Architecting Efficient Data Centers

by

David Max Meisner

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2012

ACKNOWLEDGEMENTS

There are many people I would like to thank who have supported me and made this work possible. First, a great thanks to my advisor, Tom, for his guidance. Clearly, this work would not be possible without his support. I also owe a great deal of gratitude to my coauthors: Luiz Barroso, Ricardo Bianchini, Qingyuan Deng, Brian Gold, Steven Pelley, Luiz Ramos, Chris Sadler, Jack Underwood, James VanGilder, Wolf Weber, Junjie Wu, and Pooya Zandevakili. Also, I wish to acknowledge Laura Faulk and DCO for helping obtain the PowerNap traces. Thanks so much to my family – Dad, Mom, Mandy, Peter and Rachel – for their support and patience throughout graduate school. And a great thank you to all my friends for making graduate school tolerable. Finally, I would like to thank my thesis committee for their involvement and feedback.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Architecting Efficient Data Centers

by

David Max Meisner

Chair: Thomas F. Wenisch

Data center power consumption has become a key constraint in continuing to scale Internet services. As our society's reliance on "the Cloud" continues to grow, companies require an ever-increasing amount of computational capacity to support their customers. Massive warehouse-scale data centers have emerged, requiring 30MW or more of total power capacity. Over the lifetime of a typical high-scale data center, power-related costs make up 50% of the total cost of ownership (TCO). Furthermore, the aggregate effect of data center power consumption across the country cannot be ignored. In total, data center energy usage has reached approximately 2% of aggregate consumption in the United States and continues to grow.

This thesis addresses the need to increase computational efficiency to address this growing problem. It proposes a new classes of power management techniques: *coordinated full-system idle low-power modes* to increase the energy proportionality of modern servers. First, we introduce the *PowerNap* server architecture, a coordinated full-system idle low-power mode which transitions in and out of an ultra-low power nap state to save power during brief idle periods. While effective for uniprocessor systems, PowerNap relies on full-system idleness and we show that such idleness disappears as the number of cores per processor continues to increase. We expose this problem in a case study of Google Web search in which we demonstrate that coordinated full-system active power modes are necessary to reach energy proportionality and that PowerNap is ineffective because of a lack of idleness. To recover full-system idleness, we introduce *DreamWeaver*, architectural support for deep sleep. DreamWeaver allows a server to exchange latency for full-system idleness, allowing PowerNap-enabled servers to be effective and provides a better latency-power savings tradeoff than existing approaches. Finally, this thesis investigates workloads which achieve efficiency through methodical cluster provisioning techniques. Using the popular `memcached` workload, this thesis provides examples of provisioning clusters for cost-efficiency given latency, throughput, and data set size targets.

CHAPTER 1

Introduction

Data center efficiency has quickly become a first-class design goal [33]. The emergence of *Warehouse-Scale Computers* (WSC), stadium-sized buildings with tens to hundreds of thousands of servers, has fundamentally shifted the economies of scale in running Internet-scale services. Construction and operation of these facilities is extremely costly and efficient design is essential to driving down these costs. Currently, energy and power related costs make up half of a data center's total cost of ownership [30]. Furthermore, energy and power constraints are typically the limiting factor in adding more servers to a given facility forcing the costly construction of new installations.

In aggregate, data center energy consumption is undergoing alarming growth. In 2012, U.S. data centers are projected to consume 2% of electricity usage in the United States [106]. Unfortunately, much of this energy is wasted by inefficient design. This waste is costly to corporations in operational and capital expenditures and also creates an environmental burden with an unnecessary carbon footprint.

This thesis proposes methods for efficient data center design based on the behavior of modern workloads and the inherent characteristics of server hardware. There are three major components of this thesis. First, it establishes the importance of *coordinated full-system low-power modes*. We demonstrate that this new class of low-power modes is essential in achieving energy-proportional behavior with acceptable latency for many workloads. Second, to evaluate these power savings approaches, we create an evaluation methodology for understanding data center workload behavior and the effect of low-power management on the latency of these workloads. Finally, we will address efficient cluster-level design, using the popular `memcached` workload.

1.1 Server underutilization and inefficiency

A major source of data center inefficiency is due to server idle power and non-energy-proportional operation. At idle, current servers still draw about 60% of peak power [32, 73, 114]. In typical data centers, average utilization is only 20-30% [32, 45]. Low utilization is endemic to data center operation: strict service-level agreements force operators to provision for redundant operation under peak load. Idle-energy waste is compounded by losses in the power delivery and cooling infrastructure, which increase power consumption requirements by as much as 50-100% [133]. Concern over idle-energy waste has prompted calls for a fundamental redesign of each computer system component to consume energy in proportion to utilization [32].

Ideally, we would like to simply turn idle systems off. Unfortunately, a large fraction of servers exhibit frequent but brief bursts of activity [34, 45]. User demand often varies rapidly and/or unpredictably, making dynamic consolidation and system shutdown difficult. Furthermore, many Internet-scale services distribute their data set over a cluster of machines [29, 123]; turning servers off will impact data availability [117].

1.2 PowerNap and DreamWeaver

This thesis proposes the *PowerNap* server architecture, a *coordinated full-system idle low-power mode*, in Chapter 6. With PowerNap, we design the entire system to transition rapidly between a high-performance active state and a minimal-power nap state in response to instantaneous load. Rather than requiring components that provide fine-grain power-performance trade-offs, PowerNap simplifies the system designer's task to focus on two optimization goals: (1) optimizing energy efficiency while napping, and (2) minimizing transition time into and out of the low-power nap state.

Unfortunately, because current server software architectures leverage multicore hardware using request-level parallelism, full-system idleness is growing scarce (even under light utilization) as more cores and components are integrated in a single machine. Because they are already written to process independent requests in parallel, many server applications can achieve good performance scalability by simply using additional cores to serve more concurrent requests. 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. However, increasing the number of independent requests has the side-effect of undermining the applicability of high-leverage idle low-power modes—busy and idle periods of individual cores do not align, precluding full-system sleep.

To address the lack of idleness in multicore systems, in Chapter 7 we propose *DreamWeaver*, architectural support for deep sleep. DreamWeaver is composed of a low-power Dream Processor that intercepts requests sent to a server and uses Weave Scheduling to create artificial idle periods in exchange for latency. Weaver Scheduling is based on two key concepts: (1) stall execution and nap any time that *any* core is unoccupied, but (2) constrain the maximum amount of time any request may be stalled. The key distinguishing feature of Weave Scheduling that it will *preempt execution* to enter the nap state 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.

Whereas many mechanisms required by PowerNap can be adapted from mobile and handheld devices, one critical subsystem of current blade chassis falls short of meeting PowerNap's energy-efficiency requirements: the power conversion system. PowerNap reduces total ensemble power consumption when all blades are napping to only 6% of the peak when all are active. Power supplies are notoriously inefficient at low loads, typically providing conversion efficiency below 70% under 20% load. These losses undermines PowerNap's energy efficiency.

Directly improving power supply efficiency implies a substantial cost premium. Instead, we introduce the Redundant Array for Inexpensive Load Sharing (RAILS) in Chapter 8, a power provisioning approach where power draw is shared over an array of low-capacity power supply units (PSUs) built with commodity components. The key innovation of RAILS is to size individual power modules such that the power delivery solution operates at high efficiency across the entire range of PowerNap's power demands. In addition, RAILS provides N+1 redundancy, graceful compute capacity degradation in the face of multiple power module failures, and reduced component costs relative to conventional enterprise-class power systems.

1.3 Online Data-Intensive Services

Although PowerNap is highly effective for traditional workloads, in this thesis we examine, for the first time, power management for a class of data center workloads, which we refer to as *Online Data-Intensive* (OLDI) in Chapter 9. This workload class would benefit drastically from energy proportionality because it exhibits a wide dynamic load range. OLDI workloads are driven by user queries/requests that must interact with massive data sets, but require responsiveness in the sub-second time scale, in contrast to their offline counterparts (such as MapReduce computations). Large search products, online advertising, and machine translation are examples of workloads in this class. Although the load on OLDI services varies widely during the day, their energy consumption sees little variance due to the lack of energy proportionality of the underlying machinery.

This thesis observes that energy-proportional operation can be achieved for lightly utilized servers with full-system, coordinated idle low-power modes. Such a technique works well for workloads with low average utilization and a narrow dynamic range, a common characteristic of many server workloads. Other work observes that cluster-level power management (e.g., using VM migration and selective power-down of servers) can enable energy-proportionality at the cluster level even if individual systems are far from energy proportional [165].

As we will show, full-system idle low-power modes fare poorly for OLDI services because these systems have a large dynamic range and, though sometimes lightly loaded, are rarely fully idle, even at fine time scales. Cluster-grain approaches that scale cluster size in response to load variation are inapplicable to OLDI services because the number of servers provisioned in a cluster is fixed. Cluster sizing is determined primarily based on data set size instead of incoming request throughput. For a cluster to process an OLDI data set for even a single query with acceptable latency, the data set must be partitioned over thousands of nodes that act in parallel. Hence, the granularity at which systems can be turned off is at cluster- rather than node-level.

Fundamentally, the architecture of OLDI services demands that power be conserved on a per-server basis; each server must exhibit energy-proportionality for the cluster to be energy-efficient, and the latency impact of any power management actions must be limited. We find that systems supporting OLDI services require a new approach to power management: accordingly we demonstrate the need for a *coordinate full-system active low-power mode* in Chapter 9. We demonstrate that neither power management of a single server component nor uncoordinated power management of multiple components provide desirable power-latency tradeoffs.

We report the results of two major studies to better understand the power management needs of OLDI services. First, we characterize a major OLDI workload, Google Web Search, at thousand-server, cluster-wide scale in a production environment to expose the opportunities (and non-opportunities) for active and idle low-power management. We introduce a novel method of characterization, *activity graphs*, which enable compact representation of the activity levels of server components. Activity graphs provide designers the ability to identify the potential of per-component active and idle low-power modes at various service load levels. Second, we perform a study of how latency constrains this potential, making power management more difficult. We construct and validate a performance model of the Web Search workload that predicts the 95th-percentile query latency under different low-power modes. We demonstrate that our framework can predict 95th-percentile latency within 10% error. Using this framework, we explore the power-performance tradeoffs for available and future low-power modes.

1.4 Evaluation Methodology

In order to evaluate these power management approaches, we require a new evaluation methodology. Until now, the systems community has enjoyed a large number of tools for evaluating desktop and server architectures [13, 39, 48, 120, 169, 174]. However, these tools often require hours to simulate only seconds of real time for a single machine; attempting to simulate tens, let alone thousands, of machines quickly becomes prohibitive.

The lack of scalable simulation tools has limited past WSC research to either measurement studies of existing deployments, or analysis via theoretical and statistical models. Measurement studies, though valuable, can explore only existing architectures and require access to multi-million dollar facilities. Even for the few academic and industrial research teams with access to such facilities, experimentation is typically limited to non-intrusive monitoring, since these facilities host the mission-critical operations of their owners. Analytic approaches typically require numerous simplifying assumptions and cannot capture detailed interactions among the components of a WSC. Moreover, even well-understood modeling approaches, for example queuing networks (on which our methodology is based), rapidly become analytically intractable as the size and complexity of the model grows.

In Chapter 5 we introduce *Stochastic Queuing Simulation* (SQS) a scalable, data center-level evaluation methodology. At its core, SQS is a methodology for system characterization and discrete-event simulation to enable quantitative exploration of data center-level challenges, such as performance optimization, power provisioning, power management, distributed data placement, and fault-tolerant design. SQS incorporates a number of techniques from stochastic modeling, queuing theory and statistical sampling to provide simulations that are fast enough to handle multi-thousand server complexity and provide probabilistic guarantees on its estimates. We use this methodology to evaluate a number of power management techniques to improve data center power and energy efficiency.

1.5 Efficient Cluster Provisioning

Finally, we address workloads that require intelligent cluster-level provisioning for efficiency. Recently, there has a been an explosive growth in the development and use of "Web 2.0" applications, where users tag, comment and share content with one another. Making these applications fast is difficult because of the amount of dynamic content that must be assembled in response to each user action. Though applications might retain user content in sophisticated data stores (e.g., database management systems or DBMSs), to provide programmability, durability, and consistency, these heavy-weight data stores often cannot meet the latency or throughput requirements to assemble dynamic web content on the fly. For example, generating a user's Facebook wall or Twitter feed might involve over 100 separate queries for user data—accessing a DBMS this many times with a 50ms response time could lead to an unacceptable wait time of 5 seconds for the user. Performance demands become even more staggering when one considers the 250 million visits per day Facebook receives, resulting in over 17 million requests per second.

To alleviate these performance demands, internet service architects are increasingly relying on `memcached`, a distributed key-value store that enables fast random access to small pieces of data. By aggregating the main-memory capacity of hundreds of commodity servers, datasets reaching into the terabytes can be available at ultra-high performance across a data center network. Individual `memcached` servers are each capable of servicing over 100,000 requests per second with latencies in the hundreds of microseconds. Although `memcached` architectures are relatively new, they have quickly become a critical piece of infrastructure in the modern data center. A number of companies already have sizable deployments (e.g., Facebook, Twitter, and Wikipedia) and the number and scale of `memcached` instances continues to rise—Facebook reports clusters with over 20 TB of DRAM across more than 800 servers, which service 150 million requests per second [71]. Furthermore, `memcached` clusters enable applications with no known data partitioning scheme. Whereas workloads such as web search can "divide and conquer" by partitioning data processing across servers [29], the data access patterns in a social graph prohibit simple data partitioning [163] and demand efficient communication.

While `memcached` has been successful in accelerating the speed and scalability of a number of sites, modern server systems are a poor fit for this workload and suffer from a number of inefficiencies. Despite its apparent simplicity—`memcached`'s source code comprises only 8,500 lines—we discover an astounding diversity in `memcached` performance across use cases and scale that leads to drastic differences in the most cost-effective hardware design for a cluster.

In Chapter 10 we provide a study of efficient cluster provisioning for `memcached`. We characterize the inefficiencies in individual servers and uncover the price-performance tradeoffs of building large cluster. Our optimization framework demonstrates that the latency, throughput, and data set size design space has many local optima and that no one server deployment is best.

1.6 Thesis Organization

This rest of this thesis is organized as follows: In Chapter 2 we provide background on data center workloads and power management. We address the related work in Chapter 3. In Chapter 4, we explain how data centers are utilized and the nature of idleness in these systems. We introduce our evaluation methodology used in the remainder of the thesis in Chapter 5. Chapter 6 presents the initial PowerNap server architecture and in Chapter 7 we introduce the DreamWeaver extension. The RAILS PSU system is discussed in Chapter 8. Chapter 9 contains the case study of Google web search. In Chapter 10, we investigate efficient cluster provisioning using the `memcached` workload. Finally, in Chapter 11, we conclude.

CHAPTER 2

Background

In this chapter, we review the fundamentals of efficient data centers and power management of servers. First, we outline the goals of efficient data center design and the metrics upon which we will evaluate these goals. Next, we create a taxonomy for understanding low-power modes for server power management, which we will refer to for the rest of this thesis. Finally, we discuss the challenges that make power management particularly difficult for data center deployments.

2.1 Data Center Efficiency Design Goals and Metrics

Data center design is fundamentally driven by cost; both constructing and operating large-scale data centers burdens companies with hundreds of millions of dollars in costs. Accordingly, even small changes in efficiency (e.g., 1%) can lead to millions of dollars in savings. Overall changes in cost are best captured by a data center's *total cost of ownership* (TCO). This metric establishes the aggregate cost to construct the building, provide power infrastructure, build/buy servers and install them, cool and power the equipment, and replace equipment as it fails over the entire lifetime of the data center. A data center that provides the same computational output as another at a lower TCO can be considered more efficient. Because TCO incorporates so many factors, it is not trivial to reason about why a particular data center has been built more efficiently than another. Furthermore, it does not separate *capital costs* (i.e., building a facility or buying servers) from *operational costs* (i.e., the cost of electricity or maintenance).

Another popular metric for a data centers is the *power usage effectiveness* (PUE). This quantity helps understand how much power delivered to a data center is lost before it consumed by servers. PUE is defined as:

$$\text{PUE} = \frac{\text{Power delivered to data center}}{\text{Power consumed by servers}} \tag{2.1}$$

This metric has received continual scrutiny because it potentially can be misleading, yet is still useful as a rough approximation [83].

Rather than reason about global data center efficiency metrics, it is useful to decompose efficiency into multiple components. This decomposition allows a designer to reason about the current sources of inefficiency in a given deployment and if a given technique will improve them. We address the following metrics to understand the efficiency of data centers: computational efficiency, energy proportionality, and infrastructure efficiency.

2.1.1 Computational Efficiency

For servers, the *computational efficiency* of a data center component (e.g., a server) is the amount of computational work performed per unit energy. Hence, the efficiency of a server is:

$$\text{Computational Efficiency} = \frac{\text{Throughput}}{E[\text{Power}]} = \frac{\text{Jobs}}{\text{Joule}} \tag{2.2}$$

Traditionally, most servers are designed to maximize *peak efficiency*, which is the computational efficiency of a server at peak load (e.g., 100% utilization). Similarly different processor designs (e.g., big vs. small cores) are often benchmarked at peak load. Since servers operate at non-peak load most of the time, computational efficiency will be determined by the interaction of peak efficiency and energy-proportionality of the system.

2.1.2 Energy-Proportionality

While peak efficiency measures the efficiency of a system at full load, most servers do not operate even close to 100% utilization in the common case. Servers are typically most efficient at peak load and their efficiency degrades as load is scaled down. It is common for server to consume 60% of their peak power when completely idle. The concept of *energy-proportionality* suggests how efficiently a systems scales down.

Computational energy proportionality is the concept that energy should be expended in proportion to the amount of work done by the system [29]. Fundamentally, energy-proportionality relies on removing fixed costs from a system or hiding them through amortization. In other words, with a perfectly energy-proportional system one would “only pay for what you use”.

2.1.3 Infrastructure Efficiency

The infrastructure required to house, power and cool tens of thousands of server is extremely costly, making up around 35% of a data centers TCO [30]. Accordingly, designers will wish to maximize the number of servers that can be supported by a given infrastructure installation. Hence, *infrastructure efficiency* targets *capital costs* and refer to the ratio of supported servers to the cost of the infrastructure supporting them. Techniques such as power capping [73] and power routing [141] seek to increase infrastructure efficiency by either increasing the number of server supported by a given installation or reducing the amount of power infrastructure equipment needed, respectfully.

2.2 Taxonomy of Low-Power Modes

Component-level low power modes fall into two broad classes, *idle low-power modes*, and *active low-power modes*. We briefly discuss each and introduce a generic model for describing an idealized low-power mode that encompasses both classes. Figure 2.1 illustrates a wide variety of active and idle low-power modes and classifies each according to both the spatial (fraction of the system) and temporal (activation time scale) granularity at which each operates. Darker points are used for modes that provide greater relative power savings. Coarser modes tend to save more power, but are more challenging to use because opportunity to use them is more constrained.

2.2.1 Idle Low-Power Modes

Many devices offer *idle low-power modes*, which provide even greater power savings than the most aggressive active low-power modes [78, 122]. 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 [122]. Whereas active low-power modes can increase response time significantly, small relative T_{tr} minimally alters it.

The deepest component energy savings can typically be extracted only when a component is idle. 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 [97]. The Intel Nehalem processor provides a socket-grained idle low-power mode through its “Package C6” power state, which disables some of this cir-

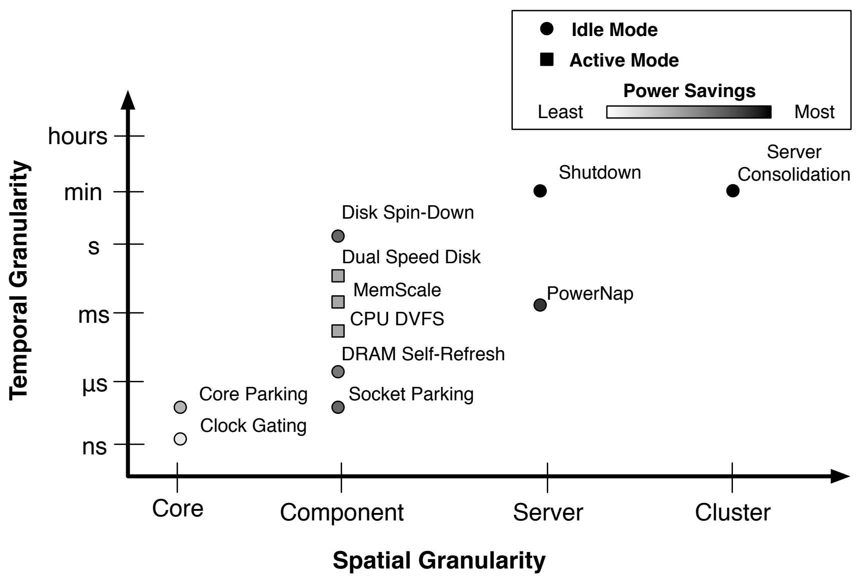


Figure 2.1: Low-Power Mode Taxonomy. Modes with the greatest power savings must be applied at coarse granularity. Modes that apply at fine granularity generally yield less power savings.

cuitry, 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.) [122]. Hence idle-power modes for other components such as power down or self-refresh for DRAM and spin-down for disks are necessary.

2.2.2 Active Low-Power Modes

Active low-power modes throttle performance in exchange for power savings, which potentially extends the time necessary to complete a given amount of work, but still allows processing to continue. Examples include *dynamic voltage and frequency scaling* for processors, MemScale [66] for the memory system and DRPM [81] for disk. Their power savings potential is often limited by support circuitry that must remain powered regardless of load. Some components offer multiple active low-power modes with increasing performance reduction and power savings (e.g., ACPI P-states).

2.3 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 multi-core scaling [90]. 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 similar high percentile) latency, not the mean. Meeting this requirement is complicated by workloads with long-tailed and unpredictable service times [85]. The majority of existing literature (particularly works that have focused on power management) has concentrated on the average latency of server systems; we will instead set targets for 95th- or 99th-percentile latency for much of our analysis.

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.

CHAPTER 3

Related Work

We now review previous work related to efficient data center design. First, we survey previous techniques for power management of servers. Second, we discuss previous methods for evaluating system designs. Finally, we summarize work related to key-value stores.

3.1 Power Management

Previous literature has demonstrated that reducing power at low-utilization is critical to increasing server efficiency [32, 122]. Numerous studies examine power management approaches for processors [70, 72, 147, 162], memory [64, 67, 72], and disk [49, 81, 143]. A detailed survey of CPU power management techniques can be found in [102]. For servers, power savings approaches fall into four broad classes: active low-power modes, idle low-power modes, cluster-grain techniques and scheduling. Additionally, efficiency may be increased through systems with higher peak efficiency.

3.1.1 Idle Low-Power Modes

Idle low-power modes have been explored in processors [117, 129], memory[67, 112], network interfaces [22], and disk [49]. Several research proposals have sought to exploit idle periods of individual memory banks to conserve memory power [64, 67, 91, 112]. During execution, if a particular bank is predicted/detected to be idle, it is transitioned to a low-power mode and re-activated upon a subsequent access. These approaches conserve memory energy during execution at a small penalty in performance, for example, one study of desktop/engineering applications reports that using RDRAM's nap mode cuts DRAM energy 60% to 85% for a few percent performance loss [112]. However, a common conclusion across several studies is that the deepest-available low-power modes (power-down in RDRAM and self-refresh in DDR DRAM) cannot be used effectively because of performance overheads of frequent mode transitions that delay many memory accesses.

3.1.2 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., DVFS) improve energy efficiency if they provide superlinear power savings for linear slowdown. DVFS is well-studied for reducing CPU power [88, 98, 104, 113, 157, 178]. Improving DVFS control algorithms remains an active research area [132, 175]. Active low-power modes have also been proposed for memory [66] and disks [49, 81].

3.1.3 Cluster-grain techniques

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 [23, 51, 55, 56, 87, 107, 144] or server consolidation and virtual machine migration [34, 165] 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 [117, 123].

Furthermore, 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 [32].

3.1.4 Scheduling for energy efficiency

Elnozahy et al investigated using request batching to leverage idle low-power modes in uniprocessors [70]. Several other prior scheduling mechanisms seek to align or construct batches of requests, for example, ecoDB [108] and cohort scheduling [110]. 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.

3.1.5 Low-power Processors

Recently, many studies have looked at using low-power, low-cost server components to improve energy-efficiency for data center workloads [109, 119, 167]. These studies have focused on improving the *peak* efficiency of server systems. We seek to also improve server-level energy-proportionality [29] through low-power modes to save power during non-peak periods.

3.2 Modeling and Simulation

Next, we address previous work on modeling of power consumption, and server and data center evaluation techniques.

3.2.1 Power Modeling

There is a significant body of work investigating processor-level power modeling [47, 59]. In particular, [59] shows power consumption of a microprocessor can be predicted accurately using various performance counters. However, for data center design, predicting CPU power alone is insufficient as it only accounts for a fraction of server power (typically 20-30%). Instead, full-system power models that account for non-CPU components (e.g., memory, disk, etc.) are needed.

Multiple studies have demonstrated that full-system average power is approximately linear with respect to CPU utilization [73, 151].

$$P_{\text{Total}} = P_{\text{Dyn}} \cdot U_{\text{Avg}} + P_{\text{Idle}} \tag{3.1}$$

Particularly when aggregated over a large number of servers, these averages are surprisingly accurate. However, this model provides only *average* power estimates and do not predict peaks.

Switched mode power supply design is well understood [145]: many models are available [168], including many that are based on signal processing [130]. However, the behavior PSUs in running systems, particularly the relationship between CPU utilization and PSU peak power, has not been characterized. We take a full system approach to server power draw, predicting peak power from the logical view of an operating system in Chapter 5.

3.2.2 Data Center Simulation

To the best of our knowledge, this is the first work to provide a rigorous methodology for data center-level simulation. Whereas we leverage numerous works in statistics, stochastic modeling and queuing theory, we refer to these in-line.

Previous studies have attempted to parallelize discrete-event simulations by executing different sections of the modeled system at the same time [76, 134]. Generally, such parallelization is difficult because the system must have a consistent state and requires explicit communication and/or locking of data structures. In contrast, our parallelization strategy, which distributes generation of independent observations for sampled output metrics, does not require synchronization, greatly reducing design complexity and communication overhead.

Alternatively, studies have used hierarchical models to represent data center systems [65]. Such models can be used in lieu of simulators or to compliment them.

Lastly, our work bears similarity to architectural simulators that use statistical simulation [69, 154] and/or sampling techniques [159, 173, 174, 176]. These methods also provide significant reduction in simulation time by either simulating with a statistical abstraction and/or by simulating only those events necessary for the desired level of statistical confidence.

3.3 Memory-resident key-value stores

Our study identifies `memcached` performance bottlenecks and investigates the impact of `memcached` server hardware architecture on performance and cost, particularly as a cluster is scaled with respect to capacity and throughput demand. Prior work has proposed using non-commodity networking hardware (e.g., infiniband) [101] to improve `memcached` performance. We focus on commodity ethernet-based systems, as ethernet currently leads to more cost-effective clusters. Alternatively, `memcached` has been studied in the context of novel many-core processor architectures [37]. Although that study demonstrates substantial performance gains, it does not enumerate the scalability bottlenecks in `memcached` we identify (e.g., the NIC, TCP/IP stack, Kernel and userspace locking) nor does it consider the economic implications of cluster design. Several studies have used `memcached` as a benchmark in evaluating various software techniques [46, 142, 158], but none of these studies focus on designing cost-effective `memcached` clusters.

Whereas `memcached` deployment is typically orthogonal to the backing store, it is important to note that there are several related software systems that integrate in-memory caching more directly with durable databases (e.g., `Cassandra`[28], `Redis` [180], `Project`

`Voldemort`[2], `memcachedb`[127], etc.). The RAMCloud project [135] takes the converse approach, eschewing a disk-based backing store entirely, instead storing all data exclusively in RAM. Although there are important implementation differences between these systems, we focus on `memcached` because it is by far the most widely used. Many of our insights can be extended to these other systems; for example, a RAMCloud cluster might be thought of as a `memcached` system with a 100% hit rate.

Several studies have investigated key-value systems that are either more efficient [27] or available [63]. However, these systems are permanent data stores that rely on file I/O using disk or Flash technology. Other studies have investigated the difference in data center workloads targeted to "wimpy" or "beefy" architectures [167]. While some of the lessons learned from these studies apply, `memcached` behavior nevertheless differs markedly from these systems because it makes no use of file I/O.

Others have looked at the scalability of TCP/IP stacks and NICs and ways to reduce packet processing overheads. There is significant interest in designing ultra-low latency networked systems, but most work has focused on removing software overheads [31, 152]. Studies considering hardware modification include TCP onloading[118, 150], Direct Cache Access (an architectural optimization to deliver packets into the CPU cache) [92], and tighter coupling between the NIC and CPU to provide "zero-copy" packet delivery [40, 41]. Any of these mechanisms might benefit `memcached`, but are not considered here because they are not available in shipping hardware.

CHAPTER 4

Understanding Server Utilization and Idleness

In this chapter we discuss the typical behavior of server utilization and why endemic low-utilization leads to inefficiency. We also investigate server idleness and its implications on idle power management.

4.1 Understanding Server Utilization

It has been well-established in the research literature that the average server utilization of data centers is low, often below 30% [34, 45, 73]. In facilities that provide interactive services (e.g., transaction processing, file servers, Web 2.0), average utilization is often even worse, sometimes as low as 10% [45]. Figure 4.1 depicts a histogram of utilization for two production workloads from enterprise-scale commercial deployments. Table 4.2 describes the workloads running on these servers. We derive this data from utilization traces collected over many days, aggregated over more than 120 severs (production utilization traces were provided courtesy of HP Labs). The most striking feature of this data is that the servers spend the vast majority of time under 10% utilization.

Low utilization creates an energy efficiency challenge because conventional servers are notoriously inefficient at low loads. Although power-saving features like clock gating and dynamic voltage and frequency scaling (DVFS) greatly reduce processor power consumption in under-utilized systems, present-day servers still dissipate about 60% as much power when idle as when fully loaded [51, 73, 114]. Processors often account for only a quarter of system power; main memory and cooling fans contribute larger fractions [113]. Figure 4.3 reproduces typical server power breakdowns for the IBM p670 [113], Sun UltraSparc T2000 [111], and a generic server specified by Google [73], respectively.

Given the poor efficiency of under-utilized servers, one obvious approach to improve overall energy efficiency is to increase average server utilization. The recent trend towards

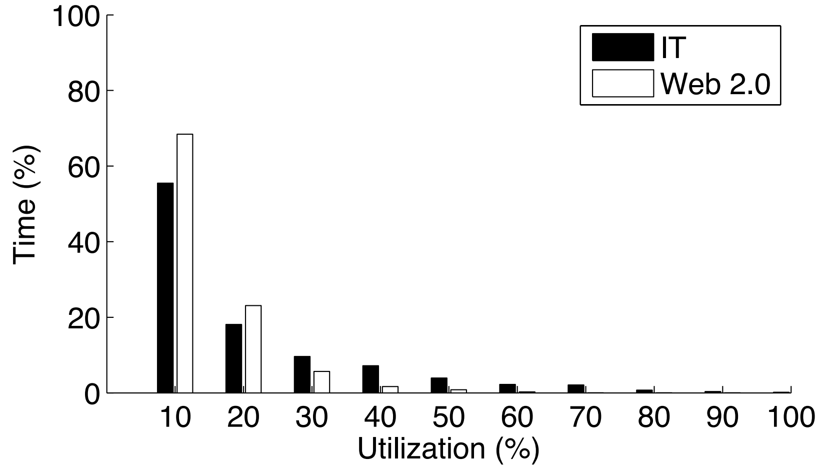


Figure 4.1: Server utilization histogram. Real data centers are under 20% utilized.
Figure 4.2: Enterprise data center utilization traces.

Workload	Avg. Utilization	Description
Web 2.0	7.4%	"Web 2.0" application servers
IT	14.2%	Enterprise IT Infrastructure apps

server consolidation [136] is partly motivated by this objective. By moving services to virtual machines, several services can be time-multiplexed on a single physical server. Consolidation allows the total number of physical servers to be reduced, thereby reducing idle inefficiency. With the availability of live migration, where virtual machines can be transferred among physical hosts during operation without service interruption, it has become possible to operate clusters where servers are brought online and shut down automatically in response to coarse-grain changes in load (e.g., diurnal patterns).

However, dynamic server consolidation cannot eliminate idle energy waste, for several reasons. First, current dynamic consolidation solutions adapt cluster size over 10's of minutes. However, load changes can be far more rapid, particularly when precipitated by an external event (e.g., web server traffic at the end of a World Cup match). For interactive services, peak loads often exceed the average by more than a factor of three [45]. Second, concerns over performance isolation, service robustness, redundancy, hardware configuration conflicts, and security often preclude consolidation of mission-critical services. Third, the software architectures of some data center workloads preclude cluster resizing. For example, both Web Search [29] and memcached [12] distribute their data sets over an entire cluster, typically without replication, to allow user queries to be processed within tight latency constraints. Under this architecture, there is no straight-forward way to resize a cluster in response to load variation. Though industry trends suggest that consolidation

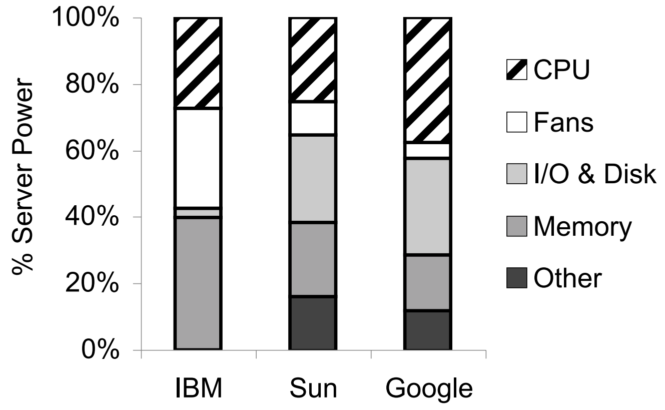


Figure 4.3: Server power breakdown. No single component dominates total system power.

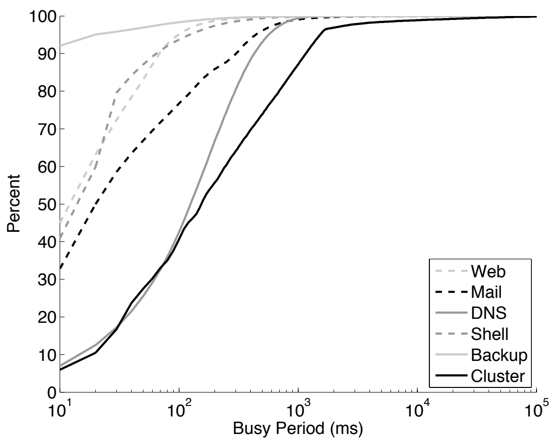
approaches can increase utilization from the 5% to 10% range that is not uncommon today, it is unlikely that utilization above 30%-50% can be achieved for even highly-tuned interactive services.

Frequent Brief Utilization. Clearly, eliminating server idle power waste is critical to improving data center energy efficiency. Engineers have been successful in reducing idle power in mobile platforms, such as cell phones and laptops. However, servers pose a fundamentally different challenge than these platforms. The key observation underlying our work is that, although servers have low utilization, their activity occurs in frequent, brief bursts. As a result, they appear to be under a constant, light load.

To investigate the time scale of servers' idle and busy periods, we have instrumented a series of interactive and batch processing servers to collect utilization traces at 10ms granularity. To our knowledge, our study is the first to report server utilization data measured at such fine granularity. We classify an interval as busy or idle based on how the OS scheduler accounted the period in its utilization tracking. The traces were collected over a period of a week from seven departmental IT servers and a scientific computing cluster comprising over 600 servers. We present the mean idle and busy period lengths, percent full-system idle time and a brief description of each trace in Table 4.6.

Figure 4.4 shows the cumulative distribution for the busy and idle period lengths in each trace (i.e., the vertical axis reflects the fraction of the count of idle periods of a given length or shorter). The key result of our traces is that the vast majority of idle periods are shorter than 1s, with mean lengths in the 100's of milliseconds. Busy periods are even shorter, typically only 10's of milliseconds.

DNS and Mail tend to exhibit the densest activity periods, as both of these frequently handle batch-like tasks (e.g., DNS zone transfers). The Mail server also exhibits the highest utilization among the departmental servers. The Web workload experiences the most



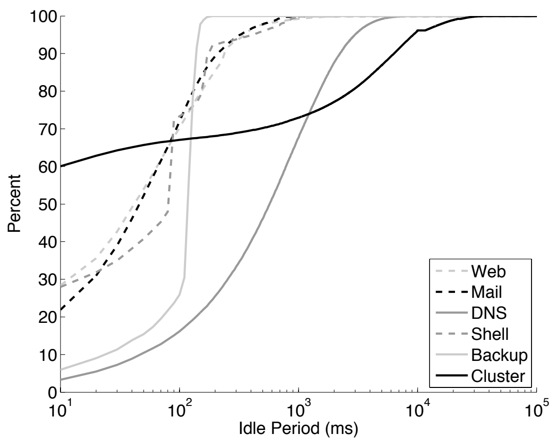
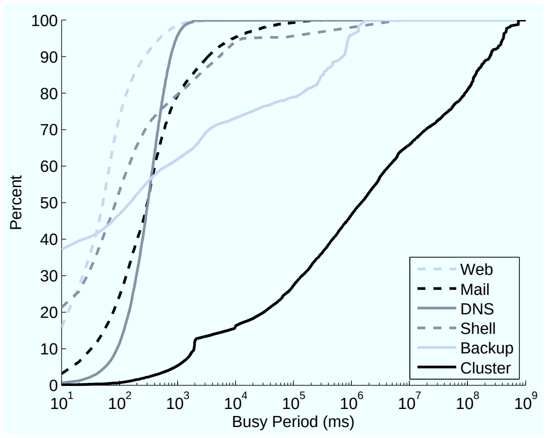


Figure 4.4: Busy and idle period cumulative distributions.



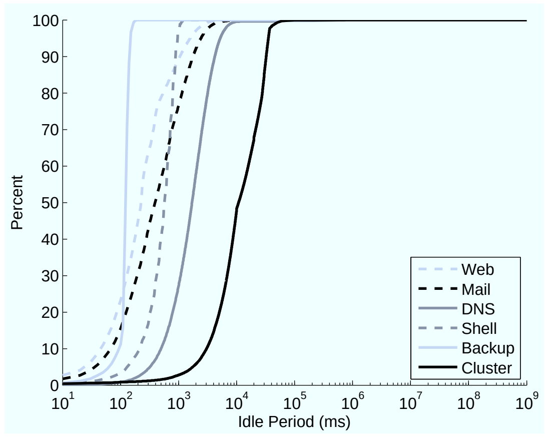


Figure 4.5: Busy and idle period weighted cumulative distributions.

Figure 4.6: Fine-grain utilization traces.

