# OPTIMIZING LOW-POWER SETTINGS IN COMPUTER SYSTEMS

by

Yanpei Liu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2016

Date of final oral examination: 10/10/16

The dissertation is approved by the following members of the Final Oral Committee:
Professor Yu-Hen Hu, University of Wisconsin Madison, Chair
Professor Ricardo Bianchini, Microsoft Research & Rutgers University
Professor Mikko Lipasti, University of Wisconsin Madison
Professor Parameswaran Ramanathan, University of Wisconsin Madison
Professor Jin-Yi Cai, University of Wisconsin Madison

*To my grandfather*
*Zhuangqing Hu*
*who brought me to the door of engineering*

# Acknowledgments

In the past a few years as a PhD student, I had the great fortune to work with some amazing people across several research institutions. On top of the list is Professor Stark Draper, my adviser who first brought me into UW-Madison's graduate program. As a student I learned a lot from his technical wisdom. As a working professional his incredible mentorship have had a long-lasting impact on my career.

I would like to thank Professor Ricardo Bianchini from Rutgers University who advised and helped me during the last a few tough PhD years. I vividly remember all the discussions we had, all the joys and frustrations we shared and all the hard work we put in together. I never forget those late night discussions in the DARK Lab (as well as the 2-hour drive back to Middle Village, NY). Countless times I emailed him about our work falling short, frustrated with the results, but almost always left motivated after reading the replies.

This thesis would not even start without Daniel Myers and Emina Soljanin. Daniel, who was then the instructor of CS 547, inspired my curiosity in queuing theory and laid down the necessary foundation for my research. The work with Emina on fork-join queue shaped my research direction even further and eventually led to the participation in the Special Focus on Energy and Algorithms at DIMACS.

My thanks go to my collaborators, Guilherme Cox and Qingyuan Deng for their tremendous contribution towards FastCap and FastEnergy, Gauri Joshi for working on fork-join queues together, Professor Nam Sung Kim for guiding me towards my first paper in ISCA, Jing Yang for demonstrating the backpressure algorithm and Professor Akbar Sayeed for co-advising my first ever academic publication.

This thesis would not end without my thesis committee members, Professor Yu-Hen Hu, Mikko Lipasti, Parameswaran Ramanathan and Jin-Yi Cai. I would like to thank especially Professor Yu-Hen Hu for the generous help in guiding me towards final defense since late 2014.

Finally, I would like to thank my all family and friends, for their undivided support on my determination to chase the dreams.

# Contents

# List of Tables

# List of Figures

# Abstract

Power and energy efficiency in computer systems has received growing attention in recent years. Striving for the balance between performance and efficiency, low-power settings in CPU, memory and other components are continuously adjusted under changing workload. It is important to adjust these settings in a coordinated manner as a lack of coordination hampers a system's ability to optimize for performance and efficiency. However over the years the number of low-power settings in a typical computer system has increased dramatically. This creates a combinatorial search space, and thus a challenging task, for optimizing the low-power settings at runtime.

In this thesis, via queuing models and performance analyses, we present three optimization techniques for managing low-power settings in modern computer systems. The first technique, SleepScale, jointly operates the active low-power modes and idle low-power states in modern CPU. The second technique, FastCap, maximizes the performance of applications under a full-system power cap while promoting fairness across applications. The third technique, FastEnergy, maximizes the full-system energy savings under pre-defined application performance loss bounds. All three techniques are developed based on novel queuing models for managing abundant low-power settings in many-core systems.

# Chapter 1

# Introduction

As the workloads allocated to data center servers increase so does the economic and environmental footprint of these servers. By the end of 2013, U.S. data centers electricity usage has exceeded 90 billion killowatt-hours (or 9 billion U.S. dollars) and is expected to increase by 47% in 2020. Further, the carbon dioxide emission due to the electricity consumed by data centers has exceeded 90 million metric tons in 2013 [9]. These numbers give material economic, societal, and environmental reasons for improving the computation efficiency.

Much power/energy consumed by servers does not go toward computation. While configured to meet service demand, servers regularly operate at lighter load levels. Even in idle a server continues to draw roughly 60% of the maximum power when running full speed [1, 10]. Although some data centers (e.g. in Google) recently report low power usage effective (PUE), benefiting from the cooling using cold water directly from rivers or seas, around 40% of costs are still associated with power consumption in other commercial and governmental data centers [11].

