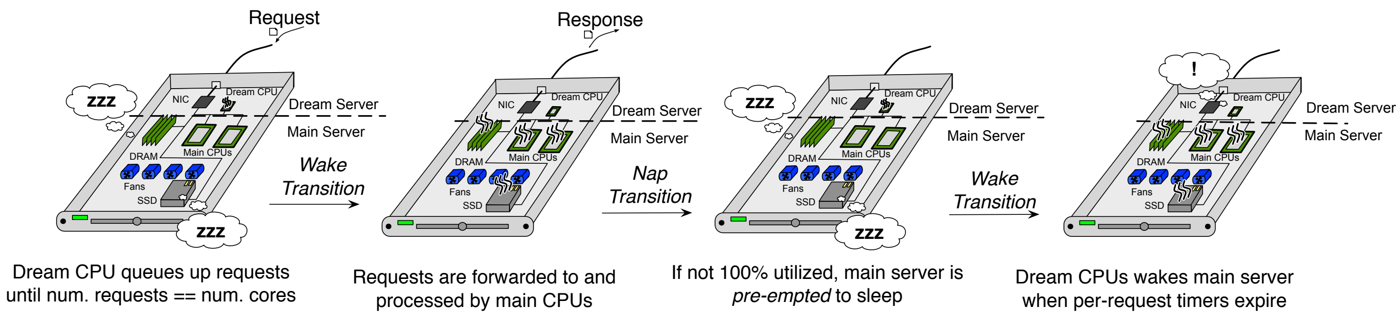

Request
Response
zzz
NIC
Dream CPU
Dream Server
Main Server
DRAM
Main CPUs
Fans
SSD
Wake Transition
Nap Transition
Wake Transition
!
Dream CPU queues up requests until num. requests == num. cores
Requests are forwarded to and processed by main CPUs
If not 100% utilized, main server is *pre-empted* to sleep
Dream CPUs wakes main server when per-request timers expire

Figure 4: DreamWeaver. The DreamWeaver system is composed of a main server with PowerNap capabilities [41] and Dream Processor that implements Weave Scheduling. The Dream Processor is a modest microcontroller that is isolated from the power state of the rest of a server. It is responsible for modulating the power state of the main system, buffering incoming requests from the network, and tracking any delay of requests while in the nap state. The nap processor resembles hardware such as in Barely-alive Servers [6] or Somniloquy [3], but requires far less processing power because it does not directly process or respond to packets.

Server-level active low-power modes. Many hardware devices offer *active low-power modes*, which trade reduced performance for power savings while a device continues to operate. Active low-power modes (e.g., VFS) improve energy efficiency if they provide superlinear power savings for linear slowdown. VFS is well-studied for reducing CPU power [27, 30, 32, 38, 48, 51]. Unfortunately, the effectiveness of VFS is shrinking with technology scaling (see Figure 1) as decreases in voltage result in increasingly disproportionate increases in circuit delay [15]. Active low-power modes have also been proposed for disks [11, 24]. Whereas active low-power modes are largely orthogonal to our study, we compare the effectiveness of DreamWeaver to voltage and frequency scaling to provide a frame of reference for our results.

Sever-level idle low-power modes. Many devices also offer *idle low-power modes*, which provide even greater power savings than the most aggressive active low-power modes [22, 41]. One of the most attractive properties of idle low-power modes is that they offer fixed latency penalties. These modes are characterized by their transition time T_{tr}: the time to enter or leave the low-power mode. When T_{tr} is small relative to the average service time, requests only experience a slight delay [41]. Whereas active low-power modes can increase the 99th-percentile response time significantly, small T_{tr} minimally alters it.

The deepest component energy savings can typically be extracted only when a component is idle. Idle low-power modes have been explored in processors [39, 45], memory [17, 18, 37], network interfaces [3], and disk [11]. Unfortunately, current per-core power modes (e.g. ACPI C-states or "core parking") save less than 1/Nth of the power in an N core processor because support circuitry (e.g., last-level caches, integrated memory controllers) remain powered to serve the remaining active cores [29]. The Intel Nehalem processor provides a socket-grained idle low-power mode through its "Package C6" power state, which disables some of this circuitry, but the incremental power savings over the per-core sleep modes is small. Nevertheless, processors typically consume only 20-30% of a server's power budget, while 70% of power is dissipated in other devices (e.g., memory, disks, etc.) [41]. PowerNap [40, 41] proposes to use full-system sleep to save energy during system idle periods, however, the prior study did not consider the implications of multicore scaling on idleness.

Alternatively, some authors have proposed scheduling background tasks or other work during primary-application idle periods [20, 23]; these mechanisms are orthogonal to our study.

2.3 Scheduling for energy efficiency

Idleness depends heavily on the workload running on a server. The amount of idleness observed at individual cores and over the system as a whole can differ drastically depending on workload characteristics. We illustrate the factors affecting idleness in Figure 3 for a four core system with a fixed utilization. If all requests arrive at the server at the same time and are of equal length (Figure 3(a)), all core-level idle periods align. Only in this degenerate case are core-level and system-level idleness equal. In Figure 3(b), the timing of request arrivals remain the same, but the request lengths vary; the amount of system-level idleness is reduced. Additionally varying request arrival timing, in Figure 3(c), further reduces system-level idleness. Finally, Figure 3(d) illustrates the effect of batch scheduling; though it is not possible to change request sizes, it is possible to alter the effective arrival pattern by delaying requests.

Elnozahy et al investigated using request batching (similar to what is shown in Figure 3(d)) to leverage idle low-power modes in uniprocessors [21]. DreamWeaver's contribution differs in three regards. First, DreamWeaver's request alignment algorithm is different; it is based on per-request stall constraints, it initiates service immediately once sufficient requests have arrived to fill all processing slots, and it suspends/resumes in-progress execution. In contrast, Elnozahy et al implement a simpler algorithm: requests are accumulated during a predefined batching window and released upon timeout. Furthermore, rather than imposing a per-request latency constraint, their approach tunes the timeout period over coarse intervals in a control-loop. Second, we consider the consequences of idleness and request skew for multicore systems; the previous study seeks to reduce transition penalties in a uniprocessor. Finally, the previous study was concerned only with processor power; one of our key observations is that non-CPU power management is critical to achieve energy-proportionality. A recent case study of request batching for Google's Web Search concludes that it provides an unappealing power-performance trade-off [42].