Workload	Full System Idle	Avg. Interval		Description
		Busy	Idle	
Cluster	36%	3.25 s	1.8 s	600-node scientific computing cluster
DNS	83%	194 ms	923 ms	Department DNS and DHCP server
Mail	45%	115 ms	94 ms	Department POP and SMTP servers
Shell	68%	51 ms	108 ms	Interactive shell and IMAP support
Web	74%	38 ms	106 ms	Department web server
Backup	78%	31 ms	108 ms	Continuous incremental backup server

frequent transitions between busy and idle, as only minimal processing is required to serve the frequent requests for static web pages. The Shell and Backup servers exhibit the largest variation in busy periods. For Shell, this variation arises because users occasionally run long, interactive jobs, whereas for Backup, the length of incremental backup tasks varies with the size of recent file modifications.

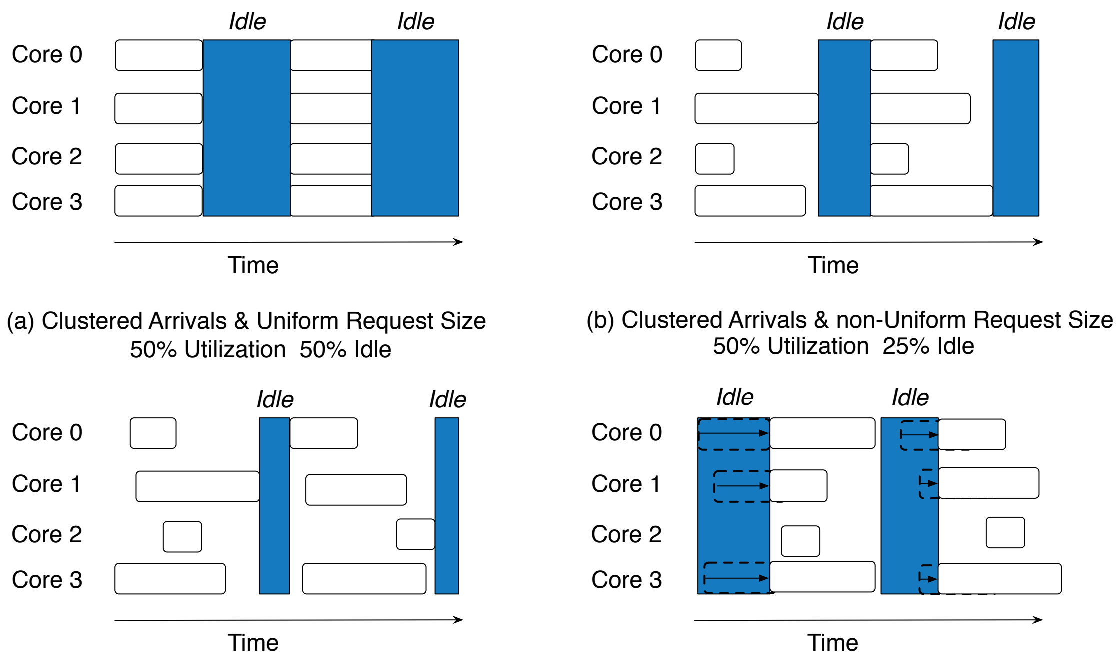
At the opposite extreme, the scientific computing cluster exhibits comparatively high utilization (in line with the results reported in [34, 73]) and an enormous variation in job lengths, from sub-second activities to jobs that run for days. Though the queue of jobs submitted to this cluster is rarely empty, because some machines in this pool are dedicated for specific job classes and users, there are also many long idle periods.

The cumulative distribution of busy and idle periods provides insight into the *frequency* of idle and busy events. However, it does not illustrate where the *time* is spent at each time scale. Figure 4.5 provides the *weighted* CDFs of idle and busy periods; these graphs show the cumulative fraction of idle time that occurs in intervals shorter than the horizontal axis value (i.e., the vertical axis reflects the total time, rather than the count, as in Figure 4.4, of idle periods). This representation demonstrates the presence of infrequent but long idle and active periods. For example, the fact that the Cluster workload spends a significant amount of time in infrequent, long jobs is immediately clear. More importantly, we can see that the majority of idle time occurs in intervals of up to 100ms. Even though Figure 4.4 suggests that most idle periods last 1-10ms, Figure 4.5 shows that the majority of time is spent in slightly longer idle intervals.

Our fine-grain utilization traces do not exhaustively represent the space of data center workloads. In particular, with the exception of the Cluster workload, we have specifically focused on interactive services, which present substantial power management challenges because of their latency constraints. The average utilization levels we observe for these workloads qualitatively match the behavior seen in the customer-provided traces of Figure 4.1 and reports from other sources [34, 45, 73]. Data centers also often run batch-oriented scientific and data intensive (e.g., MapReduce) tasks, which are more similar to our Cluster workload. Such workloads typically have looser latency constraints and are more amenable to consolidation and scheduling-based approaches, which increase average utilization and coalesce idle periods.

4.2 Quantifying and Analyzing Idleness

Our empirical workloads offer insight into idleness behavior in actual systems, but do not offer any tuning knobs to allow exploration of how the shape of the arrival and service distributions of a workload affect idleness. Typically, closed-form queuing analyses assume Poisson arrivals. Under this assumption, request arrivals are independent and interarrival times are exponentially distributed, which matches the arrival pattern generated by a large



(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 4.7: Full-system idleness varies widely as a function of arrival and request size patterns. A workload with clustered arrivals (high C_v) and uniform request sizes (low C_v) maximizes idleness. Non-clustered request arrivals (low C_v) and Non-uniform request sizes (high C_v) decrease the amount of idleness for a fixed utilization. Batching is a method that increases request latency and creates artificial idle periods.

population of independent request sources. However, we are also interested in understanding the impact of arrival processes that are more bursty (i.e., requests arrive in batches) or more uniformly spaced (i.e., requests are throttled to some tempo), corresponding to the scenarios illustrated in Figure 4.7.

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 4.7 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 4.7(a)), all core-level idle periods align. Only in this degenerate case are core-level and system-level idleness equal. In Figure 4.7(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 4.7(c), further reduces system-level idleness. Finally, Figure 4.7(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 request.

The simplest measure of the degree of burstiness/uniformity of the arrival process is its coefficient of variation (C_v), the ratio of the standard deviation and mean arrival time. Low C_v yields more deterministic, uniform arrivals. Conversely, high C_v indicates unpredictable and bursty arrivals. The exponential distribution falls in the middle of this spectrum, and has a C_v of 1.

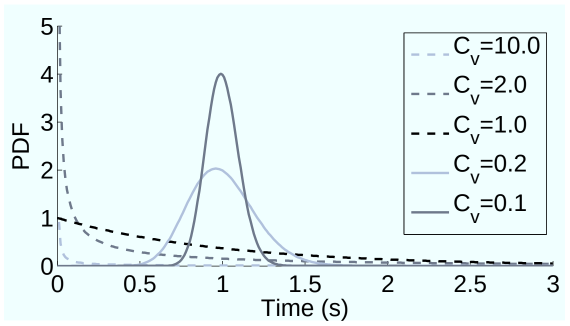
To generate synthetic arrival processes where we can smoothly control C_v, we use the gamma distribution [99]. This distribution is defined by its scale parameter θ and its shape parameter k. Figure 4.8 illustrates how the gamma distribution parameters k and θ can be altered to change the shape and C_v of the distribution without affecting its mean. These parameters allow us to investigate how variability in arrival and service time distribution affect idleness. With the gamma distribution one can "sweep" the parameter space from deterministic ($C_v < 1$), to Poisson (C_v of 1), to uniformly distributed (C_v approaching ∞) arrival/service. Note that the gamma distribution reduces to an exponential distribution when $k = 1$. The shape of the gamma distribution for a fixed mean interarrival time is shown in Figure 4.8. Note that $C_v < 1$ produces a "peaked" distribution that converges towards deterministic, whereas $C_v > 1$ produces a flattened distribution that tends towards uniformity. The table in Figure 4.8 contrasts characteristics of the gamma distribution with other commonly-used distributions.

An important property of M/G/k, and by extension G/G/k, queues is that the variance of the distributions alone does not dictate their behavior [80]. Therefore, even though our results for gamma distributions are instructive, it should be noted that higher order effects may also influence idleness.

Given these stochastic tools, we now investigate low-power mode opportunities and how they are affected by these distributions.

To gain more insight into why multicore scaling destroys full-system idleness, how workload characteristics affect idleness, and why request batching improves usable idleness, we analyze the nature of idleness using SQS. Quantifying idleness is difficult, because simple measures of utilization and other classic metrics do not provide enough information to predict the effectiveness of low power modes. For example a server with a utilization of 30% may have all its requests arrive at once and be of the same length, in which case 70% of the time is available for power savings. Alternatively the same server may have requests with wildly varying request arrival patterns and service times such that 0% of the time is available.

We explore the relationship between the arrival and service time distributions and idleness characteristics using our synthetic workload, which allows us to understand the con-



	Mean	Variance	C_v
Gamma	$k\theta$	$k\theta^2$	$\frac{1}{\sqrt{k}}$
Deterministic	$1/\lambda$	0	0
Exponential	$\frac{1}{\lambda}$	$\frac{1}{\lambda^2}$	1
Uniform	∞	∞	∞

Figure 4.8: Gamma distribution. Allows synthetic distributions with C_v of interest.

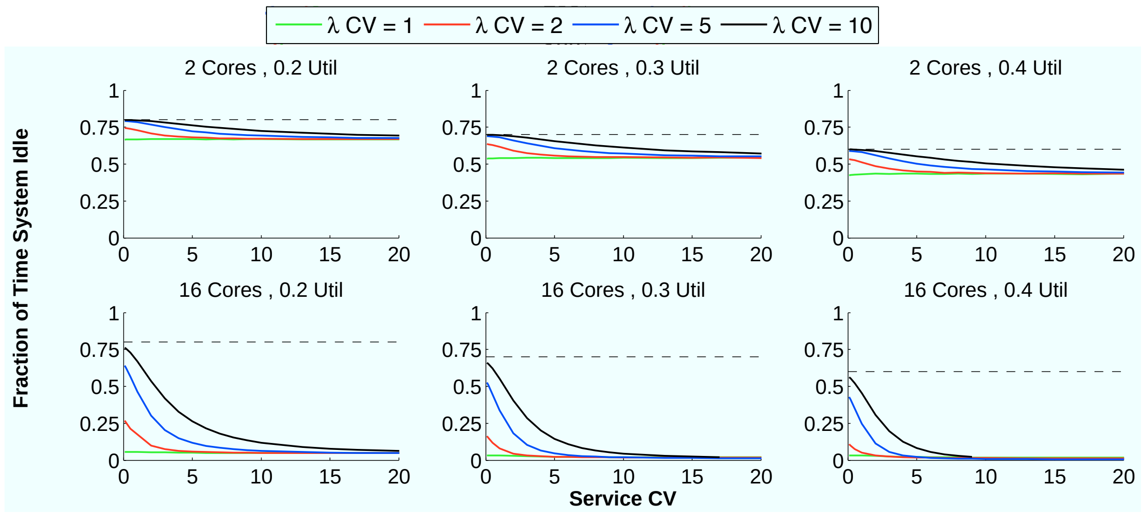


Figure 4.9: Effect of C_v on system idleness. Increasing service C_v reduces idleness, increasing arrival C_v increases idleness. Dotted line shows ideal usable idleness (i.e., core-level idleness).

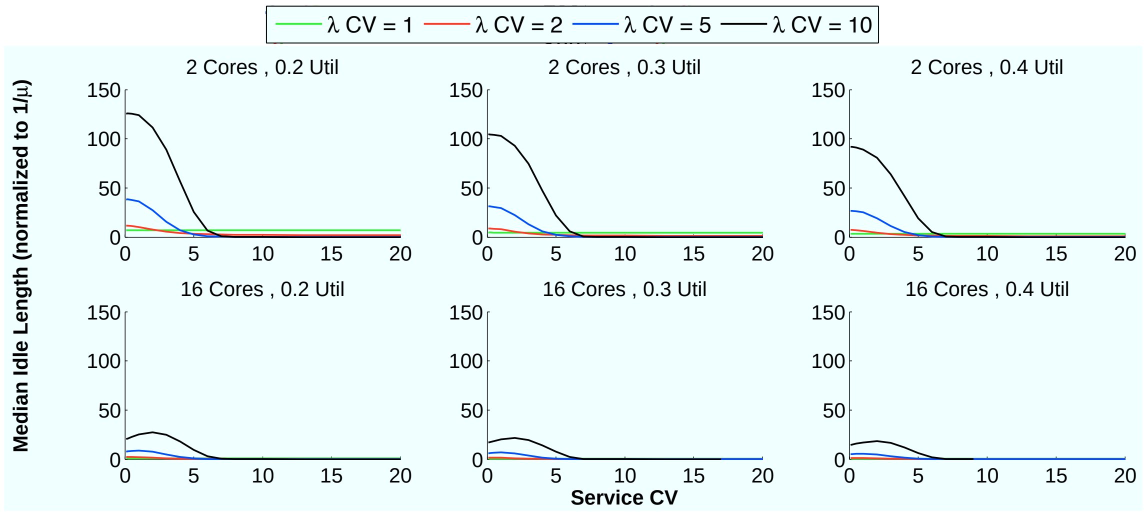


Figure 4.10: Effect of C_v on median idle period length. Increasing service C_v reduces idle lengths, increasing arrival C_v increases idle lengths. For a given utilization and workload, there is a stark reduction in the length of idle periods by integrating more cores.

nection between variability and idleness. Figure 4.9 shows the fraction of time the system is idle, and represents an upper-bound on the idle opportunity for an idle-low power mode with no transition time delay. Figure 4.10 shows the median length of an idle period. This metric is important because an idle low-power mode's usefulness is largely dependent on how its transition time compares to the available idle periods. Each plot presents a matrix of graphs showing increasing number of cores on one dimension (two and sixteen cores) and average utilization on the other (20%, 30%, and 40% utilization). The horizontal axis varies service C_v, whereas the lines vary arrival C_v. The dotted line in each sub-figure represents $1 - u$, the upper bound for idleness in each configuration (if all the requests were perfectly aligned). This upper bound decreases left to right as utilization increases.

An important initial observation is that for a uniprocessor server (one core), idleness is *always* available in full. In other words, regardless of the arrival and service distributions, the server always spends $1-u$ of it's time completely idle. However, as we add more cores, this idleness is no longer available.

Next, we observe that increasing the service C_v decreases idle opportunity, both in terms of the amount and length of idle periods. As service C_v increases, request lengths become more variable. The longest requests force the system to remain active longer and more often. Conversely, high arrival C_v increases idleness opportunity. This trend is due to the increased likelihood of requests arriving in clusters, resulting in longer periods where no request arrives.

Note that an arrival C_v of 1 results in a fixed amount of idleness independent of service C_v. This phenomenon arises because the gamma distribution reduces to a memoryless exponential distribution when C_v=1. In this case, idle periods occur only as a function of the arrival rate λ, and the expected idle length is given by the expected time until the next arrival, or $1/\lambda$ (i.e., arrivals are independent and memoryless).

We can summarize the following trends with respect to full-system idleness:

- **Increasing core density decreases the quantity and length of idleness**: The trend of integrating more cores into processors decreases the opportunity for full-system idle power modes. These modes must work on finer time scales and have less total opportunity.

- **Increased utilization decreases idleness**: Increasing utilization decreases the upper-bound on exploitable idleness (if all the requests are aligned). However, depending on arrival and service time distributions, idleness may be lost disproportionately.

- **Increased service C_v degrades idleness**: Increasing service C_v rapidly decreases the length and quantity of idleness. This trend is particularly unfortunate because many server workloads exhibit high service C_v.

- **Increased arrival C_v improves idleness**: a workload with a higher arrival C_v will have an increased probability of "bursty" requests arrivals and accordingly processes requests in batches, leaving longer periods where no requests arrive.

Our analysis leads to a critical conclusion: to increase usable idleness, we seek to transform the request arrival process to increase the effective arrival C_v. That is, we wish to make the arrival process appear more bursty by delaying arrivals to create batches. The DreamWeaver technique described in Chapter 7 achieves precisely that; by delaying requests, it transforms the actual arrival process into an apparent arrival process with higher interarrival C_v, increasing usable idleness. We provide a direct comparison of DreamWeaver to alternative power management approaches in Chapter 7.

CHAPTER 5

Evaluation Methodology

Recently, there has been an explosive growth in Cloud-based Internet services, greatly influencing both software and hardware architectures. Small mobile devices connected to large data centers are becoming increasingly important, quickly overtaking traditional workstations. The design of data centers themselves has shifted from smaller collocation centers to massive Warehouse-Scale Computers (WSC) [33], housing many thousands of servers.

Unfortunately, the research community has yet to catch up with the blistering pace of development in industry. While research in the mobile space has matured significantly, the ability to quantitatively evaluate the design of large data centers lags significantly. Research in this area has been hindered because, unlike mobile systems, researchers cannot simply use or extend existing tools and apply them to data center problems.

Until now, the systems community has enjoyed a large number of tools for evaluating desktop and server architectures [13, 39, 48, 120, 169, 174]. However, these tools often require hours to simulate only seconds of real time for a single machine; attempting to simulate tens, let alone thousands, of machines quickly becomes prohibitive.

The lack of scalable simulation tools has limited past WSC research to either measurement studies of existing deployments, or analysis via theoretical and statistical models. Measurement studies, though valuable, can explore only existing architectures and require access to multi-million dollar facilities. Even for the few academic and industrial research teams with access to such facilities, experimentation is typically limited to non-intrusive monitoring, since these facilities host the mission-critical operations of their owners. Analytic approaches typically require numerous simplifying assumptions and cannot capture detailed interactions among the components of a WSC. Moreover, even well-understood modeling approaches, for example queuing networks (on which our methodology is based), rapidly become analytically intractable as the size and complexity of the model grows.

This study presents our data center-level evaluation methodology, *Stochastic Queuing Simulation* (SQS) [124, 125], targeted specifically to investigate issues of data center design at scale. At its core, SQS is a methodology for system characterization and discrete-event simulation to enable quantitative exploration of data center-level challenges, such as performance optimization, power provisioning, power management, distributed data placement, and fault-tolerant design. SQS incorporates a number of techniques from stochastic modeling, queuing theory and statistical sampling to provide simulations that are fast enough to handle multi-thousand server complexity and provide probabilistic guarantees on its estimates. SQS is implemented in the *BigHouse* software infrastructure [126].

Our methodology hinges on the observation that designers must raise the *level of abstraction* for data center-scale simulation. Rather than simulate workloads at the granularity of an instruction, memory, or disk access as in conventional simulation tools [39, 48, 120, 169, 174], SQS is built on the theoretical framework of queuing theory, where the fundamental unit of work is a *task* (a.k.a job). Tasks are characterized by a set of statistical properties—random variables that describe their length, resource requirements, arrival distribution, or other relevant properties—which are collected through observation of real systems. SQS abstracts the data center as an interrelated network of queues and power/performance models describing the relevant behaviors of software/hardware components. The discrete event simulation uses a variety of statistical sampling techniques to provide estimates of selected output variables (e.g., 95th-percentile response time) with quantifiable measures of confidence, while enabling parallel simulation to provide strong scaling to reduce turnaround time.

SQS is not a replacement for conventional simulators; whereas existing simulation tools are still needed to refine the design for an individual server within a data center, SQS provides a framework for investigating behaviors that emerge at scale with rapid turnaround time.

5.1 Requirements of Data Center-level Evaluation

Data center research is challenging because many interesting properties emerge *at scale*. Traditionally, system designers have investigated issues such as performance, power and fault-tolerance *within* a single processor or server. Now, with the increasing importance of data center computing, studies must consider these topics *across* clusters of machines, often numbering into the tens of thousands. Numerous recent studies examine the behavior of ensembles of servers [73, 86, 141, 146, 148, 170], SQS is designed to carry out these types of studies.

One approach to investigate data center-level problems is to measure real facilities directly. However, while studies that measure tens of thousands of machines exist [73, 154], they are rare. Moreover, even researchers with access to large-scale facilities are unable to *modify* the observed system; reserving thousands of machines for every research study is simply too costly.

To enable tractable simulation of large-scale systems, we argue that it is necessary to simulate systems at coarser detail than conventional computer architecture simulation tools. In particular, for many interesting design problems (e.g., network topology design, reliability, power management), instruction-grain detail is unnecessary. An effective data center-level evaluation methodology must: 1) handle multi-thousand server complexity, 2) provide statistically rigorous results, and 3) have a reasonable turnaround time.

5.2 Shortcomings of Existing Methodologies

There is a wealth of mature evaluation techniques for computer systems. Nevertheless, the unique challenges of data center-level design require features that current methodologies do not provide. We briefly address the most popular of these techniques and explain why they are generally not appropriate as a data center-level methodology.

Analytic Modeling. When possible, analytic modeling is among the most powerful tools available to system designers. Analytic results eliminate the time-cost of simulation and have the unique advantage of providing mathematical insight. There is a rich literature supporting models based on queuing models [99] and in some cases, these can be directly applied to data center problems [77]. However, there is a large design space under which analytic modeling becomes intractable.

Many workloads have bursty arrival patterns [36, 139] and/or service times with large variances [85]. In such situations, typical queuing model assumptions, such as independence and memoryless, exponential distributions, are broken. To model such systems, approximation techniques become necessary, quickly degrading the fidelity of a model [80]. Alternatively, hierarchical models have been used to represent I/O workloads [65]. Furthermore, data center-level design requires the understanding of quantile estimates (e.g., the 95th-percentile response time). Typically, deriving the distribution of a variable such as response time is significantly more difficult than simply the expectation (i.e., the average). Finally, deriving new results from analytical models for every problem can be time-intensive and require specialized expertise. It is uncommon that an analytical model generalizes to every problem; studies like design-space explorations may benefit from a more accessible quantitative methodology.

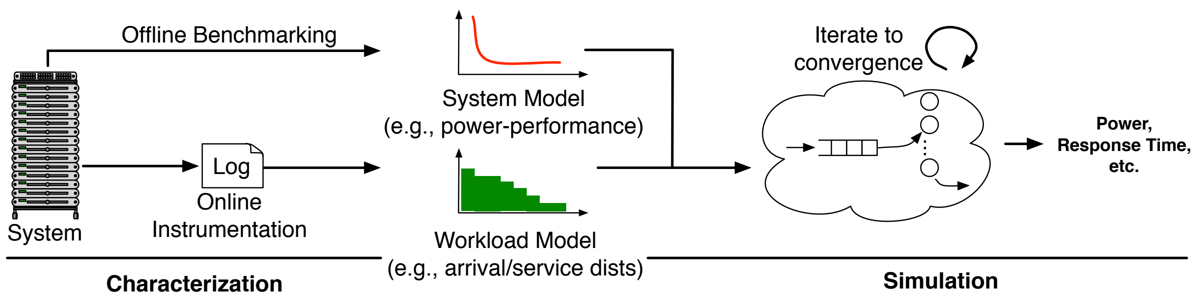

Offline Benchmarking
System
Log
Online Instrumentation
System Model (e.g., power-performance)
Workload Model (e.g., arrival/service dists)
Iterate to convergence
Power, Response Time, etc.
Characterization
Simulation

Figure 5.1: Overview of the SQS methodology. A system is a) instrumented to derive workload interarrival and service time distributions and b) characterized to create a model of system behavior (e.g., power-performance settings). From these inputs, SQS simulations derive estimates for new system designs and/or configurations.

Discrete-event Simulation. By far, discrete-event simulation is the most popular evaluation methodology for hardware system design [74]. There are a number of tools for evaluating the design of processors (e.g., Flexus [174], GEMS [120], M5 [39]), memory systems (e.g., DRAMSim[169]), disks (e.g., DiskSim [48]), and networks (e.g., NS2 [13]).

On its face, it is feasible to use these tools in conjunction to simulate each part of a server in a data center. However, the main challenge in data center design is to understand behavior at scale; the simulation time for thousands of nodes with current tools would be infeasible. Studies of data center techniques such as power-capping ask questions such as, "How often does cluster-level power exceed a threshold for various workloads?" [73, 170]. For such studies, low-level details such as processor microarchitecture are largely irrelevant or can be captured with a less detailed model; simply put, currently available tools simulate *too much detail.*

Another drawback of modern tools is that most are inherently limited to serial execution. Because of the magnitude of the systems we wish to simulate, we will need to extract simulation performance proportional to the committed resources. For example, in scaling from a simulation of 100 machines to 1000, we would like to be able to use ten times the resources to achieve the same turnaround time. Few current tools scale across available cores or machines in a cluster; instead simulations must be run on machines with the best-available single-threaded performance.

5.3 Stochastic Queuing Simulation

In this section, we present the SQS methodology, the procedure for deriving average and quantile estimates from simulations, and how workloads are modeled.

5.3.1 Overview

At its heart, SQS uses the abstractions of queuing theory to create a stochastic model describing the behavior of a data center. Rather than solve the resulting model analytically (an intractable challenge), SQS derives estimates for output variables by exercising the stochastic model with synthetic input tasks derived from empirical workload models. A variety of sampling techniques are then used to extract statistically rigorous estimates of output variables.

Throughout this description and our evaluation, we use the running example of studying a scheme for enforcing power budgets (a.k.a power capping) over server ensembles using *dynamic frequency and voltage scaling* (DVFS), a problem studied in several recent publications [73, 115, 141, 148, 169].

In the SQS abstract model, the data center is represented as a queuing network with generalized parameters—that is, arrival and service distributions, queuing discipline, etc. need not be limited to the commonly used queues that are analytically tractable (e.g., M/M/1) since the model will be exercised via simulation. A task corresponds to the most natural unit of work for the workload under study, such as a single request, transaction, query, and so on.

The SQS queuing network captures the processing steps through which tasks must proceed at a level of detail appropriate to the question under study. Each server in the queuing network is coupled to power/performance models that modulate the service rate and generate output variables of interest. For example, for our power-capping case study, each CPU core corresponds to a server in the queuing network, and the service rate of the queue is determined by the core's DVFS setting. The core model outputs task response times and the system power draw. The queue retains the backlog of tasks (e.g., web requests) at the system.

More complex multi-tier services or client-server interactions can be modeled by tasks that advance through a sequence of queues. These queue sequences can also model other constrained resources (e.g., network links, I/O subsystems). For our demonstrative case study, we model multi-core servers and their power distribution hierarchy, but examine only single-tier workloads and do not model other data center subsystems.

5.3.2 SQS Methodology

Figure 5.1 provides an overview of the SQS methodology, which comprises two parts: characterization and simulation.

Characterization. In the characterization step, we construct empirical models of workloads and systems that are used during the simulation step. A workload model comprises, at a minimum, task interarrival and service distributions. The workload model may also include distributions for other critical task parameters (e.g., tasks' network traffic if modeling network links). The system model modulates service rates and relates tasks to output variables of the simulation (e.g., in our running example, it captures the power-performance curve for DVFS states).

Characterization involves both an online and offline component. We construct empirical models of workloads online, by instrumenting a live system. Typically, this process involves instrumenting a binary such that the timing of task arrivals and their duration are recorded. Later, these traces can be processed to derive the desired distributions. It is necessary to capture these workload models online, under live traffic, because interarrival processes depend greatly on the users of an internet service.

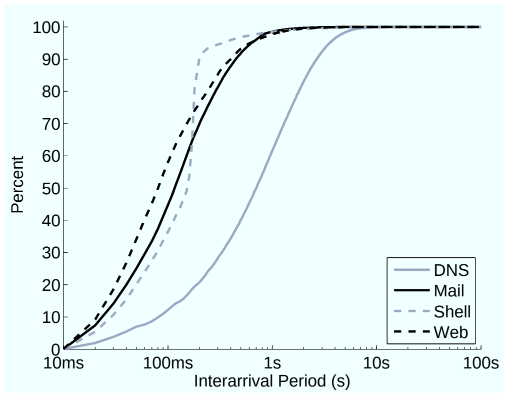
In the offline component of characterization, real systems are benchmarked to capture their modes of operation and construct system models. For our power-capping example, one would capture a server's power-performance behavior under the available DVFS settings. The model records the relative service rate and power consumption of the system as a function of frequency setting. Typically, this part of characterization must be performed offline because it would disrupt production systems.

Simulation. During simulation, SQS derives estimates for hypothetical data center configurations. For our DVFS example, various frequency transition policies for a rack of servers could be evaluated such that both latency constraints are met and rack-level power stays within a budget.

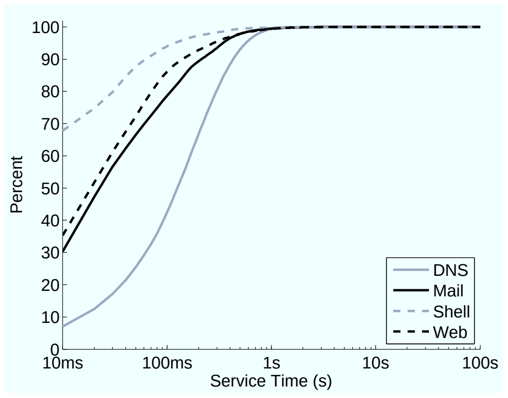
The simulation itself is a discrete-event simulation of the queuing network representing the data center. Typical events represent high-level phenomenon such as a task entering or exiting a server, a power-performance state changing, and so on. The core functionality of the SQS discrete event simulator does not differ substantially from other tools for simulating queuing networks. For a detailed survey of queuing models, we refer the reader to [84]. SQS augments conventional queuing networks with system models, such as the power-performance model used in our example.

5.3.3 Workload Models

Rather than requiring an executable binary, as in a traditional simulator, SQS workloads are defined statistically by empirical interarrival and service time distributions. This approach allows workloads to be represented compactly—a typical distribution occupies less than 1 MB, whereas event traces often require multi-gigabyte files. Furthermore, in



(a) Interarrival Time Distribution



(b) Service Time Distribution

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
Shell	186ms	796ms	4.2	46ms	725ms	15	Interactive shell and IMAP support
Web	186ms	380ms	2.0	75ms	263ms	3.4	Web server

Figure 5.2: SQS workload model. Workloads are represented by their interarrival (time between subsequent requests) and service time distributions (time to complete a given request). The example workloads have distinct properties; for example, Shell has little variance in interarrival time and short service times (99% are below 200 ms). Alternatively, DNS has a wide variance in both distributions, and a long service times (50% above 200 ms). These properties affect both the behavior of the modeled system and simulation time.

contrast to binaries, which industry is often loathe to disseminate, public dissemination of interarrival and service distributions is significantly easier, as they do not require releasing proprietary software.

Under pen-and-paper analysis of queuing models, statistics like the moments of the arrival and service distributions are used to calculate performance measures in closed form. We, and others [80], have found that easily-analyzed queuing models (e.g., M/M/1) often poorly represent internet services. More generic models, such as the G/G/1 or G/G/k queue (generalized interarrival and service time distribution and either 1 or k servers), have no known closed-form solution. Approximations can be used; however, it has been shown their accuracy is often inadequate, especially when only using a few moments [80].

Under SQS, we synthesize events from the empirical distributions directly, retaining the entire fidelity of the measured workload. Figure 5.2 presents four example workloads we will use in this study: a dynamic name service (DNS), webmail hosting (Mail), interactive login and processing (Shell), and web server (Web).

These workloads provide a diverse selection for evaluating SQS because the shape of their distributions vary substantially. The Shell workload has a small variance in its interarrival distribution, implying uniformly timed arrivals of requests to the server. On the other hand, DNS has a large variance, leading to long periods of idleness and bursts of work. Mail and Web are in between these two extremes.

The service time distributions also differ greatly. Whereas Shell requests are low-variance and generally short in duration (99% of requests are completed in less than 200ms), DNS requests have a wide variance. The service time and variance for Mail and Web again fall in between Shell and DNS.

Though it is possible to exercise the SQS discrete-event simulator by replaying traces directly (which eliminates some sampling difficulties, such as sample auto-correlation), it remains unclear how to replay traces and obtain statistically rigorous performance estimates if the simulated system differs substantially from the one where traces are collected. SQS's sampling methods are build on the assumption that event sequences are generated synthetically by random draw from the empirical distributions.

5.3.4 Output Variables

For a given simulation, in addition to the data center configuration (e.g., the number of servers, workloads, etc.), the SQS user must specify a set of *output variables*. The simulation's output variables are derived from quantities generated by the system model upon specific events, which are recorded, analyzed, and reported with statistical confidence estimates. For example, when a task is completed, its response time can be recorded and then aggregated into a mean or quantile output variable. Each output variable is specified along with a desired accuracy and confidence level for quantile and mean estimates.

Accuracy and Confidence. An estimate of an output variable has an associated *accuracy*, ϵ, and *confidence level*, $1 - \alpha$, that together form a *confidence interval*. The value ϵ defines the half-width of a confidence interval in the same units as the output variable (e.g., response time with $\pm$50ms). We normalize this value by the mean estimate, $\bar{X}$, to enable meaningful comparison across multiple output variables:

$$E = \epsilon / \bar{X} \tag{5.1}$$

With this definition, a given E describes the desired accuracy as a percentage (e.g., response time with $\pm$5%). The confidence level of an estimate describes the expected percentage of estimates that would fall within the confidence interval if the simulation were repeated a large number of times. A confidence level of 95% is common, and we use this value for the remainder of this paper.

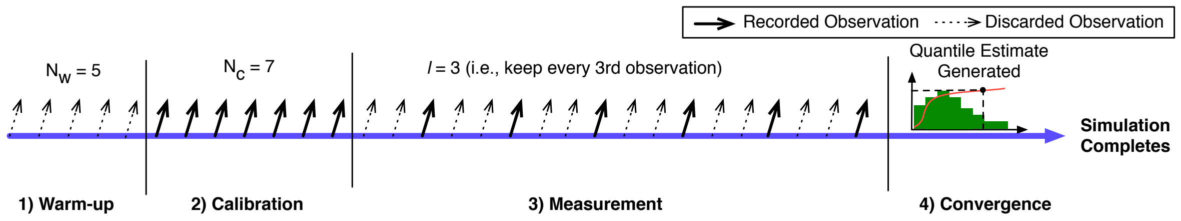


Figure 5.3: The sequence of phases in a SQS simulation. At first, all observations are thrown-away during warm-up, avoiding cold-start bias. Next, during a brief calibration phase, a small sample is collected to determine the appropriate lag spacing and histogram configuration. The majority of the simulation is spent in the measurement phase, where observations are taken with sufficient spacing to ensure independence. Finally, when the desired statistical confidence is achieved, the simulation terminates, outputting quantile and mean estimates.

Mean Estimates. To determine the confidence interval for mean estimates (e.g., mean response time), we leverage standard techniques for large-sample analysis. According to the central limit theorem, the sampling distribution of a mean value estimate tends towards the normal distribution as sample size increases. Hence, we can determine the sample size needed for a given confidence by:

$$N_m = \frac{Z^2_{1-\alpha/2} \cdot \sigma^2}{\epsilon^2} \tag{5.2}$$

Where $Z_{1-\alpha}$ comes from the standard normal: it is the value of the standard normal distribution at the $(1-\alpha/2)^{\text{th}}$ quantile and is 1.96 for 95% confidence. σ is the sample standard deviation and ϵ is the half-width of the desired confidence interval.

Quantiles. Confidence intervals for quantiles (e.g., the 95th-percentile latency) can also be derived using the central limit theorem [53].

$$N_q = \frac{Z^2_{1-\alpha/2} \cdot q(1-q)}{\epsilon^2} \tag{5.3}$$

The notation is the same as for mean estimates with the addition of q as the desired quantile. To find an exact quantile, one would need to record and sort all observations in the sample. The sample size required for even a single output variable can be quite large. Accordingly, recording and sorting the entire sample sequence to determine quantiles imposes a large burden. However, space-efficient approximations using online algorithms are described in [52, 53]. We use the method presented in [53] to maintain a histogram representation of an observed variable, drastically reducing memory overhead. This method requires the histogram binning parameters to be determined in advance; we do so during the calibration phase of the simulation sequence (see below).

Typically, it is useful to know both the mean and at least one quantile of a given output variable. In this case, the required sample size for the desired confidence will be N=max(N_m, N_q). Furthermore, if multiple quantiles are desired, the maximum N_q value from Equation 5.3 should be used. If the entire distribution is desired, q=0.5 as this maximizes the quantity $q \cdot (1 - q)$ in Equation 5.3.

5.3.5 Simulation Sequence

SQS simulations proceed by exercising the discrete-event queuing simulation, creating task arrival events through random draws according to the distributions captured in the workload model. We refer readers to the literature for details on implementing such a simulator [74]. We focus our discussion on the sampling methods at work in SQS, detailing the progression of a simulation from the perspective of an observed output variable (e.g., server response time or power consumption). The phases of an SQS instance, illustrated in Figure 5.3, are:

1. Warm-Up - A simulation begins in an initial *transient state*, where observations are biased by the initial simulation state (e.g., all queues are empty). To avoid this cold-start effect, the simulation must undergo a *warm-up* phase and is exercised for N_w observations, during which all observations are discarded. Unfortunately, a reliable method for determining N_w has been the subject of years of debate [138]. To date, no rigorous method for automatically detecting steady-state is available and N_w must be explicitly specified. We conservatively choose large values for N_w (much larger than any busy interval we have observed in simulation).

2. Calibration - One of the key challenges that must be addressed when drawing a sample from a discrete event simulation is ensuring independence among the sampled observations. Using successive observations from a queuing-based simulation has been shown to introduce bias into estimates because observations tend to be autocorrelated (i.e., nearby observations are not independent) [54]. However, it has also been demonstrated that if observations are spaced sufficiently apart—by keeping only every lth sample—they can be treated as independent [53]. Determining this minimum spacing, l, is accomplished with the *runs-up* test detailed in [105]. The major consequence of this approach is that steady-state simulation length is inflated by a factor of l. Though a sample size of $N = n$ observations may be sufficient to achieve a given confidence in an i.i.d. draw, since $l - 1$ observations are discarded for every 1 taken, a total of $N = l \cdot n$ events must be simulated to achieve the target sample size. A small caveat is that this method often increases sample variance [61], further increasing n.

During the *calibration* phase, we perform the runs-up test to determine the *lag spacing*,

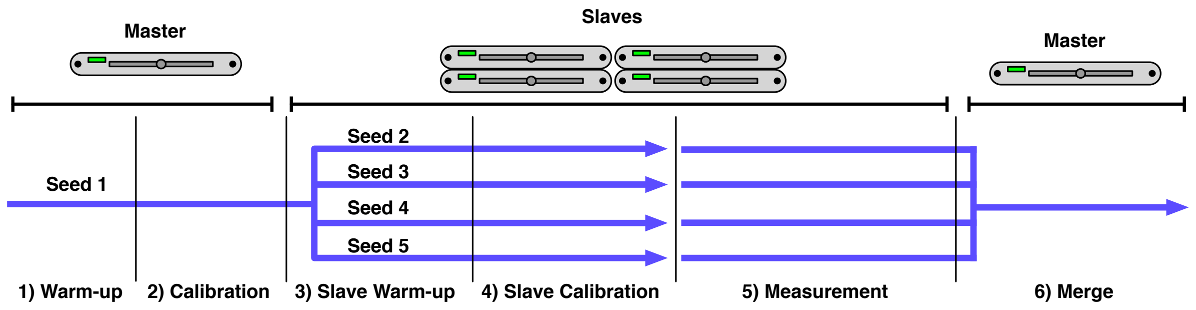


Figure 5.4: Parallel execution on a cluster. First, the simulation undergoes a warm-up and calibration phase on the master. A histogram is generated from the calibration sample and the bin scheme is sent to the slaves. Each slave then executes its own warmup and calibration phase using a unique random seed to achieve steady-state and determine its own lag-spacing. Samples are collected at each slave until their aggregate size is sufficient to achieve the desired accuracy. Finally, in the merge phase, each slave sends its histogram to the master, which aggregates the histograms and reports estimates.

l, between observations and the proper histogram binning parameters to enable quantile estimates.