To maximize power/energy efficiency, servers have incorporated an increasing number of idle low-power states (such as CPU sleep states) and active low-power modes (such as CPU dynamic voltage and frequency scaling, or DVFS). While active low-power modes offer flexibility in balancing performance and efficiency during normal execution, idle low-power states allow systems to conserve power during idle period. These low-power settings are also integrated in other system components such as main memory [12, 13], disk drives [14, 15] and interconnects [16].

Both active low-power modes and idle low-power states are well-studied in recent literature [17, 18, 19, 3, 20, 21]. George and Harrison [22] studied how to define the service rates that minimize a general cost function (combining waiting and service costs) in a single-server queuing system. Bansal *et al.* [23] studied minimizing energy via speed scaling without violating task deadlines. Past work [24, 25, 26] has shown that memory active low-power modes are particularly useful for workloads when the memory bus is underutilized for long periods. On idle low-power states, in [3] the authors proposed a method for eliminating idle power in servers by quickly transitioning between a high-performance active mode and a single idle low-power state. In [8] the authors warned of potential problems of using these low-power states and suggest "guarded" mechanisms to avoid negative power savings. The use of different idle low-power states for memory has also been extensively studied in [20, 21]. In Chapter 6 we provide a complete literature survey related to this thesis.

However, as these low-power settings grow in numbers, it is important to manage them in a coordinated manner as operating in separation can hamper a system's ability to optimize for performance and efficiency. To see an example of such behavior, consider a scenario in which our goal is to maximize the full-system energy savings within a pre-selected bound on the acceptable application performance loss. Further, suppose that the CPU cores are stalled waiting for the memory a significant fraction of the time. In this situation, the CPU power manager might predict that lowering voltage/frequency will improve energy efficiency while still keeping performance within the bound and effect the change. The lower core frequency would reduce traffic to the memory subsystem, which in turn could cause its (independent) power manager to lower the memory frequency. After this latter frequency change, the performance of the server as a whole may dip below the CPU power manager's projections, potentially violating the target performance bound. So, at its next opportunity, the CPU manager might start increasing the core frequency, inducing a similar response from the memory subsystem manager. Such interactions between CPU and memory frequencies cause performance oscillations. These unintended behaviors suggest that it is essential to coordinate power-performance management techniques across system components to ensure that the system is balanced to yield maximal energy savings.

The need for coordinated management creates a combinatorial search space for all low-power setting configurations. In a typical 16-core server with 10 active low-power modes for each core, the total number of possible low-power mode combinations is $10^{16}$ – traversing this large space in search for the optimal configuration is difficult. Existing works often rely on some simplified exhaustive search [27] or sub-optimal heuristics [28, 13] which may not perform well for future many-core servers.

We now summarize the challenges in effectively optimizing low-power settings. First, it is important to understand the complex interaction between different low-power settings in system components. Adjusting them in isolation may cause unpredictable performance and/or efficiency losses. Second, since the number of low-power setting combinations can be large, selecting the best low-power settings should be done with low overhead while still achieving optimal or near optimal solution. Third, the best low-power setting should adapt to workload at runtime. This challenge is especially acute for servers that run many applications in parallel (each with a potentially different behavior), since it is unlikely that the power mode selected for a core running one application can be used for a core running another one.

In the next section we summarize how our work address these challenges.

## 1.1 Contribution

### 1.1.1 Problem statement and overview

In this dissertation, we tackle the problem of optimizing low-power settings in computer systems for power/energy efficiency and application performance. We abstract system operations using queuing models. The queuing models allow us to quantitatively characterize the workload behavior under different low-power settings. For a given set of workloads, the goal is to develop effective algorithms which adjust the low-power settings at runtime to achieve the best power/energy efficiency under performance constraints (or vice versa). In particular, we focus on three specific problems:

a) Management of active low-power modes and idle low-power states in processors to maximize efficiency under performance constraints.

b) Management of active low-power modes in both processors and memory to maximize performance under a given power cap.

c) Management of active low-power modes in both processors and memory to maximize energy efficiency under performance constraints.