Several other prior scheduling mechanisms bear similarities to DreamWeaver in that they seek to align or construct batches of requests, for example, ecoDB [34] and cohort scheduling [36]. EcoDB introduces two techniques: using DVFS and delaying requests to batch SQL requests with common operators that can be amortized. Cohort scheduling seeks to maximize performance by scheduling similar stages of multiple requests together to increase the effectiveness of data caching. In contrast, DreamWeaver introduces delays to increase usable idleness; it is agnostic of the underlying software (i.e., the requests need not be similar) and depends only on statistical effects. All of these techniques take advantage of

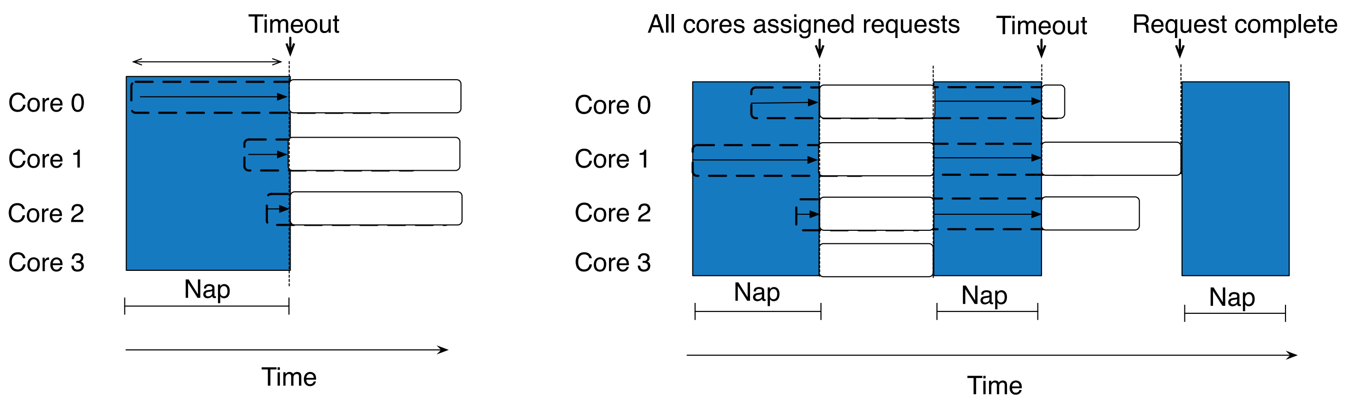


Figure 5: Weave Scheduling example. Weave Scheduling is an algorithm for intelligently delaying, preempting, and executing requests to maximize the fraction of time a multicore CPU is fully utilized while providing an upper-bound on per-request latency increase. The example on the left demonstrates an individual request exceeding its maximum delay. Although the system is underutilized, the system transitions out of the nap state because Core 0's request experienced a timeout. On the right, we demonstrate an example of preemption. At first, requests are delayed until all cores can be occupied and then the system transitions out of the nap state. The system remains active until Core 3's request finishes and then the system preempts the unfinished requests. Finally, Core 1's request experiences a timeout and the system resumes to meet the maximum delay constraint.

the insight that handling requests as they arrive may not be optimal for performance or energy efficiency.

3. DreamWeaver

DreamWeaver increases usable idleness by batching requests to maximize server utilization whenever it is active while ensuring that each request incurs at most a bounded delay. Our approach builds on PowerNap [40, 41], which allows a server to transition rapidly in and out of an ultra-low power nap state. PowerNap places an entire system (including memory, motherboard components, and peripherals) in an application-software–transparent deep sleep state during idle periods. PowerNap reduces power consumption by up to 95% while sleeping. Though PowerNap already approaches energy-proportionality (energy consumption proportional to utilization) in uniprocessor servers, it requires full-system idleness. As shown in Figure 2, there is little, if any, opportunity for PowerNap in lightly- to moderately-utilized large-scale multicore servers.

3.1 Hardware mechanisms: the Dream Processor

The baseline PowerNap design requires a sever (and, hence, all of its components) to transition between active and idle states in millisecond timeframes. Furthermore, it requires an operating system without a periodic timer tick, and software/hardware support to schedule wake-up in response to software timer expiration. The original PowerNap study [41] outlines these software and hardware requirements in greater detail, we focus here on new requirements.

DreamWeaver presents several additional implementation challenges. The largest challenge lies in handling the expiration of request timeouts and arrival of new work while the system is napping. Under PowerNap, handling the arrival of new work is simple—the system wakes up. Under DreamWeaver, however, the system must keep track of the number of idle cores and be able to defer arriving requests (while tracking their accumulated delay) without waking. A second challenge lies in preempting in-process execution to enter the nap state.

DreamWeaver addresses these requirements through the addition of a dedicated *Dream Processor* that coordinates with the operating system on the main processor(s) to manage sleep and wake transitions. The functionality of the Dream Processor is summarized in Figure 4. During operation, the primary OS uses the Dream Processor to track the assignment of requests to cores and the accumulated delay of each request. The primary OS notifies the Dream Processor each time a new request is created (e.g., because an incoming packet is processed), assigned one or more cores for execution, or completes. When a core becomes idle, the primary OS is responsible for preempting work on all cores and triggering a sleep transition. Upon transition, the primary OS passes the Dream Processor a list of active requests, the accumulated delay for each and the number of idle cores. Then, it hands control to the Dream Processor, which tracks the passage of time and continues to operate the network interface, while tracking the accumulated delay for each request. Using its own hardware timers, the Dream Processor wakes the system when any request's accumulated delay reaches the threshold.

Network packets that arrive during nap are received and queued by the Dream Processor. When the system wakes, the Dream Processor returns the accumulated delay of each request to the primary OS and then replays the delivery of queued packets through the network interface. Each arriving packet is assumed to create a new single-core request, and the Dream Processor wakes the system when the number of queued packets equals the number of idle cores. Hence, the number of queued packets is bounded by the number of cores and never grows large. While the Dream Processor could operate a complete TCP/IP stack, this is not necessary; only a layer-2 interface is needed to receive and log arriving packets. A more sophisticated Dream Processor may be able to identify packets that require minimal processing or can be deferred (e.g., TCP ack packets).