3. Measurement - Once the simulation enters steady-state, observations are collected to populate the histogram representation of the output variable. The majority of simulation runtime is spent in this phase; the other three phases impose insignificant runtime overhead (in single-threaded simulations; however, their overheads can grow dominant in parallel simulations. See Section 5.4.1).

4. Convergence - An output variable is considered *converged* once the observed sample size is sufficient to achieve the desired confidence interval. If the sample has been generated using distributed computation (Section 5.4.1), it is coalesced at this point. Finally, estimates of quantiles and averages can be reported.

Observing Multiple Variables. Typically, multiple variables are observed in a single simulation. For simplicity, we have explained a sequential procedure and illustrated the sequence in Figure 5.3 in terms of a single output variable. However, it is important to understand that there is an associated sequence for each output variable in the simulation. There are two important constraints on the simulation progression when targeting multiple variables. First, the simulation may not progress out of the warm-up phase until N_w observations have been collected for all output variables. This constraint ensures that measurement does not take place until the entirety of the model is warm. Second, the simulation may not terminate until *all* variables have a sufficient sample size to reach convergence. Again, the slowest variable will determine simulation runtime.

5.4 Parallelization

The complexity of data center systems can require complicated models, leading to long simulation time. We now detail how SQS can overcome lengthy simulations by parallelizing across a cluster of machines.

5.4.1 Distributed Simulation

The procedure of a distributed SQS simulation is outlined in Figure 9.3. A simulation cluster comprises a single *master* and many *slave* machines. First, the master executes just the warmup and calibration phase of a serial SQS simulation. After calibration, the master constructs the appropriate histogram bin structure, which is forwarded to the slaves.

Next, the master broadcasts the histogram setup and simulation configuration and each slave begins their own SQS instance. Each slave must use a unique seed for their random number generator. The SQS process at the slave is nearly identical to a single-machine SQS simulation, requiring warmup, calibration and steady-state measurement, except that the slave's calibration phase does not determine the histogram setup. Also, slaves do not determine when the simulation converges; the master monitors the slaves' progress and signals convergence when aggregate sample size is sufficient across the entire cluster.

Once the aggregate sample is large enough, the master collects all the histograms and combines them to form a single estimate. In a number of ways, the master-slave relationship resembles the MapReduce framework [62]—a single program is executed with high fan-out across a number of slave machine (map) with different inputs (the random seed). After completion, their results are then merged (reduce) to form a result.

5.5 Evaluation

In this section, we demonstrate the utility of SQS with a case study of cluster-level power capping. We report on the ability of SQS to scale to simulate many servers, its sensitivity to simulation inputs, and our ability to parallelize simulation.

5.5.1 Case Study - Power Capping

Power capping is technique that allows a data center to deploy more servers than its provisioned power infrastructure can support at peak. It has been observed that—especially in large installations—servers rarely draw peak power concurrently [73, 115, 148, 170]. Because a cluster's aggregate power draw is typically significantly less than the potential

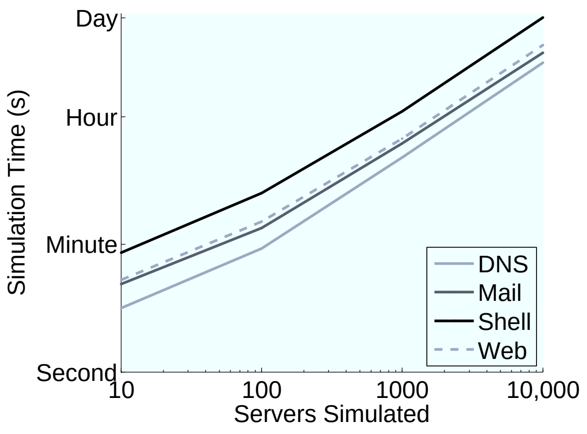


Figure 5.5: Simulation time scaling. Simulation time required for convergence scales roughly linearly with the number of servers simulated. Scaling simulation size typically does not increase the variance of the output variables, so the required sample size does not increase significantly. Instead, the overhead of maintaining the discrete-event-simulation state is the main cause of increased runtime.

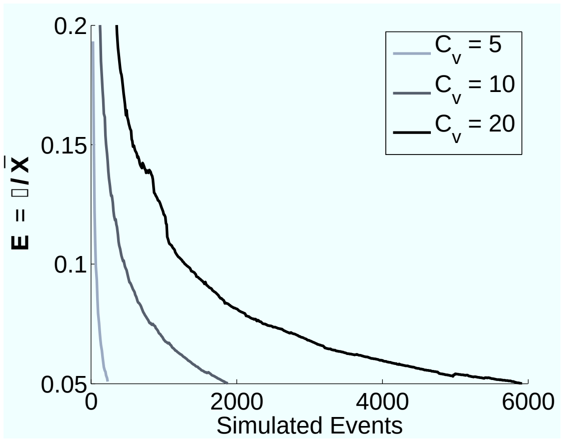


Figure 5.6: Sensitivity to workload distribution variance. Increasing service distribution coeff. of variation (C_v) leads to increased variance in the target variables, requiring a disproportionate increase in simulation time.

sum of all its servers' peak power, provisioning the number of servers based on peak power is wasteful.

To amortize the high cost of power infrastructure, it is desirable to provision servers based on their average power consumption. While such a scheme might work in the common case, rare power spikes across many machines do occur, which can exceed the provisioned capacity of the power infrastructure, blowing a circuit breaker and taking the cluster offline. Power capping solves this problem by assigning hard limits, or "caps", to each server's power consumption. These limits are enforced by throttling a server's performance thereby reducing its power consumption.

To evaluate the scalability of SQS, we model power capping for a server cluster with quad-core machines. Our case study uses a relatively simple power capping scheme; we wish to demonstrate the utility of our methodology rather than explore sophisticated power capping strategies. Servers are assigned a *power budget*, the maximum power they may draw over a given interval. We use a fair, proportional budgeting mechanism such that every server gets a budget in proportion to its current utilization at each budgeting interval. Budgets are calculated every second. At each budgeting epoch, the *capping* level can be observed and is defined as how much more power a server would draw, beyond its budget,

Table 5.1: Power model assumptions.

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

without a cap. We use idealized DVFS as the power-performance throttling mechanism.

Power-Performance Model. To simulate power capping, we require a baseline server power model and a model for power savings and performance loss under DVFS. We use the linear model validated by [73] and [151]:

$$P_{\text{Total}} = P_{\text{Dynamic}} \cdot U + P_{\text{Idle}} \tag{5.4}$$

Where U is the average server utilization, P_{Dynamic} represents the dynamic range of the server's power, and P_{Idle} the idle power. Our power model is based on typical server specification from industry [33] and is summarized in Table 5.1. For simplicity, we assume that the CPU is the only component with a dynamic power range such that:

$$P_{\text{CPU}} \propto \left(\frac{f}{f_{\text{Max}}}\right)^3 \tag{5.5}$$

Where f is the operating clock frequency of the CPU. We assume that this frequency can be continuously scaled from $f = 1.0$ to $f = 0.5$, even though in practice these setting are discrete. The exact scaling of DVFS with respect to frequency has been receiving increasing scrutiny [42]; however, since our focus is on simulator performance rather than power capping efficacy, we assume the classic cubic scaling.

Next, we require a performance model to understand the slowdown imposed by various DVFS settings. The slowdown in service rate due to DVFS can be modeled as:

$$\mu' = \mu \cdot \alpha \cdot \left(\frac{f}{f_{\text{Max}}}\right) + \mu \cdot (1 - \alpha) \tag{5.6}$$

For some α, which represents how "CPU-bound" an application is. We assume an α of 0.9, which would be typical of a CPU-intense application (e.g., LINPACK).

The power model given here is a simple example of the kind of model that can be used with SQS; the particular details of this model are not critical to the simulation approach.

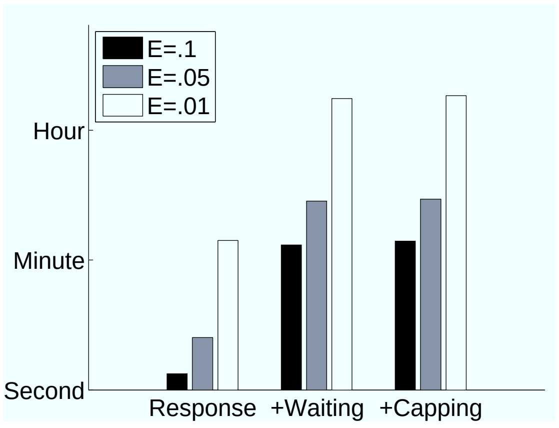


Figure 5.7: Sensitivity to accuracy and target variables Runtime is affected by the selected output variables and the desired confidence intervals. Monitoring response and waiting time (+Waiting) increase simulation time over monitoring response time alone, because most requests do not experience queuing, which makes "waiting" observations more rare. Additionally including power capping as an output variable (+Capping) further increases runtime because capping epochs occur much less frequently than request completions.

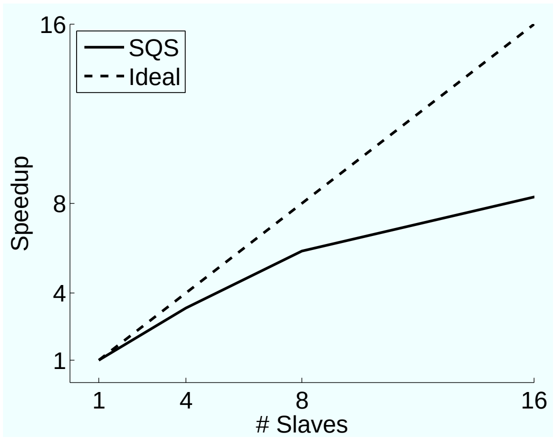


Figure 5.8: Parallel simulation. SQS achieves speedup by parallelizing measurement across multiple slaves. The primary limiting factor to parallel scalability is the calibration phase, which requires 5000 observations for the runs-up test on each slave. Since this simulation requires a sample size just under 40,000, calibration imposes an Amdahl bottleneck.

5.5.2 Performance

Unless otherwise specified, all simulations are run to achieve 95% confidence of E=.05 for both the average value and a 95th-percentile quantile.

In Figure 5.5 we demonstrate how simulation time scales with the size of the simulated cluster. Simulation of a ten-server system is trivial, taking no longer than a minute. As we increase the number of servers, simulation time increases roughly linearly. While simulation time across our workloads varies, the scaling relationship is the same. Even at three orders of magnitude greater cluster size (10,000 servers), simulations take hours rather than days.

It is important to note that the primary cause of increased simulation time is the overhead of maintaining and updating the enlarged state of the discrete-event simulation. The sample size required for convergence, however, depends only on the variance of the output variables and may be reduced slightly due to averaging effects in larger clusters (as in the case of power capping).

For a given system size, simulation time is strongly dependent on the workload model.

To understand this effect, we simulate a system where the workload's service distribution is adjusted to a desired coefficient of variation, C_v (the standard deviation normalized by the mean). We use the response time as the sole output variable because it is most dependent on the C_v parameter. Figure 5.6 shows how the accuracy of an output variable, E, reaches a target value of .05 with the number of simulated events for three values of C_v. For larger values of E, the difference in the number of simulated events across values of C_v is small; however, at .05, the required number of simulated events becomes pronounced. This phenomenon is a direct implication of Equations 5.2 and 5.3—simulation time increases quadratically with increased accuracy and the standard deviation of the worst case across output variables. In our example, the C_v of the service time strongly affects response time variance; however, in more complex systems the relationship may not be as clear.

Finally, we evaluate how the selected output variables impact runtime. We use the same power capping system as before, but vary the set of output variables and their desired accuracy. First, we monitor only response time ("Response"). Increasing the desired accuracy drastically increases runtime, but simulations require at most a few minutes. Adding a wait time ("+Waiting") output variable greatly increases runtime. This increase occurs because wait events are much less frequent than request completion events (i.e., queuing is relatively infrequent). Finally, additionally monitoring power capping ("Capping") results in a further, slight increase in runtime (note that results are on a log scale).

5.5.3 Parallel Simulation

We demonstrate the ability for SQS to parallelize across multiple slaves using our power capping example. We run the simulation with $E = .01$ so that it is sufficiently long to gain benefit from parallel execution. Figure 5.8 demonstrates the speedup gained by using an increasing number of slaves. We distribute the slaves across 4 machines such that each machine has an equal number of slaves (e.g., with 8 total slaves, each machine has 2 slaves).

A system with perfect parallel scaling would achieve a speedup equal to the number of slaves ("Ideal"). SQS demonstrates good scaling up to 8 slaves ("SQS"), but Amdahl effects limit scalability beyond 16 slaves. Each slave must execute a 5000-observation calibration phase to complete the runs-up test. As this particular simulation problem requires a sample size around 40,000, calibration overhead becomes dominant beyond 16 slaves.

5.6 Peak-Power Modeling

Power- and energy-related costs make up almost 50% of data center lifetime costs and are increasing [33]. Whereas energy costs have received significant attention lately, the

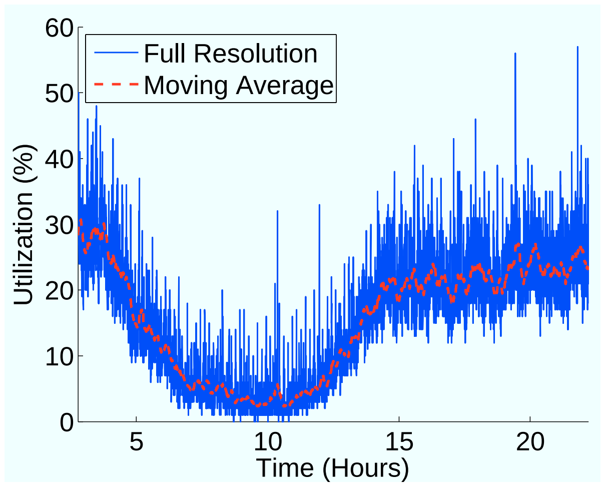


Figure 5.9: Example data center trace: Average utilization masks important spikes.

infrastructure investment required to power thousands of servers has received less attention and remains high [82]. Modeling these systems accurately is critical for large-scale evaluation [124, 140]. Designing power infrastructure requires understanding the aggregate *peak power* of multiple servers at the rack, cluster and data center level. Monitoring the power consumption of individual servers can be costly, requiring power meters at each server. Rack-level monitoring can provide more economic monitoring, but masks individual server behavior. As an alternative to direct measurement, prior work has shown that CPU utilization can provide an accurate proxy for average power, as average utilization is roughly proportional to average power [73, 151]. However, estimation approaches that average utilization at a coarse-grain are not sufficient to predict peak power spikes.

Today, it is not unusual for data center operators to collect utilization traces with sampling intervals of tens of minutes to hours; finer-grained sampling is prohibitive for tens of thousands of servers due to storage and processing overheads. For example, for a 1000-node cluster, sampling at the granularity of the OS scheduler (100Hz) would produce 225 GB of data per week.

Figure 5.9 shows the utilization of a production data center server running a web 2.0 service. The trace was collected at ten-second granularity ("Full Resolution") and has a wide dynamic range. Unfortunately, most utilization traces are not collected with such fine detail. When a ten-minute average is used instead ("Moving Average"), significant detail is lost. For example, though there appears to be little demand around hour ten when examining the coarse average, the fine-grain trace shows there are still brief spikes that exceed the maximum of the coarse trace.

We show that to determine a server's peak power, it is critical to understand the behavior of server switched-mode power supply units (SMPSUs). These devices are highly efficient,

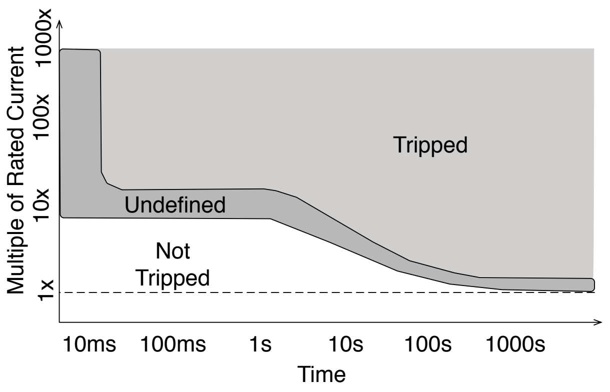


Figure 5.10: Example PDU circuit breaker curve.

but rely on a switching and charge storage mechanism that introduces RC behavior into the power draw. While SMPSUs are well understood, our contribution is to connect the operating system view of a server to the peak power draw at the power outlet. We construct an analytic model of a server's power draw that can be understood using signal processing techniques.

Finally, we introduce an easily-collected operating system-level metric that can be used determine peak power draw over a time epoch. By leveraging our model, we are able to incorporate the RC behavior of SMPSUs and track peak power with low overhead. This mechanism can enable logging of peak power over time and will facilitate large-scale data center power-provisioning research.

5.6.1 Data Center Power Provisioning

Provisioning power infrastructure for data centers is extremely costly; typical installations incur \$10-\$20 per provisioned watt [33]. A large fraction of this cost is associated with installing power distribution units (PDUs), which provide power to groups of servers. Often, total PDU capacity is overprovisioned [73, 114, 141]. Data center designers typically use conservative estimates for the maximum power draw of servers. However, in aggregate, racks and clusters of servers rarely draw their peak power at the same time [73]. At the PDU level, this conservatism means that PDUs are rarely fully loaded; the provisioned capacity at each PDU is well above its average load.

One method to reduce power infrastructure cost is to *oversubscribe* a data center's power infrastructure with more servers than it can support [73]. Oversubscribing power infrastructure introduces the possibility of exceeding the maximum rated power for a PDU; this scenario can throw a circuit breaker and take a section of the data center offline. Figure 5.10 depicts a simplified version of a typical PDU circuit breaker curve [60]. Tripping a breaker is not an instantaneous event; the PDU can tolerate brief current overloads. However, since several servers might incur power spikes at the same time, to maintain availability, a design must guarantee that the total power draw at each server remain below a predetermined limit.

Power capping is a data center-level technique to set hard limits on servers' peak power consumption (e.g., using a control loop) [73, 114]. Throttling server power (via frequency/voltage scaling) is used as a safety mechanism to ensure maximum power levels are not exceeded and circuit breakers are not tripped. With this mechanism in place, PDUs and other power provisioning infrastructure can be oversubscribed, reducing the effective capital cost. Since load/power spikes are rare, little performance is lost to throttling. Capital costs can be further reduced by using *Power Routing* [141], which allows load to be shifted among PDUs during imbalances.

All of these techniques require software mechanisms to track and predict peak power, to manage power budgets at each server, circuit, and PDU, while minimizing performance throttling. Though peak power could be tracked with explicit metering and logging, assessing peak power directly from operating system-level metrics can drastically reduce costs. To infer and record peak power from OS level metrics, we must understand the operation of server power supplies and its relationship to utilization.

5.6.2 Understanding SMPSU Behavior

In this section, we explore the behavior of SMPSU devices in servers and its connection to OS-observed utilization. To ensure our observations generalize, we study two different kinds of systems: a smaller system with a cheap, commodity PSU ("Commodity") and a larger system with an enterprise class PSU ("Server"). Since SMPSU designs vary, these systems exhibit some differences in behavior; nevertheless, aspects relevant to predicting peak power draw are similar. First, we briefly describe the operation of SMPSU devices; a detailed description of such devices may be found in [145]. Next, we describe our experimental methodology for characterizing these devices and measuring the important behaviors of SMPSUs. Finally, we develop a high-level signal processing model to predict peak power.

5.6.2.1 Operation

Modern servers use some form of SMPSU to convert from 120/240V AC to 12V DC power. SMPSUs are far more efficient than, for example, linear regulators, but are also more complicated in their design. While the design and operation of these devices is well understood, our contribution is to understand how the processor's logical view of utilization

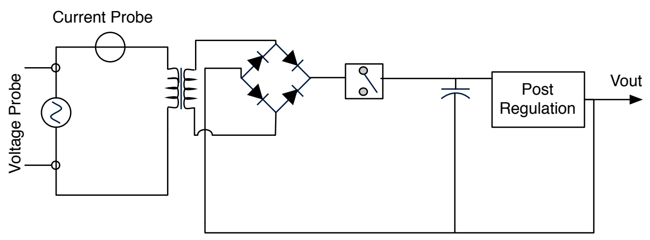


Figure 5.11: Simplified switched-mode power supply design and instrumentation to measure power.

maps to the physical power draw at an outlet. This connection is important because, as we show, the design of typical SMPSU devices impacts the manner in which we should model server power.

Figure 5.11 illustrates the topology of a typical SMPSU. In the first stage, line AC voltage is rectified and passed to a storage element (i.e., a capacitor). The second stage typically includes some form of regulation to maintain a DC voltage. As the demands of the DC devices powered by the SMPSU change, the SMPSU controller adjusts the duty cycle of its switching to transfer more or less charge.

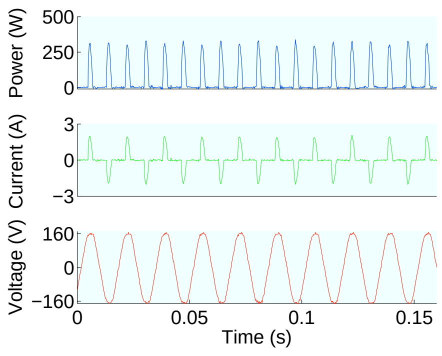
Because of its design, a SMPSU does not draw power continuously. Instead, there are spikes of current during each charging cycle. During these spikes, current is transferred from the high-voltage supply to the SMPSU capacitor. Example measurements of power draw in idle systems are shown in Figure 5.12. While the basic principle of operation is the same, the Commodity PSU clearly transfers current in more pronounced spikes than the Server. This difference is due to extra switching regulation in the first stage, common in higher-end devices, used to produce a more continuous current draw.

Because of the capacitor used to store and transfer charge, this circuit exhibits RC behavior. We would like to know the effect a typical SMPSU has on the frequency response and phase of power consumption with respect to processor utilization. Such an understanding will allow us to better determine at what granularity we must track utilization.

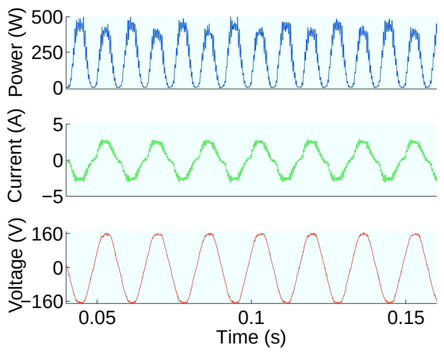
5.6.2.2 Experimental Methodology

We measure the power consumed by a server at the wall outlet to observe its behavior with respect to utilization. We accomplish this by simultaneously measuring the instantaneous voltage over and current entering the PSU as illustrated in Figure 5.11. A simple power probe is not sufficient for this measurement because these devices typically report average RMS power, masking the phenomenon we are attempting to observe. We record detailed traces of the instantaneous signals from both these probes.

We measure the two machine configurations described earlier: a smaller, inexpensive



(a) **Commodity**



(b) **Server**

Figure 5.12: Systems at idle. Switch-mode power supplies draw power in discrete spikes.

system ("Commodity") and an enterprise, dual socket system ("Server"). Comparing these systems allow us to determine if the size or price class of the machines influences their behavior. The measured idle and maximum power consumptions of these machines are provided in Table 5.2.

To understand how utilization and the PSU interact, we wish to characterize two effects. First, we investigate how the frequency at which utilization varies is reflected in the power draw at the wall outlet. Intuitively, we expect that utilization variations that occur faster than some cut-off frequency will be filtered by the PSU behavior and not be reflected at the outlet; measuring utilization at a granularity finer than this cut-off is not necessary for accurate peak power prediction. The precise cut-off frequency has not previously been characterized. Second, we wish to determine the latency between utilization changes and a corresponding change in the SMPSU power draw; in other words, how rapidly a step function in utilization affects power draw at the PSU.

To observe the effect of the frequency of utilization variation, we use a synthetic workload we refer to as SQUARE. This workload produces a square wave in system utilization by switching cores between a matrix multiplication kernel designed to maximize CPU power draw and an idle mode where the processors enter a power-save mode. The duty cycle of the workload is fixed at 50%, producing an average utilization of 50%. We vary the frequency of the square wave and observe the response at the PSU.

To characterize the latency between a utilization change and the PSU response, we idle the system and wait until the PSU behavior reaches steady state. We then trigger execution of the matrix multiply kernel on all cores. We refer to this synthetic workload as STEP. Because CPU utilization is not directly observable externally, we send a signal (using general purpose I/O that is significantly faster than the expected SMPSU response) immediately before the transition to initiate timing at our oscilloscope.

	RMS Power (W)		Dyn. Range (max/min)
	Idle	Max	
Commodity	57 W	188 W	3.3
Server	212 W	355 W	1.7

Table 5.2: Systems under test.

5.6.2.3 Measured Behavior

We now present results for frequency and phase delay behavior.

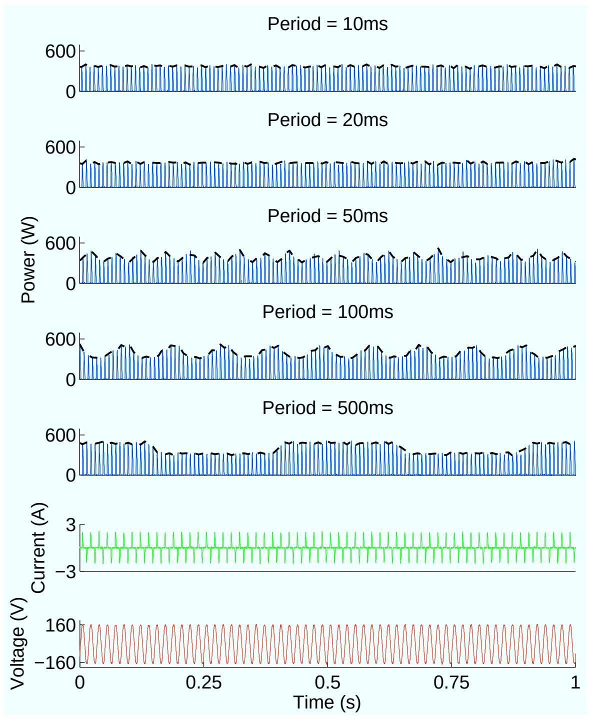
Frequency Response. To understand the relationship between the frequency of utilization and power, we ran the SQUARE benchmark on both test systems with varying frequencies. Figure 5.13 shows the observed instantaneous power at each frequency on both systems. We used 100 Hz as the maximum frequency we investigate because we found that the Linux kernel could not reliably schedule faster than this frequency (in general this will depend on the OS kernel configuration). The current draw and voltage of an idle system are provided for reference. The dotted line ("Envelope"), connects the peaks of the power waveform and functions as an envelope detector. The varying utilization modulates the instantaneous power waveform of a system at idle; the envelope detector reveals the modulated signal.

The results in Figure 5.13 show that the frequency of modulation has a strong influence on the observed power waveform. As long as the utilization of the CPU is modulated slowly, the envelope of power draw roughly resembles a square wave, matching the CPU behavior. However as the frequency is increased, the power draw becomes more uniform.

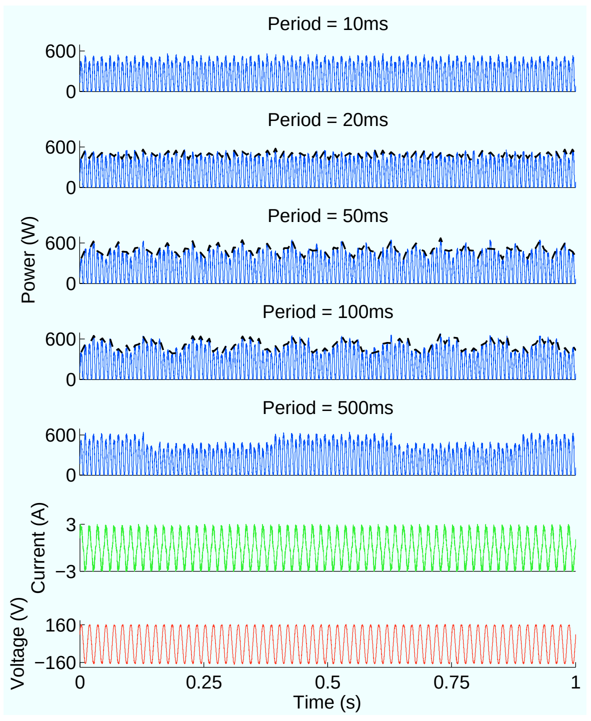
We draw several conclusions from this result. First, the SMPSU effectively acts as a low-pass filter with respect to utilization. We construct a model for this behavior in Section 5.6.2.4. Second, to faithfully model the peak power of an SMPSU, it is necessary to monitor utilization at fine granularity (near the kernel scheduling interval for many systems). Averages that use coarser windows lose information. However, monitoring utilization at a time-scale finer than 50 Hz is unnecessary: the variations in the 50 Hz (20 ms period) and 100 Hz (10 ms period) waveforms are filtered.

To give a better sense of the filtering in the SMPSU system, we construct a Bode-style plot of the systems in Figure 5.15. The figure illustrates the attenuation of the modulating signals. We show the Commodity and Server frequency responses compared to a idealized first-order RC low-pass filter ("Ideal"). We find that these systems are closely approximated by a filter with a frequency cutoff of 30Hz.

Phase Delay. Next, we investigate the phase delay of SMPSU power load using the STEP workload. The step function response of both test machines is provided in Figure

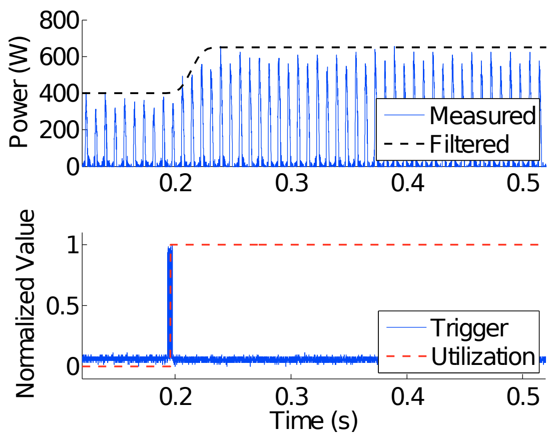


(a) **Commodity**

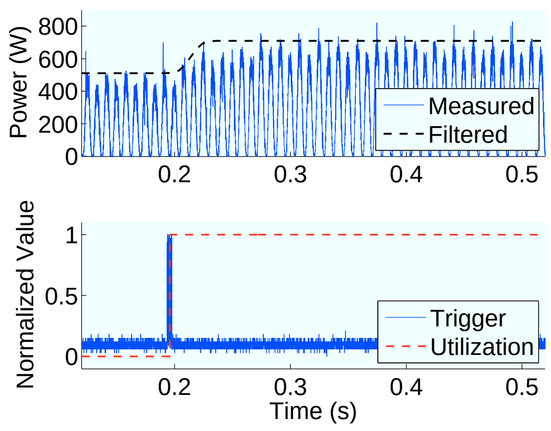


(b) **Server**

Figure 5.13: Effect of modulation frequency. All examples have the same *average* utilization, but exhibit different *peak* power.



(a) **Commodity**



(b) **Server**

Figure 5.14: Delay of a step function in utilization.

5.14. There is a delay in the instantaneous power response, which rises as one would expect of a step function with RC filtering. We report the I/O signal indicating the utilization transition ("Trigger"), as well as the implied utilization waveform ("Utilization"). Finally, we show a filtered ("Filtered") step function that fits the observed rising waveform. This signal is produced from a first order RC filter with a frequency cutoff of 30 Hz.

5.6.2.4 Model

The goal of our investigation has been to model the peak power draw of SMPSU for server systems. Accordingly, we now construct an analytic model of the PSU behavior using signal processing. We start with the observation that the PSU is effectively exhibiting *amplitude modulation* (AM). An idle system demonstrates the carrier signal: the periodic spikes in current consumption. This signal is then modulated by changes in utilization. The block system diagram for our model is illustrated in Figure 5.16.

We can describe the observed power draw of an SMPSU as a signal:

$$p_{\text{wall}}(t) = p_{\text{dyn}} \cdot c(t) \cdot (h(t) * x(t)) + p_{\text{idle}} \tag{5.7}$$

Where $p_{\text{wall}}(t)$ is the observed time varying power consumption, $c(t)$ is the SMPSU current carrier waveform, $h(t)$ is the transfer function for a low-pass filter and $x(t)$ is the *instantaneous* fractional utilization. Note that $x(t)$ can only take on one of 2^N values, where N is the number of cores in the system. Finally, p_{dyn} and p_{idle} are constants that are the same as in Equation 1 and are provided for our system in Table 1. While we use a relatively simple power model, if more sophisticated models are needed (e.g., to model the use of low-power modes), they can easily fit within this framework; we leave such extensions to future work.

We have observed that a first order low-pass filter is quite accurate; therefore, the transfer function is:

$$h(t) = \frac{1}{\tau} e^{-t/\tau} \tag{5.8}$$

Where τ is the time constant and is approximately 33 ms for our system (alternatively, the cutoff frequency is 30 Hz).

5.6.3 Server Peak Power Accounting

A key result of our measurement study is that utilization must be monitored at a granularity below 30 Hz to predict peak power. However, finer-grained variation is filtered by the RC behavior of the power supply and need not be monitored. With our new understanding of the operation of SMPSUs and their relationship with server utilization, we construct a

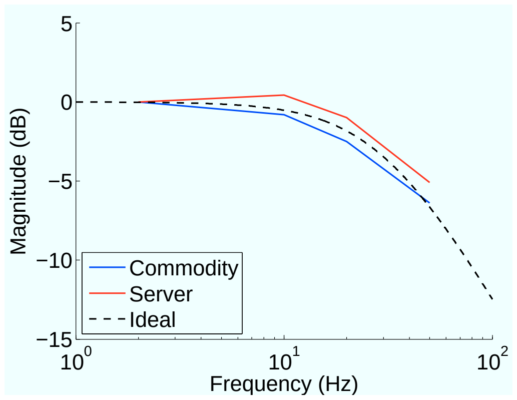


Figure 5.15: Frequency response.

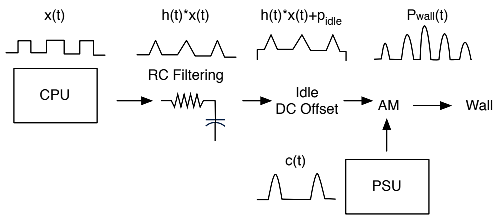


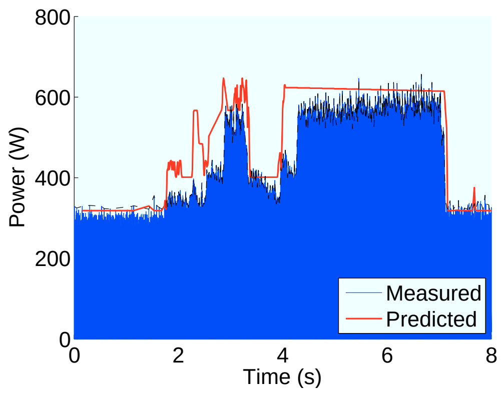
Figure 5.16: Simplified Server SMPSU Model.

low-overhead method to infer peak power from utilization in the operating system kernel. We then validate our model using real machines and show we can predict the peak power trace with an error below 20%.

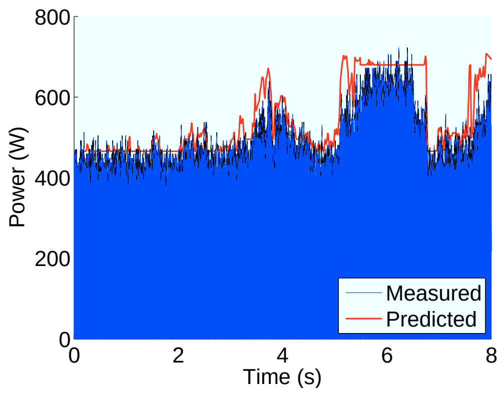
5.6.3.1 A Compact Metric for Peak Power

We propose a new operating system-level metric to track spikes in peak power. We use the model presented in Section 5.6.2.4 to filter fine-grain utilization signals (idle/busy transitions) collected in the OS scheduler to determine and record maximum power draw over a time epoch T. To obtain accurate estimates of power draw spikes, we must know the RC response of a particular PSU, which may vary among PSUs. (However, the fact that the two PSUs we study, which differ drastically in their design, exhibit similar RC response provides some evidence that other PSUs will have similar behavior). Over the course of each epoch, we evaluate the estimated power at each sampling interval and retain the peak value.

Most current releases of the Linux kernel are tickless [160]; that is, they operate without a periodic timer interrupt, and can have a variable scheduling interval. Variations in the length of idle/busy periods complicate construction of the input utilization signal to our filter-based model; we must correct for these variations prior to calculation. Note that



(a) **Commodity**



(b) **Server**

Figure 5.17: Predicted peak power closely follows measured value.

these corrections do not lose information, as idle/busy transitions cannot occur without invoking the scheduler. We detect scheduler transitions at each core and compute utilization in sampling intervals of 4 ms each.

At the operating system level, the view of a processor is binary; in other words, it tracks if in a given scheduling intervals, measured in *jiffies*, the processor was active or idle. In fact, because floating point operations are not available in the Linux kernel, the number of idle and active jiffies are collect and most averaging is performed by user-level tools such as top. In order to determine the peak power, it may seem reasonable to collect the peak utilization during a time interval or to count the longest series of consecutive active jiffies. However, as shown earlier, because of the RC behavior of the system, these are not sufficient. A brief spike of 100% utilization for a jiffy, will be filtered and well short of 100% of power draw. Similarly, a system that oscillates between 100% and 0% utilization at a high frequency will be indistinguishable from one which is 50% utilized. Therefore, a metric that helps us know peak power must take the RC behavior of the system into account

To construct an estimator for peak power, we transform the utilization signal using an in-kernel finite impulse response (FIR) filter of the form:

$$y[n] = \sum_{i=0}^{N} b_i x[n-i] \tag{5.9}$$

This processing allows us to model the RC behavior of the PSU. Since our tracking and processing takes place in the scheduling subsystem of the OS kernel, it must be light-weight and use fixed point arithmetic [179]. We have found that a 10th order FIR filter captures the behavior well. This filter can compute our metric easily, it requires only the last 10 utilization observations and 10 multiply-accumulate operations per update.