We present three methods, namely SleepScale, FastCap and FastEnergy to tackle these three problems. For problem a), we develop a novel queuing simulator in SleepScale to select the best low-power settings at runtime. The queuing simulator allows us to identify the optimal pair of active low-power mode and idle low-power state for the best efficiency/performance. For problem b), we develop a novel many-core queuing model and formulate a convex optimization problem. We deduce the FastCap algorithm – the fastest algorithm among the state-of-the-arts which produces the best application performance under a given power budget. For problem c), we further extend the many-core queuing model and formulate a non-convex optimization problem. We deduce the FastEnergy algorithm which, when compared with the state-of-the-arts, produces the same level of energy efficiency and application performance at lowest algorithmic complexity.

The models used in SleepScale, FastCap and FastEnergy are different. FastCap and FastEnergy consider active low-power modes in both processors and memory. The queuing models developed capture workload characteristics at a fine-grained micro-architectural level. SleepScale consider active low-power modes and idle low-power states in processors. The queuing model developed captures workload characteristics at a system/server level. In Chapter 7.1 we further discuss how the models can be extended for other use cases.

We now discuss the aforementioned contributions in details.

### 1.1.2 SleepScale

Our first contribution, SleepScale, solves the problem of optimizing active low-power modes and idle low-power states in processors. SleepScale challenges the conventional wisdom that active low-power modes and idle low-power states can be operated independently. It also challenges the idea of always using fixed active and idle low-power settings as suggested by many prior literature. SleepScale suggests that the optimal choice of the active low-power modes and idle low-power states depends on workload and hardware characteristics, and should be adjusted at runtime in response to workload changes.

To understand the interaction between active low-power modes and idle low-power states, we first build a queuing simulator to model job executions in different DVFS and sleep state settings. Through extensive simulation, we verify that different DVFS choices should be used together with different sleep states. We show that the conventional "race-to-halt" method [3], which runs the CPU at the maximum speed when busy and uses a fixed sleep state when idle is a suboptimal choice. To construct low overhead algorithms and operate at runtime, SleepScale uses a workload predictor to predict the workload statistics online. Then the policy manager uses the predicted statistics to determine the best DVFS and sleep state setting by running the aforementioned light-weight queuing simulator. We design SleepScale to run in epochs and our results show that SleepScale offers significant power savings over some conventional methods while respecting the same QoS constraints.

### 1.1.3 FastCap

Our second contribution, FastCap, solves the problem of optimizing active low-power modes in processors and memory under a given power budget. FastCap challenges the current solutions which operate low-power settings in processors and memory independently. By optimizing jointly for processors and memory, FastCap significantly advances the state-of-the-arts in both algorithm complexity and application performance.

To understand the interaction in low-power settings between processors and memory, we first develop a queuing model that effectively captures the workload dynamics in a many-core system. Based on the queuing model, we formulate a convex optimization framework for maximizing application performance under a given power budget. To construct low overhead algorithm, we make a key observation that core frequencies can be determined optimally in linear time for a given memory frequency. We develop the FastCap algorithm and implement it to operate at runtime. The operating system runs the algorithm periodically (once per time quantum, by default), and feeds a few performance counters as inputs to it. Our results show that FastCap maintains overall system power under budget and produces the best application performance among the state-of-the-art methods.

### 1.1.4 FastEnergy

Our third contribution, FastEnergy, solves the problem of optimizing active low-power modes in cores and memory for energy conservation. FastEnergy adopts the same queuing model developed in FastCap. Based on the queuing model, the FastEnergy algorithm has the lowest complexity among the state-of-the-arts while achieving the same level of energy efficiency. The work of FastEnergy further demonstrates the applicability of the queuing models for managing abundant active low-power modes in many-core systems.

To understand the interaction in low-power settings between processors and memory, we formulate a non-convex optimization framework for minimizing the full-system energy under application performance constraints. As in FastCap, we observe that core frequencies can be determined optimally in linear time for a given memory frequency. This observation allows us to develop a low overhead algorithm to operate at runtime. Our results show that FastEnergy produces significant energy savings across a range of core counts, especially when users are willing to accept greater performance degradation. In addition, our results show that FastEnergy consistently scales to larger core counts than the state-of-the-art design.

## 1.2 Thesis structure

The rest of the thesis is organized as follows. In Chapter 2 we provide an overview of system low-power settings and introduce key performance metrics. In Chapter 3, Chapter 4 and Chapter 5 we introduce SleepScale, FastCap and FastEnergy respectively. The discussion of related work is in Chapter 6 and we conclude the thesis in Chapter 7.