Since the Dream Processor operates continuously (including in the nap state), it is essential that its power requirements are low. Hence, it operates using its own dedicated memory and does not access any system peripherals except the network interface. The Dream and main processors communicate through programmed I/O (i.e., no shared memory). As the Dream Processor performs relatively simple tasks, it can be implemented with a low-power microcontroller. Several recent studies have evaluated auxiliary processors and network interfaces with similar capabilities, for example, Barely-alive Servers [6] and Somniloquy [3]. Our Dream Processor also is similar, albeit with considerably simpler requirements, to the service processors in existing IBM and HP server systems. These service processors perform a variety of environmental, temperature, and performance monitoring, maintenance, failure logging, and system management functions. They usually operate a complete TCP/IP stack to provide integrated lights-out functionality in contrast to the simple layer-2 and programmed I/O interfaces of the Dream Processor.

3.2 Weave Scheduling

Weave Scheduling improves energy efficiency by aligning service and idle times as much as possible, such that all cores are simultaneously active or idle. Our key intuition is to *stall service any time that any cores are unoccupied*, even if that means preempting requests that are in progress to go to sleep. During stalls, we invoke PowerNap to save energy. By allowing execution only when all cores are busy, DreamWeaver maximizes energy efficiency—the power required to operate the system is amortized over the maximum amount of concurrent work. If strictly implemented, this policy guarantees that all core-grain idleness is exploited at the system level.

Of course, such an approach could result in massive (potentially unbounded) increases in response time. To limit the impact on response time, we *constrain the maximum amount of time any request may be stalled*. Hence, if not all cores are occupied, but at least one request in the system has accrued its maximum allowable stall time, we resume service and allow all cores to execute until that request completes. When service proceeds due to exhausting a request's allowable stall time, some core-grain idleness is lost (cannot be used to conserve energy). However, the maximum stall threshold bounds the response time increase from Weave Scheduling; we simply choose this bound based on the amount of slack available between the current 99th-percentile response time and that required by the SLA.

We illustrate the operation of Weave Scheduling in a 4-core system in Figure 5. On the left, we demonstrate the stall threshold mechanism. Service is initially stalled and the system is napping. Then, the request at Core 0 reaches its maximum allowable delay (timeout). Request processing then resumes and all current requests are released (even though Core 3 is idle) until the request at Core 0 finishes. Subsequently, the system will again stall and nap. On the right, we demonstrate the behavior when all cores become occupied. The system is initially stalled and napping. Then a request arrives at Core 3, occupying all cores and starting service. As soon as the first request completes (at Core 3), the system again stalls and returns to nap. Shortly after, the request at Core 1 reaches timeout. Hence, service resumes and continues until the request at Core 1 is finished.

4. Prototype Evaluation

We evaluate DreamWeaver in two steps. In this section, we investigate its performance impact with a proof-of-concept prototype. We use these results to validate the performance predictions of our simulation approach. In Section 5, we use simulation to explore DreamWeaver's impact on power consumption.

4.1 Methodology

To assess the performance impact of DreamWeaver, we have constructed a software prototype that implements Weave Scheduling. Our prototype models the functionality of the Dream Processor with a software proxy that executes on the main CPU. Because servers with PowerNap capabilities are not currently commercially available, we cannot directly measure power savings from DreamWeaver; we defer this investigation to our simulation-based studies.

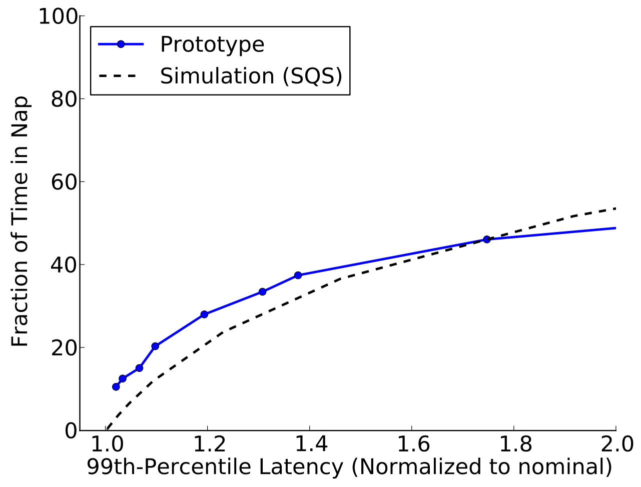


Figure 6: DreamWeaver prototype vs. simulation validation. This figure illustrates the accuracy of our simulation environment to predict the fraction of time a DreamWeaver server spends in the nap state. As we increase the predefined maximum delay a request can experience, the available full-system idleness increases as a function of 99th-percentile latency. One can see that the simulation ("Simulation") makes reasonable estimates of our prototype system ("Prototype").

We study the impact of DreamWeaver on a Web Search system modeled after that studied in [42] using the Solr Web Search platform. Solr is a full-featured Web indexing and search system used in production by many enterprises to add local search capability to their Web sites. Our system serves a Web index of the Wikipedia site [2], which we query using the AOL query set [1]. We believe this is the best approximation of a commercial Web search system that can be achieved using open source tools without access to proprietary binaries and data.

We emulate the behavior of the Dream Processor through a software proxy. Instead of sending queries directly to the Solr system, queries are sent to the proxy, which controls their admission to Solr. The software logic in the proxy mirrors that of the Dream Processor, however, the code runs on a core of the main CPU rather than a dedicated Dream Processor. The proxy tracks the number of active queries in the system and the accumulated delay of each query. When the system is awake, queries are passed immediately from the proxy to Solr via TCP/IP. We have confirmed that the addition of the proxy has negligible impact on the response time or throughput of Solr. When the system emulates nap, the proxy buffers incoming packets and uses timers to monitor accumulated delay. We implement the preemptive sleep called for by Weave Scheduling using Linux's existing process suspend capabilities; whenever the system enters the nap state, a suspend signal is sent to all Solr processes. The proxy assumes that all incoming TCP/IP packets correspond to a new query for the purposes of determining when to awake from nap. A Resume signal is sent to Solr upon a wake transition. Transition delays are emulated through busy waits in the proxy.

4.2 Results

We now present the results of our prototype system and compare it to our simulation infrastructure used in Section 5. Specifically we compare the sleep-latency tradeoff of the two evaluation methodologies. In Figure 6 we provide the time spent in sleep as a function of 99th-percentile latency as provided by our prototype ("Imple-

Table 1: Server Power Model. Based on data from Google [9] and HP [50].

Power (% of Peak)	CPU	Memory	Disk	Other
Max	40%	35%	10%	15%
Idle	15%	25%	9%	10%

mentation") and our simulation infrastructure ("SQS"). When allowed a 1.5x slack on 99th-percentile response time, DreamWeaver allows the prototype system to sleep 40% of the time. In contrast, the opportunity to sleep with PowerNap alone is negligible. Furthermore, the figure clearly demonstrates that the performance predictions of our simulation model agree well with the actual behavior of the prototype DreamWeaver system.