5.6.3.2 Validation

We validate our models against the power consumption of the two server configurations presented in Table 5.2. Two representative traces of measured power are presented in Figure 5.17. These traces were collected while the system executed a parallel compile of the Linux kernel, a workload that produces a chaotic, bursty utilization pattern. The instantaneous power ("Measured") is measured the same way as described in Section 4.2.

We overlay our predicted power ("Predicted"), which tracks peak power well, but can still overshoot occasionally. Fortunately, this model tends to be conservative, and overestimates power more than it underestimates. Hence, it will provide conservative estimates in, for example, studies of power budgeting/capping. In this example, the Commodity and Server machines exhibit a normalized root mean square deviation (NRMSD) of 14% and 19% respectively.

CHAPTER 6

The PowerNap Server Architecture - A Coordinated Idle Low-Power Mode

In this chapter, we propose an energy-conservation approach, called *PowerNap*, that is attuned to server utilization patterns. With PowerNap, we design the entire system to transition rapidly between a high-performance active state and a minimal-power nap state in response to instantaneous load. Rather than requiring components that provide fine-grain power-performance trade-offs, PowerNap simplifies the system designer's task to focus on two optimization goals: (1) optimizing energy efficiency while napping, and (2) minimizing transition time into and out of the low-power nap state.

Based on the PowerNap concept, we develop requirements and outline mechanisms to eliminate idle power waste in a high-density blade server system. Through modeling and analysis of actual data center workload traces, we demonstrate:

- **Energy efficiency and response time bounds.** Through queuing analysis, we establish bounds on PowerNap's energy efficiency and response time impact. Using our models, we determine that PowerNap is effective if state transition time is small compared to the average request service time. For the workloads we evaluate, transition time should be below 10ms, and incurs no overheads below 1ms. Furthermore, we show that PowerNap provides greater energy efficiency and lower response time than solutions based on DVFS.

- **Experimental validation of response time impact.** By instrumenting a kernel to emulate PowerNap's transition delays, we validate the response time predictions of our analytic model, confirming that neither CPU caching effects, nor 1ms transitions significantly impact workload response time.

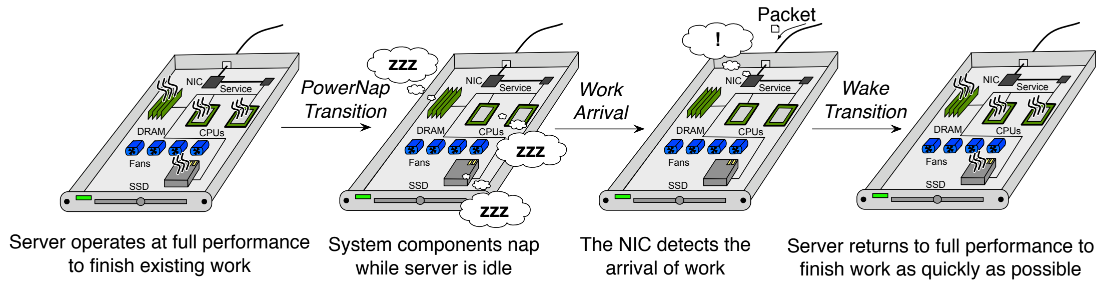

NIC
Service
DRAM
CPUs
Fans
SSD
PowerNap Transition
zzz
Work Arrival
Packet
!
Wake Transition
Server operates at full performance to finish existing work
System components nap while server is idle
The NIC detects the arrival of work
Server returns to full performance to finish work as quickly as possible

Figure 6.1: PowerNap.

6.1 PowerNap

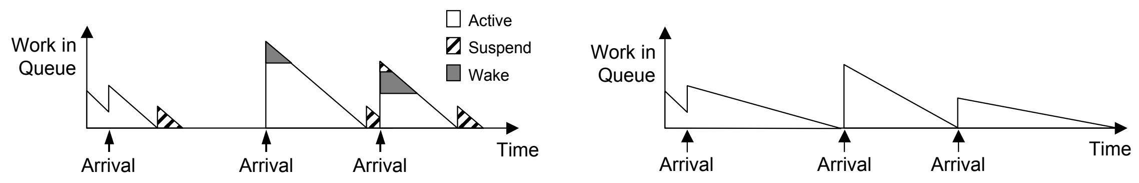
Although servers spend most of their time idle, conventional energy-conservation techniques are unable to exploit these brief idle periods. Hence, we propose an approach to power management that enables the entire system to transition rapidly into and out of a low-power state where all activity is suspended until new work arrives. We call our approach *PowerNap*.

Figure 6.1 illustrates the PowerNap concept. Each time the server exhausts all pending work, it transitions to the nap state. In this state, nearly all system components enter sleep modes, which are already available in many components (see Section 6.3). While in the nap state, power consumption is low, but no processing can occur. System components that signal the arrival of new work, expiration of a software timer, or environmental changes, remain partially powered. When new work arrives, the system wakes and transitions back to the active state. When the work is complete, the system returns to the nap state.

PowerNap is simpler than many other energy conservation schemes because it requires system components to support only two operating modes: an active mode that provides maximum performance and a nap mode that minimizes power draw. For many devices, providing a low-power nap mode is far easier than providing multiple active modes that trade performance for power savings. Any level of activity often implies fixed power overheads (e.g., bus clock switching, power distribution losses, leakage power, mechanical components, etc.) We outline mechanisms required to implement PowerNap in Section 6.3.

6.1.1 PowerNap Performance and Power Model

To assess PowerNap's potential, we develop a queuing model that relates its key performance measures—power consumption and response time penalty—to workload parameters and PowerNap implementation characteristics. We contrast PowerNap with a model of the upper-bound energy-savings possible with DVFS. The goal of our model is threefold: (1) to gain insight into PowerNap behavior, (2) to derive requirements for PowerNap



(a) PowerNap **(b) DVFS**

Figure 6.2: PowerNap and DVFS analytic models.

implementations, and (3) to contrast PowerNap and DVFS.

We model both PowerNap and DVFS under the assumption that each seeks to minimize the energy required to serve the offered load. Hence, both schemes provide identical throughput (matching the offered load) but differ in response time and energy consumption.

PowerNap Model. We model PowerNap as an M/G/1 queuing system with arrival rate λ, and a generalized service time distribution with known first and second moments $E[S]$ and $E[S^2]$. Figure 6.2(a) shows the work in the queue for three job arrivals. Note that, in this context, work also includes time spent in the wake and suspend states. Average server utilization is given by $\rho = \lambda E[S]$. To model the effects of PowerNap suspend and wake transitions, we extend the conventional M/G/1 model with an exceptional first service time [172]. We assume PowerNap transitions are symmetric with latency T_t. Service of the first job in each busy period is delayed by an initial setup time I. The setup time includes the wake transition and may include the remaining portion of a suspend transition as shown for the rightmost arrival in Figure 6.2(a). Hence, for an arrival x time units from the start of the preceding idle period, the initial setup time is given by:

$$I = \begin{cases} 2T_t - x & \text{if } 0 \leq x < T_t \\ T_t & \text{if } x \geq T_t \end{cases}$$

The first and second moments $E[I]$ and $E[I^2]$ are:

$$\begin{aligned} E[I] &= \int_0^\infty I \lambda e^{-\lambda x} dx = 2T_t + \frac{1}{\lambda} e^{-\lambda T_t} - \frac{1}{\lambda} \\ E[I^2] &= \int_0^\infty I^2 \lambda e^{-\lambda x} dx \\ &= 4T_t^2 - 2T_t^2 e^{-\lambda T_t} - \left(\frac{4T_t}{\lambda} + \frac{2}{\lambda^2} \right) \left[1 - (1 + \lambda T_t) e^{-\lambda T_t} \right] \end{aligned}$$

We compute average power as

$$P_{avg} = P_{nap} \cdot Pr(\text{nap}) + P_{max}(1 - Pr(\text{nap}))$$

where the fraction of time spent napping $Pr(\text{nap})$ is given by the ratio of the expected length of each nap period $E[N]$ to the expected busy-idle cycle length $E[C]$:

$$\begin{aligned} Pr(\text{nap}) &= \frac{\int_0^{T_t}(0)\lambda e^{-\lambda t}dt + \int_{T_t}^{\infty}(t - T_t)\lambda e^{-\lambda t}dt}{\frac{E[S]+E[I]}{1-\lambda E[S]} + \frac{1}{\lambda}} \\ &= \frac{e^{-\lambda T_t}(1 - \lambda E[S])}{1 + \lambda E[I]} \end{aligned}$$

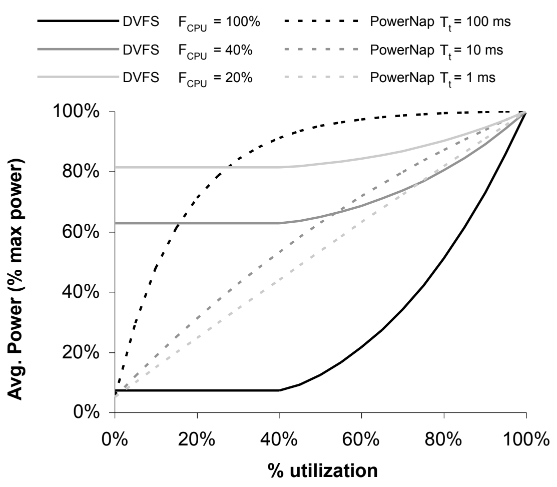
The response time for an M/G/1 server with exceptional first service is due to Welch [172]:

$$E[R] = \frac{\lambda E[S^2]}{2(1-\lambda E[S])} + \frac{2E[I]+\lambda E[I^2]}{2(1+\lambda E[I])} + E[S]$$

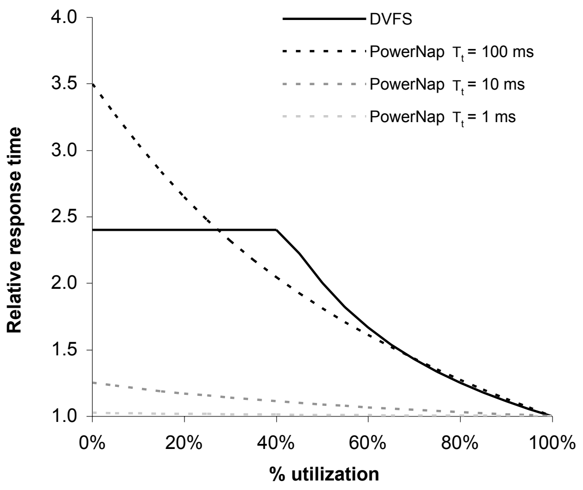
Note that the first term of $E[R]$ is the Pollaczek-Khinchin formula for the expected queuing delay in a standard M/G/1 queue, the second term is additional residual delay caused by the initial setup time I, and the final term is the expected service time $E[S]$. The second term vanishes when $T_t = 0$.

DVFS model. Rather than model a real DVFS frequency control algorithm, we instead model the upper bound of energy savings possible with DVFS. For each job arrival, we scale instantaneous frequency f to stretch the job to fill any idle time until the next job arrival, as illustrated in Figure 6.2(b), which gives $E[f] = f_{max}\rho$. This scheme maximizes power savings, but cannot be implemented in practice because it requires knowledge of future arrival times. We base power savings estimates on the theoretical formulation of processor dynamic power consumption $P_{CPU} = \frac{1}{2}CV^2Af$. We assume C and A are fixed, and choose the optimal f for each job within the range $f_{min} < f < f_{max}$. We impose a lower bound $f_{min} = f_{max}/2.4$ to prevent response time from growing asymptotically when utilization is low. We chose a factor of 2.4 between f_{min} and f_{max} based on the frequency range provided by a 2.4 GHz AMD Athlon. We assume voltage scales linearly with frequency (i.e., $V = V_{max}(f/f_{max})$), which is optimistic with respect to current DVFS implementations. Finally, as DVFS only reduces the CPU's contribution to system power, we include a parameter F_{CPU} to control the fraction of total system power affected by DVFS. Under these assumptions, average power P_{avg} is given by:

$$P_{avg} = P_{max}(1 - F_{CPU}(\frac{E[f]}{f_{max}})^3)$$



(a) Power Scaling



(b) Response Time Scaling

Figure 6.3: PowerNap and DVFS power and response time scaling.

Response time is given by:

$$E[R] = E\left[\frac{R_{base}}{f}\right]$$

where R_{base} is the response time without DVFS.

6.1.2 Analysis

Power Savings. Figure 6.3(a) shows the average power (as a fraction of peak) required under PowerNap and DVFS as a function of utilization. For DVFS, we show power savings for three values of F_{CPU}. F_{CPU} = 100% represents the upper bound if DVFS were applicable to all system power. $20\% < F_{CPU} < 40\%$ bound the typical range in current servers. For PowerNap, we construct the graphs with $E[s] = 38ms$ and $E[s^2] = 3.7E[s]$, which are both estimated from the observed busy period distribution in our Web trace. We assume P_{nap} is 5% of P_{max}. We vary λ to adjust utilization, and present results for three values of T_t: 1ms, 10ms, and 100ms. We expect 10ms to be a conservative estimate for achievable PowerNap transition time. For transition times below 1ms, transition time becomes negligible and the power savings from PowerNap varies linearly with utilization for all workloads. We discuss transition times further in Section 6.3.

When F_{CPU} is high, DVFS clearly outperforms PowerNap, as it provides cubic power savings while PowerNap's savings are at best linear in utilization. However, for realistic values of F_{CPU} and transition times in our expected range ($T_t \leq 10ms$), PowerNap's savings rapidly overtake DVFS. As transition time increases, the break-even point between DVFS and PowerNap shifts towards lower utilization. Even for a transition time of 100 ms, PowerNap can provide substantial energy savings when utilization is below 20%.

Response time. In Figure 6.3(b), we compare the response time impact of DVFS and PowerNap. The vertical axis shows response time normalized to a system without power management (i.e., that always operates at f_{max}). For DVFS, response time grows rapidly when the gap between job arrivals is large, and reaches the f_{min} floor below 40% utilization. DVFS response time penalty is independent of F_{CPU}, and is bounded at 2.4 by the ratio of f_{max}/f_{min}. For PowerNap, the response time penalty is negligible if T_t is small relative to average service time E[S], which we expect to be the common case (i.e., most jobs last longer than 10ms). However, if T_t is significant relative to E[S], the PowerNap response time penalty grows as utilization shrinks. When utilization is high, the server is rarely idle and few jobs are delayed by transitions. As utilization drops, the additional delay seen by each job converges to T_t (i.e., every job must wait for wake-up).

Per-Workload Energy Savings. Finally, we report the energy savings under simulated PowerNap and DVFS schemes for our workload traces. Because these traces only contain busy and idle periods, and not individual job arrivals, we cannot estimate response time impact. For each workload, we perform a trace-based simulation that assumes busy periods will start at the same time, independent of the current PowerNap state (i.e., new work still arrives during wake or suspend transitions). We assume a PowerNap transition time of 10ms and nap power at 5% of active power, which we believe to be conservative estimates (see Section 6.3). For DVFS, we assume $F_{CPU} = 25\%$. Table 6.1 shows the results of these simulations. All workloads except Mail and Cluster hit the DVFS frequency floor, and, hence, achieve a 23% energy savings. In all cases, PowerNap achieves greater energy savings. Additionally, we extracted the average arrival rate (assuming a Poisson arrival process) and compared the results in Table 6.1 with the M/G/1 model of $Pr(\text{nap})$ derived above. We found that for these traces, the analytic model was within 2% of our simulated results in all cases. When arrivals are more deterministic (e.g., Backup) than the exponential we assume, the model slightly overestimates PowerNap savings. For more variable arrival processes (e.g., Shell), the model underestimates the energy savings.

6.1.3 Implementation Requirements

Based on the results of our analytic model, we identify two key PowerNap implementation requirements:

Table 6.1: Per-workload energy savings and response time penalty.

Workload	PowerNap		DVFS	
	Energy Savings	Δ Latency	Energy Savings	Δ Latency
Cluster	34%	0.2%	18%	156%
DNS	77%	5.1%	23%	240%
Mail	35%	11%	21%	181%
Shell	55%	13%	23%	240%
Web	59%	13%	23%	240%
Backup	61%	7.6%	23%	240%

Fast transitions. Our model demonstrates that transition speed is the dominant factor in determining both the power savings potential and response time impact of PowerNap. Our results show that transition time must be less than one tenth of average busy period length. Although a 10ms transition speed is sufficient to obtain significant savings, 1ms transitions are necessary for PowerNap's overheads to become negligible. To achieve these transition periods, a PowerNap implementation must preserve volatile system state (e.g., memory) while napping—mass storage devices transfer rates are insufficient to transfer multiple GB of memory state in milliseconds.

Minimizing power draw in nap state. Given the low utilization in most enterprise deployments, servers will spend a majority of time in the nap state, making PowerNap's power requirements the key factor affecting average system power. Hence, it is critical to minimize the power draw of napping system components. As a result of eliminating idle power, PowerNap drastically increases the range between the minimum and maximum power demands on a blade chassis. Existing blade-chassis power-conversion systems are inefficient in the common case, where all blades are napping. Hence, to maximize PowerNap potential, we must re-architect the blade chassis power subsystem to increase its efficiency at low loads.

Although PowerNap requires system-wide modifications, it demands only two states from each subsystem: active and nap states. Hence, implementing PowerNap is substantially simpler than developing energy-proportional components. Because no computation occurs while napping, many fixed power draws, such as clocks and leakage power, can be conserved.

6.2 Emulating PowerNap Transition Performance Impact

The PowerNap architecture can impact application response time in two ways: transitions in and out of the nap state delay responses and some processors may flush on-chip caches when transitioning. To investigate these effects in greater detail, we have instrumented a Linux kernel to insert transition delays and flush CPU caches when exiting from idle, emulating PowerNap's performance impact. Using this emulation, we have examined PowerNap's impact on the response time of a web serving benchmark.

6.2.1 Cache Effects

The static power of processor caches consumes a large and potentially growing fraction of overall CPU power budget, particularly when idle. Accordingly, sleep modes available in some CPUs may turn off caches, flushing their contents. The ACPI standard leaves it unspecified whether cache contents are preserved during ACPI sleep states, and implementations vary across vendors and processor generations. We wish to characterize the performance impact of discarding cache contents during PowerNap transitions, to determine if it is important for PowerNap to use only cache-state-preserving sleep modes.

To produce the effect of flushing the cache, we instrument the kernel to issue the x86 `WBINVD` instruction (which writes back and then invalidates the entire contents of CPU caches [6]) when emulating a PowerNap transition. We have tested our modified kernel using a microbenchmark that strides over L1 and L2-sized data structures to confirm that the `WBINVD` instruction discards the contents of both the L1 and L2 caches. We test the effect of flushing the cache each time the server becomes idle (i.e., upon entry to the OS idle loop) for the SPECweb and SPECpower benchmarks [16, 17]. Figure 6.4 show's that the average response time for these benchmarks does not change appreciably as the cold-start cache effect is small relative to the average response time.

6.2.2 Transition Latency

We further investigate the impact of PowerNap transition time to understand how various values of T_t affect a workload. To emulate a wide spectrum of delays, we instrument the Linux kernel to artificially insert delays when exiting the idle loop. The instrumentation tracks the time since the end of the last job such that the delay is I as described in the model in Section 3 (i.e., it accounts for both sleep and wake transitions, falling between T_t and $2T_t$).

Figure 6.5 reports the average response time of SPECweb and SPECpower for a T_t of 1, 10 and 100ms including cache flush effects. Our measurements confirm the model pre-

Figure 6.4: Cache effect.

Benchmark	Relative Response Time
SPECweb	1.01
SPECpower	1.00

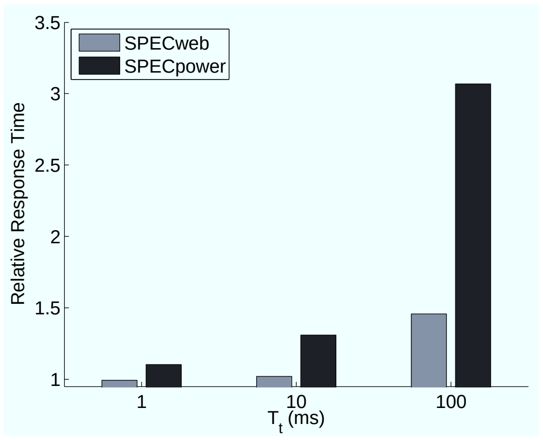


Figure 6.5: Emulating PowerNap.

dictions, showing that a 100ms transition time has a considerable response time impact. However, a T_t of 10ms results in tolerable delay and 1ms incurs a negligible performance impact. Furthermore, we see that SPECpower is more sensitive to transition latency because of it's shorter average service time.

6.3 PowerNap Mechanisms

We outline the design of a PowerNap-enabled blade server system and enumerate required implementation mechanisms. PowerNap requires nap support in all hardware subsystems that have non-negligible idle power draws, and software/firmware support to identify and maximize idle periods and manage state transitions.

6.3.1 Hardware Mechanisms

At the component level, the sleep states required by PowerNap are already available in many products, particularly those targeted to mobile devices. However, few of these mechanisms are exploited in existing servers, and some are omitted in current-generation server-class components. Moreover, the operating system APIs that control sleep/wake transitions in current desktops and laptops introduce enormous overheads that dominate the transition latency, making them inapplicable for PowerNap.

For each hardware subsystem, we identify existing mechanisms or outline requirements for new mechanisms necessary to implement PowerNap. Furthermore, we provide estimates of power dissipation while napping and transition speed. We summarize these estimates, along with our sources, in Table 6.2. Our estimates for a ”Typical Blade” are based on HP's c-series half-height blade designs; our PowerNap power estimate assumes a two-

Table 6.2: Component power consumption.

Component	Power			Transition	Sources
	Active	Idle	Nap		
CPU chip	80-150W	12-20W	3.4W	30 μs	[95] [94]
DRAM DIMM	3.5-5W	1.8-2.5W	0.2W	$< 1\mu$s	[128] [93]
NIC	0.7W	0.3W	0.3W	no trans.	[161]
SSD	1W	0.4W	0.4W	no trans.	[153]
Fan	10-15W	1-3W	-	independent	[116]
PSU	50-60W	25-35W	0.5W	300 μs	[156]
Typical Blade	450W	270W	10.4W	300 μs	

CPU system with eight DRAM DIMMs.

Processor: ACPI S3 "Sleep" state. The ACPI standard defines the S3 "Sleep" state for processors that is intended to allow low-latency transitions. Although the ACPI standard does not specify power or performance requirements, some implementations of S3 are ideal for PowerNap. For example, in Intel's mobile processor line, S3 preserves last-level cache state and consumes only 3.4W [95]. These processors require approximately 30 μs for PLL stabilization to transition from sleep back to active execution [94].

If S3 is unavailable, clock gating can provide substantial energy savings. For example, Intel's Xeon 5400-series power requirements drop from 80W to 16W upon executing a halt instruction [96]. From this state, resuming execution requires only nanosecond-scale delays.

DRAM: Self-refresh. DRAM is typically the second-most power-hungry system component when active. However, several recent DRAM specifications feature an operating mode, called self-refresh, where the DRAM is isolated from the memory controller and autonomously refreshes DRAM content. In this mode, the memory bus clock and PLLs are disabled, as are most of the DRAM interface circuitry. Self-refresh saves more than an order of magnitude of power. For example, a 2GB SODIMM (designed for laptops) with a peak power draw above 5W uses only 202mW of power during self- refresh[128]. Transitions into and out of self- refresh can be completed in less than a microsecond [93].

Mass Storage: Solid State Disks. Solid state disks draw negligible power when idle, and, hence, do not need to transition to a sleep state for PowerNap. A recent 64GB Samsung SSD consumes only 0.32W while idle [153].

Network Interface: Wake-on-LAN. The key responsibility PowerNap demands of the network interface card (NIC) is to wake the system upon arrival of a packet. Existing NICs already provide support for Wake-on-LAN to perform this function. Current implementations of Wake-on-LAN provide a mode to wake on any physical activity. This mode forms a basis for PowerNap support. Current NICs consume only 400mW while in this mode [161].

Environmental Monitoring & Service Processors: PowerNap transition management. Servers typically include additional circuitry for environmental monitoring, remote management (e.g., remote power on), power capping, power regulation, and other functionality. These components typically manage ACPI state transitions and would coordinate PowerNap transitions. A typical service processor draws less than 10mW when idle.

Fans: Variable Speed Operation. Fans are a dominant power consumer in many recent servers. Modern servers employ variable-speed fans where cooling capacity is constantly tuned based on observed temperature or power draw. Fan power requirements typically grow cubically with average power. Thus, PowerNap's average power savings yield massive reductions in fan power requirements. In most blade designs, cooling systems are centralized in the blade chassis, amortizing their energy cost over many blades. Because thermal conduction progresses at drastically different timescales than PowerNap's transition frequency, chassis-level fan control is independent of PowerNap state (i.e., fans may continue operating during nap and may spin down during active operation depending on temperature conditions).

Power Provisioning: RAILS. PowerNap fundamentally alters the range of currents over which a blade chassis must efficiently supply power. In Chapter 8, we explain why conventional power delivery schemes are unable to provide efficient AC to DC conversion over this range, and present RAILS, our power conversion solution.

6.3.2 Software Mechanisms

Existing software support for sleep modes in desktop and laptop (e.g., ACPI) fails to meet the needs of PowerNap in several ways. First, system-wide sleep transitions are exceedingly slow (often requiring seconds) because individual devices are transitioned among modes sequentially through elaborate driver interfaces. To achieve acceptable transition latencies, devices must transition in parallel without complex operating system interactions. Second, existing APIs contain complexity and features (e.g., support for multiple power modes and per-device state management) that are not needed for PowerNap and introduce unnecessary overheads. Third, current state transitions are not software-transparent—most operating systems notify applications prior to a state change and have numerous visible side-effects (e.g., closing active network connections). Finally, these APIs do not provide adequate mechanisms to schedule the system to wake from sleep at a specific time in the future.

For schemes like PowerNap, the periodic timer interrupt used by legacy OS kernels to track the passage of time and implement software timers poses a challenge. As the timer interrupt is triggered every 1ms, conventional OS time keeping precludes the use of PowerNap. The periodic clock tick also poses a challenge for idle-power conservation on laptops and for virtualization platforms that consolidate hundreds of OS images on a single hardware platform. Hence, the Linux kernel has recently been enhanced to support "tickless" operation, where the periodic timer interrupt is eschewed in favor of hardware timers for scheduling and time keeping [160]. PowerNap depends on a kernel that provides tickless operation.

PowerNap's effectiveness increases with longer idle periods and less frequent state transitions. Some existing hardware devices (e.g., legacy keyboard controllers) require polling to detect input events. Current operating systems often perform maintenance tasks (e.g., flushing disk buffers, zeroing memory) when the OS detects significant idle periods. These maintenance tasks may interact poorly with PowerNap and can induce additional state transitions. However, efforts are already underway (e.g., as described in [160]) to redesign device drivers and improve background task scheduling.

6.3.3 Evidence of PowerNap Capabilities in the Wild

Our analysis of PowerNap is based upon the best estimates we can obtain for transition time and power consumption of system components as reported by data sheets. Even if there is some uncertainty in the exact values of these quantities, we believe the insights from our study hold and there is evidence "in the wild" that the PowerNap concept works. In particular, the Si0x power states recently developed by Intel [57, 75] demonstrate a PowerNap-like feature in smartphones that one can translate into the server space. These systems can enter the Si01 state in 600 μs and exit in 1.2ms, consuming only 8mW while in this low-power state. Our analysis shows that such transition times are acceptable for most server workloads.

6.4 Shortcomings

PowerNap is able to provide near energy-proportional operation for workloads with characteristics similar to those we have studied (i.e., average service times near 100ms). Though PowerNap is well suited for under-utilized services, there are a few potential shortcomings we enumerate in this section.

6.4.1 Multicore Servers

Current trends indicate that more and more cores will be integrated into a CPU. The workloads we analyzed were run on servers with only a few cores (i.e., 1-4). The model presented in Section 3 assumes a uniprocessor system; our queuing model predicts the performance for a single server (M/G/1) system. This queuing system is not appropriate for many-core servers, for which an M/G/k queue would be more appropriate. Unfortunately, it is not clear how to extend our model to an M/G/k system as Welch's derivation for exceptional first service [172] does not apply; to date, the M/G/k variant remains analytically intractable.

However, even straight-forward analysis of multicore scaling suggests that PowerNap, or similar techniques that rely on full-system idleness, will grow increasingly difficult to apply if server software architectures do not change. Data center designers already find idle periods difficult to exploit with current multicore hardware [171]. Current server workloads leverage multicore scaling through *weak scaling*, that is, they exploit additional cores by servicing additional, independent user requests. For example, if the number of cores doubles, roughly twice as much traffic can be directed to a single server, doubling throughput without increasing utilization. Unfortunately, because these requests are independent and their arrivals/completions are staggered, idle periods fail to align across cores, and PowerNap cannot be employed. Figure 7.2 illustrates this effect using a simple M/M/k analysis of a multicore server as the number of cores per socket is scaled. Even under only 10% utilization (u=0.1), all cores in a 16-core system are concurrently idle less than 20% of the time.

To continue to gain the high-leverage power savings of PowerNap, we must either schedule jobs in an attempt to align idle periods, or rearchitect server software to leverage strong scaling. Prior work has proposed scheduling, using simple timeouts to control performance impact, to reduce the overhead of transitioning to/from idle low-power modes in single/dual-core CPUs [25, 70] and memory DIMMS [137]. However, to recover a substantial fraction of idleness, these scheduling approaches require large delays which come at a steep response time penalty. Strong scaling, where multiple cores cooperate to reduce

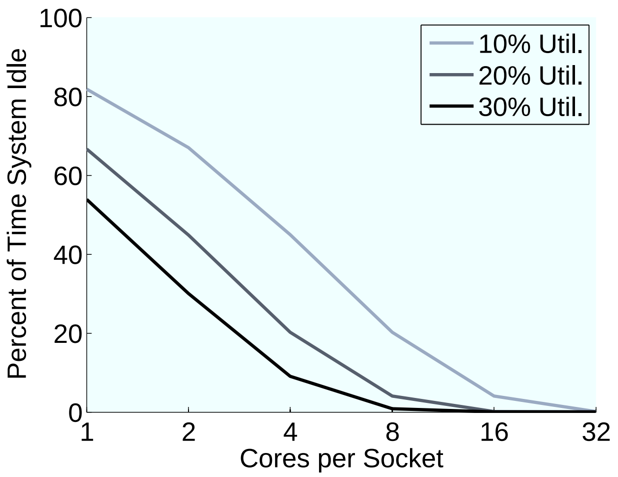


Figure 6.6: M/M/k analysis of full-system idleness under weak scaling. Because idle periods do not align across cores, full system idleness rapidly vanishes. Smarter scheduling or new models of parallelism that enable strong scaling are needed to continue to exploit idleness.

the latency of a single request, has the side-effect of aligning core busy and idle periods (if the parallelism is well-balanced), extending the applicability of PowerNap. However, rearchitecting services to leverage intra-request parallelism is challenging. We provide a solution to this challenge in Chapter 7, where we introduce DreamWeaver.

6.4.2 Highly Utilized Services

The PowerNap architecture provides excellent power savings with minimal latency penalty for lightly utilized servers. However, there are a few service that are highly utilized. Web search, rendering farms and batch processing, for example, have higher average utilization than the workloads we explore. Figure 6.3 shows that at high utilization PowerNap loses its advantage over throttling techniques such as DVFS. This change occurs because, as utilization approaches 100%, idle periods (and hence time spent in the nap state) become rare. Furthermore, at higher utilization DVFS will incur a significantly smaller latency penalty whereas PowerNap still incurs the same transition time. Therefore, we believe active low-power modes may be more appropriate for highly utilized services. For typical services where utilization is low, however, PowerNap remains a superior power management option.

6.4.3 Reliability

It is important to consider that PowerNap may affect the reliability of server components. Because the system quickly transitions between power extremes, PowerNap may increase component wear. However, two insights suggest the increased stress may be limited. First, PowerNap does not rapidly modulate the operation of mechanical components. Fans and disks are both likely to suffer reduced lifetimes from frequent spin-up and spin-down; hence, PowerNap uses SSDs instead of disks and modulates fan speed independent of wake/nap transitions (in response to temperature instead). Second, the transition time we demand of most components is far longer than their designed capabilities. For instance, CPUs incur substantial power transitions (due to clock gating or HLT instructions) on the nano- and microsecond scale. PowerNap requires only millisecond-scale transitions. A rigorous study of the component-level reliability implications of PowerNap is left to future work.

CHAPTER 7

DreamWeaver: Architectural Support for Deep Sleep

Modern data centers suffer from low energy efficiency due to endemic under-utilization [32]. 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) [88, 98, 104, 178] due to the shrinking gap between nominal and threshold voltages [68], 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 [58]. Figure 7.1 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 [22, 64, 70, 109, 112, 121, 122, 137]. In uniprocessors, the deep sleep possible with full-system idle low-power modes (e.g., PowerNap [121, 122]) 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 7.2 illustrates the poor scalability of PowerNap for a Web serving workload when CPU utilization is fixed at 30% (i.e.,

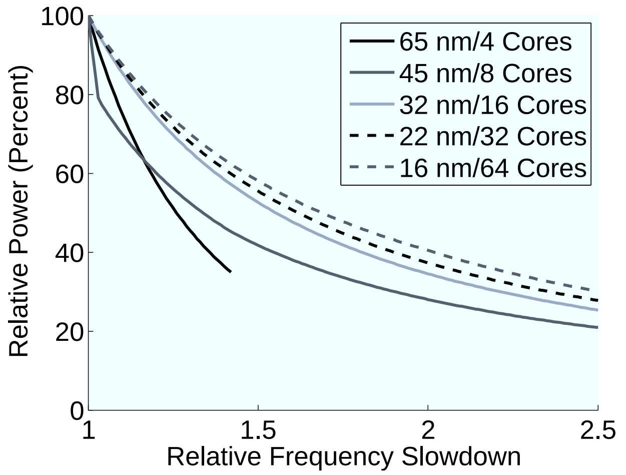


Figure 7.1: Voltage and frequency scaling. Future technology nodes require a disproportionate reduction in clock frequency for a given voltage reduction, breaking the classic assumption that dynamic power scales down cubically with frequency. Hence, VFS is becoming less effective: a 16nm processor requires a 2x slowdown for 50% power savings compared to 1.25x at 65nm. Data from [58].

load is scaled with the number of cores to maintain constant utilization; see Section 7.3.1 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 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 [25, 70, 137]. 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 [85], 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 [123]. Naïve batching is not effective because it either (1) incurs

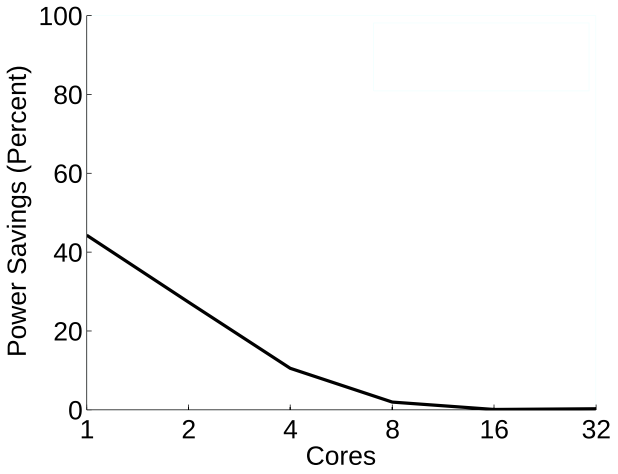


Figure 7.2: Full-system idle low-power mode. Power savings for a Web server at 30% utilization using a full-system idle low-power mode (e.g., PowerNap [122]). System-level idleness disappears with multicore integration, rendering coarse-grain power savings techniques ineffective.

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 [26] and Somniloquy [22], 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

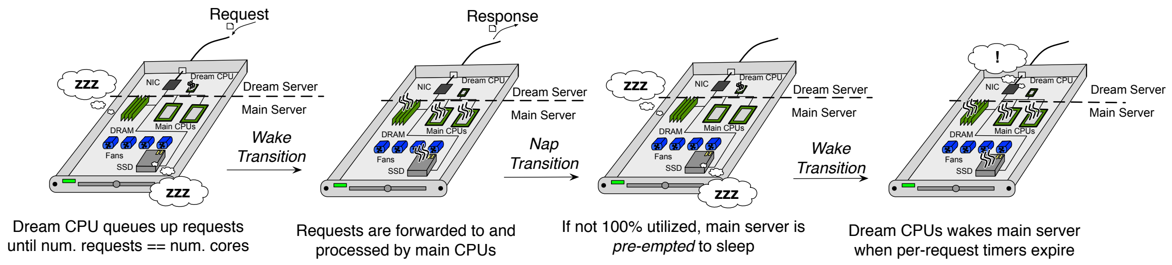

Request
Response
zzz
NIC
Dream CPU
Dream Server
Main Server
DRAM
Main CPUs
Fans
SSD
zzz
Wake Transition
Nap Transition
Wake Transition
!
Dream CPU queues up requests until num. requests == num. cores
Requests are forwarded to and processed by main CPUs
If not 100% utilized, main server is *pre-empted*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

Figure 7.3: DreamWeaver. The DreamWeaver system is composed of a main server with PowerNap capabilities [122] 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 [26] or Somniloquy [22], but requires far less processing power because it does not directly process or respond to packets.

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)* [126], 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.

7.1 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 [121, 122], 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 7.2, there is little, if any, opportunity for PowerNap in lightly- to moderately-utilized large-scale multicore servers.

7.1.1 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 [122] 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 7.3. 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 [26] and Somniloquy [22]. 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.

7.1.2 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 7.4. On

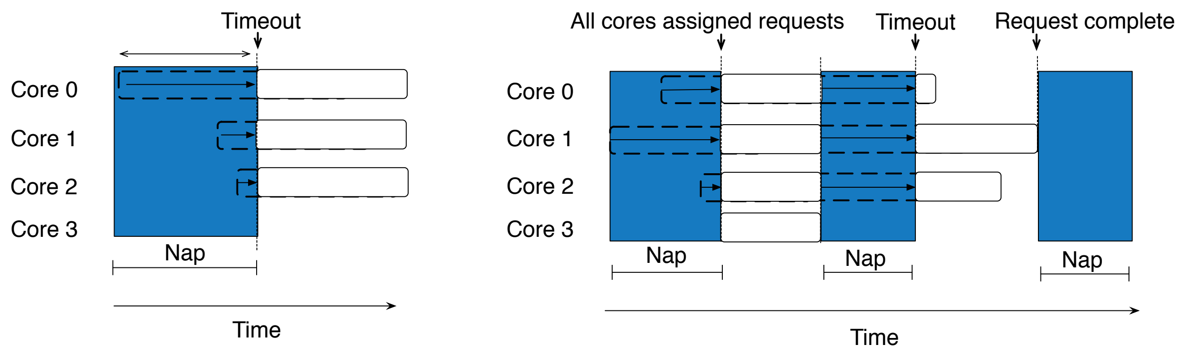


Figure 7.4: Weave Scheduling example. Weave Scheduling is an algorithm for intelligently delaying, preempting, and executing requests to maximize the fraction of time a multicore CPU is fully utilized while providing an upper-bound on per-request latency increase. The example on the left demonstrates 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 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.