# Chapter 2

# Background

## 2.1 Server power consumption

Data centers spend a large amount of capital on power usage and other associated infrastructures. Around 40% of total operation cost is related to power distribution, cooling and electricity bills [11]. In data centers, most power is consumed by servers themselves – the server power consumption is around 47% of total power distributed to data centers [29].

Within each server, power is further distributed to system components such as CPU, memory and disks. Figure 2.1, excerpted from [6], shows the power breakdown in different components in Google's $\times 86$ production servers running at different utilization levels. As shown in Figure 2.1, power spent in CPU and memory account for more than 60% of total power draw.

We now describe different power saving techniques (i.e., low-power settings) commonly implemented in modern computer systems.

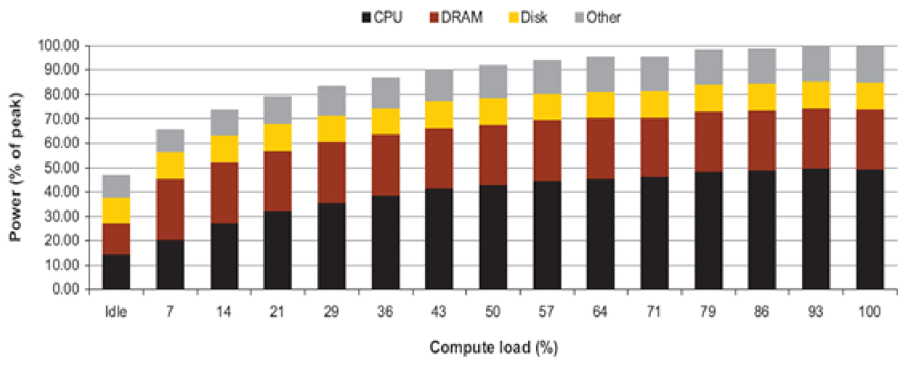


Figure 2.1: Server power breakdown. Figure drawn from [6].

## 2.2 Low-power settings

In this thesis, we focus on two types of low-power settings: active low-power modes and idle low-power states. Active low-power modes are power saving methods used when a device is in normal operating state. Idle low-power states are the ones used when a device is in idle.

### 2.2.1 CPU

**Active low-power modes.** In modern processors, active low-power modes are provided via the dynamic voltage and frequency scaling (DVFS) during normal processor operation. DVFS adjusts the frequency and voltage on-the-fly to conserve power and prevent processors from overheating. In DVFS, voltage is typically adjusted together with the frequency to provide desired trade-off between performance and power. In advanced configuration and power interface (ACPI) specifications, DVFS settings are also referred to as P-states. Different P-states correspond to different DVFS levels and all P-states are referred to as the C0 operating state [2]. Some industrial implementations include Intel's SpeedStep [30] and AMD's Cool'n'Quite [31]. When frequency is changed, the speed at which the processor executes instructions is also changed accordingly. Maximum frequency provides fastest instruction execution at the cost of highest power consumption. While for workload that is not as CPU intensive, setting the frequency low can be beneficial for power savings.

In active low-power modes, the total power of the CPU is the sum of the dynamic and static power. The dynamic power depends on the frequency and voltage of the CPU. Approximately, it can be expressed as $CV^2f$ [32], where $V$ and $f$ are the operating voltage and frequency of the CPU. The parameter $C$ is some scaling factor which relates to the capacitance. In DVFS, the voltage $V$ is changed together with the frequency thus the CPU dynamic power scales cubically with the frequency. The static power consists of all power components that do not depend on either voltage and frequency. It is mainly due to various leakage currents. It is also possible to adjust the frequency only while the voltage is held constant (thus DFS) or to adjust the voltage only with constant frequency (thus DVS). However they are not used as widely as DVFS due to their subpar efficiency for achieving the same level of performance.

**Idle low-power states.** Besides the C0 operating mode, ACPI also specifies a series of idle low-power states such as C1/C1E, C2, C3 and C6 states. These states are applied when the processor is not executing instructions. In C1/C1E state, the frequency/voltage are reduced to the minimum and it is often used when the processor just stops executing instructions. Transition to C0 state from C1/C1E can happen almost instantaneously with little wake-up latency. Under longer idle period, the processor can enter C2 or C3 state with more power savings but at the cost of higher wake-up latencies. In C2 state, the clock is stopped and in C3 state, the private cache is also flushed. Some processors even have very deep idle low-power states such as C6 state [30]. In such state, the architectural information is saved to RAM and voltage is set to zero. The power consumption in C6 state is very low but the wake-up latency is often quite high. The number of idle low-power states has been growing as new modern processors are manufactured; the recent Intel's Haswell platform has up to C10 state [33].