5. Power Savings Evaluation

While our prototype allows us to validate the performance impacts of DreamWeaver, the lack of PowerNap support in existing servers precludes measuring power savings. In this section, we use simulation to investigate DreamWeaver's power-performance impact on a variety of workloads over several multicore server generations.

5.1 Methodology

We evaluate the power savings potential of DreamWeaver and contrast it with other power management approaches using the BigHouse simulator [44]. This simulator leverages *Stochastic Queuing Simulation (SQS)*, a validated methodology for rapidly simulating the power-performance behavior of data center workloads. SQS is a framework for stochastic discrete-time simulation of a generalized system of queuing models driven by empirical profiles of a target workload. In SQS, empirical interarrival and service distributions are collected from measurements of real systems at fine time-granularity. Using these distributions, synthetic arrival/service traces are generated and fed to a discrete-event simulation of a G/G/k queuing system that models server active and idle low-power modes through state-dependent service rates. SQS allows real server workloads to be characterized on one physical system, but then studied in a different context, for example on a system with vastly more cores (by varying k), or at different levels of load (by scaling the interarrival distribution). Furthermore, SQS enables analysis of queuing systems that are analytically intractable. Performance measures (e.g., 99th-percentile response time) are obtained by sampling the output of the simulation until each reaches a normalized half-width 95% confidence interval of 5%. Further details of the design and statistical methods used in SQS appear in [43, 44]. SQS has been previously used to model Google's Web search application [42], and its latency and throughput predictions have been validated against a production Web search cluster.

SQS does not model the details of what active system components are doing (e.g., which instructions are executing, what memory locations are accessed). However, these are not relevant to understanding idle periods and scheduling effects, hence, more detailed simulation models (e.g., instruction or cycle-accurate simulators) are unnecessary.

Low-Power Modes. Our power model assumptions for the system (Table 1) are based on the breakdowns from Google [9] and HP [50] and published characteristics of Intel Nehalem [29]. We model idle low-power modes through exceptional first service; that is, when a system is napping, the service rate of the corresponding server in the queuing model is set to zero and a latency penalty is incurred when the first request is serviced after idle.

As a point of comparison, we also model voltage and frequency scaling (VFS), by varying the service rate. We map core count to a corresponding technology node and power-performance scaling curve as shown in Figure 1, using data from [15]. We explore a range of power-performance settings by exhaustively sweeping static frequency and corresponding voltage settings. It is important to note that we optimistically allow the system to pick any arbitrary voltage/frequency setting although most processors only provide a few discrete points. Our VFS results should be viewed as an estimate of the potential of voltage and frequency scaling, they do not model any particular policy for selecting voltages. It is possible that a scheme that dynamically tunes frequency could improve slightly over our VFS estimates, though we expect such gains to be minimal because our experiments operate a server at a steady utilization.

Workloads. We collect empirical interarrival and service distributions from several production server workloads. These distributions are derived from week-long traces of departmental servers and from the Google Web Search test cluster as described in [42, 44]. Table 2 describes each workload and summarizes important interarrival and service statistics: for each distribution, we include the mean (Avg.), standard deviation (σ), and coefficient of variation (C_v).

Using these empirical distributions, we can independently scale arrival and service rates (i.e., without changing the distributions' shapes) to simulate higher and lower utilization. Moreover, we can replay stochastically-generated request sequences against arbitrary queuing models, including models for multicore chips far larger than are built today.

5.2 Results

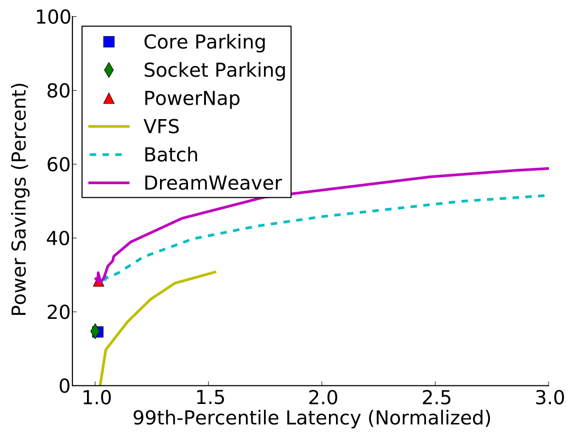
Power-latency tradeoff compared to other techniques. We first contrast DreamWeaver with alternative power management approaches. We consider systems assuming a fixed throughput and evaluate the latency-power tradeoffs. It is important to note that nearly any power savings techniques will undoubtedly increase latency. If latency-at-any-cost is paramount, the best system design may discard power management. Instead, the question we pose is: Given an allowable threshold to increase 99th-percentile response time, what is the best way to save energy and how much can we save?

We contrast our mechanism ("DreamWeaver") with four other power management approaches. First, we compare against PowerNap as proposed in [41] ("PowerNap"). We initially assume an aggressive transition latency of 100 μs for both PowerNap and DreamWeaver because the goal of this work is to evaluate the ability of these techniques to exploit multicore idleness, not to mitigate transition latencies. We examine sensitivity to longer transition latencies below. Second, we compare it against Core Parking ("Core Parking"). We optimistically assume that cores can be parked during all core-grain idle time, ignoring transition penalties. Under this assumption, Core Parking subsumes approaches that consolidate tasks onto fewer cores to reshape core-grain idle periods (e.g., to lengthen them). Furthermore, we compare against a timeout-based batching mechanism ("Batch") based on the approach of Elnozahy et al [21]. Finally, we compare to voltage/frequency scaling ("VFS"), as described in Section 5.1.

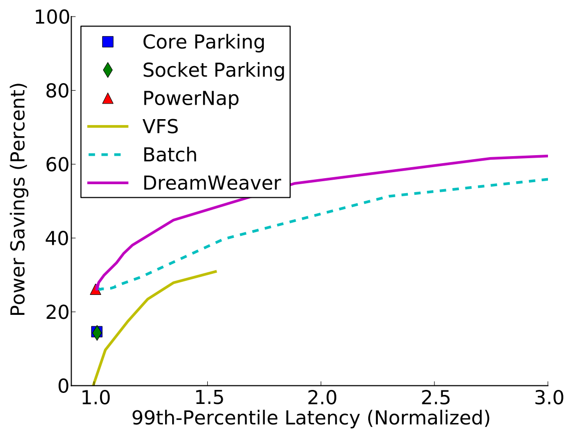
4-Core Server. We first show the results for a server with four cores. The relative power savings of each of the considered power savings techniques is shown in Figure 7. Core Parking, Socket Parking, and PowerNap each yield only a single latency-performance point per system configuration and workload. In contrast, DreamWeaver, Batch, and VFS each produce a range of latency-power options. We present each of the four workloads with load scaled such that the server operates at 30% average utilization. The horizontal axis on each graph shows 99th-percentile latency

Table 2: Workload Characteristics.