7.2 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 7.3, we use simulation to explore DreamWeaver's impact on power consumption.

7.2.1 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.

We study the impact of DreamWeaver on a Web Search system modeled after that studied in [123] 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 [7], which we query using the AOL query set [3]. 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.

7.2.2 Results

We now present the results of our prototype system and compare it to our simulation infrastructure used in Section 7.3. Specifically we compare the sleep-latency tradeoff of the two evaluation methodologies. In Figure 7.5 we provide the time spent in sleep as a function of 99th-percentile latency as provided by our prototype ("Implementation") and our simulation infrastructure ("SQS"). When allowed a 1.5x slack on 99th-percentile response

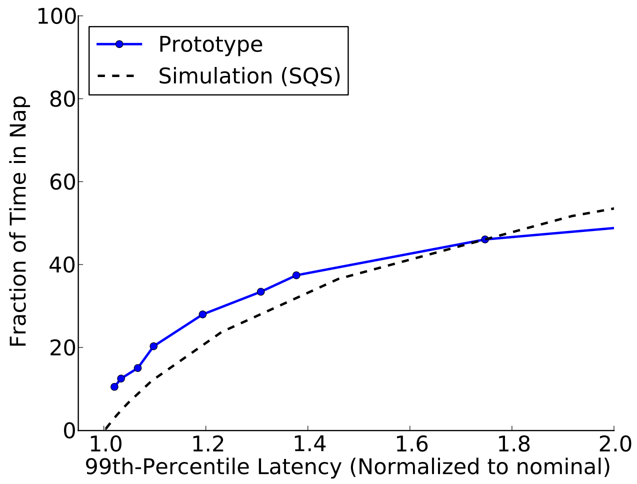


Figure 7.5: DreamWeaver prototype vs. simulation validation. This figure illustrates the accuracy of our simulation environment to predict the fraction of time a DreamWeaver server spends in the nap state. As we increase the predefined maximum delay a request can experience, the available full-system idleness increases as a function of 99th-percentile latency. One can see that the simulation ("Simulation") makes reasonable estimates of our prototype system ("Prototype").

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.

7.3 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.

7.3.1 Methodology

We evaluate the power savings potential of DreamWeaver and contrast it with other power management approaches using the SQS simulation methodology established in Chpater 5. 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

Table 7.1: Server power model. Based on data from Google [33] and HP [166].

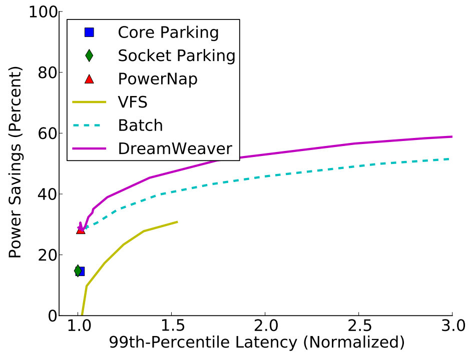
Power (% of Peak)	CPU	Memory	Disk	Other
Max	40%	35%	10%	15%
Idle	15%	25%	9%	10%

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 can be found in Chapter 5. SQS has been previously used to model Google's Web search application [123], and its latency and throughput predictions have been validated against a production Web search cluster.

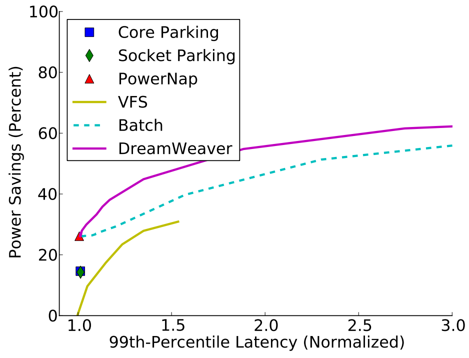
SQS does not model the details of what active system components are doing (e.g., which instructions are executing, what memory locations are accessed). However, these are not relevant to understanding idle periods and scheduling effects, hence, more detailed simulation models (e.g., instruction or cycle-accurate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

Low-Power Modes. Our power model assumptions for the system (Table 9.2) are based on the breakdowns from Google [33] and HP [166] and published characteristics of Intel Nehalem [97]. 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.

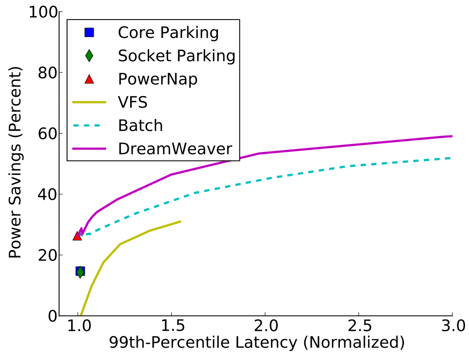
As a point of comparison, we also model voltage and frequency scaling (VFS), by varying the service rate. We map core count to a corresponding technology node and power-performance scaling curve as shown in Figure 7.1, using data from [58]. We explore a range of power-performance settings by exhaustively 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.



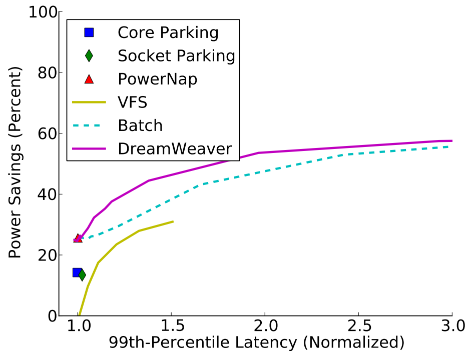
(a) Apache



(b) DNS



(c) Mail

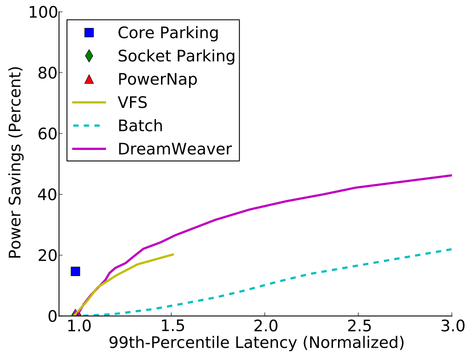


(d) Search

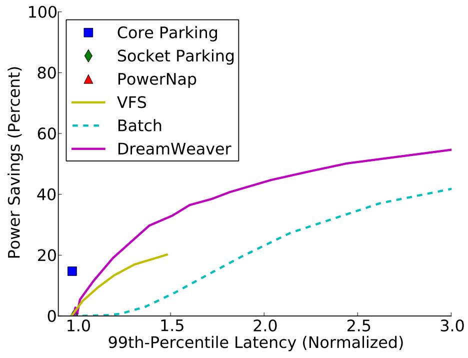
Figure 7.6: Comparison of power savings for 4-core system. This figure demonstrates the power savings of low-power modes as a function of 99th-percentile latency for a 4 core server. Per-core power gating ("Core Parking") can save a modest amount of power for a small latency increase because its transition 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.

7.3.2 Results

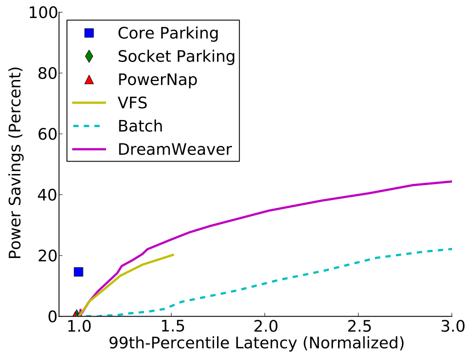
Power-latency tradeoff compared to other techniques. We first contrast DreamWeaver



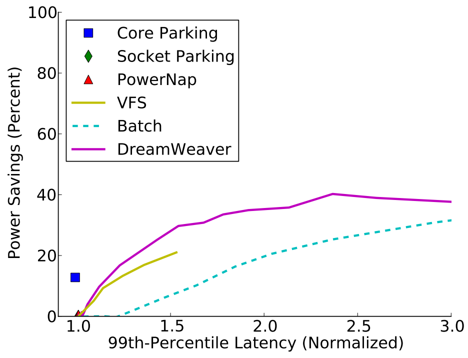
(a) Apache



(b) DNS



(c) Mail



(d) Search

Figure 7.7: 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.

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 [122] ("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 [70]. Finally, we compare to voltage/frequency scaling ("VFS"), as described in Section 7.3.1.

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 7.6. 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 normalized to the nominal latency (i.e., no power management). As discussed in Section 10.1, 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 substan-

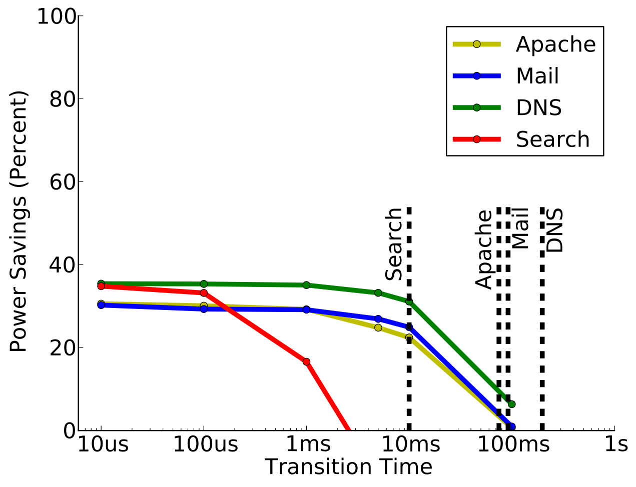


Figure 7.8: Sensitivity to transition time. DreamWeaver is less effective as the transition time in and out of PowerNap increases. Dotted vertical lines denote the average service time of each workload. The majority of power savings is realized by providing a transition time of about one order of magnitude less than the average service time of the workload.

tial 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 7.7 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, 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 7.8 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 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

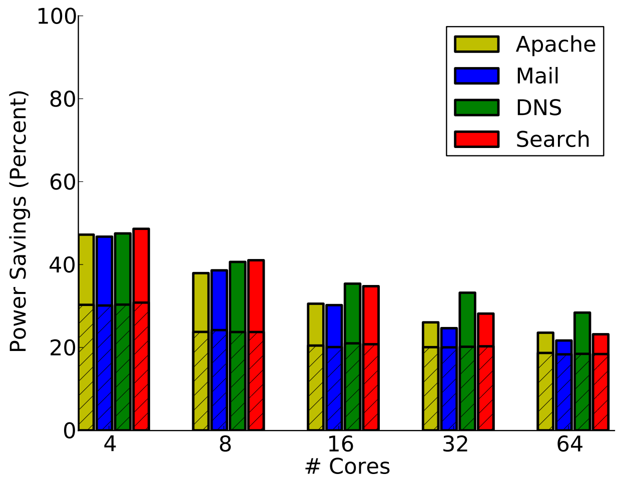


Figure 7.9: Sensitivity to number of cores. Solid bars represent DreamWeaver savings and hatched bars represent VFS savings. DreamWeaver is less effective as the number of cores increase, but always provides greater savings than VFS.

smaller than the average service time. Consistent with PowerNap [122], 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 7.9 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 [29]. Accordingly, DreamWeaver provides greater power savings at lower utilization. In Figure 7.10 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.

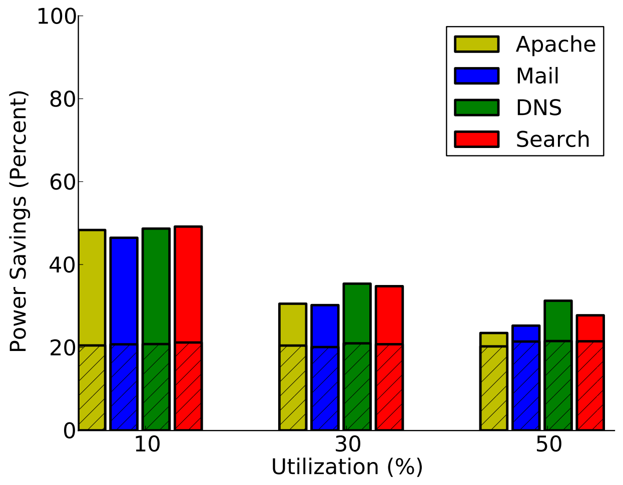


Figure 7.10: Sensitivity to utilization. Solid bars represent DreamWeaver savings and hatched bars represent VFS savings. DreamWeaver provides more savings in all cases.

7.3.3 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 [103] 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 7.2 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.

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.

CHAPTER 8

Redundant Array for Inexpensive Load Sharing (RAILS)

Whereas many mechanisms required by PowerNap are available in existing server components, one critical subsystem of current blade chassis falls short of meeting PowerNap's energy-efficiency requirements: the power conversion system. PowerNap reduces total ensemble power consumption when all blades are napping to only 6% of the peak when all are active. Power supplies are notoriously inefficient at low loads, typically providing conversion efficiency below 70% under 20% load [1]. These losses undermine PowerNap's energy efficiency.

Directly improving power supply efficiency implies a substantial cost premium. Instead, in this chapter, we introduce the *Redundant Array for Inexpensive Load Sharing* (RAILS), a power provisioning approach where power draw is shared over an array of low-capacity power supply units (PSUs) built with commodity components. The key innovation of RAILS is to size individual power modules such that the power delivery solution operates at high efficiency across the entire range of PowerNap's power demands. In addition, RAILS provides N+1 redundancy, graceful compute capacity degradation in the face of multiple power module failures, and reduced component costs relative to conventional enterprise-class power systems.

AC to DC conversion losses in computer systems have recently become a major concern, leading to a variety of research proposals [89, 116], product announcements (e.g., HP's Blade System c7000), and standardization efforts [1] to improve power supply efficiency. The concern is particularly acute in data centers, where each watt wasted in the power delivery infrastructure implies even more loss in cooling. Because PowerNap's power draw is substantially lower than the idle power in conventional servers, PowerNap demands conversion efficiency over a wide power range, from as few as 300W to as much as 7.2kW in a fully-populated enclosure.

In this section, we discuss why existing power solutions are inadequate for PowerNap and present RAILS, our power solution. RAILS provides high conversion efficiency

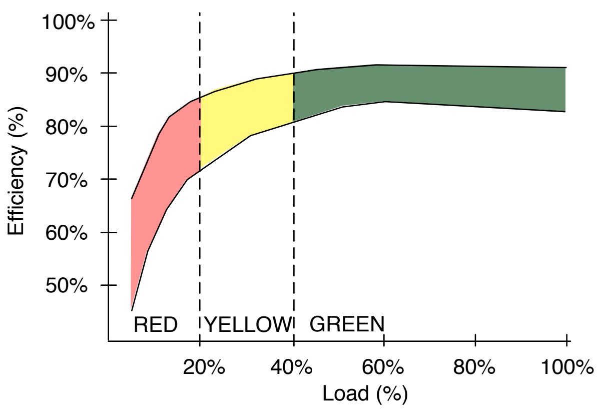


Figure 8.1: Power supply efficiency.

across PowerNap's power demand spectrum, provides N+1 redundancy, allows for graceful degradation of compute capacity when PSUs fail, and minimizes costs by using commodity PSUs in an efficient arrangement.

8.1 Power Supply Unit background

Poor Efficiency at Low Loads. Although manufacturers often report only a single efficiency value, most PSUs do not have a constant efficiency across electrical load. A recent survey of server and desktop PSUs reported their efficiency across loads [1]. Figure 8.1 reproduces the range of efficiencies reported in that study. Though PSUs are often over 90% efficient at their optimal operating point (usually near 75% load), efficiency drops off rapidly below 40% load, sometimes dipping below 50% (i.e., >2W in for 1W out). We divide the operating efficiency of power supplies into three zones based on electrical load. Above 40% load, the PSUs operate in the "green" zone, where their efficiency is at or above 80%. In the 20-40% "yellow" zone, PSU efficiency begins to drop, but typically exceeds 70%. However, in the "red" zone below 20%, efficiency drops off precipitously.

Two factors cause servers to frequently operate in the "yellow" or "red" efficiency zones. First, servers are highly configurable, which leads to a large range of power requirements. The same server model might be sold with only one or as many as 20 disks installed, and the amount of installed DRAM might vary by a factor of 10. Furthermore, peripherals may be added after the system is assembled. To simplify ordering, upgrades, testing, and safety certification, manufacturers typically install a power supply rated to exceed the power requirements of the most extreme configuration. Second, servers are often configured with 2N redundant power supplies (i.e., twice as many as are required for a worst-case configuration). The redundant supplies typically share the electrical load to minimize PSU temperature and to ensure current flow remains uninterrupted if a PSU fails. However, the EPRI study [1] concluded that this load-sharing arrangement often shifts PSUs from “yellow”-zone to “red”-zone operation.

Recent Efficiency Improvements. A variety of recent initiatives seek to improve server power efficiency:

- **80+ certification.** The EPA Energy Star program has defined the “80+” certification standard [4] to incentivize PSU manufacturers to improve efficiency at low loads. The 80+ incentive program is primarily targeted at the low-peak-power desktop PSU market. 80+ supplies require considerably higher design complexity than conventional PSUs, which may pose a barrier to widespread adoption in the reliability-conscious server PSU market. Added circuit components and tighter tolerances add to the cost of the PSU. Furthermore, despite their name, the 80+ specification does not require energy efficiency above 80% across all loads, rather, only within the typical operating range of conventional systems. This specified efficiency range is not wide enough for PowerNap.

- **Single voltage supplies.** Unlike desktop machines, which require five different DC output voltages to support legacy components, server PSUs typically provide only a single DC output voltage, simplifying their design and improving reliability and efficiency [89]. Although PowerNap benefits from this feature, a single output voltage does not directly address inefficiency at low loads.

- **DC distribution.** Recent research [89] has called for distributing DC power among data center racks, eliminating AC-to-DC conversion efficiency concerns at the blade enclosure level. However, the efficiency advantages of DC distribution are unclear [149] and deploying DC power will require multi-industry coordination.

- **Dynamic load-sharing.** Blade enclosures create a further opportunity to improve efficiency through dynamic load-sharing. HP’s Dynamic Power Saver [116] feature in the HP Blade Center c7000 employs up to six high-efficiency 2.2kW PSUs in a single enclosure, and dynamically varies the number of PSUs that are engaged, ensuring that all active supplies operate in their “green” zone while maintaining redundancy. Although HP’s solution is ideal for the idle and peak power range of the

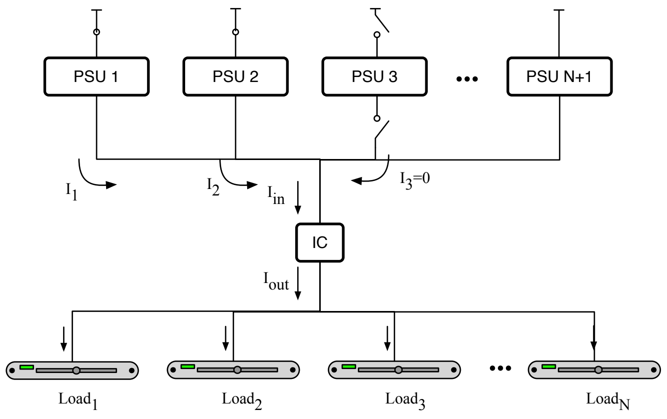


Figure 8.2: RAILS PSU design.

c-class blades, it requires expensive PSUs and provides insufficient granularity for PowerNap.

While all these solutions improve efficiency for their target markets, none achieve all our goals of efficiency for PowerNap, redundancy, and low cost.

8.2 RAILS Design

We introduce a new power delivery solution tuned for PowerNap: the Redundant Array for Inexpensive Load Sharing (RAILS). The central idea of our scheme is to load- share over multiple inexpensive, small PSUs to provide the efficiency and reliability of larger, more expensive units. Through intelligent sizing and load-sharing, we ensure that active PSUs operate in their efficiency sweet spots. Our scheme provides 80+ efficiency and enterprise-class redundancy with commodity components.

RAILS targets three key objectives: (1) efficiency across the entire PowerNap dynamic power range; (2) N+1 reliability and graceful degradation of compute capacity under multiple PSU failure; and (3) minimal cost.

Figure 8.2 illustrates RAILS. As in conventional blade enclosures, power is provided by multiple PSUs connected in parallel. A conventional load-sharing control circuit continuously monitors and controls the PSUs to ensure load is divided evenly among them. As in Dynamic Smart Power [116], RAILS disables and electrically isolates PSUs that are not necessary to supply the load. However, our key departure from prior designs is in the granularity of the individual PSUs. We select PSUs from the economic sweet spot of the high-sales-volume market for low-wattage commodity supplies.

Table 8.1: Relative PSU density.

	microATX	ATX	Custom Blade
Density (Normalized W/vol.)	675.5	1000	1187

We choose a power supply granularity to satisfy two criteria: (1) A single supply must be operating in its "green" zone when all blades are napping. This criterion establishes an upper bound on the PSU capacity based on the minimum chassis power draw when all blades are napping. (2) Subject to this bound, we size PSUs to match the incremental power draw of activating a blade. Thus, as each blade awakens, one additional PSU is brought on line. Because of intelligent sizing, each of these PSUs will operate in their optimal efficiency region. Whereas current blade servers use multi-kilowatt PSUs, a typical RAILS PSU might supply 500W.

RAILS meets its cost goals by incorporating high-volume commodity components. Although the form-factor of commodity PSUs may prove awkward for rack-mount blade enclosures, precluding the use of off-the-shelf PSUs, the power density of high-sales-volume PSUs differs little from high-end server supplies. Hence, with appropriate mechanical modifications, it is possible to pack RAILS PSUs in roughly the same physical volume as conventional blade enclosure power systems.

RAILS meets its reliability goals by providing fine-grain degradation of the system's peak power capacity as PSUs fail. In any N+1 design, the first PSU failure does not affect compute capacity. However, in conventional blade enclosures, a subsequent failure may force shutdown of several (possibly all) blades. Multiple-failure tolerance typically requires 2N redundancy, which is expensive. In contrast, in RAILS, where PSU capacity is matched to the active power draw of a single blade, the second and subsequent failures each require the shutdown of only one blade.

8.3 Evaluation

We evaluate the power efficiency and cost of PowerNap with four power supply designs, commodity supplies ("Commodity"), high-efficiency 80+ supplies ("80+"), dynamic load sharing ("Dynamic"), and RAILS ("RAILS"). We evaluate all four designs in the context of a PowerNap-enabled blade system similar to HP's Blade Center c7000. We assume a fully populated chassis with 16 half-height blades. Each blade consumes 450W at peak, 270W at idle without PowerNap, and 10.4W in PowerNap (see Table 6.2). We assume the blade enclosure draws 270W (we neglect any variation in chassis power as a function of

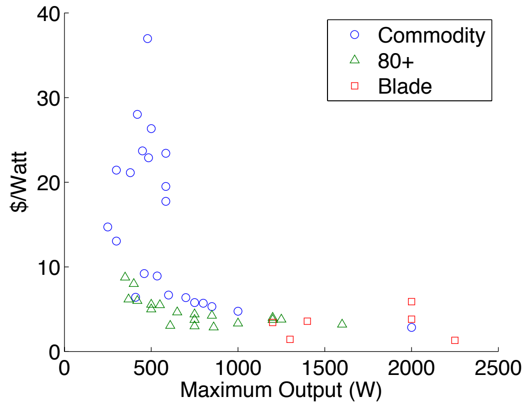


Figure 8.3: Power supply pricing.

the number of active blades). The non-RAILS systems employ 4 2250W PSUs (sufficient to provide N+1 redundancy). The RAILS design uses 17 500W PSUs. We assume the average efficiency characteristic from Figure 8.1 for commodity PSUs.

Cost. Server components are sold in relatively low volumes compared to desktop or embedded products, and thus, command premium prices. Some Internet companies (e.g., Google), have eschewed enterprise servers and instead assemble systems from commodity components to avoid these premiums. PSUs present another opportunity to capitalize on low-cost commodity components. Because desktop ATX PSUs are sold in massive volumes, their constituent components are cheap. A moderately-sized supply can be obtained at extremely low cost. Figure 8.3 shows a survey of PSU prices in Watts per dollar for a wide range of PSUs across market segments. Price per Watt increases rapidly with power delivery capacity. This rise can be attributed to the proportional increase in required size for power components such as inductors and capacitors. Also, the price of discrete power components grows with size and maximum current rating. Presently, the market sweet spot is around 500W supplies. Both 80+ and blade server PSUs are substantially more expensive than commodity parts. Because RAILS uses commodity PSUs with small maximum outputs, it takes advantage of PSU market economics, making RAILS far cheaper than proprietary blade PSUs.

Power Density. In data centers, rack space is at a premium, and, hence, the physical volume occupied by a blade enclosure is a key concern. AILS drastically increases the number of distinct PSUs in the enclosure, but each PSU is individually smaller. To confirm the feasibility of RAILS, we have compared the highest power density available in commodity PSUs, which conform to one of several standard form-factors, with that of PSUs

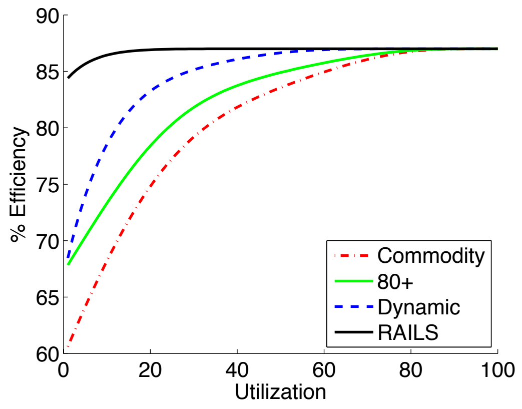


Figure 8.4: Power Delivery Solution Comparison.

designed for blade centers, which may have arbitrary dimensions. Table 8.1 compares the power density of two commodity form factors with the power density of HP's c7000 PSUs. We report density in terms of Watts per unit volume normalized to the volume of one ATX power supply. The highly-compact microATX form factor exhibits the worst power density—these units have been optimized for small dimensions but are employed in small form-factor devices that do not require high peak power. Though they are not designed for density, commodity ATX supplies are only 16% less dense than enterprise-class supplies. Furthermore, as RAILS requires only a single output voltage, eliminating the need for many of a standard ATX PSU's components, we conclude that RAILS PSUs fit within blade enclosure volumetric constraints.

Power Savings and Energy Efficiency. To evaluate each power system, we calculate expected power draw and conversion efficiency across blade ensemble utilizations. As noted in Section 4.1, low average utilization manifests as brief bursts of activity where a subset of blades draw near-peak power. The efficiency of each power delivery solution depends on how long blades are active and how many are simultaneously active. For each utilization, we construct a probability mass function for the number of simultaneously active blades, assuming utilization across blades is uncorrelated. Hence, the number of active blades follows a binomial distribution. From the distribution of active blades, we compute an expected power draw and determine conversion losses from the power supply's efficiency-versus-load curve. We obtain efficiency curves from the Energy Star Bronze 80+ specification [4] for 80+ PSUs and [1] for commodity PSUs.

Figure 8.4 compares the relative efficiency of PowerNap under each power delivery solution. Using commodity ("Commodity") or high efficiency ("80+") PSUs results in the lowest efficiency, as PowerNap's low power draw will operate these power supplies in the

Table 8.2: Power and cost comparison.

	Web 2.0			Enterprise		
	Power	Efficiency	Power costs	Power	Efficiency	Power costs
Blade	6.4 kW	87%	\$29k	6.6 kW	87%	\$30k
PowerNap	1.9 kW	67%	\$10k	2.6 kW	70%	\$13k
PowerNap with RAILS	1.4 kW	86%	\$6k	2.0 kW	86%	\$9k

"Red" zone. RAILS ("RAILS") and Dynamic Load-Sharing ("Dynamic") both improve PSU performance because they increase average PSU load. RAILS outperforms all of the other options because its fine-grain sizing best matches PowerNap's requirements.

8.4 Cost Analysis

We presented *PowerNap*, a method for eliminating idle power in servers by quickly transitioning in and out of an ultra-low power state. We have constructed an analytic model to demonstrate that, for typical server workloads, PowerNap far exceeds DVFS's power savings potential with better response time. Because of PowerNap's unique power requirements, we introduced RAILS, a novel power delivery system that improves power conversion efficiency, provides graceful degradation in the event of PSU failures, and reduces costs.

To conclude, we present a projection of the effectiveness of PowerNap with RAILS in real commercial deployments. We construct our projections using the commercial high-density server utilization traces described in Table 4.2. Table 8.2 presents the power requirements, energy-conversion efficiency and total power costs for three server configurations: an unmodified, modern blade center such as the HP c7000; a PowerNap-enabled system with large, conventional PSUs ("PowerNap"); and PowerNap with RAILS. The power costs include the estimated purchase price of the power delivery system (conventional high-wattage PSUs or RAILS), 3-year power costs assuming California's commercial rate of 11.15 cents/kWh [5], and a cooling burden of 0.5W per 1W of IT equipment [133].

PowerNap yields a striking reduction in average power relative to Blade of nearly 70% for Web 2.0 servers. Improving the power system with RAILS shaves another 26%. Our total power cost estimates demonstrate the true value of PowerNap with RAILS: our solution provides power cost reductions of nearly 80% for Web 2.0 servers and 70% for Enterprise IT.

CHAPTER 9

Online Data-Intensive Services - The Need for Coordinated Active Low-Power Modes

In this chapter we examine, for the first time, power management for a class of data center workloads, which we refer to as *Online Data-Intensive* (OLDI), that would benefit drastically from energy proportionality because it has a wide dynamic load range. OLDI workloads are driven by user queries/requests that must interact with massive data sets, but require responsiveness in the sub-second time scale, in contrast to their offline counterparts (such as MapReduce computations). Large search products, online advertising, and machine translation are examples of workloads in this class. As shown in Figure 9.1, although the load on OLDI services vary widely during the day, their energy consumption sees little variance due to the lack of energy proportionality of the underlying machinery.

Previous research has observed that energy-proportional operation can be achieved for lightly utilized servers with full-system, coordinated *idle low-power modes* [122]. Such a technique works well for workloads with low average utilization and a narrow dynamic range, a common characteristic of many server workloads. Other work observes that cluster-level power management (e.g., using VM migration and selective power-down of servers) can enable energy-proportionality at the cluster level even if individual systems are far from energy proportional [165].

As we will show, full-system idle low power modes fare poorly for OLDI services because these systems have a large dynamic range and, though sometimes lightly loaded, rarely are fully idle, even at fine time scales. Cluster-grain approaches that scale cluster size in response to load variation are inapplicable to OLDI services because the number of servers provisioned in a cluster is fixed. Cluster sizing is determined primarily based on the data set size instead of the throughput of incoming requests. For a cluster to process an OLDI data set for even a single query with acceptable latency, the data set must be partitioned over thousands of nodes that act in parallel. Hence, the granularity at which

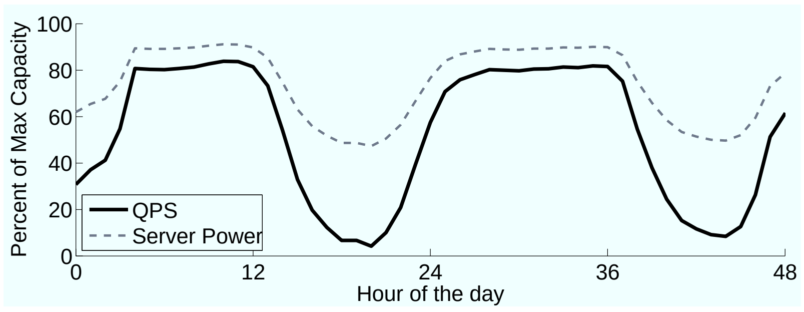


Figure 9.1: Example diurnal pattern in queries per second (QPS) for a Web search cluster. Non-peak periods provide significant opportunity for energy-proportional servers. For a perfectly energy proportional server, the percentage of peak power consumed and peak QPS would be the same. Server power is estimated for systems with 45% idle power.

systems can be turned off is at cluster- rather than node-level.

Fundamentally, the architecture of OLDI services demands that power be conserved on a per-server basis; each server must exhibit energy-proportionality for the cluster to be energy-efficient, and the latency impact of any power management actions must be limited. We find that systems supporting OLDI services require a new approach to power management: coordination of *active low-power modes* across the entire utilization spectrum. We demonstrate that neither power management of a single server component nor uncoordinated power management of multiple components provide desirable power-latency tradeoffs.

We report the results of two major studies to better understand the power management needs of OLDI services. First, we characterize a major OLDI workload, Google Web Search, at thousand-server, cluster-wide scale in a production environment to expose the opportunities (and non-opportunities) for active and idle low-power management. We introduce a novel method of characterization, *activity graphs*, which enable compact representation of the activity levels of server components. Activity graphs provide designers the ability to identify the potential of per-component active and idle low-power modes at various service load levels. Second, we perform a study of how latency constrains this potential, making power management more difficult. We construct and validate a performance model of the Web Search workload that predicts the 95th-percentile query latency under different low-power modes. We demonstrate that our modeling framework can predict 95th-percentile latency within 10% error. Using this framework, we explore the power-performance tradeoffs for available and future low-power modes.

We draw the following conclusions about power management for major server components:

1. **CPU active low-power modes provide the best *single* power-performance mech-**

anism, but are not sufficient for energy-proportionality. Voltage and frequency scaling (VFS) provides substantial power savings for small changes in voltage and frequency in exchange for moderate performance loss (see Figure 9.14). Looking forward, industry trends indicate that VFS power savings will be reduced in future technology generations as the gap between circuits' nominal supply and threshold voltages shrink [43], suggesting that power savings may not be realized from VFS alone. Furthermore, we find that deep scaling yields poor power-performance tradeoffs.

2. **CPU idle low-power modes are sufficient at the core level, but better management is needed for shared caches and on-chip memory controllers.** We find that modern CPU cores have aggressive clock gating modes (e.g., C1E) that conserve energy substantially; power gating modes (e.g., core C6) are usable, but provide little marginal benefit at the system level (see Figure 9.15). However, we observe that non-core components such as shared caches and memory controllers must remain active as long as *any* core in the system is active. Thus, we find opportunity for full socket idle management (e.g, socket C6) is minimal.

3. **There is great opportunity to save power in the memory system with active low-power modes during ample periods of underutilization.** We observe that the memory bus is often highly underutilized for periods of several seconds. There is a great opportunity to develop an active low-power mode for memory and we demonstrate that it would provide the greatest marginal addition to a server's low-power modes. Because the memory system is so tightly coupled to CPU activity, it is extremely rare for DRAM idle periods to last long enough to take advantage of existing idle low-power modes (e.g., self-refresh) (see Figure 9.6).

4. **Unlike many other data center workloads, full-system idle power management (e.g., PowerNap) is ineffective for OLDI services.** Previous research has demonstrated that energy-proportionality can be approached by rapidly transitioning between a full-system high-performance active and low-power inactive state to save power during periods of brief idleness [122]. Whereas such a technique works well for many workloads, we demonstrate that it is inappropriate for the ODLI workload class. Because periods of full-system idleness are scarce in an OLDI workloads, we evaluate batching queries to coalesce idleness, but find that the latency-power tradeoffs are not enticing.

5. **The only way to achieve energy-proportional operation with acceptable query**

latency is coordination of full-system active low-power modes. Rather than requiring deep power savings out of any one component, we observe that systems must be able to leverage moderate power savings in all their major components. Since full-system idleness is scarce, power savings must be achieved with active low-power modes. Furthermore, system-wide coordination of power-performance states is necessary to maintain system balance and achieve acceptable per-query latency.

The rest of this paper is organized as follows. In Section 2, we discuss the unique aspects of OLDI services, how these impact power management, and prior work. To identify opportunities for power management, we present our cluster-scale Web Search characterization in Section 3. In Section 4, we develop and validate a Web Search performance model to determine which of these opportunities are viable from a latency perspective, and draw conclusions from this model in Section 5. Finally, in Section 6, we conclude.

9.1 Online Data-Intensive Services

Online data-intensive services are a relatively new workload class and have not undergone an extensive characterization study in the systems community. Perhaps the defining feature of these workloads is that latency is of the utmost importance. A user's experience is highly dependent on perceived speed; even delays of less than 400ms in page rendering time have a measurable impact on web search queries per user and ad revenue [155]. Rather than focusing on the mean, search operators often express performance objectives over a quantile of queries, for example, a constraint on 95th-percentile latency ensuring users experience low latency 95% of the time. Because queries can exhibit drastically varying processing requirements and a wide variance in latency, achieving these targets can be quite difficult. It is particularly important to consider the tail of the latency distribution when considering power management schemes, as many power management approaches impact tail latencies more than they affect the mean. Figure 9.2 shows a cumulative distribution of query latency (normalized to the mean latency) for a web search cluster operating at 65% of its peak capacity. There is a 2.4x gap between mean and 95th-percentile latency.

9.1.1 Understanding Online Data-Intensive Services

Cluster Processing in Web Search. The sheer size of the Web index and the complexity of the scoring system requires thousands of servers working in parallel in order to meet latency requirements. Figure 9.3 depicts the distributed, multi-tier topology of a Web search cluster [29]. Queries arrive at a Web server front-end, which forwards them to a

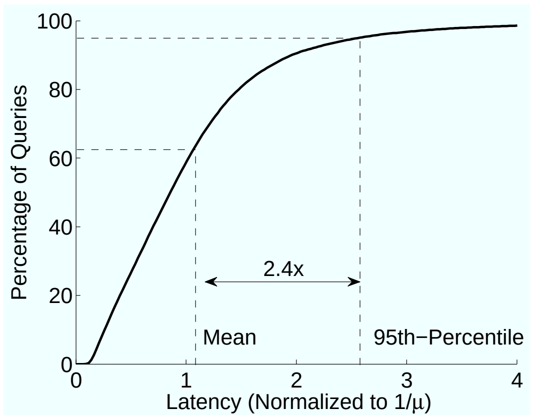


Figure 9.2: Example leaf node query latency distribution at 65% of peak QPS.

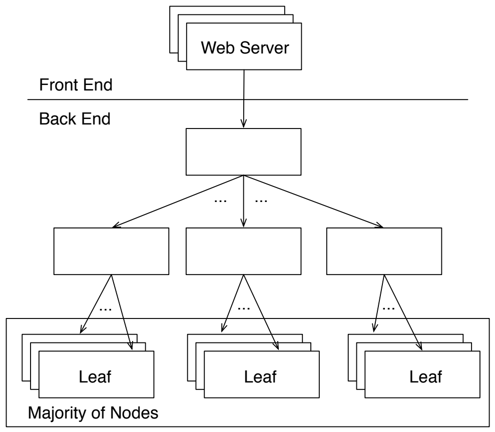


Figure 9.3: Web search cluster topology.

large collection of *leaf* nodes using a distribution tree of intermediary servers. The index is *sharded* (i.e., distributed) among all the leaf nodes, so that each performs the search on its fragment of the index. Most of the index is not replicated across leaves, hence, each search query must be processed at each leaf to maximize result relevance. The query distribution tree aggregates all results and returns the highest-scoring ones back to the front-end server.