The power consumption in idle low-power states vary from state to state. Generally speaking the power is less dependent on the voltage and frequency thus can often be treated as a constant. In Chapter 3 we discuss in details about the power consumed in idle low-power states.

### 2.2.2 Memory

For modern memory architecture, memory power consists of memory background power, register/phase-lock loop (PLL) power, memory controller (MC) power and power associated with other activities such as activation/precharge, read/write, etc. Typically the background, register/PLL and MC account for $70-95\%$ of the total memory power depending on the workloads [34].

**Active low-power modes.** Recently, researchers have proposed using DVFS to adjust the frequency of the memory [25, 12]. This involves adjusting the frequency of the memory bus, dual in-line memory modules (DIMMs), and dynamic random-access memory (DRAM) chips. Changing the frequency of the memory affects the power and performance. Lowering the frequency increases the response time of the memory as it makes data burst longer on the memory bus. It also makes the MC slower thus results in large delay due to queuing effects. Lowering the frequency also leads to the linear reduction in the background and register/PLL power. Together with the frequency, it is also possible to reduce the voltage of the MC thus MC power can be reduced cubically. A detailed discussion on memory power and frequency relationship can be found in [34].

**Idle low-power states.** When a rank of DRAM chips is idle, all chips within that rank can be transitioned into idle low-power states. Different idle low-power states have different wake-up latencies. For example, the fast-exit precharge powerdown offers short wake-up latency but limited power savings while slow-exit precharge powerdown provides more power savings at longer wake-up cost.

Unlike in CPU, idle low-power states in memory are not often used as powering down an entire rank can be very costly [34]. To justify the powerdown, sufficiently long idleness for the entire rank should be observed, which rarely occurs in modern multi-core systems. As the result we do not consider idle low-power modes in memory in this thesis.

## 2.3 Performance metrics

Whenever low-power settings are changed, it is useful to quantitatively understand what aspect of the performance is affected and by how much. In this thesis, we mainly focus on two performance metrics: *response time* and *throughput*.

**Response time.** Response time is a random variable which measures the time a job spent in the system from its arrival to its departure. It consists of delays due to queuing and the service time of the job itself. The response time averaged across jobs is often called mean response time in queuing theory. The mean response time of the system indicates how much delay a job experiences on average.

However, mean response time only reflects the average behavior in the system. It is very likely for a system to have small mean response time yet with many jobs experiencing significant delay. Thus the probability of the response time exceeding a given threshold is also important; it gives quantitative measure on the performance of those outlier jobs. In Chapter 3, we demonstrate how response time is used to measure the performance of a server system.

**Throughput.** The throughput is the rate at which the system can finish its jobs. It measures the capacity of the system, i.e., how much work can be done within a given time interval. In Chapter 4 and Chapter 5, we demonstrate how we use throughput as the performance metric for multi-core systems.

# Chapter 3

# SleepScale

## 3.1 Introduction

In this chapter, we introduce *SleepScale*, a low-power setting manager that jointly operates active low-power modes and idle low-power states for CPU. For active low-power modes, we consider a series of CPU DVFS settings. For idle low-power states, we consider various sleep states. Following the ACPI specification [2], we use $C0_{(a)}$ to denote the processor operating mode. In this mode, DVFS settings can be adjusted. For sleep states, we consider $C0_{(i)}$, C1, C3 and C6 states. Different processor low-power settings are supported under different platform low-power settings (e.g., $S0_{(a)}$, $S0_{(i)}$ and S3). In Table 3.1 and Table 3.2 we show the processor low-power settings and associated platform settings based on the Intel Xeon/Atom family of processors.