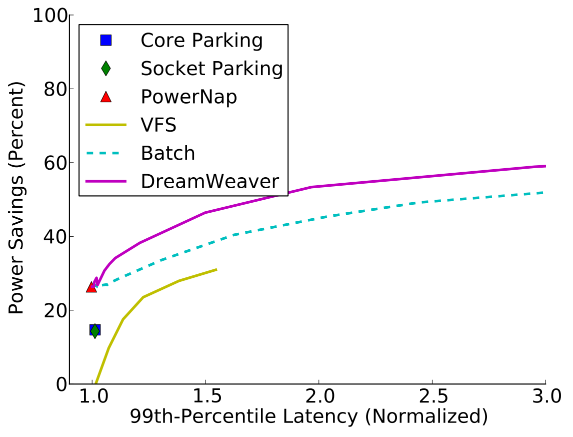
Workload	Interarrival			Service			Description
	Avg.	σ	C_v	Avg.	σ	C_v	
DNS	1.1s	1.2s	1.1	194ms	198ms	1.0	DNS and DHCP server
Mail	206ms	397ms	1.9	92ms	335ms	3.6	POP and SMTP servers
Search	319μs	376μs	1.2	4.2ms	4.8ms	1.1	Google Web Search [42]
Apache	186ms	380ms	2.0	75ms	263ms	3.4	Web server



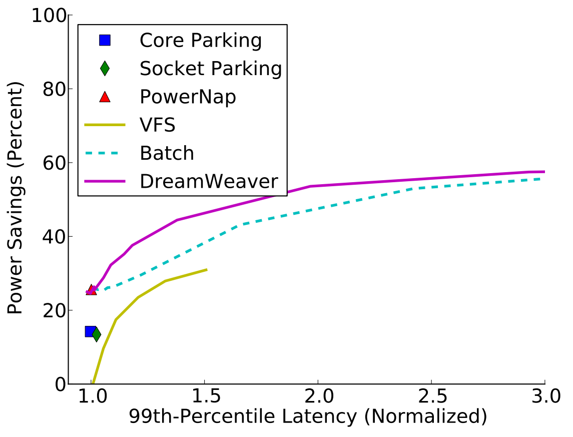
(a) Apache



(b) DNS

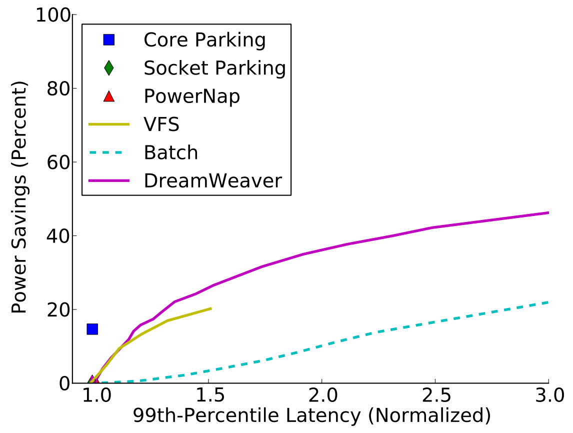


(c) Mail

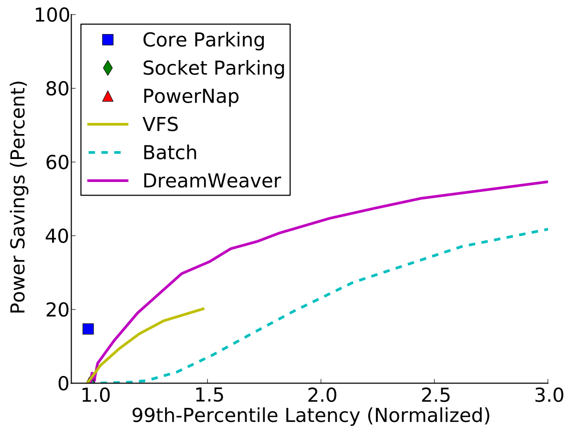


(d) Search

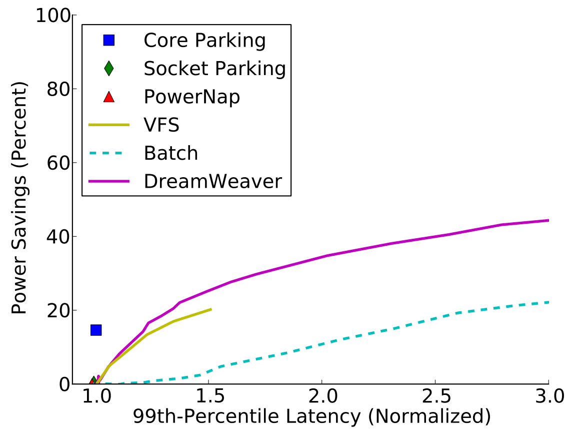
Figure 7: Comparison of power savings for 4-core system. This figure demonstrates the power savings of low-power modes as a function of 99th-percentile latency for a 4 core server. Per-core power gating ("Core Parking") can save a modest amount of power for a small latency increase because its transition latency is low, however it cannot reduce power in non-core components (e.g., last-level caches or the memory system). Attempting to put an entire socket into a low-power sleep mode ("Socket Parking") provides roughly the same benefit as per-core power gating; less idleness is available at socket granularity but this reduction is offset by the increase in power savings. Using a full-system low-power mode such as PowerNap ("PowerNap") exploits as much idle time as socket parking, but saves significantly more power. Processor voltage and frequency scaling ("VFS") provides significant savings for the CPU, but does not alter non-processor power (e.g., the memory system, I/O buses etc.). Greater power savings can be achieved by using a full-system idle low-power mode. Creating idleness by batching ("Batch"), provides even more power savings than PowerNap in exchange for increased latency due to delaying requests. An even better power-latency tradeoff is achieved by DreamWeaver ("DreamWeaver"), because of its hardware support to track requests and intelligent scheduling.