Leaf nodes are the most numerous in a search cluster. Because of the high fan-out of the distribution tree, leaf nodes account for both the majority of query processing and cluster power consumption. Hence, we restrict our study to power savings for these servers. Note, however that the importance of the 95% latency becomes even more relevant when studying performance at the leaf-node level, since the entire service latency can be dominated by the latency of a single slow leaf. Although the effect of slow leaf nodes can be limited using latency cutoffs, in practice these tend to be conservative—prematurely terminating a search can compromise query result relevance.

Though other data-intensive workloads share similarities with OLDI services, there are some critical differences that make OLDI a particularly challenging target for power and energy optimizations. For example, personalized services such as Web mail and social networking also manipulate massive amounts of data and have strict latency requirements for user interaction. A key difference between these workloads and OLDI services is the fraction of data and computing resources that are used by every single user request. For a Web search cluster, a query typically touches all of the machines in the cluster. By contrast, a Web mail request will access primarily the users' active mailbox and a small fraction of the machines in the serving cluster. MapReduce-style computations can touch all of the machines and data in a cluster, but their focus on throughput makes them more latency tolerant, and therefore an easier target for energy optimizations, which tend to increase latency.

Diurnal Patterns. The demands of an OLDI service can vary drastically throughout the day. The cluster illustrated in Figure 9.1 is subject to load variations of over a factor of four. To simplify our discussion, we will focus on three traffic levels based on percentage of peak achievable traffic: low (20%), medium (50%) and high (75%). We choose 75% of peak as our high traffic point because, although clusters can operate near 100% capacity when needed, it is common for operators to set normal maximum operating points that are safely below those peaks to ensure service stability.

9.2 Cluster-scale Characterization

In this section we characterize the Web Search workload at 1000-node cluster scale. Opportunities for power savings often occur at very fine time-scales, so it is important to understand the magnitude and time-scale at which components are utilized [122]. We present the results of our characterization, and analyze its implications on opportunity for power management, using a new representation, which we call *activity graphs*. Our characterization allows us to narrow the design space of power management solutions to those that are applicable to OLDI services. In Section 5 and 6, we further analyze promising approaches from a latency perspective.

9.2.1 Experimental Methodology

We collected fine-grained activity traces of ten machines from a Web search test-cluster with over a thousand servers searching a production index database. The test workload executes a set of anonymized web search queries, with the query injection rate tuned to produce each of the QPS loads of interest (low, medium, and high). The behavior of the test runs was validated against live production measurements, however the latency and QPS levels reported here do not correspond exactly to any particular Google search product—there are multiple production search systems at Google and their exact behavior is a result of complex trade-offs between index size, results quality, query complexity and caching behavior. We are confident that the parameters we chose for our test setup faithfully represents the behavior and energy-performance trade-offs of this class of workloads.

We collect raw utilization metrics for CPU, memory bandwidth, and disk I/O bandwidth with in-house kernel and performance counter-based tracing tools that have a negligible overhead and do not have a measurable impact on application performance. CPU utilization is aggregated across all the processors in the system. Our memory and disk instrumentation

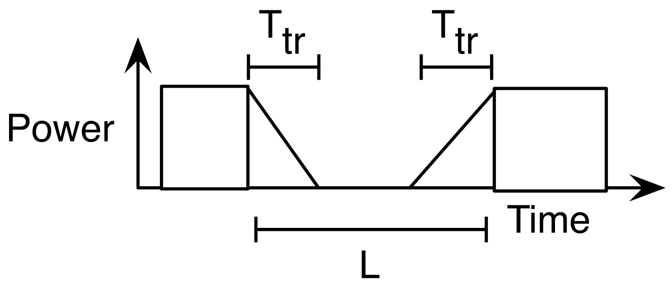


Figure 9.4: Idealized low-power mode. L is the length of the idle period and T_{tr} is the time required to transition in and out of the low power state.

computes utilization as the fraction of peak bandwidth consumed over a given window; our resolution is limited to 100 ms windows.

9.2.2 Activity Graphs: Compactly representing component-level utilization

Activity graphs compactly describe the opportunity for power savings at a range of utilizations and time-scales of interest. Figures 9.5-9.7 show the activity graphs for a Web search workload at multiple traffic levels (denoted by queries-per-second or QPS). These activity graphs depict a function $A(L, U)$ that gives the fraction of time a component spends at or below a given utilization U for a time period of L or greater:

$$A(L, U) = Pr(l \geq L, u \leq U) \tag{9.1}$$

This formulation allows us to determine the fraction of time where any particular power mode might be useful. Perhaps as importantly, it lets us quickly rule out low-power modes that will not be applicable. We assume that an oracle guides transitions in and out of the mode at the start and end of each period L. Most modes do not save power during this transition and may halt processing during transitions; accordingly, for our study we assume that T_{tr} must be an order of magnitude less than L for a low-power mode to be effective.

Example: Reading Figure 9.5(a). We now demonstrate how to interpret the results of an activity graph, using Figure 9.5(a) as an example. Suppose we wish to evaluate a CPU active low-power mode that incurs roughly a 25% slowdown (i.e., it can only be used below 1/1.25 = 80% utilization) and has a transition time of 1 ms (i.e., it is only useful for L of about 10ms or greater). We can see that at 20% QPS, this mode is applicable nearly 80% of the time. Conversely, if transitions latency limited this mode to periods of 1 second or greater, there would be almost no opportunity for the mode.

Abstract Power Mode Model. Figure 9.4 illustrates the abstract model we use to determine upper bounds on power savings. A power mode can be activated for a period L in which utilization is less than or equal to a threshold U. For idle low-power modes, U is zero. For active low-power modes, U is bounded by the slowdown incurred when operating under the low power mode. For example, a low power mode that incurs a factor of two slowdown may not be used when $U > 50\%$ as the offered load would then exceed the processing capacity under the mode. To save power, a component must transition in and out of a mode with latency T_{tr}.

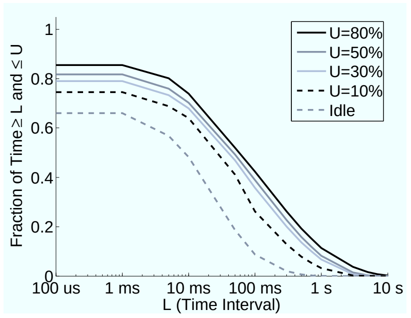
9.2.3 Characterization Results

We now discuss the implications of the activity graphs shown in Figures 9.5, 9.6, and 9.7, for CPU, memory and disk respectively. Each figure has a separate result for low, medium and high QPS.

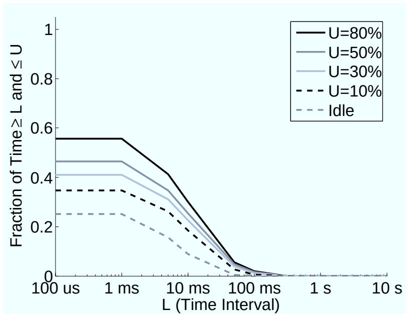
CPU. Figure 9.5 show activity graphs for CPU. There is almost no opportunity for CPU power management at a time-scale greater than one second even at low loads; instead power modes must be able to act well into the sub-second granularity to be effective. Perhaps most interestingly, the 1 ms time scale captures nearly all the savings opportunity regardless of the utilization threshold, suggesting that it is unnecessary to design modes that act on granularities finer than 50-100μs. Additionally, while increased QPS reduces the overall opportunity for power management at each utilization level, it does not change the time-scale trends.

Memory. Figure 9.6 presents the activity for memory. Two features are prominent. First, we observe that the memory bus is greatly underutilized, with many long periods during which utilization is below 30%. Hence, active low power modes that trade memory bandwidth for lower power are applicable even if they have relatively large transition times. Second, the memory bus is never idle for 100ms or longer. Unfortunately, our instrumentation cannot examine sub-100ms granularities. However, by estimating finer-grained memory idleness from our CPU traces (by assuming the memory system is idle when all CPUs are idle), we find that idle periods longer than 10μs are scarce even at low QPS values. We conclude that DRAM idle low-power modes require sub-μs transitions times to be effective for these workloads, which is an order of magnitude faster than currently available DRAM modes.

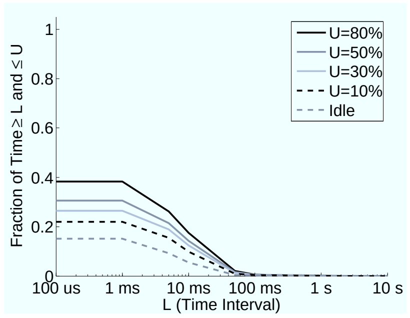
Disk. Finally, Figure 9.7 shows the activity graphs for disk. The activity trends for disk differ from CPU and memory because varying QPS shifts the time-scales over which utilization levels are observed. Whereas the majority of time spent at or below 30% utilization is captured in 10 second intervals at 20% QPS, this opportunity shifts to an order of magnitude finer time-scale when load increases to 75% QPS. As with memory, disk bandwidth is



(a) 20% QPS

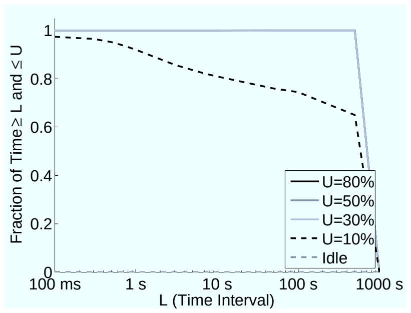


(b) 50% QPS

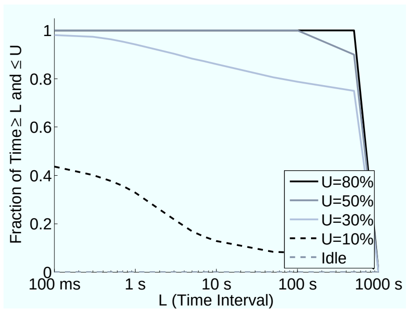


(c) 75% QPS

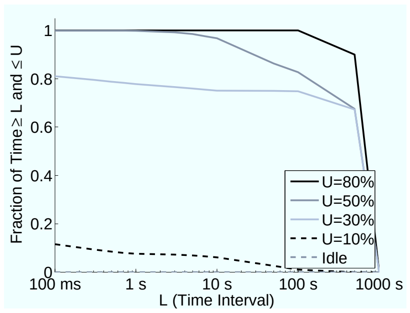
Figure 9.5: CPU activity graphs. Opportunities for CPU power savings exist only below 100 ms regardless of load.



(a) 20% QPS

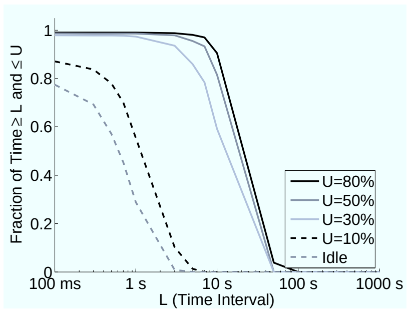


(b) 50% QPS

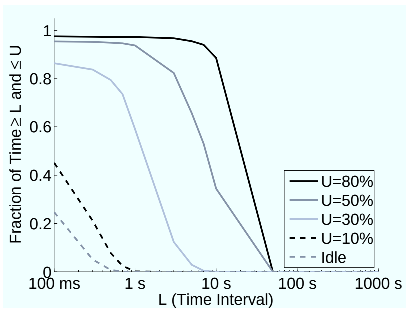


(c) 75% QPS

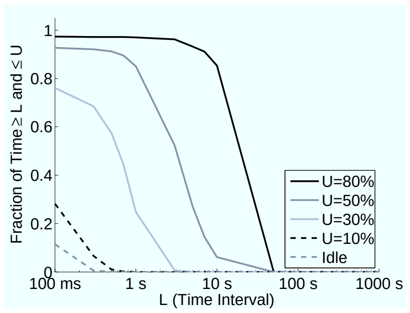
Figure 9.6: Memory activity graphs. Memory bandwidth is undersubscribed, but the subsystem is never idle.



(a) 20% QPS



(b) 50% QPS



(c) 75% QPS

Figure 9.7: Disk activity graphs. Periods of up to tens of seconds with moderate utilization are common for disks.

not heavily utilized even at high QPS; the majority of time is spent below 50% utilization even at a 1 second granularity. Unlike memory, disk exhibits periods of idleness at 100ms to 1s time-scales, especially for low QPS.

A particularly interesting property of disk activity patterns is that disks rarely see peak utilization—moderate utilization levels persist for long periods. This observation suggests that active low-power modes that provide less than 2-3x performance degradation (i.e., that can be used in periods of 30-50% utilization) at time scales of a few seconds may be quite useful. Unfortunately, most available or proposed power modes for disks act on time scales substantially greater than one second [81, 143]. Accordingly, this data suggests that the main challenge in saving disk power lies in reducing power mode transition time. Alternatively, a non-volatile storage solution that could trade bandwidth for power would be useful.

9.2.4 Analysis: Evaluating Available/Potential Power Modes

Using the activity graphs, we now examine power-savings effectiveness, that is, the product of applicability and power reduction, for existing and proposed power modes. Table 9.1 lists the idle and active power modes we consider for CPU, memory, and disk, as well as our assumptions for transition time (T_{tr}), utilization threshold ($u_{\text{Threshold}}$), and normalized power savings, while using a power mode ($\frac{\Delta P_{\text{Mode}}}{P_{\text{Nominal}}}$). We use these parameters to derive an estimate of the potential per-component power savings at each QPS level. Here P_{Nominal} is the power consumed under normal operation and P_{Mode} the power when the mode is active. For the ideal V_{dd} scaling modes, we assume that $\frac{P_{\text{Mode}}}{P_{\text{Nominal}}} = \left(\frac{f}{f_{\text{Max}}}\right)^3$, which is an optimistic upper bound on the potential of frequency and voltage scaling. Note that, to our knowledge, no current memory device can scale supply voltage during operation; however, we include it as a proxy for other potential memory active low-power modes. Disk active mode savings are taken from [143]. The quantity $\frac{P_{\text{Mode}}}{P_{\text{Nominal}}}$ for idle power modes is simply the ratio of the idle mode power to the power at idle without the mode. We calculate the normalized per-component power savings using the following model:

$$\begin{aligned}\frac{P_{\text{Saving}}}{P_{\text{Nominal}}} &= \frac{P_{\text{Nominal}} - (1 - A(L,U)) \cdot P_{\text{Nominal}} + A(L,U)) \cdot P_{\text{Mode}}}{P_{\text{Nominal}}} \\ &= 1 - ((1 - A) + A \cdot \frac{P_{\text{Mode}}}{P_{\text{Nominal}}}) = A \cdot \frac{\Delta P_{\text{Mode}}}{P_{\text{Nominal}}}\end{aligned} \tag{9.2}$$

Figure 9.8 reports the power savings opportunity for each low-power mode at each QPS level. The ACPI C3 state provides minimal opportunity because it does not lower CPU power much below the savings already achieved by C1E, which current processors auto-

Table 9.1: Low-power mode characteristics.

Power Mode	T_{tr}	$u_{\text{threshold}}$	$\frac{\Delta P_{\text{Mode}}}{P_{\text{Nominal}}}$
C1E $\rightarrow$ ACPI C3	10 μs	Idle	2%
C1E $\rightarrow$ ACPI C6	100 μs	Idle	44%
Ideal CPU V_{dd} Scaling	10 μs	50%	88%
Ideal Memory V_{dd} Scaling	10 μs	50%	88%
Dual-Speed Disk	1 sec	50%	59%
Disk Spin-Down	10 sec	Idle	77%

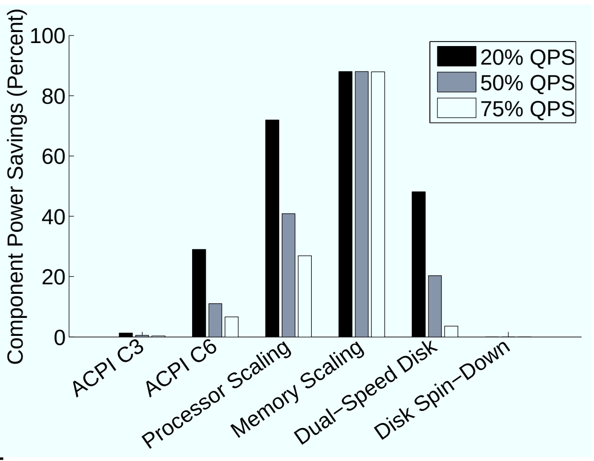


Figure 9.8: Power savings potential for available low-power modes.

matically enter upon becoming idle [97] On the other hand, ACPI C6 (also referred to as power-gating or core-parking for the Nehalem architecture) can provide moderate savings, especially at low-utilization. Using scaling for processor and memory provide substantial opportunity for benefit. Though the opportunity for CPU scaling decreases with increasing load, the opportunity for memory scaling remains large. Disk spin-down is inapplicable to this workload due to its prohibitive transition latency. However, a dual-speed disk has moderate power savings potential, especially at low QPS. Improving transition latency could further increase the opportunity for dual-speed disks, but we leave this study for future work.

9.3 Leaf Node Performance Model

The activity graphs reveal the potential of low-power modes with respect to throughput constraints, but do not consider whether their impact on service latency will be acceptable. As noted in Section 9.1, OLDI services place tight constraints on 95th-percentile latency. To predict impacts on query latency, in this section, we develop a performance model that relates per-component slowdowns to the overall query latency distribution. Our validation against hardware measurements shows that our model estimates 95% query latency to within 9.2% error. In Section 9.4, we will apply this model to estimate the power-performance pareto frontiers of a variety of current and hypothetical CPU and memory low power modes for various *service level agreement* (SLA) targets on 95th-percentile query latency. We elected not to explore disk power/performance trade-offs based on the results in Section 9.2.3.

Query latency is composed of time a query spends in service and the time it must wait to receive service:

$$L_{\text{Query}} = L_{\text{Service}} + L_{\text{Wait}} \tag{9.3}$$

We first construct a model to predict the impact of reduced component performance on L_{service}, the expected time to execute a single query without queuing. We then incorporate the service model into a queuing model to estimate L_{wait} and determine the total latency, L_{Query} as a function of low-power mode settings and contention.

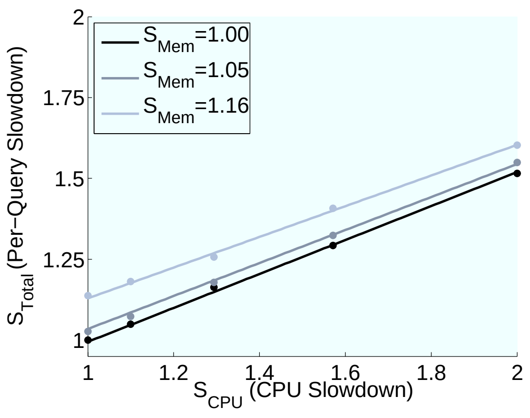
9.3.1 Modeling L_{Service}

Rather than model specific low power modes, we instead capture directly the relationship between per-component performance and query service time; our model can then be applied to existing/hypothetical per-component low power modes by measuring/assuming a specific power-latency tradeoff for the mode. We develop the query service time model empirically by measuring the impact of per-component slowdowns in a controlled test environment. We then perform a linear regression on the measured performance results to derive the overall service time relationship.

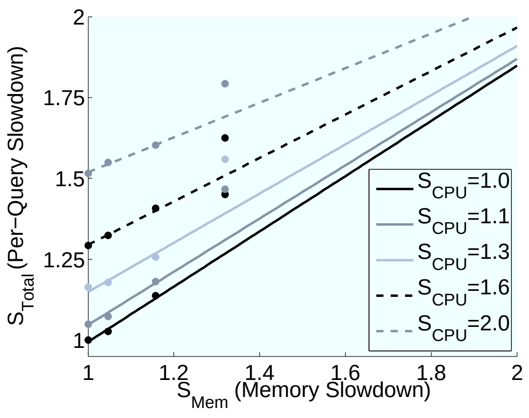
We replicate the behavior of a production leaf node on a 16-core x86 server with 32GB of memory. Our test harness reproduces an arrival process, query set, and web index comparable to production environments. Our test server allows us to vary the CPU and memory performance and measure their impact on average query execution time. We vary processor performance using frequency scaling; we scale all cores together at the same clock frequency.

The most effective method to vary memory latency in our test environment is to decrease the interconnect speed between the processor and DRAM. Varying interconnect speed has the potential to introduce both latency and bandwidth effects. To construct a platform agnostic model, we wish to exclude bandwidth effects (which are specific to the memory system implementation of our test harness) and capture only latency effects in our model; the latency model can be augmented with bandwidth constraints as appropriate. Bandwidth effects are generally small at low utilization [164]. However, for this workload and test system, we have determined that memory bandwidth becomes a constraint at the slowest available interconnect speeds. Hence, we have chosen to exclude these settings from our regression, but report the measured values in our figures for interested readers. We also include measurements from one of the excluded settings in our validation to expose these bandwidth-related queuing effects.

Given component slowdowns S_{CPU} and S_{Mem} (the relative per-query latency increase in



(a) Avg. query slowdown as a function of CPU slowdown (holding memory slowdown constant).



(b) Avg. query slowdown as a function of memory slowdown (holding CPU slowdown constant). Data points at S_{mem} = 1.3 exhibit test-harness-specific memory bandwidth effects and are excluded from the regression.

Figure 9.9: Per-query slowdown regression. Dots represent measured values and lines represent values predicted by the regression model.

the processor and memory) we would like to determine the function $S_{\text{Total}} = f(S_{\text{CPU}}, S_{\text{Mem}}) = \frac{L_{S_{\text{CPU}},S_{\text{Mem}}}}{L_{\text{Nominal}}}$ where S_{Total} is the overall per-query slowdown.

First, we measured the latency of queries delivered to the leaf node one at a time (hence eliminating all contention effects). This experiment confirmed our intuition that the relationship between CPU and memory slowdown and per-query slowdown is linear, providing a sanity check on our use of a linear regression model.

Next we measured the $S_{\text{Total}}, S_{\text{CPU}}, S_{\text{Mem}}$ triple for each available processor and memory setting while loading the leaf node to capacity, but only issuing new queries once outstanding queries complete. This experiment captures the interference effects (e.g., in on-chip caches) of concurrent queries while isolating service latency from queuing latency. Because of the previously-discussed bandwidth effects, we can use only three memory-link-speed settings to approximate the slope of the S_{Mem} relationship. Whereas the absolute accuracy of S_{Total} estimates will degrade as we extrapolate S_{Mem} beyond our measured results, we nevertheless expect that general trends will hold.

We empirically fit the measured data using a linear regression on an equation with the form:

$$S_{\text{Total}} = \mathbf{S}\boldsymbol{\beta}, \quad \boldsymbol{S} = [1,\ S_{\text{CPU}},\ S_{\text{Mem}},\ S_{\text{CPU}} \cdot S_{\text{Mem}}] \quad \boldsymbol{\beta} = [\beta_0,\ \beta_1,\ \beta_2,\ \beta_3]^T \qquad (9.4)$$

Using a least-squares regression, we find that the $\boldsymbol{\beta}$ vector has the values:

$$\boldsymbol{\beta} = [-0.70,\ 0.84,\ 1.17,\ -0.32]^T \tag{9.5}$$

This formulation predicts S_{Total} with an average error of 1% and a maximum error of 1.2%.

The values of the β parameters yield intuition into the workload's responds to component slowdown. The β_1 and β_2 values show a strong linear relationship between application slowdown and CPU and memory slowdown. Interestingly, the negative value of β_3 indicates that slowing CPU and memory together may be beneficial.

Figures 9.9(a) and 9.9(b) provide two views of how CPU and memory slowdown affect average query latency. They further illustrates how our regression predictions (represented by solid lines) compare to measured values (represented by points).

9.3.2 Modeling L_{Wait}

To capture query traffic and contention effects, we augment our per-query service time model with a queuing model describing the overall behavior of a leaf node. We model the leaf node as a G/G/k queue—a system with an arbitrary interarrival and service time distribution, average throughput λ (in QPS), average service rate μ (in QPS), k servers, average load $\rho = \frac{\lambda}{\mu \cdot k}$ and a unified queue. Whereas this model does not yield a closed-form analytic expression for latency, it is well-defined and straight-forward to simulate. We use the approach of Chapter 5 to simulate the leaf-node queuing system.

Interarrival Distribution. We have found that the distribution of query arrival times at the leaf node can greatly influence 95th-percentile latencies. Naïve loadtesting clients tend to send queries at regular intervals, resulting in fewer arrival bursts and fewer, longer idle intervals than what is seen with actual user traffic. Figure 9.10 shows how different assumptions for interarrival distribution influence predictions for 95th-percentile latency; the difference is most pronounced at moderate QPS. We have found that actual user traffic tends to show higher coefficient of variation ($\text{C}_\text{v} = 1.45$) than an exponential distribution and considerably higher than a naïve (Low C_v) arrivals. We have modified our loadtesting client to replay the exact distribution seen by live traffic and were able to validate these effects. Our validation demonstrates the perils of using naïve loadtesting clients, and suggests that in absence of real user traces, exponential distributions could be used with reasonably small errors.

Figure 9.11 depicts the production system arrival distribution at a leaf node. The interarrival distribution has distinguishing notches at $.007(1/\mu)$ and $.01(1/\mu)$. This shape is an artifact of the multi-tier server topology shown in Figure 9.3. We have also observed that

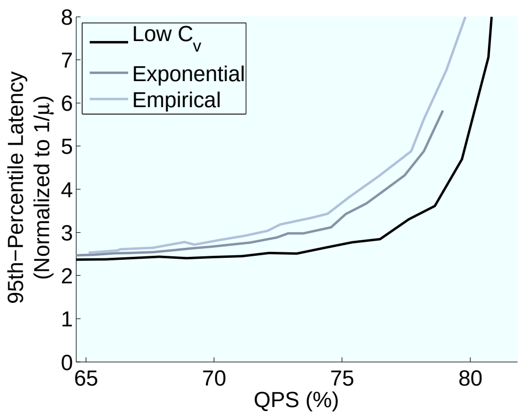


Figure 9.10: Arrival process greatly influences quantile predictions.

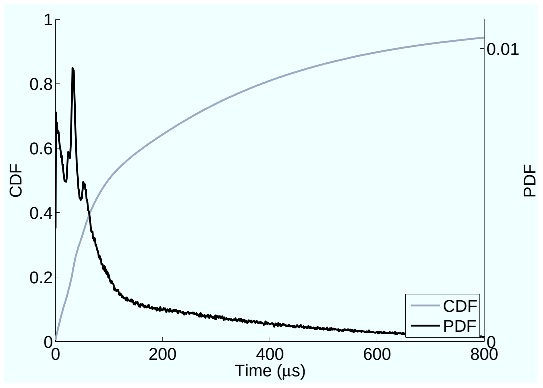


Figure 9.11: Distribution of time between query arrivals.

the shape of the interarrival distribution does not change significantly with varying QPS, only its scale. This property allows us to use the distribution measured at a single QPS and scale it to achieve an arbitrary throughput for our model.

Service Distribution. The service time distribution describes the amount of time spent executing each query. It is parameterized by μ, the average service rate (in QPS); $1/\mu$, the average query execution time, is given by the L_{Service} model described in Section 9.3.1. Since multiple queries are serviced concurrently, the aggregate service rate is $k \cdot \mu$. In our case, k is 16.

The service time distribution measured is shown in Figure 9.12. Though the variance of this distribution is not particularly high (C_v=1.12), the 95th-percentile of the distribution is nearly 3 times greater than the mean. This variability is largely intrinsic to the computing requirements of each individual query.

The service distribution shape, as with the interarrival distribution, does not change significantly with average query latency changes, allowing us to model power modes' effects as a service rate modulation, $\mu' = \mu / S_{\text{Total}}$.

Autocorrelation. Our simulation approach detailed in Chapter 5 generates interarrival and service times randomly according to the scaled empirical distributions, which assumes that the arrival/service sequences are not autocorrelated (i.e., consecutive arrivals/services are independent and identically-distributed). We have validated this assumption in traced arrival and service time sequences (there is less than 5% autocorrelation on average). This independence is important to establish, as it allows us to generate synthetic interarrival/service sequences rather than replaying recorded time traces.

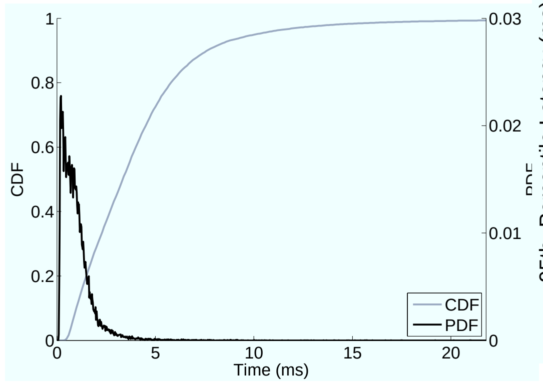


Figure 9.12: Distribution of query service times.

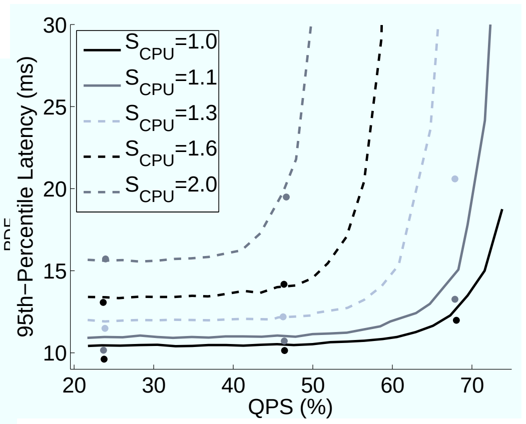


Figure 9.13: Performance model validation. Dots represent values measured on the test node and lines represent modeled values.

9.3.3 Validation

To validate our performance model, we measured the performance of the web search workload at the three QPS levels of interest and all applicable performance settings. Figure 9.13 compares 95th-percentile latency predicted by our model (lines) against measured values on our test node (points). At low QPS, the predicted latency is primarily a function of the L_{Service} performance model, as queuing rarely occurs. As QPS reaches the maximum level, queuing effects and our model for L_{Wait} increase in importance; our overall model predicts 95th-percentile latency accurately in both regions, achieving an average error of 9.2%.

9.4 Evaluating Latency-Power Tradeoffs

Our performance model allows us to quantify the trade-offs between power and latency for both proposed and currently available power modes for our OLDI workload. In this section, our analysis will draw the following conclusions: (1) Full-system, coordinated low-power modes provide a far better latency-power tradeoff than individual, uncoordinated modes. (2) Idle low-power modes do not provide significant marginal power savings over C1E. (3) There is a need to develop full-system, coordinated active low-power modes, because full-system idleness is difficult to find.

We assume similar server power breakdown and power scaling as in [33]; these assumptions are summarized in Table 9.2. In this analysis, we consider processor and memory voltage and frequency scaling, processor idle low-power modes, and batching queries to create and exploit full-system power modes (i.e., PowerNap [122]).

9.4.1 Active system power modes: Voltage and frequency scaling

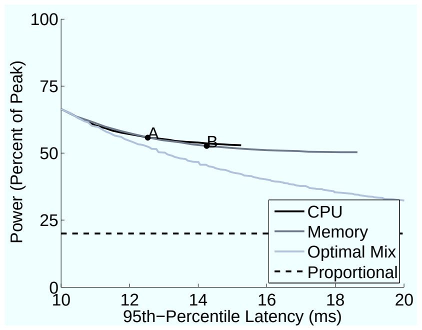
First, we investigate the applicability of active low-power modes for CPU and memory; in particular, we assess the power savings of frequency and voltage scaling. In an ideal CMOS processor, voltage scales proportionally to frequency. Accordingly, power consumption is reduced cubically with respect to processor frequency ($P \propto f^3$). In reality, however, reducing processor frequency does not allow an equivalent reduction in processor voltage [44, 162]. Divergence from the ideal model is due to the inability to scale voltage with the required ideal linear relationship to frequency, and the increasingly large static power consumed by modern processors. This divergence is much more pronounced in high-end server processors than low-end processors, because by definition they are optimized for performance (low threshold voltage and high supply voltage) rather than power (high threshold voltage and low supply voltage).

In addition, limited voltage scaling and automated idle power modes (e.g., C1E) lead to essentially constant idle power, no matter what voltage/frequency mode is selected. To provide an optimistic scenario, we evaluate the expected power savings assuming ($P \propto f^{2.4}$), which was derived from an embedded processor [44] and matches theoretical predictions of practical scaling limits [177]. For this case, we also optimistically assume that the ratio of static power to active power remains constant. Data from today's high end servers [8] suggests much more modest power savings associated with DVFS, with energy savings approaching zero for some DVFS modes.

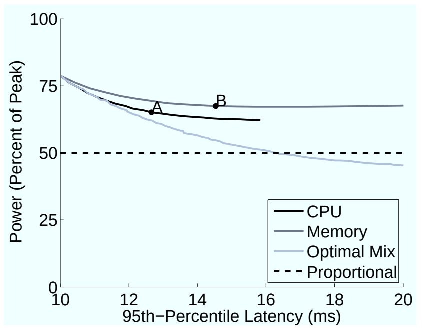
Although full-featured voltage and frequency control (e.g., DVFS) is not currently available for modern memory systems, the power consumption of these devices follow similar relationships with respect to voltage and frequency [100]. We therefore hypothesize a memory power mode that resembles the optimistic processor frequency and voltage scaling scenario.

Figure 9.14 shows total server power as a function of 95th-percentile latency for each QPS level. Three scenarios are considered: CPU scaling alone ("CPU"), memory scaling alone ("Memory") and a combination of CPU and memory scaling ("Optimal Mix"). Since there are many permutations of CPU and memory settings, we show only the pareto-optimal results (the best power savings for a given latency).

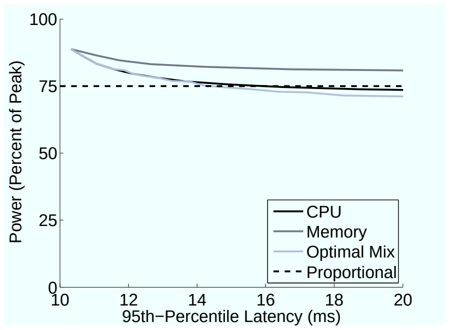
In isolation, the latency-power tradeoff for CPU scaling dominates memory scaling. Memory slowdown impacts overall performance more and uses less power at higher QPS. However, at 20% one can see that memory scaling may do better at times. Different server configurations may shift this balance; in particular, recent trends indicate that the fraction



(a) 20% QPS.



(b) 50% QPS.



(c) 75% QPS.

Figure 9.14: System power vs. latency trade-off for processor and memory scaling ($P \propto f^{2.4}$). The point ”A” represents $S_{\text{CPU=1.5}}$ and ”B” represents $S_{\text{Mem}} = 1.5$. “A” and “B” do not appear in the 75% QPS graph because the latency exceeds 20 ms.

Table 9.2: Typical server power breakdown.

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

of power due to memory will increase. Our results show that coordinated scaling of both CPU and memory yields significantly greater power reductions at all target latencies than scaling either independently. This observation underscores the need for coordinated active low-power modes: components must scale together to maintain system balance. With our optimistic scaling assumption, we can achieve better than energy-proportional operation at 50% and 75% QPS. At 20% QPS, power consumption becomes dominated by other server components with large idle power (e.g., disk, regulators, chipsets, etc.).

9.4.2 Processor idle low-power modes

Next, we explore idle CPU power modes, where the CPU enters a low-power state but other components remain active. Modern CPUs already use the ACPI C1 or C1E state whenever the HLT instruction is executed. We would like to understand the benefit of using deeper idle low-power modes. For comparison, we evaluate using ACPI C6, which can be applied either at the core or socket level. At the core level, ACPI C6 or “Core Parking” uses power gating to eliminate power consumption for that core. It has a transition time of less than 100 μs [**?**]. At the socket level (“Socket Parking”), we assume that an integrated memory controller must remain active but all caches may be powered down yielding a 50% reduction in non-core power consumption. This mode requires socket-level idleness and incurs a 500 μs transition time. For both modes, we assume that the mode is invoked

Table 9.3: Processor idle low-power modes.

	Power (% of Peak)			
	Active	Idle (HLT)	Parking	
			Core	Socket
Per-Core (x4)	20%	4%	0%	0%
Uncore	20%	20%	20%	0%

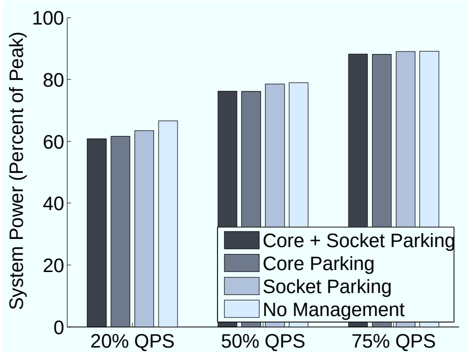


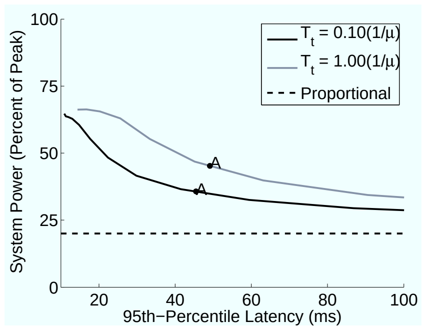
Figure 9.15: System power savings for CPU idle low-power modes. Core Parking only yields marginal gains over C1E. Power savings from socket parking is limited by lack of per-socket idleness.

immediately upon idleness and that transition to active will occur immediately upon a query requiring the resource. The power consumption of a socket in each state is summarized in Table 9.3.

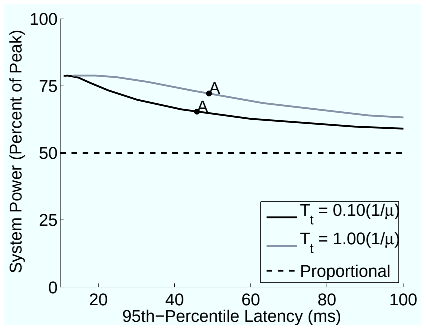
Figure 9.15 depicts the power savings opportunity for processor idle low-power modes. The transition time of these modes is sufficiently low that the 95th-percentile is not affected significantly. We observe that ACPI C6 applied at the core level (“Core Parking”), socket level (“Socket Parking”) or in combination (“Core + Socket Parking”) does not provide a significant power reduction. Since modern processors enter C1E at idle (i.e., the HLT instruction), the idle power of the processor is not very high. Therefore, compared to a baseline scenario (“No Management”), using core or socket parking does not provide much benefit. Larger power savings are likely possible if work is rescheduled amongst cores to coalesce and thus extend the length of the idle periods, enabling the use of deeper idle power modes.

9.4.3 Full-system Power Modes: Query batching with PowerNap

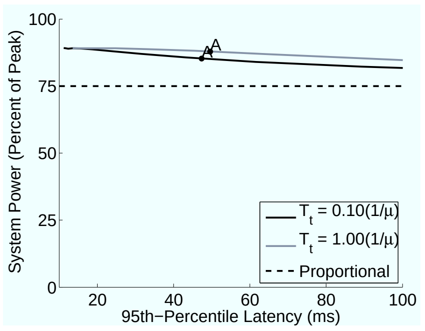
Finally, we investigate using a full-system idle low power mode to recover power consumed by all server components. We evaluate PowerNap, which provides a coordinated, full-system low-power mode with a rapid transition time to take advantage of idle periods



(a) 20% QPS.