SleepScale uses a set of power management policies (i.e., the choice of DVFS setting and sleep states) for given workload under target quality-of-service (QoS) constraints. To develop effective policies we consider a system model that takes

| Setting | Operation & Characteristics |
|---|---|
| $C0_{(a)}$ | Operating active mode: there is work to do, voltage & frequency setting adjusted dynamically by DVFS |
| $C0_{(i)}$ | Operating idle state: there is no work to do, voltage & frequency held constant at last DVFS setting |
| C1 | Halt state: clock stops |
| C3 | Sleep state: cache flushed, architectural state maintained, clock stopped |
| C6 | Deep sleep state: architectural state saved to RAM, voltage set to zero |

Table 3.1: CPU low-power settings [1].

| Setting | Operation & supported CPU state(s) |
|---|---|
| $S0_{(a)}$ | Active mode: associated with $C0_{(a)}$ only |
| $S0_{(i)}$ | Idle state: associated with other CPU states |
| S3 | Sleep: RAM powered, associated with C6 only |

Table 3.2: Platform low-power settings [2].

into account important QoS constraints and processor, platform, and workload characteristics (Section 3.2). Then, to develop engineering insight, we study the effectiveness of various policies for the aforementioned system model under idealized setting (Section 3.3). In contrast to some previous studies, we demonstrate that there is not a single optimal policy; the optimum policy heavily depends on QoS constraints, as well as on the characteristics of processor, platform, and workload. Through intensive simulation and analytic study, we observe that (1) there exists an optimal joint choice of frequency setting and sleep state; (2) the best sleep state depends on the performance constraint at low utilization; (3) the best sleep state also depends on the job size; (4) the delay to enter a sleep state should be jointly

| **Components** | **Operating** | **Idle** | **Sleep** | **Deep sleep** | **Deeper Sleep** |
|---|---|---|---|---|---|
| CPU×1 [1] | $130V^2f$ *W* (C0$_{(a)}$) | $75V^2f$ *W* (C0$_{(i)}$) | $47V^2$ *W* (C1) | 22 *W* (C3) | 15 *W* (C6) |
| Chipset×1 [1] | 7.8 *W* | 7.8 *W* | 7.8 *W* | 7.8 *W* | 7.8 *W* |
| RAM×6 [1] | 23.1 *W* | 10.4 *W* | 10.4 *W* | 10.4 *W* | 3.0 *W* |
| HDD×1 [36] | 6.2 *W* | 4.6 *W* | 4.6 *W* | 4.6 *W* | 0.8 *W* |
| NIC×1 [37] | 2.9 *W* | 1.7 *W* | 1.7 *W* | 1.7 *W* | 0.5 *W* |
| Fan×1 [3] | 10 *W* | 1 *W* | 1 *W* | 1 *W* | 0 *W* |
| PSU×1 [3] | 70 *W* | 35 *W* | 35 *W* | 35 *W* | 1 *W* |
| Platform total | 120 *W* (S0$_{(a)}$) | 60.5 *W* (S0$_{(i)}$) | 60.5 *W* (S0$_{(i)}$) | 60.5 *W* (S0$_{(i)}$) | 13.1 *W* (S3) |

Table 3.3: Power consumption for different components in a computer system.

determined with frequency and (5) service time dependency on CPU frequency matters.

SleepScale selects the best combination of frequency and sleep states at runtime. It predicts the characteristics of given workload and determines the optimal power management policy with low overhead. While we note that accurately predicting the workload characteristics of a computing system (i.e., the utilization, service time and inter-arrival time distributions) is the key to effective runtime power management, we also demonstrate that simple prediction techniques can offer sufficient accuracy for real-world utilization traces.

## 3.2 System model

We now present a system model that accounts for both DVFS and sleep states. We later use the model to study performance and power consumption under various workloads.

**Power model**

We discuss how we model (i) the power consumed by the processor, (ii) the power consumed by peripheral (non-CPU) components such as DRAM, hard disk drive (HDD), network interface card (NIC), and (iii) the latencies involved in transitioning between power states.

First, consider processor power. A CPU in the active $C0_{(a)}$ mode (and also in the idling $C0_{(i)}$ state) consumes dynamic power. Power consumption will be proportional to $V^2 f$, where $V$ is the supply voltage and $f \in [0, 1]$ is the DVFS clock frequency scaling factor. In our study, we choose a linear DVFS scenario, where both voltage and frequency are scaled linearly. This assumption falls within the scope of some existing processors (see the datasheet in [38]). Dynamic power consumption in $C0_{(a)}$ and $C0_{(i)}$ will therefore scale cubically with frequency. In the sequel we consider only the frequency parameter $f$ and assume $V$ to be proportional to $f$. In sleep state C1 the clock signal is gated. Thus, only leakage power is consumed. Platform components also have low-power settings and each supports a subset of the CPU low-power settings. Table 3.2 lists the platform low-power settings and the CPU setting (or settings) that each supports.