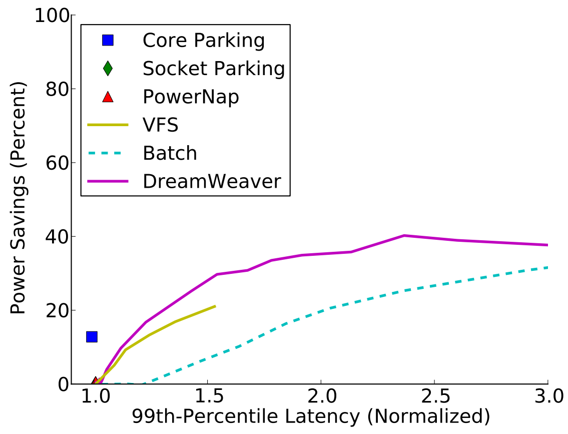
(a) Apache



(b) DNS



(c) Mail



(d) Search

Figure 8: Comparison of power savings for 32-core system. Most low-power modes are less effective when moving to future systems (smaller transistor feature size and higher core count) because voltage scaling requires greater frequency reductions and coarse-grain idleness is more difficult to capture (See Figures 1 and 2). Per-core power gating ("Core Parking") does not rely on coarse-grain idleness and is just as effective as for a 4 core system (see Figure 6). However, both Socket Parking ("Socket Parking") and PowerNap ("PowerNap") require that all cores are simultaneously idle. At 32 cores, the system is almost never entirely idle and there is no opportunity to use these low-power modes. Voltage and frequency scaling ("VFS") saves less power because it requires a larger slowdown for a given voltage reduction. Batching ("Batch") at 32 cores is quite ineffective requiring inordinate latency increases to save appreciable power. DreamWeaver's effectiveness is reduced at 32 cores ("DreamWeaver"), but generally provides the greatest power savings for all but the tightest latency constraints.

normalized to the nominal latency (i.e., no power management). As discussed in Section 2, we focus our evaluation on 99th-percentile latencies as these are the more difficult constraints to meet; DreamWeaver's impact on mean latency follows the same trends. The vertical axis shows power savings relative to a nominal system without any of these power management features (but with clock gating on HLT instructions as in Nehalem); higher values indicate greater power savings.

Over the range from nominal to a 2x increase in 99th-percentile latency, DreamWeaver strictly dominates the other power management techniques. When the user configures DreamWeaver to allow no additional performance degradation on the 99th-percentile latency (i.e., a timeout of zero), DreamWeaver converges to PowerNap as expected; with a 2x increase in latency, DreamWeaver can offer roughly 25% better power savings than PowerNap and nearly 30% more than VFS. Also important, Batching can provide substantial power savings, and provides a roughly linear trade-off of 99th-percentile latency vs. power. However, its range of latency-power settings, while also better than VFS, is strictly inferior to DreamWeaver.

32-Core Server. Next, we consider a server with 32 cores. The results are presented in Figure 8 and parallel the previous study. First, as expected, we highlight that PowerNap is ineffective. Because there is no naturally occurring full-system idleness, there is no opportunity for PowerNap and it saves no power (nor incurs any latency). Next, we observe that Core Parking is still effective, but as before only provides power savings of less than 20%. A striking difference is that, unlike our four core study, Batch has become largely ineffective. The latency-power tradeoff for this technique is unattractive; it saves far less power than Core Parking,

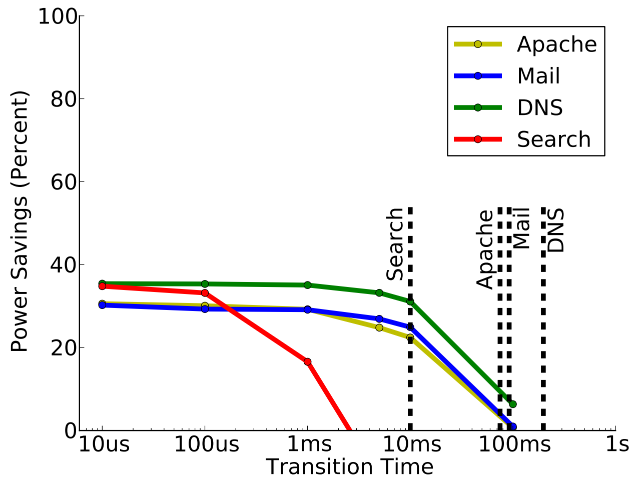


Figure 9: Sensitivity to transition time. DreamWeaver is less effective as the transition time in and out of PowerNap increases. Dotted vertical lines denote the average service time of each workload. The majority of power savings is realized by providing a transition time of about one order of magnitude less than the average service time of the workload.

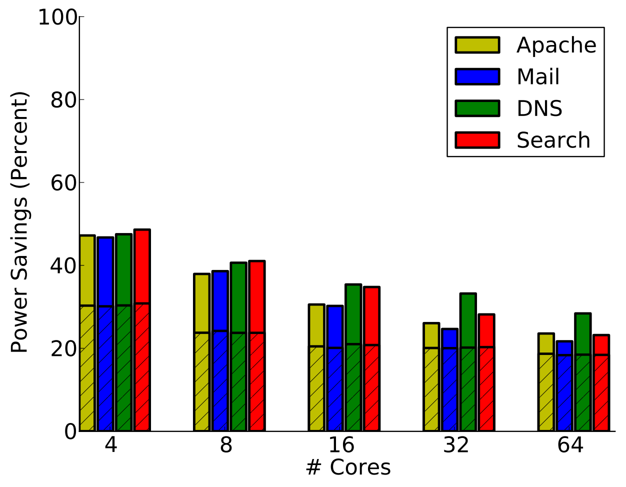


Figure 10: Sensitivity to number of cores. Solid bars represent DreamWeaver savings and hatched bars represent VFS savings. DreamWeaver is less effective as the number of cores increase, but always provides greater savings than VFS.

while incurring much greater delays. As with the 4-core system, DreamWeaver dominates the alternative approaches.