(b) 50% QPS.



(c) 75% QPS.

Figure 9.16: System power vs. latency trade-off for query batching with PowerNap. "A" represents a batching policy that holds jobs for periods equal to 10x the average query service time.

and has shown great promise for non-OLDI server applications [122]. We assume that while in PowerNap mode, the server consumes 5% of peak but cannot process queries.

It is extremely rare for full-system idleness to occur naturally for our workload, even at low QPS values. Instead, we coalesce idle periods by batching queries to the leaf node. Queries are accumulated at a higher level node in the search tree and released after a given timeout to the leaf node. Increasing the timeout allows for longer periods in the PowerNap mode. Our approach is similar to the technique described in [70].

Figure 9.16 shows the power-latency tradeoff given a transition time, T_t, of both one tenth and equal to the average query processing time $(1/\mu)$. (Note the larger scale of the horizontal axis in this figure relative to Figure 9.14). We find that in order to save an appreciable amount of power, PowerNap requires relatively long batching intervals incurring large latency penalties. The point ("A") represents a batching policy with a timeout period equal to 10x the average query service time. Even with a relatively short transition time, the latency-power trade-off is not enticing across all the QPS levels.

Clearly, using a simple batching method to artificially create idleness is unattractive from a latency perspective. A more intelligent architecture is required, and we have proposed such a system, DreamWeaver, in Chapter 7. We direct the reader to Chapter 7 for an evaluation of DreamWeaver with respect to the Web search workload.

9.4.4 Comparing Power Modes

Out of the power modes explored thus far, we would like to identify the best mode given an SLA latency bound. For this evaluation, we define our initial SLA to be the 95th-percentile latency at 75% QPS. Figure 9.17 compares the power saving potential of system active low-power modes ("Scaling") from Section 5.1, processor idle low-power modes

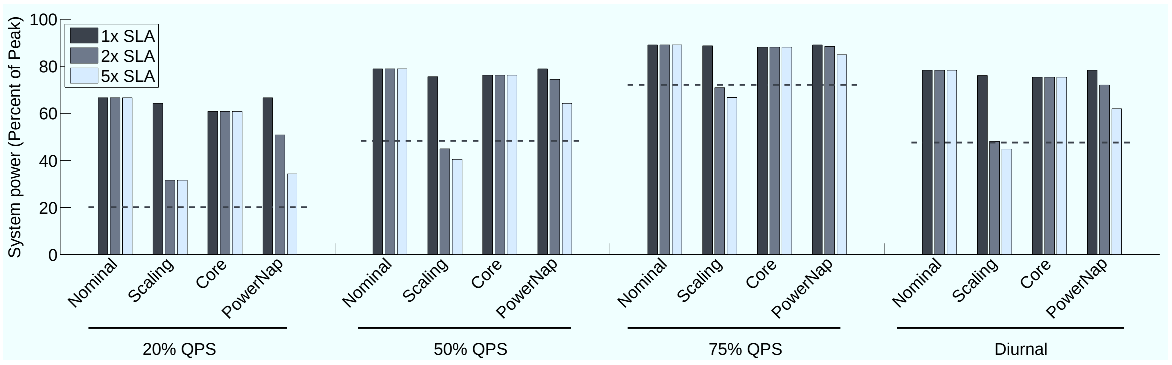


Figure 9.17: Power consumption at each qps level for a fixed 95th-percentile increase. The dotted line at each QPS level represents the power consumption of an energy proportional system."Diurnal" represents the time-weighted daily average from Figure 9.1. An energy-proportional server would use 49% of its peak power on average over the day.

("Core") from Section 5.2, and system idle low-power modes ("PowerNap") from Section 5.3 given a latency bound of 1x, 2x, and 5x of this initial SLA. At 20% QPS, none of the power management schemes can achieve energy-proportionality (a power consumption of 20% of peak) even at a 5x latency increase, although scaling falls just short. For 50% and 75% QPS, coordinated scaling can achieve slightly better than energy proportional power consumption for a 2x latency increase, but other techniques cannot regardless of SLA.

To understand the overall daily potential for power savings, we show the time-weighted average power consumption ("Diurnal") using the QPS variations shown in Figure 9.1. Once again, only scaling is able to achieve a power consumption at or better than energy proportional operation for a 2x or greater latency increase. This result supports our claim that OLDI services can achieve energy-proportionality only with coordinated, full-system scaling active low-power modes; other power management techniques simply do not provide a suitable power-latency tradeoff for the operating regions of this workload class.

CHAPTER 10

Architecting Cost-Effective Data Center Memcached Systems

In this chapter, we examine the architecture of both an individual `memcached` server and an entire cluster to determine how to design the most cost-effective cluster as a function of the cluster's workload and scale. In doing so, we discover stark inefficiencies in modern systems that hamper `memcached` performance. We develop a load-testing methodology and infrastructure to allow us to reproduce precisely-controlled object size, popularity, and load distributions to mimic the traffic a `memcached` server receives from a large client cluster. Furthermore, we develop a `memcached` benchmark suite that captures a range of use cases modeled after popular sites using data sets captured off the web. We then carry out an extensive measurement study using performance counters and profiling tools to investigate the microarchitectural and system-level behavior of each `memcached` use case across two system and three network interface designs to shed light on system bottlenecks. In most cases, a `memcached` server is unable to saturate available network bandwidth due to a range of software and hardware inefficiencies.

We categorize these inefficiencies in Figure 10.1, which we use to structure the first part of our study. This figure demonstrates `memcached`'s throughput scaling with respect to core count for fixed-size objects. If this server could saturate its network bandwidth capacity, it would be able to service just over one million requests per second ("Theoretical 1GbE"). However, our measurements of a high-performance Xeon server ("Base System") fall short of this throughput by nearly an order of magnitude. There are two causes of this performance gap: (1) `memcached` does not scale well with core count on current systems and (2) modern processor microarchitectures perform poorly on `memcached`, primarily (and perhaps surprisingly) due to poor front-end performance. We identify and quantify the three components of non-scalability: a lack of load-balancing hardware support in the network interface controller (NIC) ("NIC Scalability"), lock contention in the `memcached`

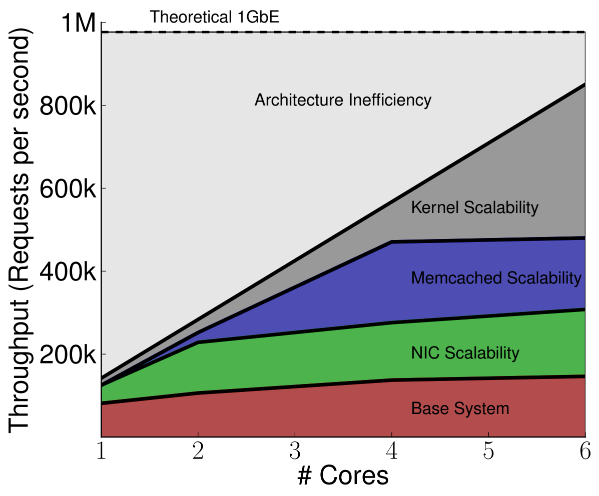


Figure 10.1: `Memcached` performance decomposition. A Xeon-class server ("Base System") can only achieve 15% of the throughput potential of a 1GbE NIC and does not make good use of additional cores. This figure enumerates the factors preventing `memcached` from saturating a 1GbE link when caching 128-byte objects. Advanced NIC features (discussed in Section 10.3.3) double achievable throughput ("NIC Scalability"). Resolving userspace contention (e.g., locks within `memcached`) improves throughput another 75%. Improving kernel scalability could enable another 75% gain. The remaining gap between linear scaling (w.r.t. core count) and the theoretical 1GbE bandwidth arises from the inefficiency of modern server CPU architectures when executing TCP/IP-intensive workloads, which we explore in Section 10.3.1. Note that, with all scalability bottlenecks removed, a modest 6-core system falls just short of saturating a 1GbE link. Looking forward to 10GbE NICs, saturating available network bandwidth would require nearly 64 Xeon-class cores or vast improvements to TCP/IP processing efficiency.

application ("Memcached Scalability"), and bottlenecks in the Linux Kernel ("Kernel Scalability"). A perfectly scalable system would be able to reach nearly 80% of theoretical throughput, but still falls short due to microarchitectural inefficiencies (e.g., caching stalls, VM translation overheads and unpredictable branches). This shortfall should be concerning to architects—Xeon cores are quite capable, yet six of them cannot even saturate a 1GbE link. Looking forward to the adoption of 10GbE, it would take 64 Xeon cores to saturate available network bandwidth.

In a designing a `memcached` cluster, we find these deficiencies greatly influence how architects should select server hardware to economically meet performance goals. The second part of our study investigates the design and scaling of cost-optimal `memcached` clusters. Based on our extensive characterization data, we create a design space optimization framework and perform an exhaustive search over a large design space of cluster architectures to reveal trends in how the cost-optimal server and cluster design varies across workloads, latency targets and scale. Our study exposes an important challenge in provisioning clusters—there are many locally optimal server configurations among the three primary scaling dimensions (data set size, throughput and latency) precluding prescribing a single server configuration for all clusters.

From our single-server characterization and cluster-level study, we highlight our central observations:

Modern processor microarchitectures perform poorly for `memcached` and suffer from numerous frontend stalls. We find that processor cycles per instruction (CPI) can be as poor as 8 for Atom and 2 for Xeon cores. Frequent trips into the TCP/IP stack, kernel, and library code are characterized by poor instruction supply due to ICache misses, virtual memory (VM) translation overheads and branch unpredictability. ICache and ITLB performance is often an order of magnitude worse relative to benchmarks commonly used for processor design. Furthermore, large last-level caches seem to provide little benefit except to bolster instruction supply.

High performance high-core-count systems are not cost effective because of `memcached`'s non-scalability. Due to bottlenecks in the `memcached` application, the Linux kernel and networking hardware itself, additional cores provide marginal gains in throughput. While many workloads benefit from amortizing server costs with multiple cores, our study suggests higher-core-count server models often do not pay for themselves. Instead, configurations with a modest number of cores tend to be favored. Contrary to intuition, using these system with *less* RAM per server allows *higher* performance operation.

Networking hardware features are as important for `memcached` as raw bandwidth and their performance benefits often justify their cost premium. As demonstrated in Figure 10.1, `memcached` servers often cannot even saturate a 1GbE NIC. However, a number of premium features such as Receive-side Scaling, more commonly found in premium and 10GbE NICs, provide a significant performance gain. We find that this benefit is significant enough to justify their steep price premium in a number of scenarios.

Cluster-level performance objective (size, throughput and latency) impose constraints that shift the economically optimal server design. While there are clear trends (e.g., `memcached` favors low core counts), there is no dominant server configuration for all performance goals. Tighter latency constraints in particular necessitate premium components such as Xeon-class processors and enterprise-grade NICs. These shifts make provisioning an optimal cluster difficult as these objectives can grow unpredictably and in different proportions.

10.1 Background

We first briefly describe the operation of `memcached` clusters and discuss related work.

10.1.1 Memcached Clusters

A `memcached` cluster is a high-performance distributed key-value store, implemented through a distributed hash table. Typically, `memcached` is used as a caching layer to reduce response time for a slower backing store (e.g., an SQL DBMS) that provides more robust availability and durability guarantees. The strength of the `memcached` architecture lies in its simplicity—individual servers are independent and do not communicate with one another or the backing store; rather, the cache is managed entirely by library software on the clients. Simplicity enables the scalability of `memcached` clusters to high aggregate capacity and throughput, and allows clusters to achieve impressive response times below 500 μs.

`Memcached` servers manage key-value pairs (a.k.a. *objects*). Under normal usage, each `memcached` server is a best-effort store, and will discard objects under an LRU replacement scheme as main memory capacity is exhausted. Keys may be at most 250 bytes in length, while the stored values must be 1 MB or smaller. Because typical object sizes are small, servers can cache many objects; for example given 4 KB values, a server with 16 GB of DRAM could cache nearly 4 million objects.

Clients access data stored in the `memcached` cluster over the network, maintaining an independent connection to each server. Though `memcached` provides both TCP and UDP interfaces, anecdotal evidence suggests that most large-scale installations use TCP. The `memcached` interface comprises a small number of simple key-value operations, the most important of which are described below.

Figure 10.2 (a) shows a typical client-server data store without `memcached`. A client asks the backing store for the value corresponding to a key (e.g., the friend list for a given user-id). Using robust, yet heavyweight systems such as a SQL DBMS, such requests may take hundreds of milliseconds to complete.

Get. Figure 10.2 (b) illustrates how `memcached` improves data access performance in the case of a cache hit. The client hashes a key to identify the (single) `memcached` server responsible for that key, and sends it a "get" request. Typically, clients use *consistent hashing* (similar to the method in [63]), providing a key-to-node hash that load-balances and can adapt to node insertions and deletions while ensuring that each key maps to only a single server at any time. If available, the server returns the corresponding value within

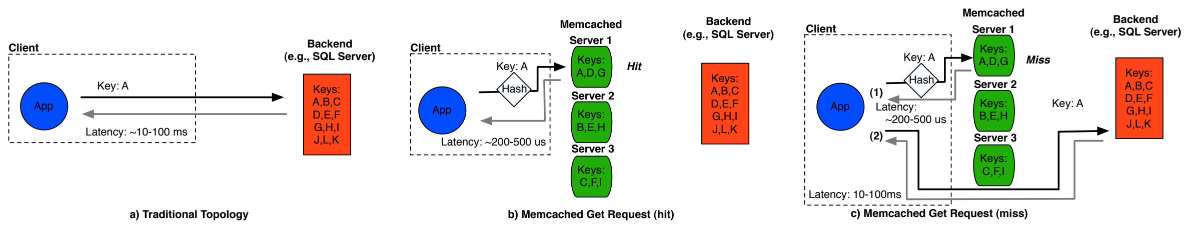


Figure 10.2: Example cluster operations. (a) illustrates data access without `memcached`. Requests often take tens to hundreds of milliseconds. (b) illustrates a `memcached` hit. The client hashes the key to determine which `memcached` server to contact, and sends it the requested key. The server responds within hundreds of microseconds. (c) illustrates a cache miss. Once the `memcached` server reports a miss, the client must fall back to accessing the backend.

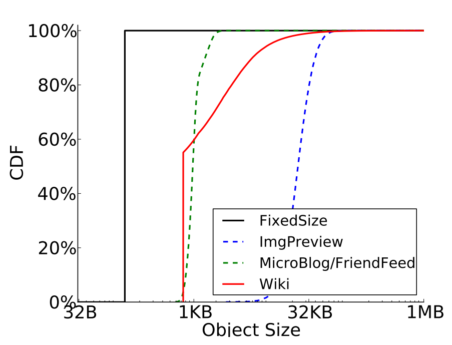
hundreds of microseconds. Figure 10.2 (c) illustrates a cache miss. The server will quickly indicate that the object was not found and the client must then access the backing store. Note that the hash function ensures that it is unnecessary to consult any other `memcached` servers.

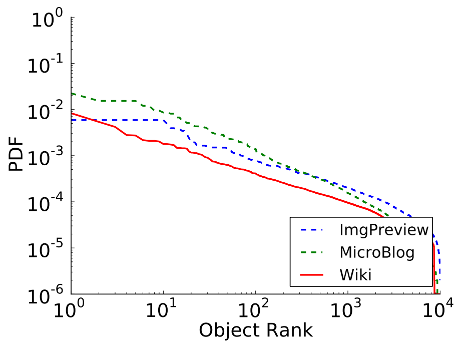
Set. A client may add an object to the cache via a "set" request. The key is hashed to identify the correct server, and the object is stored in DRAM. If no space is available at the server, another key is evicted to make room via LRU replacement.

Multiget. Often, applications wish to access numerous objects at once (e.g., retrieving the status of all a user's friends in a social networking application). To minimize the network overheads of requesting many objects and simplify the `memcached` library interface, the client software may pass the `memcached` client library a list of keys to retrieve. The library then hashes each key, sorts them according to their destination server, and sends each server a single "multiget" request for the objects it might hold. These multigets are then processed in parallel (both within and across servers). Returned values (and miss notifications) are provided in a single response packet per server and the aggregate list is returned to the client software after the last response is received.

10.2 Methodology

We next describe our methodology for stress-testing `memcached` performance and our efforts to develop workloads that approximate real-world `memcached` deployments. Previous studies of `memcached` have used simple load testing tools, such as `memblaster` or `memslap`. As our results will demonstrate, the manner in which `memcached` is loaded drastically alters its behavior and hardware requirements. In particular, we find that the typical object size has a large impact on system behavior. Existing load testing tools use





Name	Single/Multi-get	% Writes	Object Size				Type
			Avg.	Std. Dev.	Min	Max	
FixedSize	Single-get	0%	128 B	0 B	128 B	128 B	Plain Text
MicroBlog	Single-get	20%	1 KB	0.26 KB	0.56 KB	2.7 KB	Plain Text
Wiki	Single-get	1%	2.8 KB	10.4 KB	0.30 KB	1017 KB	HTML
ImgPreview	Single-get	0%	25 KB	12.4 KB	4 KB	908 KB	JPEG Images
FriendFeed	Multi-get	5%	1 KB	0.26 KB	0.56 KB	2.7 KB	Plain Text

Figure 10.3: Workload characteristics. We construct a set of workloads to expose the broad range of `memcached` behavior. The object size distributions of the workloads differ substantially. Our measurements show that popularity for most web objects follows a zipf-like distribution, although the exact slope may vary.

either a fixed or uniform object size distribution and a uniform popularity distribution. Our measurements using real objects and popularity statistics (captured from public web sites known to use `memcached`) suggest simple synthetic load generators can yield misleading conclusions.

10.2.1 Understanding `memcached` as a workload

Though its interface is quite simple, `memcached` behavior can vary considerably based on the size and access frequency of the objects it stores (the actual values are opaque byte strings and do not affect behavior). Hence, we have designed a suite of five `memcached` workloads, each based on usage scenarios from Web 2.0 and social networking applications, that capture a wide range of behavior. We emphasize that our goal is not to replicate the precise workloads of existing web sites, but rather to create a set of easy-to-understand yet realistic micro-benchmarks that expose `memcached` performance sensitivity.

A workload is defined by an object size distribution, popularity distribution, fraction of set requests, and whether or not multi-get requests are used. Summary statistics for each of our workloads are provided in Figure 10.3. The measured popularity distributions are shown in Figure 10.3 (probability as a function of rank). The zipf-like distribution is consistent with a previous study by Cha et al [50].

FixedSize. The "FixedSize" workload is the simplest, using a fixed object size of 128 B and uniform popularity distribution. We include this workload because small objects place the greatest stress on `memcached` performance; anecdotal evidence suggests production clusters frequently cache numerous small objects.

MicroBlog. Our "MicroBlog" workload represents a number of social networking sites that distribute brief user status updates. We base the object size and popularity distribution on a sample of "tweets" (brief messages shared between Twitter users) collected from Twitter. The text of a tweet is restricted to 140 characters, however, associated meta-data brings the average object size to 1KB with little variance. Even the largest tweet objects are under 2.5KB in size. As we will show, the small, tight object size distribution leads this workload to favor architectures with lower total network bandwidth but higher packet processing rates than other workloads.

Wiki. Our "Wiki" workload caches articles from Wikipedia.org [14]. We use the entire Wikipedia database, which has over 10 million entries. Each object represents an individual article in HTML format. Articles are relatively small, 2.8 KB on average, but have a notable variance because some articles have significantly more text. This workload exhibits the highest normalized object size variance (relative to the mean) of any of our workloads. Object popularity is derived from the page view count for each article.

ImgPreview. The "ImgPreview" workload represents photo objects used in photo-sharing sites. We collect a sample of 873 thousand photos and associated view counts from Flickr [9]. In particular, Flickr provides a set of "interesting" photos every day; we collect these photos because they are likely to be accessed frequently. Photo sharing sites often offer the same images in multiple resolutions. In a typical web interaction, a user will view many low-resolution thumbnails before accessing a high-resolution image. We collect a sample of these thumbnails, which are 25 KB, on average, in size. We focus on thumbnails because larger files (e.g., hi-resolution images) are typically served from data stores other than `memcached` (e.g., Facebook's Haystack system [35]).

FriendFeed. We construct the "FriendFeed" workload to understand the implications of heavy use of multi-get requests. Facebook has disclosed that multi-gets play a central role in its use of `memcached` [71]. This workload seeks to emulate the requests required to construct a user's Facebook Wall, a list of a user's friends' most recent posts and activity. Because Facebook prohibits crawling their network, we reuse the distributions from our MicroBlog workload, as we believe Facebook status updates and tweets have similar size and popularity characteristics. We assume a typical Facebook user has 100 friends, each with numerous recent posts and activity. Hence, our workload issues multi-get requests of O(100) keys.

	Xeon	Atom
System	Dell R610	Supermicro 5015A-EHF
Processor	1x 2.25 GHz 6-core Xeon (Westmere L5640) 12 MB L3 Cache	1.6 Ghz Atom D510 Dual-core 2x SMT 1 MB L2 Cache
DRAM	3x 4GB DDR3-1066	2x 2GB DDR2-800 SDRAM
Commodity	Realtek RTL8111D Gigabit	–
Enterprise	Broadcom NetXtreme II Gigabit	Intel 82574L Gigabit
10GbE	Intel X520-T2 10GbE NIC	Intel X520-T2 10GbE NIC

Table 10.1: System under test (SUT).

10.2.2 Load Testing Framework

We next describe the load-testing infrastructure we developed to investigate the performance of a `memcached` server under load from a large client cluster. Because `memcached` servers do not interact, we can characterize the performance of only a single server and use this data to derive cluster scaling effects. To be able to fully saturate the target server, we employ several load generator clients that each emulate 100 `memcached` clients via separate TCP/IP connections. Throughout our experiments, we explore the tradeoff between server throughput and response time by varying the request injection rate on our load generator clients.

The load-testing infrastructure first populates and then accesses the `memcached` server according to the specific workload's object size and popularity distributions described in Section 10.2.1. Our implementation leverages the same ultra-fast, asynchronous libevent framework [10] as `memcached`. To maximize efficiency, we use `memcached`'s binary network protocol [11]. Furthermore, we disable Nagle's algorithm, which causes packet buffering, to minimize network latency [131].

The primary objective of our investigation is to determine which aspects of system architecture impact `memcached` performance. Accordingly, we intentionally decouple our study from network switch saturation, since many other studies have investigated efficient techniques to avoid switch over-subscription in data center networks [21, 24, 79]. All experiments use dedicated switches or cross-over links between the load generators and test servers.

10.2.3 Systems under tests

As a key focus of our work is to determine what kind of server is most cost-effective when building a `memcached` cluster of a given scale (in terms of total capacity and desired throughput). Hence, we study two drastically different server systems, a high-end Xeon-based server and a low-cost Atom-based server, and three different classes of network interface cards (NIC), spanning low-cost consumer-grade, manufacturer-installed, and high-end 10GbE NICs. The details of each system are shown in Table 10.1. By exchanging NICs and selectively disabling cores, we evaluate a total of 21 different system configurations.

The Xeon-class system, from Dell, is typical of the `memcached` servers described in media reports. Though our test system includes only 12GB of RAM (lower than is typically reported), we have confirmed that memory capacity has no direct effect on `memcached` latency or throughput. (Note that, as we explore in detail later, memory capacity per node indirectly affects performance because the share of a cluster's overall load directed to a particular server is proportional to the server's share of the overall cluster memory capacity). The Atom system represents a low-cost alternative; we include it because several recent studies have suggested the use of "wimpy" cores for data center workloads [167]. We study several NICs to demonstrate that the feature set of the NIC and driver, rather than the theoretical peak network bandwidth, primarily affect `memcached` performance, particularly for small object sizes.

All systems run Ubuntu 11.04 (2.6.38 kernel) with `memcached` 1.4.5. We gather utilization, response time, bandwidth, and microarchitectural statistics using our load generators, `sysstat` [18], and `perf` [15].

10.3 Single-server Characterization

In this section, we characterize the microarchitectural behavior of a single `memcached` server to show how the bottlenecks identified in Figure 10.1 prevent it from saturating the network link. Our single-server study demonstrates that:

- **Modern processors perform poorly for `memcached`, primarily due to poor front-end performance.** We find that `memcached` exhibits exceedingly poor CPIs on modern cores, as seen in Figure 10.4. The most critical bottlenecks lie in the processor front-end, though address translation misses also play a role. Larger L2 and L3 caches do not provide much benefit, nor would additional memory bandwidth. We explore microarchitectural bottlenecks in Section 10.3.1.

- **There are significant scalability bottlenecks in `memcached`, the Linux kernel and NIC hardware that prevent effective multicore scaling.** Adding additional cores to a server provides little additional throughput, even when network bandwidth is underutilized. We explore multicore scalability in greater detail in Section 10.3.2.

- **Selecting the correct NIC hardware can be as important as selecting the right CPU—networking *quality* (i.e., NIC hardware features) is often more important**

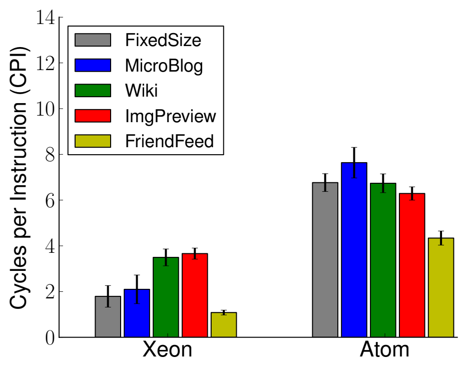


Figure 10.4: Microarchitectural inefficiency with `Memcached`. Modern processors exhibit unusually high CPI for Xeon and Atom-based servers. Xeon-class systems operate at less than one eighth of their theoretical instruction throughput. Atom-class systems fare even worse, providing as little as 1/16th of the theoretical instruction throughput. Compared to other workloads (e.g., SPEC CPU), these CPIs are quite high and demonstrate that current microarchitectures are a poor match for `memcached`. We identify specific microarchitectural hurdles in 10.3.1.

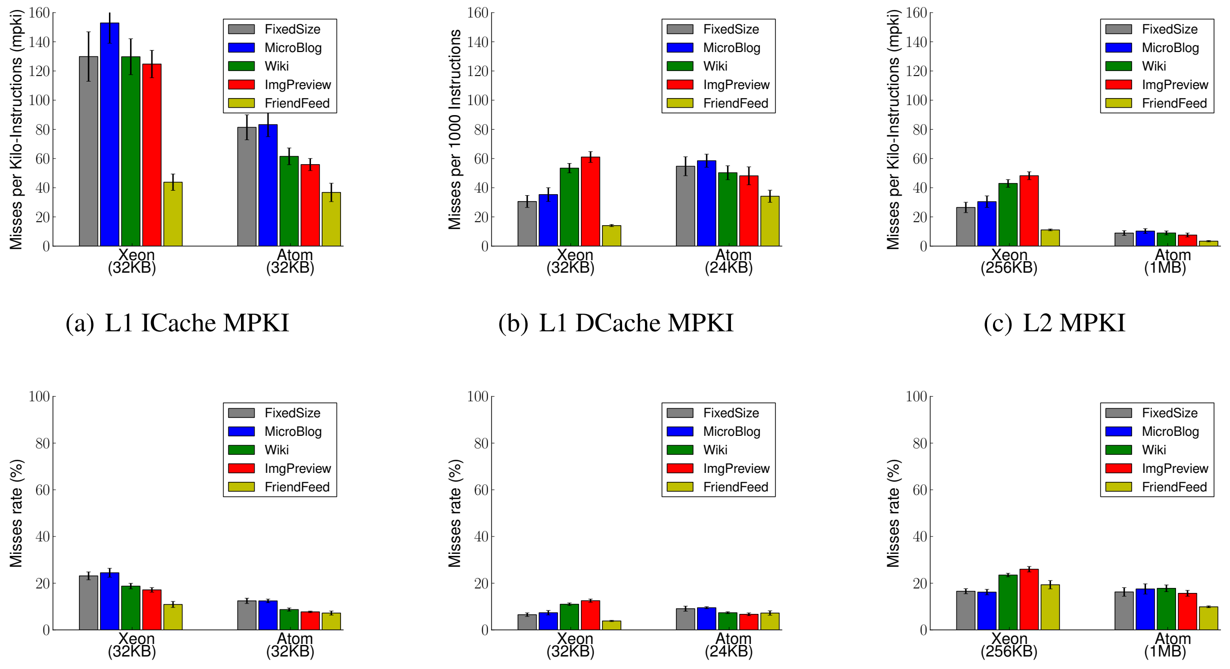
than *quantity* (i.e., network bandwidth). Our measurements demonstrate that NIC hardware plays an important role in increasing efficiency and scalability. NICs with the same theoretical bandwidth exhibit drastically differing performance. Often paying a premium for a better NIC pays for itself in increased performance due to more efficient packet processing. We identify critical NIC features in Section 10.3.3.

10.3.1 Microarchitecural Inefficiency

We first investigate what microarchitectural bottlenecks cause the poor CPI we observe in `memcached`. We gather performance counter data for a variety of microarchitectural structures on both Atom and Xeon cores. We report results for each workload averaged across load levels, with error bars indicating one standard deviation from the mean.

Broadly, our results suggest that modern processors (whether Xeon-class or Atom-class) are ill-suited for `memcached`: Xeons achieve only one eighth and Atoms only one sixteenth of their theoretical peak instruction throughput. Prior to undertaking this microarchitectural study, our expectation was that `memcached` might be memory bandwidth bound, with performance limited primarily by the speed of copying data to outgoing network packets. In fact, measuring bandwidth to main memory, we find it is massively underutilized, always falling under 15% of max throughput and often much less (e.g., 5% for MicroBlog).

Surprisingly, the most significant bottlenecks lie in the processor front-end, with poor instruction cache and branch predictor performance. Neither increased memory bandwidth,



(a) L1 ICache MPKI (b) L1 DCache MPKI (c) L2 MPKI

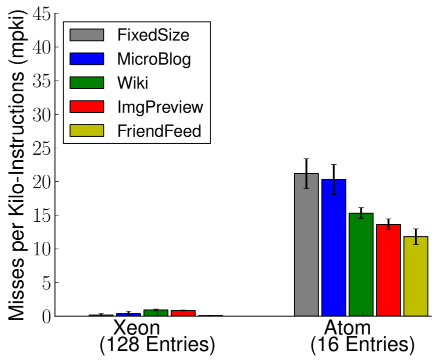
(d) L1 ICache miss rate (e) L1 DCache miss rate (f) L2 miss rate

Figure 10.5: Caching behavior. `Memcached` performance suffers from an inordinate number of ICache misses (over an order of magnitude worse than SPEC benchmarks). L1 data caching performs significantly better. L2 caching performs moderately well, although many of the hits are to service instructions. The Xeon also has a 12MB L3 (data not shown), but it provides little benefit, exhibiting miss rates as high as 95%.

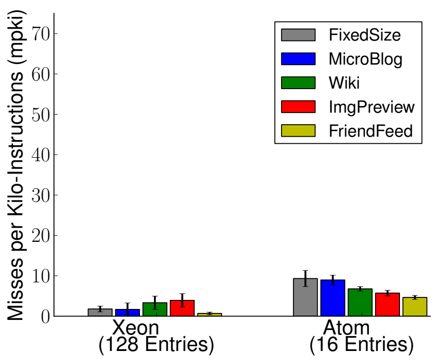
nor larger data caches are likely to improve performance. Future architectures will need to address these bottlenecks to achieve near-wire-speed processing rates. We address caching, address translation, and branch prediction behavior in greater detail.

Caching bottlenecks. Figure 10.5 presents cache performance metrics. Our most surprising finding is that the instruction cache performance of `memcached` is drastically worse than typical workloads. A typical SPEC CPU 2006 integer benchmark incurs at most ten misses per thousand instructions. In contrast, Figure 10.5(a) reveals rates up to 15x worse. Our result is surprising because `memcached` itself comprises little code, fewer than 10,000 source lines. The poor instruction behavior is due to the massive footprint of the Linux kernel—specifically the TCP/IP stack.

L1 data cache behavior is more typical of other workload classes (e.g., SPEC). There are numerous compact but hot data structures accessed as a packet traverses the TCP/IP stack that can be effectively cached in L1. L2 caches are moderately effective, missing between 15-25% of the time as seen in Figure 10.5(f). However, the majority of L2 accesses are



(a) Instruction TLB MPKI



(b) Data TLB MPKI

Figure 10.6: Virtual memory behavior. The Atom microarchitecture suffer from numerous translation misses due to its ITLB mere 16 entries. DTLB misses fall within a nominal range for both processor classes.

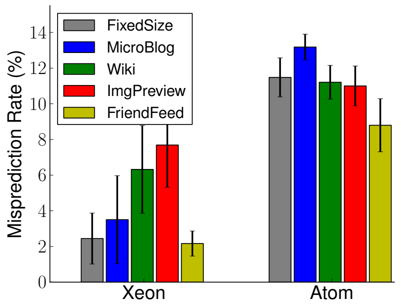
for instructions that do not fit in the L1 instruction cache; if these are removed, the L2 data cache miss rates are substantially higher, in excess of 50%. Unlike the Atom, the Xeon processor also has a large L3 cache. We find the L3 to be highly inefficient, with miss rates from 60% to nearly 95% (we omit graphs due to space constraints), a finding that suggests that increasing data cache sizes will yield little gain.

We observe roughly twice as many instruction cache misses in Xeon than Atom, despite similar instruction cache organizations. This disparity arises because the Atom CPU is slower relative to the NIC than the Xeon, which, for a given packet arrival rate, results in more packet batching at the NIC before delivery to the kernel. Processing packets in batches improves kernel code locality.

Overall, our analysis suggests that `memcached` requires far more instruction cache capacity than cores currently provide (primarily to hold the OS networking stack), but will not gain from larger data caches.

Virtual memory translation bottlenecks. In most applications, TLB misses are rare events and the overhead of virtual memory is hidden. A recent characterization of the Parsec Benchmark suite [38] finds that ITLB miss rates typically occur at a rate of 3x10^{-4} MKPI to 1 MPKI. DTLB miss rates fall generally in the range of 1x10^{-2} to 10 MPKI, but can be as high as 140 MKPI. In Figure 10.6(a) and Figure 10.6(b), we see we see comparable TLB behavior for Xeon, but find that Atom provides an insufficient ITLB (16 entries vs. 128 for Xeon), which contributes significantly to its instruction fetch stalls. DTLB pressure is not a problem for either class of core.

Branch prediction bottlenecks. Figure 10.7 demonstrates the front-end bottlenecks seen



Function		%Time	Misprediction Rate
tcp_sendmsg	(k)	2.12%	10.74%
copy_user_generic_string	(k)	1.81%	10.97%
pthread_mutex_lock		1.04%	37.56%
event_handler		0.82%	13.68%
memcached main		0.77%	18.68%
tcp_clean_rtx_queue	(k)	0.77%	15.23%
kmem_cache_alloc_node	(k)	0.72%	11.43%
dev_queue_xmit	(k)	0.71%	12.19%
e1000_clean_rx_irq	(k)	0.71%	18.27%
sys_sendmsg	(k)	0.63%	12.96%

Figure 10.7: Branch prediction. The tables list the top ten functions with misprediction $>$ 10% out of fifty that consume the most execution time for the Microblog workload running on the Atom system.

in the instruction caches also extend to the branch predictor. Despite numerous bulk memory copies with tight, predictable loops, branch misprediction rates are significant, particularly on the Atom, which has a considerably less capable branch predictor than Xeon. As seen for cache locality, the batching effect of multiget requests in FriendFeed also leads to better branch predictability.

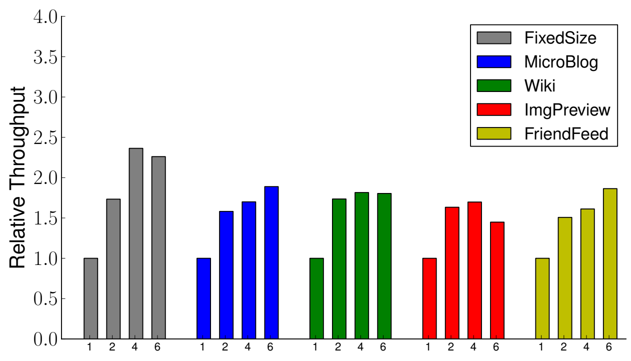
The table in Figure 10.7 lists the top functions by execution time that also have misprediction rate of $> 10\%$. Many of the memory copies that dominate the total execution time are not present in the list. While they have many branches, they're highly predictable with mispredict rates of $< 1\%$.

The entire networking stack is well represented in the table from the entry into the kernel from userspace `sys_sendmsg`, through the tcp stack `tcp_*`, the device layer `dev_*`, and finally the NIC driver `e1000_*`. The networking stack has significant control flow because of error and stateful protocol requirements that must be met. In addition the kernel memory management functions are mainly utilized in this system to allocate and free buffers for network packets.

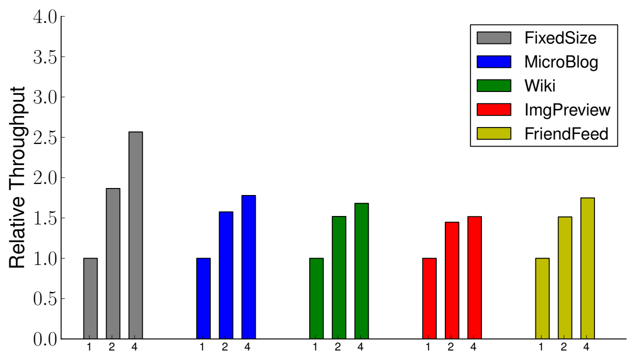
Usercode also contains some highly unpredictable branches. Several synchronization functions such as `_pthread_mutex_lock` figure prominently in the time breakdown, and have poor branch predictability because branch outcomes depend on synchronization races for contended locks. Other `memcached` functions contain case statements that change the behavior based on the current connection state or request which are difficult to predict.

10.3.2 Multicore non-scalability

We next examine the scalability of `memcached` with the number of cores, shown in Figure 10.8. For each workload and core type, we show relative throughput normalized to



(a) Xeon Scaling



(b) Atom Scaling

Figure 10.8: Memached's lack of scalability.

the single-core case. In each experiment, we tune the offered load until the 95th-percentile response time is 5ms so that all configurations can meet the target, and then report the normalized throughput.