The power consumption of the entire system is the sum of CPU power and platform power. In the following we use the term “state” or “low-power state” to encompass both CPU and platform modes and denote the combined states by concatenating their notations, e.g., $C0_{(i)}S0_{(i)}$. Table 3.3 tabulates power consumption numbers for the Xeon family of CPUs and associated platform components. As an example, the power consumption in low-power state $C0_{(i)}S0_{(i)}$ is $75V^2 f + 52.7\ W$.

| | $C0_{(a)}$ | $C0_{(i)}$ | C1 | C3 | C6 |
|---|---|---|---|---|---|
| $S0_{(a)}$ | 0 s | – | – | – | – |
| $S0_{(i)}$ | – | 0 s | $1 - 10$ μs | $10 - 100$ μs | $0.1 - 1$ ms |
| S3 | – | – | – | – | $1 - 10$ s |

Table 3.4: Wake-up latencies for different states [3, 4].

Table 3.4 summarizes the delay incurred by the various possible states in returning to active operation. We note that while it is possible to consider a platform shut-down in which the entire system is turned off, the wake-up latency incurred will be enormous, and thus should be considered at a coarser time granularity than is the focus of this work.

**Operation model**

We assume jobs arrive at the system according to some random process with rate $\lambda$ and are served based on the first-come-first-serve (FCFS) order. The server is equipped with a DVFS mechanism, which affect the service time. In active operation the clock frequency can be scaled by a factor $f \in [0, 1]$ and the time it takes to process each job is scaled correspondingly. For CPU-bound jobs, the resulting (scaled) service times are modeled as a random variable with mean $\frac{1}{\mu f}$ where $\mu$ is the max service rate. Setting the frequency to the maximum $f = 1$ yields maximum processing speed $\frac{1}{\mu}$ and setting $f = 0$ stops the server from processing jobs, i.e., the server is in a clock-gated mode. For memory-bound jobs, the service time is modeled as independent of frequency, thus with mean $\frac{1}{\mu}$. Finally, we note that the "utilization" factor $\rho = \frac{\lambda}{\mu}$ is the expected fraction of time the server has jobs to process.

As discussed earlier, in active state $C0_{(a)}S0_{(a)}$ power varies cubically in $f$; hence the power is $P_0 f^3$ for some maximum power $P_0$. The system also has $n$ low-power states indexed by $i$, $1 \leqslant i \leqslant n$. Each time the server becomes idle it enters a sequence of low-power states, staying in each for a pre-set amount of time. The server enters state $i$ some $\tau_i$ seconds after its queue empties. Naturally, $\tau_1 < \tau_2 < \ldots < \tau_n$. Formally, the $i$th low-power state is characterized by the three-tuple $(P_i, \tau_i, w_i)$ where:

- $P_i$ is the power consumed in state $i$,
- $\tau_i$ is the time at which the server enters state $i$, measured from the time the queue empties, and
- $w_i$ is the average wake-up latency that the system incurs to return to the active state $C0_{(a)}S0_{(a)}$ from state $i$.

A job arrival interrupts the low-power state and wakes up the system from its current low-power state. We assume a job arrival immediately triggers the wake-up process, during which no job can be served. Deeper sleep states consume less power but take longer to wake up from so $P_1 > P_2 > \ldots P_n$ but $w_1 < w_2 < \ldots < w_n$. For simplicity and conservative evaluation we assume the power consumption during the wake-up is the same as the power consumed in active operation.

The low-power states that we study include $C0_{(i)}S0_{(i)}$, $C1S0_{(i)}$, $C3S0_{(i)}$, $C6S0_{(i)}$ and C6S3. We also analyze sequences of pairs of low-power states such as entering

$C0_{(i)}S0_{(i)}$ first then C6S3, and also sequences many more low-power states, e.g., entering $C0_{(i)}S0_{(i)}$, $C1S0_{(i)}$, $C3S0_{(i)}$, $C6S0_{(i)}$ and C6S3 in sequence.

## 3.3 Workload-dependent Optimal Policy

In this section we study the model of Section 3.2 under the idealized assumptions of Poisson arrivals and exponentially distributed service times. We present our methodology for evaluating various policy choices and discuss engineering lessons.