Sensitivity to transition time. To understand the utility of DreamWeaver for various server scenarios, we provide three sensitivity studies. First, we characterize the effectiveness of DreamWeaver for varying sleep transition times. Figure 9 illustrates how power savings diminishes for increasing transition time. We present results for a 16-core system at 30% utilization, with a performance constraint of 1.5x increase in 99th-percentile latency relative to nominal. We annotate the average service time of each workload

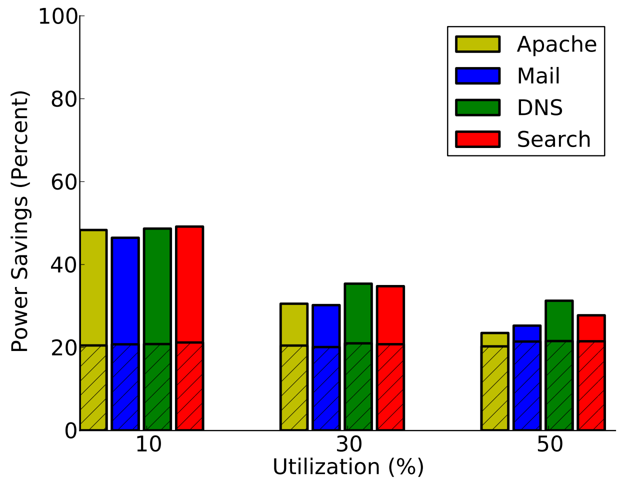


Figure 11: Sensitivity to utilization. Solid bars represent DreamWeaver savings and hatched bars represent VFS savings. DreamWeaver provides more savings in all cases.

along the time axis. As with PowerNap, when transition time becomes large relative to average service time, less power is saved. Ideally, transition time should be roughly an order of magnitude smaller than the average service time. Consistent with PowerNap [41], we find that the slowest transition time that is useful across all workloads is 1ms and designers should target the 100 μs to 1ms range.

Sensitivity to core count. In the next two sensitivity studies, we directly compare DreamWeaver to a system using VFS to save power. Figure 10 contrasts the power savings of DreamWeaver (solid bars) and VFS (hashed subset within each bar) when both are allowed a 1.5x slack on 99th-percentile latency. We vary the number of cores and the corresponding assumption for technology generation (65nm down to 16nm). Even for 64-core systems, DreamWeaver still provides power savings over 20%. DreamWeaver provides greater savings than VFS at all core counts, though its advantage shrinks as the number of cores grows.

Sensitivity to utilization. DreamWeaver is designed for low utilization, which is the common-case operating mode of servers [7]. Accordingly, DreamWeaver provides greater power savings at lower utilization. In Figure 11 we again contrast DreamWeaver (solid) and VFS (hashed) for a 16-core system as a function of utilization, under a 1.5x 99th-percentile response time slack. DreamWeaver still saves roughly 25% of power at utilization as high as 50%. Across the utilization spectrum, DreamWeaver saves more power than VFS, though its advantage is small for some workloads.

5.3 Discussion

Power Management in the 1000-Core Era. DreamWeaver is an effective means to enable full-system idle low-power modes for core counts that we foresee in the next three process generations (to 16nm). However, recent research has proposed 1000-core systems [31] and if transistor scaling beyond the 16nm node continues to double core counts, eventually, massively multicore architectures may become mainstream. The power management challenges we have identified will reach near-asymptotic limits in such a scenario. As we have observed, VFS effectiveness is shrinking at each technology node due to transistor scaling. Similarly, if servers continue to leverage weak scaling, full-system idleness will clearly disappear altogether with 1000 concurrent requests. The hardware and software models for 1000-core systems remain unclear; however, if we continue under current server software paradigms, we conclude that these power management techniques may become ineffective.

The Potential of Strong Scaling. Existing data center workloads rely on request-level parallelism to achieve performance scalability on multicore hardware. This parallelism strategy is a form of *weak scaling* (i.e, solving a larger problem size in a fixed amount of time, as opposed to *strong scaling* where a fixed problem size is solved in a reduced amount of time)—scalability is achieved by increasing request bandwidth rather than per-request speedup. A potential solution to the inefficacy of power management in a 1000-core system is for server software architectures to adopt strong scaling. Whereas in current systems each incoming request is assigned to a single core, under strong scaling multiple cores work together to service a single request faster. The aggregate throughput under strong scaling stays the same, but per-request latency is reduced; the downside is that the software engineering overhead for such architectures is likely to be significantly higher, as engineers must identify intra-request parallelism. Strong scaling makes power management easier because the number of concurrent independent requests is reduced—idle and busy periods naturally align across cooperating cores. As a result, the trends observed in Figure 2 will be reversed. In the limit, if all cores are used to service a single job, the system will behave (with respect to idleness) as if it were a uniprocessor. However, it is likely that Amdahl bottlenecks will preclude using 1000 cores for one request; instead clusters of cores might cooperate. Under this scenario, there will be a moderate number of clusters, and the effectiveness of DreamWeaver will resemble a weak-scaling system with the corresponding moderate number of cores. Unfortunately, the effectiveness of VFS does not change with better parallel software and its effectiveness will continue to decline unless better circuit techniques are developed.

6. Conclusion

As technology continues to scale and core counts increase, effective power management is becoming increasingly difficult. The effectiveness of voltage and frequency scaling is diminishing due to fundamental scaling trends. Because current-generation server software relies on weak scaling to use additional cores, full-system idleness is becoming increasingly scarce. DreamWeaver offers one mechanism to trade latency for power savings from idle low-power modes despite the challenges posed by multicore scaling. We show that DreamWeaver outperforms alternatives such as VFS, Core and Socket Parking, and past batching approaches while providing a smooth trade-off of 99th-percentile latency for power savings. Furthermore, should the community succeed in rearchitecting server systems to leverage strong scaling through intra-request parallelism, the advantages of DreamWeaver over other power management schemes grow even larger. We hope that our work serves as a warning that past approaches to power management are under threat given present scaling trends, and as a call to arms to redesign server software for strong scaling.

Acknowledgements

The authors would like to thank Ricardo Bianchini and Philip Wells for feedback on early drafts of the paper and the anonymous reviewers for their feedback. This work was supported in part by grants from Google and NSF grants CNS-0834403, CCF-0811320, and CCF-0815457.

References

[1] AOL Query Log, 2006.

[2] A Solr index of Wikipedia on EC2/EBS. 2010.

[3] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce PC energy usage. *NSDI '09: Networked Systems Design and Implementation*, 2009.

[4] F. Ahmad and T. Vijaykumar. Joint optimization of idle and cooling power in data centers while maintaining response time. *ASPLOS '10: Architectural Support for Programming Languages and Operating Systems*, 2010.