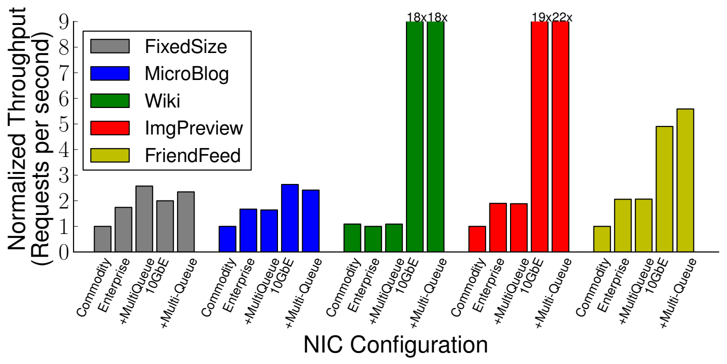
Our central observation is that `memcached` performance scalability on multiple cores is far from linear. In general, adding a second core helps. However, additional cores provide rapidly diminishing returns (or even slowdowns in some cases). These results are surprising because none of these configurations saturate the 10GbE network interface, and hence are not I/O bound in the conventional sense. Rather, the scalability limit arises due to synchronization stalls and load imbalance in both `memcached` and the kernel. The NIC also contributes to load imbalance among cores, as it may not always balance interrupt delivery.

`Memcached` uses nine different classes of lock which guard access to the connection queue, cache operations, statistics, and others. We use SystemTap [19] to inspect these locks and found that only the cache operations lock, which guards access and updates to the hash table, is highly contended and that contention increases with the square of the request rate. For 150,000 requests-per-second, the lock is contended about 55,000 times per second for an average time of $11\mu s$. While difficult to do, some type of finer grained locking would greatly improve the scalability of `memcached`.

While there has been some recent work to improve Linux scalability [46], widely used distributions still suffer from scalability bottlenecks. Improving the core-scaling behavior of `memcached` will require effort at the application, kernel, and even NIC hardware layers.

10.3.3 I/O subsystem: Quality vs. quantity

While it may seem obvious that `memcached` performance depends on networking performance, it is somewhat surprising that the quality of a NIC (i.e., its efficiency on a



	Commodity	Enterprise	10GbE
Approximate Price	$10	$145	$850
Segment Offload[1]	X	✓	✓
Multi-Queue[a]	X	✓	✓
Software Batch	X	✓	✓
Hardware Batch	X	X	✓

Figure 10.9: NIC features. NICs have a number of key features that distinguish their performance beyond pure bandwidth. The data presented is the maximum achievable throughput with the various NICs and a latency constraint of 5ms with all 6-cores of the Xeon enabled. We demonstrate the performance of each of two classes of 1GbE NIC (Commodity and Enterprise) and a 10GbE model. We also selectively disable and enable multiple-queues(MQ). Note that adding these features can be expensive as is apparent with the switch between commodity and enterprise. However, these features can be quite powerful, for example with FixedSize Enterprise and MQ provide over a 2.5x performance boost. Wiki and ImgPreview suffer from being network bandwidth bound and get inordinate gains from the switch to 10GbE alone.

per-packet basis) is more important than raw bandwidth. We find that the choice of NIC is critical—not all NICs are created equal. We now explain the features that differentiate NIC performance and quantify their benefit.

There are a variety of features that advanced NICs can support, listed in Figure 10.9. *Segment offloading* allows the driver to provide the NIC a payload that is larger than can be sent on the wire. The NIC will partition and transmit the payload across multiple packets, saving the CPU time of generating additional packet headers. *Software batching*, also called Generic Receive Offload (GRO), batches multiple smaller requests between the driver and the TCP stack, which ultimately reduces TCP stack invocations, reducing processing time. These features coupled with the more optimized interface between the driver and device (many fewer programmed I/O requests (PIO)) in the Enterprise NIC result in a 70% - 110% performance increase in all workloads except Wikipedia as can be seen in Figure 10.9.

Multiple-queue(MQ) support allows the NIC to communicate with a driver running on more than one core. The receive portion of this, Receiver-Side-Scaling (RSS), hashes packet header fields in the NIC to choose among the available receive queues, each assigned to a different core. The hashing ensures that a single core processes all packets received on a particular flow, improving locality. Transmit scaling allows multiple CPUs to enqueue packets for transmission simultaneously without the need for locks. For the Fixed-Size workload, an impressive performance improvement of 47% is achieved with multiple queues. The gains are so pronounced in this workload because its average packet size is the smallest. Most of the other workloads see little gain at 1Gb/s line speeds because a single CPU core is able to keep up with the resulting request rate, because of the larger average packet size.

The 10GbE and 10GbE +MultQueue bars in each cluster show the effectiveness of using a 10Gb/s NIC (with and without multiple-queues, respectively). The impact of greater network bandwidth, 10GbE bar, varies drastically. For workloads with large average object size (Wiki and ImgPreview), gains are up to 22x[2]. Unlike the Enterprise NIC, the 10GbE NIC supports *hardware batching*, or Receiver Side Coalescing, which performs similar similar batching in hardware. Enabling multi-queue support further increases these gains. Microblog and FriendFeed workloads are able to utilize 1.4Gbps, 2.2Gbps respectively, only a small fraction of the bandwidth available.

10.4 Designing cost-optimal `memcached` clusters

In this section we explore how internet service architects should best design `memcached` clusters to achieve the lowest cost for given performance objectives. We find that `memcached`'s unique set of microarchitectural deficiencies and scalability bottlenecks make the space of cost-optimal designs complex with multiple local optima. To navigate this space, we construct a design space optimization framework and then discuss the insights we have gained from the decisions made by the optimization. This study exposes a number of surprising trends including that high-performance high-core-count systems are economically ineffective and that it is sometimes necessary to *reduce* RAM per server to scale performance.

10.4.1 Design Space Exploration

Using the empirically measured performance data from Section 10.3, we perform an exhaustive design space exploration to find the optimally cost-efficient cluster design given

[2]Both ports of the 10GbE NIC are utilized for testing and thus the maximum bandwidth from the 10Gb NIC is actually 20Gb

a set of design constraints. An architect must consider three primary factors in designing a `memcached` cluster: the target request latency, the aggregate cluster bandwidth and the required memcache hit rate. The desired cache hit rate determines the required DRAM capacity of the cluster based on the size of the data set and the shape of the key popularity distribution. We express a desired latency constraint in terms of the 95th-percentile response latency across all servers. Whereas the latency requirements of `memcached` deployments likely fall within a fixed, narrow range (e.g., 200μs to 5ms), as a service grows, the required data set size and throughput will increase in a non-trivial manner depending on user activity.

Figure 10.10 illustrates how the optimal server configuration changes as the cluster requirements vary. Initially, the needs of a `memcached` cluster may be small enough to be satisfied by a single machine with a simple Atom-like processor. If data set size is scaled (while holding throughput targets fixed), DRAM can simply be added to this server (current Atom-like machines cannot support large DRAM capacities; we remove this limitation for the sake of the study). At some point, the per-server DRAM capacity will cease to increase because it is more economical to purchase a cluster of machines (DRAM density costs increase non-linearly). Alternatively, if the throughput target is increased (holding data set size fixed), one may first scale-up the capabilities of a single machine by using a more powerful processor (e.g., a Xeon) or adding more advanced NIC hardware. However, since the increased cost of better hardware will quickly overtake the increased throughput, a cluster-based solution will again prevail. Realistically, the data set size and throughput requirements will scale together in a non-trivial manner necessitating an optimization framework.

10.4.2 Understand `memcached` cluster economics

Our design exploration considers a number of server designs over a large range of processor performance, memory capacity, I/O performance and price. Server prices are determined by current market prices[3]; the cheapest configuration is less than \$200 (a two-core Atom with 6GB RAM and 1GbE NIC) and the most expensive is over \$4000 (a six-core Xeon with 48GB RAM and 10GbE NIC). We summarize our economic assumptions in Table 10.2. Although the exact pricing of these components will likely change over time, we believe that the insights we garner from this study transcend current pricing; the reasons for changing architectures will remain the same even if the absolute performance points shift.

[3]We estimated server prices from popular online vendors at the time of the study.

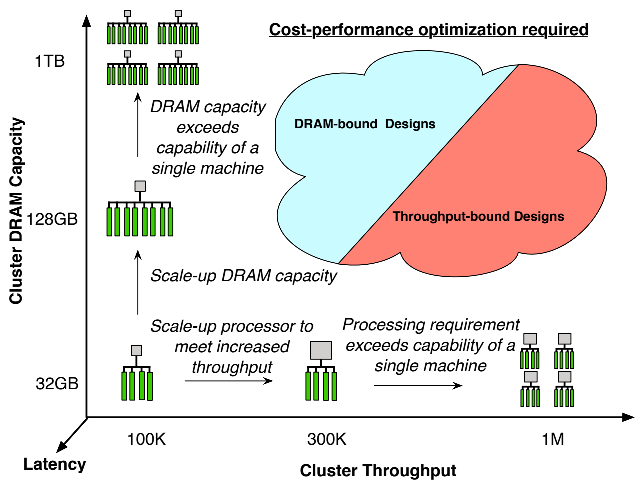

Cost-performance optimization required
1TB
128GB
32GB
Cluster DRAM Capacity
DRAM capacity exceeds capability of a single machine
DRAM-bound Designs
Throughput-bound Designs
Scale-up DRAM capacity
Scale-up processor to meet increased throughput
Processing requirement exceeds capability of a single machine
100K
300K
1M
Latency
Cluster Throughput

Figure 10.10: Scaling a `memcached` cluster. As a service grows, the data set size and throughput requirements of `memcached` will also grow. Reasoning about each scaling dimension in isolation is straight-forward. As data set size grows (while throughput targets remain fixed), the cost-optimal system will use an inexpensive processor with as much RAM per system as is economical. Conversely, when scaling throughput in isolation, servers will first leverage more capable processors (and possibly NICs) before expanding the cluster. However, an actual internet service will grow along both axes in a manner unique to each service. We create an optimization framework to explore the optimal system configuration for various points in this deign space and find that cost-optimal designs tend to bifurcate into clusters provisioned for capacity ("DRAM-bound") and for performance ("Throughput-bound").

We find that the cost-optimal design space is complex and it is helpful to explain a few aspects of why certain designs are favored. In Figure 10.11 (a) and (b) we show how cost efficiency, expressed in throughput per dollar, changes as a function of 95th-percentile latency target. For web services that are latency tolerant, Atom systems provide the most cost-efficient solution. Above 1ms latency, these systems are nearly twice as cost effective. However, many web services demand `memcached` latency in the range of 200μs to 1ms. This space is not clearly dominated by one design, but generally favors 2-core Xeon configurations. For the FriendFeed workload, we find that the premium 10GbE NIC pays for itself. Most importantly, as the latency constraint of the system changes, the optimal system configuration passes through many designs. The cost premium of reducing latency is steep and non-linear.

Compared to a DBMS system, `memcached` costs are quite low. A highly-optimized TPC-C deployment delivers less than 0.001 transactions per second per dollar [20]. `Memcached` clusters are four orders of magnitude more cost efficient, which explains their popularity.

We demonstrate how a cluster scales up with respect to desired aggregate throughput in Figure 10.11 (c). The number of required servers is the greater of (1) the number needed to

	1GbE				10GbE			
Base Machine	6GB	12GB	24GB	48GB	6GB	12GB	24GB	48GB
4-Core Atom	$199 $33.2/GB	$236 $19.7/GB	$410 $17.1/GB	$710 $14.8/GB	$1249 $208.2/GB	$1286 $107.2/GB	$1460 $60.8/GB	$1760 $36.7/GB
2-Core Xeon	$1418 $236.3/GB	$1455 $121.2/GB	$1629 $67.9/GB	$1929 $40.2/GB	$2468 $411.3/GB	$2505 $208.8/GB	$2679 $111.6/GB	$2979 $62.1/GB
4-Core Xeon	$1889 $314.8/GB	$1926 $160.5/GB	$2100 $87.5/GB	$2400 $50.0/GB	$2939 $489.8/GB	$2976 $248.0/GB	$3150 $131.2/GB	$3450 $71.9/GB
6-Core Xeon	$2489 $414.8/GB	$2526 $210.5/GB	$2700 $112.5/GB	$3000 $62.5/GB	$3539 $589.8/GB	$3576 $298.0/GB	$3750 $156.2/GB	$4050 $84.4/GB

Table 10.2: Server economics. We estimate server cost by first selecting a base system from the desired kind and number of processor cores. Next, RAM and NIC are added to this base system. The 10GbE NIC cost captures both the NIC itself and the increased cost of the top-of-the-rack switch.

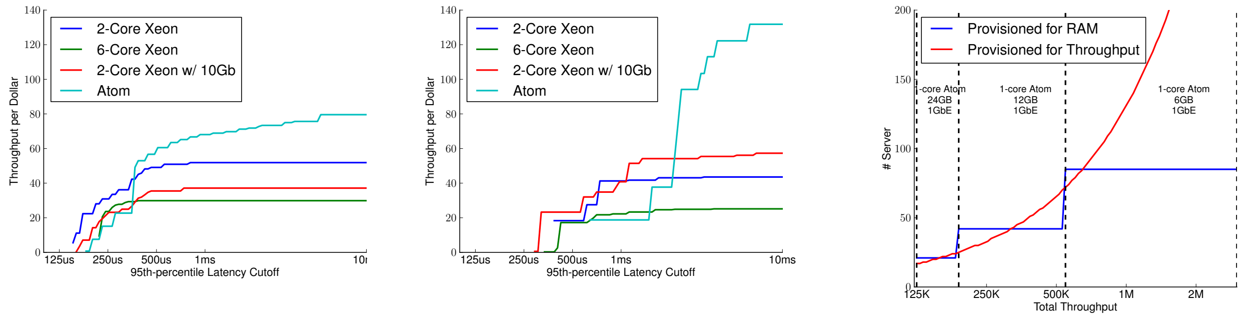
provision sufficient DRAM or (2) the number needed to achieve a given aggregate throughput. We demarcate the regions in which a particular server configuration is optimal as the target throughput scales. As throughput requirements grow, it generally becomes more cost-effective to increase the number of servers and provision less DRAM in each, thus reducing the fraction of cluster load each server must support.

10.4.3 Implications of the optimal design space

We highlight a number of interesting observations regarding optimal cluster configurations. We do not have space to report every facet of our study, so we highlight the most important insights we have gained.

Figure 10.12 illustrates the results of our study for each of our five workloads. Each subfigure reports the optimal server configuration for a given latency and throughput target. All results are for a cluster with an aggregate of 512GB of DRAM. We indicate which server configuration tends to dominate in various regions of the optimization space. System configurations are represented by color and DRAM configuration is annotated in regions of dominance. We find that the regions favoring high-performance NICs are easily divisible and are demarcated by a dotted line.

High-performance, high-core-count systems are economically unappealing because of the poor scaling demonstrated in Section 4. Often, integrating more cores into a server system is economically advantageous because workloads can scale throughput with core counts and the non-processor components can be amortized. We find that the cost premium of adding Xeons cores frequently does not pay for itself. In Figure 10.12 we find that the 6-core Xeon system is never preferred. Furthermore, the 4-core Atom and 2-core Xeon



(a) MicroBlog throughput per dollar (b) FriendFeed throughput per dollar (c) Effect of throughput scaling

Figure 10.11: Understanding design space optimization decisions. Subfigures (a) and (b) demonstrate how the throughput achieved per dollar varies as a function of 95th-percentile latency for various machine configurations. For MicroBlog, low-latency designs are dominated by 2-core Xeons whereas great price-efficiency can be gained under loose latency constraints (our observation here mirrors those for Web search in [167]. High core counts and expensive networking equipment do not pay for themselves. For FriendFeed, we see that a 10GbE NIC is useful across a large latency space. In Subfigure (c), we demonstrate how cluster sizing is determined as a function of aggregate cluster throughput. The number of required servers is the greater of (1) the number needed to provision sufficient DRAM or (2) the number needed to achieve a given aggregate throughput. We annotate the regions in which certain designs are optimal. Whether DRAM or throughput is the limiting factor alternates multiple times as the throughput is increased. This scaling also illustrates our observation that less DRAM per machine can lead to better performance. As the throughput requirements increase, it is cost-optimal to decrease the DRAM per server (from 24GB to 12GB to 6GB) and spread load over additional servers.

system dominate the majority of design points.

***Less* RAM per server leads to *higher* performance.** A particularly counter-intuitive behavior of memcached is that adding more RAM to a server actually makes latency *worse*. The reason for this peculiarity is that more objects are stored in a server with more RAM and, therefore, a larger fraction of a cluster's requests will map to that server, increasing its load. Because of this behavior, we find scenarios where the cost-optimal design provisions more lower-performance hardware with less RAM per server to improve latency. We see this behavior often; we previously demonstrated this in Figure 10.11 (c). We also see this behavior in Figure 10.12. Higher-performance constraints (lower-right region of each space) tend to favor much lower DRAM capacities than high-latency, low-throughput regions (upper-left).

The cost-premium of 10GbE is sometimes justified, even in deployments that do not saturate network bandwidth. Currently, adding 10GbE hardware to servers comes at a steep price premium. Since cost efficiency is paramount in high-scale data centers, it

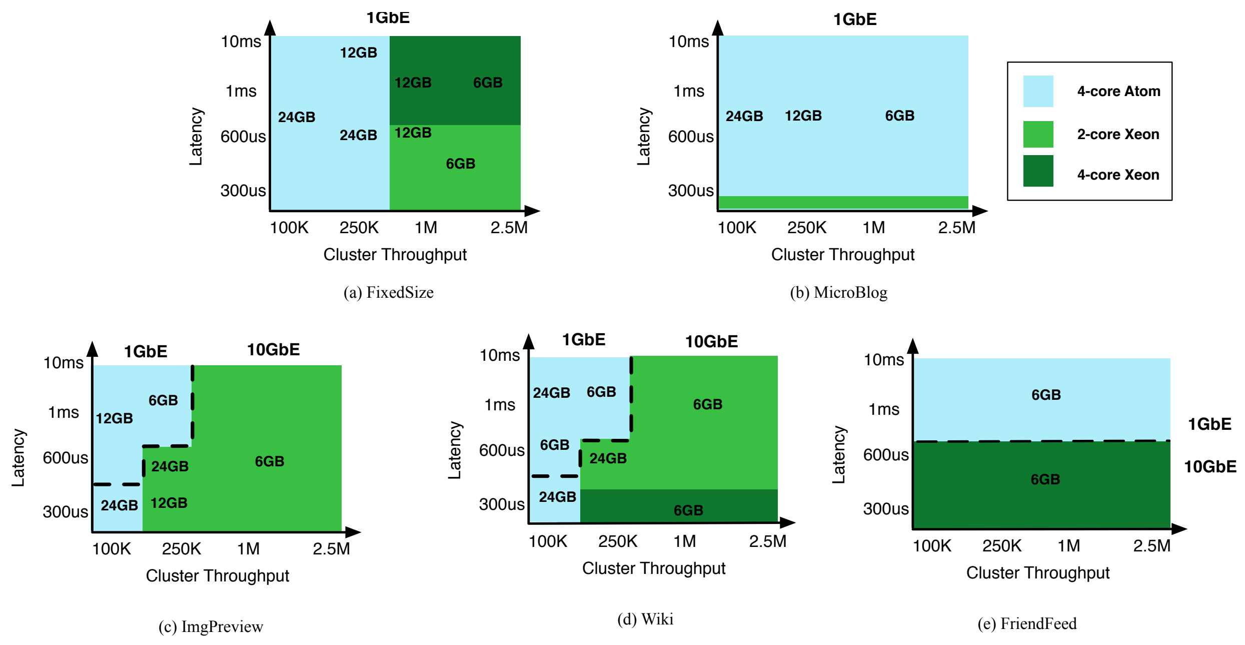


Figure 10.12: Memcached cluster design space exploration. Our optimization framework finds the cost optimal server configuration for a range of latency and throughput goals. Designs are clustered by base system. RAM capacity and networking are annotated within these clusters.

is not surprising that 10GbE networking is not pervasive—in many scenarios, commodity 1GbE links are preferred. Nevertheless, we find that certain cluster design points do justify the use of high-performance I/O. For example, the high-performance regions of the ImgPreview, Wiki and FriendFeed workload in Figure 10.12 call for 10GbE Ethernet even though there are other feasible alternatives.

Decreasing latency comes at a much higher cost-premium than increasing throughput. As we have seen from Figure 10.11, decreasing latency disproportionately decreases the cost-efficiency of servers. One of the main benefits of `memcached` is its ultra-fast response time. We find that low latency comes at a cost premium and that the costs rise much faster than scaling data set size or throughput.

10.4.4 The challenge of scaling a cluster

Although we have provided a set of guidelines, these rules of thumb may not be sufficient if a designer wishes to maximize cost effectiveness. As a service's requirements grow, the cost optimal cluster design will shift. Accordingly, simply finding the best design at a service's current operating point and buying more of the same server will lead to a suboptimal cluster in the future. Take the Wikipedia workload as an example. At 100,000 RPS and 48GB data set size requirements, the optimal system is 2 4-core Xeons with 24GB of RAM and a 10GB NIC (\$3150/server). If the throughput is scaled in isolation to 10 million RPS, and the data set size stays the same, the optimal cluster has 183 4-core xeons with 6GB of RAM and a 10GB NIC (\$2939/server). Using the original server design in this case would lead to a cluster that is 7% more expensive than optimal. Alterantively, if the data set size is scaled in isolation to 512GB, the optimal configuration is 21 Atom-based systems with 24GB of RAM and a 10GB NIC (\$410/server). This scenario would lead to a cluster that is 2.09 times more expensive than the optimal. These two examples demonstrate that scaling a cluster may range from either a modest to quite large waste in capital.

10.5 Summary

`Memcached` has rapidly become one of the central tools in the design of scalable web services. On its face, `memcached` might appear to be a simple workload, however we find that its common use cases have drastically different hardware requirements. We have characterized the microarchitectural deficiencies that lead to instruction throughputs as low as 1/16 of peak, and scalability bottlenecks that yield as little as 2x the throughput for 6x the cores. These trends have a fundamental impact on the construction of cost-optimal `memcached` clusters. Through design space exploration, our study demonstrates that there is no single server configuration which dominates the space and that different uses of `memcached` demand different configurations to be cost optimal.

CHAPTER 11

Conclusion

As the Internet continues to grow, so too does the importance of the data centers which support it. These systems are extremely costly and the fraction of this cost due to energy and power provisioning continue to rise in large-scale data centers. This thesis proposes a new approach towards improving the energy proportionality of servers: coordinated full-system low power modes. We propose the PowerNap server architecture, a coordinated full-system idle low-power mode which takes advantage of natural idle periods to save power. For uniprocessor systems, PowerNap approaches energy proportional operation.

However, modern servers use multicore processors with a high degree of concurrency due to the request-parallel nature of the software they run. As we expose in a study of Google Web search, trying to create full-system idle periods with simple batching mechanism yields a poor latency-power tradeoff. For this class of workload, there is a large opportunity for coordinated full-system active low-power modes. By combining voltage and frequency scaling for the processor and memory system in a coordinated fashion, a more appealing latency-power tradeoff can be achieved.

Unfortunately, current projections show that voltage and frequency scaling will decrease in efficacy with successive technology generations. Coarse-grain idle low-power modes do not decrease in their power savings, but we have shown they are more difficult to use with many concurrent, independent requests. To remedy the lack of idleness, we propose DreamWeaver, architectural support for creating artificial full-system idle periods. We demonstrate that DreamWeaver provides a more appealing latency-power tradeoff than existing power modes.

Finally, we have investigated increasing the price efficiency of `memcached`, a workload which is rapidly rising in popularity. We expose the many inefficiencies in modern processor microarchitecture and software stack. Our study demonstrates that this class of workload must be reasoned about as a cluster and that provisioning an efficient `memcached` cluster requires a design space optimization framework to achieve the best cost efficiency.

Bibliography

[1] Efficient power supplies for data center.

[2] Project Voldemort. A distributed key-value storage system.

[3] AOL Query Log, 2006.

[4] Energy Star computer specification v. 4.0, 2007.

[5] Average retail price of electricity to ultimate customers by end-use sector, by stat, 2008.

[6] Intel 64 and ia-32 architectures software developers manual volume 3b: System programming guide, 2009.

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

[8] AMD Family 10h Server and Workstation Processor Power and Thermal Data Sheet Rev 3.15, 2010.

[9] Flickr - Photo Sharing, 2010.

[10] libevent - an event notification library, 2010.

[11] Memcache Binary Protocol, 2010.

[12] memcached - a distributed memory object caching system, 2010.

[13] The Network Simulator - ns-2, 2010.

[14] Wikipedia, 2010.

[15] Linux Perf, 2011.

[16] SPEC power_ssj2008, 2011.

[17] SPECweb2005, 2011.

[18] Sysstat, 2011.

[19] Systemtap, 2011.

[20] TPC-C - Top Ten Price/Performance Results, 2011.

[21] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy proportional datacenter networks. *ISCA '10: International Symposium on Computer Architecture*, 2010.

[22] 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.

[23] 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.

[24] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera : Dynamic Flow Scheduling for Data Center Networks. *NSDI '10: Networked Systems Design and Implementation*, 2010.

[25] 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.

[26] 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.

[27] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: a fast array of wimpy nodes. *SOSP '09: Symposium on Operating Systems Principles*, 2009.

[28] Apache. The Apache Cassandra Project. 2010.

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

[30] L. A. Barroso. Warehouse-scale Computing. Keynote address at SIGMOD, 2010.

[31] L. A. Barroso. Warehouse-scale computing: Entering the teenage decade. Federated Computing Research Conference Plenary Speaker, 2011.

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

[33] 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*, Jan. 2009.

[34] 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.

[35] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. *OSDI '10: Operating systems design and implementation*, 2010.

[36] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *WREN '09: workshop on research on enterprise networking*, 2009.

[37] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. *IGCC '11: International Green Computing Conference*, 2011.

[38] A. Bhattacharjee and M. Martonosi. Characterizing the TLB behavior of emerging parallel workloads on chip multiprocessors. *PACT '09: International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[39] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 2006.

[40] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Performance analysis of system overheads in tcp/ip workloads. PACT '05, pages 218–230, 2005.

[41] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. *ASPLOS '06: Architectural support for programming languages and operating systems*, 2006.

[42] D. Blaauw, S. Das, and Y. Lee. Managing variations through adaptive design techniques, 2009.

[43] D. Blaauw, S. Das, and Y. Lee. Managing variations through adaptive design techniques. ISSCC Tutorial, 2010.

[44] D. Blaauw, S. Das, and Y. Lee. Managing variations through adaptive design techniques, 2010.

[45] P. Bohrer, E. N. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. Mcdowell, and R. Rajamony. The Case for Power Management in Web Servers. *Power Aware Computing*, 2002.

[46] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. *OSDI '10: Conference on operating systems design and implementation*, 2010.

[47] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *ISCA '00: International Symposium on Computer Architecture*, 2000.

[48] J. S. Bucy and G. R. Ganger. The DiskSim Simulation Environment Version 3.0 Reference Manual, 2003.

[49] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving disk energy in network servers. *Proceedings of the 17th annual international conference on Supercomputing - ICS '03*, page 86, 2003.

[50] M. Cha, A. Mislove, and K. P. Gummadi. A measurement-driven analysis of information propagation in the flickr social network. *WWW*, 2009.

[51] 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.

[52] E. J. Chen and W. D. Kelton. Simulation-Based Estimation of Quantiles. *Winter Simulation Conference*, 1999.

[53] E. J. Chen and W. D. Kelton. Quantile and histogram estimation. *Winter Simulation Conference*, 2001.

[54] E. J. Chen and W. D. Kelton. Determining simulation run length with the runs test. *Simulation Modelling Practice and Theory*, 11(3-4), 2003.

[55] 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.

[56] 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.

[57] A. A. Chien. Digital transformation. Talk at Intel Developer Forum, 2008.

[58] 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.

[59] G. Contreras and M. Martonosi. Power prediction for intel XScale processors using performance monitoring unit events. *ISLPED '05: International Symposium on Low Power Electronics and Design*, 2005.

[60] T. T. Control. Typical circuit breaker trip curve.

[61] R. Conway. Some Tactical Problems in Digital Simulation. *Management Science*, 10(1), 1963.

[62] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *OSDI '04: Operating System Design and Implementation,*, 2004.

[63] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo : Amazons Highly Available Key-value Store. *SOSP '07: Symposium on Operating Systems Principles*, 2007.

[64] 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.

[65] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis. Decoupling datacenter studies from access to large-scale applications: A modeling approach for storage workloads. *IISWC '11: IEEE International Symposium on Workload Characterization*, 2011.

[66] 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.

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

[68] 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.

[69] L. Eeckhout, S. Nussbaum, J. Smith, and K. De. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, pages 26–38, 2003.

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

[71] Facebook. Memcached Tech Talk with Mark Zuckerberg, 2010.

[72] X. Fan, C. S. Ellis, and A. R. Lebeck. The Synergy between Power-aware Memory Systems and Processor Voltage Scaling. *Workshop on Power-Aware Computing Systems*, 2002.

[73] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. *ISCA '07: International Symposium on Computer Architecture*, 2007.

[74] G. S. Fishman. *Discrete-event simulation*. Springer-Verlag, 2001.

[75] B. Flemming. Squeezing a ia computer in a smartphone. 2010 Asia Academic Forum, 2011.

[76] R. Fujimoto. Parallel Discrete Event Simulation. *Winter Simulation Conference*, 1989.

[77] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. *SIGMETRICS*, 2009.

[78] A. Gandhi, C. Lefurgy, and J. O. Kephart. Power Capping Via Forced Idleness. *Analysis*, 2009.

[79] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2 : A Scalable and Flexible Data Center Network. *SIGCOMM*, 2009.

[80] V. Gupta, M. Harchol-Balter, J. G. Dai, and B. Zwart. On the inapproximability of M/G/K: why two moments of job size distribution are not enough. *Queueing Systems: Theory and Applications*, 64(1):5–48, Aug. 2009.

[81] 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.

[82] J. Hamilton. Internet-Scale Service Infrastructure Efficiency, 2009.

[83] J. Hamilton. PUE is Still Broken and I still use it, 2010.

[84] M. Harchol-balter. Theory of Performance Modeling, 2005.

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

[86] T. Heath, A. Centeno, P. George, L. Ramos, Y. Jaluria, and R. Bianchini. Mercury and freon: temperature emulation and management for server systems. *ASPLOS '06: Architectural Support for Programming Languages and Operating Systems*, 2006.

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

[88] 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.

[89] U. Hoelzle and B. Weihl. High-efficiency power supplies for home computers and servers, 2006.

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

[91] H. Huang, K. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making DRAM less randomly accessed. *ISLPED '05: International Symposium on Low Power Electronics and Design*, 2005.

[92] R. Huggahalli, R. Iyer, and S. Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *Proceedings of the 32nd annual international symposium on Computer*

Architecture, ISCA '05, pages 50–59, Washington, DC, USA, 2005. IEEE Computer Society.

[93] Hynix. Hynix-DDR2-1Gb. 2008.

[94] Intel. Intel Pentium M processor with 2-MB L2 cache and 533-MHz front side bus, 2005.

[95] Intel. Intel Pentium dual-core mobile processorIntel Pentium dual-core mobile processor, 2007.

[96] Intel. Quad-core Intel Xeon processor 5400 series, 2008.

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

[98] 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.

[99] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[100] J. Janzen. Calculating Memory System Power for DDR SDRAM. *Designline*, 10(2):1–12, 2001.

[101] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached design on high performance rdma capable interconnects. *International Conference on Parallel Processing*, 2011.

[102] S. Kaxiras and M. Martonosi. Computer Architecture Techniques for Power-Efficiency. *Synthesis Lectures on Computer Architecture*, 3(1):1–207, 2008.

[103] 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.

[104] 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.

[105] D. Knuth. *The Art of Computer Programming - Voll II.* Addison-Wesley, 1981.

[106] J. Koomey. Growth in data center electricity use 2005 to 2010. A report by Analytics Press, completed at the request of The New York Times, 2011.

[107] 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.

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

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

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

[111] J. Laudon. UltraSPARC T1: A 32-threaded CMP for servers, 2006.

[112] 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.

[113] 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.

[114] C. Lefurgy, X. Wang, and M. Ware. Server-level power control. *ICAC'07: International Conference on Autonomic Computing*, June 2007.

[115] C. Lefurgy, X. Wang, and M. Ware. Power capping: a prelude to power shifting. *Cluster Computing*, 11(2):183–195, Nov. 2008.

[116] K. Leigh and P. Ranganathan. Blades as a general-purpose infrastructure for future system architectures: Challenges and solutions, 2007.

[117] 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.

[118] G. Liao and L. Bhuyan. Performance measurement of an integrated nic architecture with 10gbe. In *Proceedings of the 2009 17th IEEE Symposium on High Performance Interconnects*, Washington, DC, USA, 2009. IEEE Computer Society.

[119] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. *ISCA '08: International Symposium on Computer Architecture*, 2008.

[120] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM SIGARCH Computer Architecture News*, 33(4):92, Nov. 2005.

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

[122] 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.

[123] 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.

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

[125] D. Meisner, J. Wu, and T. F. Wenisch. Towards a Scalable Data Center-level Evaluation Methodology. *ISPASS '11: International Symposium on Performance Analysis of Systems and Software*, 2011.

[126] 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.

[127] Memcachedb. MemcacheDB: A distributed key-value storage system designed for persistent.

[128] Micron. DDR2 SDRAM SODIMM. 2004.

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

[130] R. Middlebrook and S. Cuk. A general unified approach to modelling switching-converter power stages. *Power Electronics Specialists Conference*, 1976.

[131] G. Minshall, Y. Saito, J. C. Mogul, and B. Verghese. Application performance pitfalls and TCP's Nagle algorithm. *SIGMETRICS Performance Evaluation Review*, 27(4):36–44, 2000.

[132] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical Power Slope : Understanding the Runtime Effects of Frequency Scaling. *ICS '02: International Conference on Supercomputing*, 2002.

[133] J. Moore, J. Chase, P. Ranganathan, and R. Sharma. Making scheduling cool: Temperature-aware workload placement in data centers. In *USENIX Annual Technical Conference*. USENIX Association, 2005.

[134] D. M. Nicol. Parallel Simulation Of FCFS Stochastic Queueing Networks. *PPoPP '88: Principles and Practice of Parallel Programming*, 1988.

[135] J. Ousterhout, M. Rosenblum, S. M. Rumble, E. Stratmann, R. Stutsman, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, and G. Parulkar. The case for RAMClouds. *ACM SIGOPS Operating Systems Review*, 43(4):92, Jan. 2010.

[136] P. Padala, X. Zhu, Z. Wang, S. Singhal, and K. Shin. Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 2007.

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

[138] K. Pawlikowski. Steady-state simulation of queueing processes: survey of problems and solutions. *ACM Computing Surveys (CSUR)*, 22(2), 1990.

[139] V. Paxson and S. Floyd. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3), June 1995.

[140] S. Pelley, D. Meisner, T. F. Wenisch, and J. W. VanGilder. Understanding and abstracting total data center power. *WEED '09: Workshop on Energy-Efficient Design*, 2009.

[141] S. Pelley, D. Meisner, P. Zandevakili, T. F. Wenisch, and J. Underwood. Power Routing : Dynamic Power Provisioning in the Data Center. *ASPLOS '10: Architectural Support for Programming Languages and Operating Systems*, 2010.

[142] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. *Proceedings of the 5th European conference on Computer systems - EuroSys '10*, page 335, 2010.

[143] E. Pinheiro and R. Bianchini. Energy conservation techniques for disk array-based servers. *ICS '04: International Conference on Supercomputing*, 2004.

[144] 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.

[145] A. Pressman. *Switching Power Supply Design*. 1998.

[146] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu. No Power Struggles : Coordinated Multi-level Power Management for the Data Center. *ASPLOS '08: Architectural Support for Programming Languages and Operating Systems*, 2008.

[147] K. Rajamani, C. Lefurgy, S. Ghiasi, and J. Rubio. Power management for computer systems and datacenters. *Tutorial at ISLPED '08: International Symposium on Low-Power Electronic Desig*, 2008.

[148] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. *ISCA '06: International Symposium on Computer Architecture*, 34(2):66–77, May 2006.

[149] N. Rasmussen. AC vs. DC power distribution for data centers, 2007.

[150] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. Tcp onloading for data center servers. *Computer*, 37(11):48 – 58, nov. 2004.

[151] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A comparison of high-level full-system power models. *HotPower*, 2008.

[152] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout. It's time for low latency. *HotOS '11: Hot topics in operating systems*, 2011.

[153] Samsung. SSD SATA 3.0Gbps 2.5 data sheet. 2008.

[154] B. Schroeder, E. Pinheiro, and W. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[155] E. Schurman and J. Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. *Velocity*, 2009.

[156] N. Semiconductor. Introduction to power supplies, 2002.

[157] 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.

[158] N. Sharma, S. Barker, D. Irwin, and P. Shenoy. Blink: managing server clusters on intermittent power. *ASPLOS '11: Architectural support for programming languages and operating systems*, 2011.

[159] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *ASPLOS '002: Architectural Support for Programming Languages and Operating Systems*, 2002.

[160] S. Siddha, V. Pallipadi, and A. Ven. Getting maximum mileage out of tickless. *Proceedings of the Linux Symposium. Intel Open Source Technology Center*, 2007.

[161] SMSC. LAN9420/LAN9420i single-chip ethernet controller with HP Auto-MDIX support and PCI interface, 2008.

[162] D. Snowdon, S. Ruocco, and G. Heiser. Power management and dynamic voltage scaling: Myths and facts. *Workshop on Power Aware Real-time Computing*, 12, 2005.

[163] J. Sobel. Building Facebook: Performance at Massive Scale, 2010.

[164] S. Srinivasan, L. Zhao, B. Ganesh, B. Jacob, M. Espig, and R. Iyer. CMP Memory Modeling: How much does accuracy matter? *MoBS '09: Workshop on Modeling, Benchmarking and Simulation*, 2009.

[165] 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.

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

[167] VJ Reddi, Benjamin Lee, Trishul Chilimbi, and Kushagra Vaid. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. 2010.

[168] V. Vorperian. Simplified analysis of PWM converters using model of PWM switch. II. Discontinuous conduction mode. *Aerospace and Electronic Systems, IEEE Transactions on*, 26(3):497–505, 1990.

[169] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):107, 2005.

[170] X. Wang, M. Chen, C. Lefurgy, and T. W. Keller. SHIP: Scalable Hierarchical Power Control for Large-Scale Data Centers. In *PACT '09: Parallel Architectures and Compilation Techniques*. Ieee, Sept. 2009.

[171] W.-D. Weber. Energy-saving approaches for warehouse-scale computing, 2010.

[172] P. Welch. On a Generalized M/G/1 Queuing Process in Which the First Customer of Each Busy Period. *Operations Research*, 12(5):736–752, 1964.

[173] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. *ISPASS '06: International Symposium on Performance Analysis of Systems and Software*, 2006.

[174] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4), 2006.

[175] Q. Wu, P. Juang, M. Martonosi, L. Peh, and D. Clark. Formal control techniques for power-performance management. *Micro, IEEE*, 2005.

[176] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: accelerating microarchitecture simulation via rigorous statistical sampling. *ISCA '03: International Symposium on Computer Architecture*, 2003.

[177] F. Xie, M. Martonosi, and S. Malik. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(3):323–367, 2004.

[178] 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.

[179] R. Yates. Practical Considerations in Fixed-Point FIR Filter Implementations. *Digital Signal Labs, Technical Reference*, 2007.

[180] J. Zawodny. Redis: Lightweight key/value Store That Goes the Extra Mile, 2010.