[5] H. Amur, R. Nathuji, M. Ghosh, K. Schwan, and H. Lee. IdlePower: Application-aware management of processor idle states. *MMCS '08: Workshop on Managed Many-Core Systems*, 2008.

[6] V. Anagnostopoulou, S. Biswas, A. Savage, R. Bianchini, T. Yang, and F. Chong. Energy Conservation in Datacenters through Cluster Memory Management and Barely-Alive Memory Servers. *WEED '09: Workshop on Energy-Efficient Design*, 2009.

[7] L. A. Barroso, J. Dean, and U. Hölzle. Web Search for A Planet: The Architecture of the Google Cluster. *IEEE Micro*, 2003.

[8] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, (December):33–37, 2007.

[9] L. A. Barroso and U. Hölzle. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, Morgan Claypool, 2009.

[10] C. Bash and G. Forman. Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. *USENIX Annual Technical Conference*, 2007.

[11] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, 2003.

[12] J. S. Chase, D. C. Anderson, P. N. Thakar, A. M. Vahdat, and R. P. Doyle. Managing Energy and Server Resources in Hosting Centers. *SOSP '01: Symposium on Operating Systems Principles*, Dec. 2001.

[13] G. Chen, W. He, J. Liu, S. Nath, L. Rigas, L. Xiao, and F. Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. *NSDI '08: Networked Systems Design and Implementation*, 2008.

[14] Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. *SIGMETRICS '05: International Conference on Measurement and Modeling of Computer Systems*, 2005.

[15] M. Cho, N. Sathe, M. Gupta, S. Kumar, S. Yalamanchilli, and S. Mukhopadhyay. Proactive power migration to reduce maximum value and spatiotemporal non-uniformity of on-chip temperature distribution in homogeneous many-core processors. *Semiconductor Thermal Measurement and Management Symposium*, 2010.

[16] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin. DRAM energy management using software and hardware directed power mode control. *HPCA '01: High-Performance Computer Architecture*, 2001.

[17] Q. Deng, D. Meisner, T. F. Wenisch, and R. Bianchini. MemScale : Active Low-Power Modes for Main Memory. *ASPLOS '11: Architectural Support for Programming Languages and Operating Systems*, 2011.

[18] B. Diniz, D. Guedes, W. Meira Jr., and R. Bianchini. Limiting the Power Consumption of Main Memory. *ISCA '07: International Symposium on Computer Architecture*, 2007.

[19] R. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming Moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2), Feb. 2010.

[20] L. Eggert, J. D. Touch, and M. Rey. Idletime scheduling with preemption intervals. *SOSP '05: Symposium on Operating Systems Principles*, Oct. 2005.

[21] M. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. *USENIX Symposium on Internet Technologies and Systems-Volume 4*, 2003.

[22] A. Gandhi, M. Harchol-Balter, R. Das, J. Kephart, and C. Lefurgy. Power Capping Via Forced Idleness. *WEED '09: Workshop on Energy Efficient Design*, 2009.

[23] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. *USENIX Annual Technical Conference*, 1995.

[24] S. Gurumurthi, A. Sivasubramaniam, and M. Kandemir. DRPM: dynamic speed control for power management in server class disks. *ISCA '03: International Symposium on Computer ArchitectureA*, 2003.

[25] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. *ACM Transactions on Computer Systems*, 15(3), Aug. 1997.

[26] T. Heath, B. Diniz, and E. Carrera. Energy conservation in heterogeneous server clusters. *PPoPP '05: Principles and Practice of Parallel Programming*, 2005.

[27] S. Herbert and D. Marculescu. Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors. *ISLPED '07: International Symposium on Low Power Electronics and Design*, 2007.

[28] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[29] Intel. Intel Xeon Processor 5600 Series. Datasheet, Volume 1. 2010.

[30] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. *Micro '06: International Symposium on Microarchitecture*, 2006.

[31] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel : An Architecture and Scalable Programming Interface for a 1000-core Accelerator. *ISCA '09: International Symposium on Computer Architecture*, 2009.

[32] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. *HPCA '08: High Performance Computer Architecture*, 2008.

[33] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. Culler, and R. Katz. NapSAC : Design and Implementation of a Power-Proportional Web Cluster. *Green Networking*, 2010.

[34] W. Lang and J. Patel. Towards eco-friendly database management systems. *CIDR '09: Conference on Innovative Data Systems Reasearch*, 2009.

[35] W. Lang and J. M. Patel. Energy Management for MapReduce Clusters. *VLDB*, 2010.

[36] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *ACM SIGPLAN Notices*, volume 36, 2001.

[37] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power Aware Page Allocation. *ASPLOS '00: Architectural Support for Programming Languages and Operating Systems*, 2000.

[38] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, Dec. 2003.

[39] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis. Power Management of Datacenter Workloads Using Per-Core Power Gating. *IEEE Computer Architecture Letters*, 8(2):48–51, Feb. 2009.

[40] D. Meisner, B. T. Gold, and T. F. Wenisch. The PowerNap server architecture. *ACM Transactions on Computer Systems (TOCS)*, 29(1), 2011.

[41] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. *ASPLOS '09: Architectural Support for Programming Languages and Operating Systems*, Feb. 2009.

[42] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-Intensive Services. *ISCA '11: International Symposium on Computer Architecture*, 2011.

[43] D. Meisner, and T. F. Wenisch. Stochastic Queuing Simulation for Data Center Workloads. *EXERT '10: Exascale Evaluation and Research Techniques Workshop*, 2010.

[44] D. Meisner, J. Wu, and T. F. Wenisch. BigHouse: A simulation infrastructure for data center systems. *ISPASS '12: International Symposium on Performance Analysis of Systems and Software*, 2012.

[45] Microsoft. Improved data center power consumption and streamlining management. 2010.

[46] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-Aware Memory Energy Management. *HPCA '06: High-Performance Computer Architecture*, 2006.

[47] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. *Workshop on Compilers and Operating Systems for Low Power*, 2001.

[48] J. Sharkey, A. Buyuktosunoglu, and P. Bose. Evaluating design tradeoffs in on-chip power management for CMPs. *ISLPED 07: International symposium on Low power electronics and design*, 2007.

[49] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering Energy Proportionality with Non Energy-Proportional Systems Optimizing the Ensemble. *HotPower '08: Workshop on Power-Aware Computing Systems*, 2008.

[50] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. *SIGMOD*, 2010.

[51] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage scaling: an exact algorithm and a linear-time heuristic approximation. *International Symposium on Low Power Electronics and Design*, page 287, 2005.