### 3.3.1 Methodology

To evaluate a candidate policy we generate $N = 10,000$ jobs and evaluate the policy at each possible frequency setting based on our model presented in Section 3.2. The simulated maximum frequency is $f = 1$ and the minimum is the one that the system is barely stable, i.e., $f = \rho + 0.01$ with step size of 0.01. (We take such a fine step size only to generate smooth plots, in a real system there would be about 10 distinct frequencies.) For policies that consist of a sequence of low-power states, when the processor queue empties the processor enters the low-power states one after another. The arrival of the next job wakes up the processor, incurring some latency. From our simulations we gather results on mean response time $\mathbb{E}[R]$ and average power $\mathbb{E}[P]$. An example of our simulator is provided in Algorithm 1 for single low-power state with $\tau_1 = 0$. Simulating one policy, i.e., one frequency and low-power state combination takes on average 6.3 $ms$ on an Intel i5 2.6 GHz machine using Matlab. This can take even less time when an optimized code is

**Algorithm 1** Simulation under $\rho = \frac{\lambda}{\mu}$ and frequency $f$

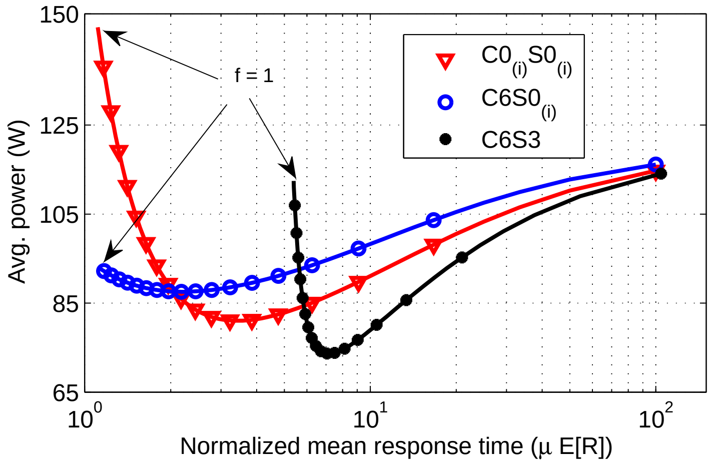
```
1: Generate jobs with job size and inter-arrival time sampled from probability
   distributions with mean $\frac{1}{\mu f}$ and $\frac{1}{\lambda}$ respectively (assuming CPU-bound).
2: for job j in 1 to N do
3:   if job j arrives before $j-1$ departs then
4:     Active += service time of j.
5:     Delay of j = departure time of j - arrival time of j.
6:   else
7:     Active += service time of j + wake-up latency.
8:     Idle += arrival time of j - departure time of $j-1$.
9:     Delay of j = service time of j + wake-up latency.
10:  end if
11: end for
12: Compute delay by taking average of all j jobs.
13: Compute power by the ratio of active and idle periods.
```

dedicated to run Algorithm 1.

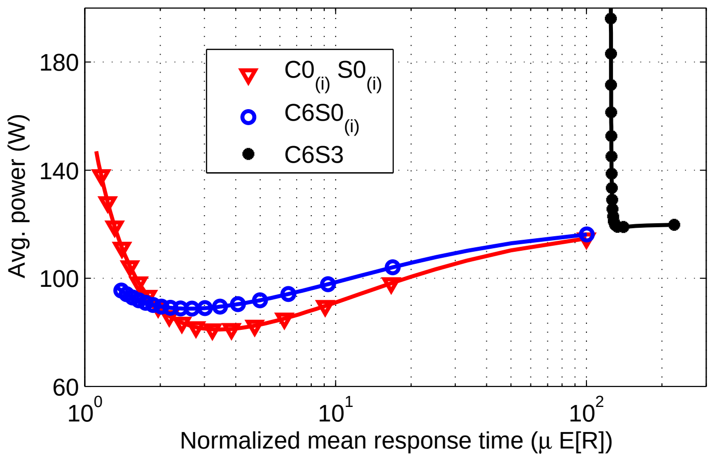
### 3.3.2 Engineering lessons

We first consider $\tau_1 = 0$, i.e., whenever the server completes all jobs in its queue the server immediately enters a low-power state from the active state $C0_{(a)}S0_{(a)}$. Second, to investigate how job size effect the choice of policy, we consider two different sizes: "Google-like" ($\frac{1}{\mu} = 4.2$ ms) and "DNS-like" ($\frac{1}{\mu} = 194$ ms). In Section 3.4.1, when we use traces from data centers for our analysis, the former is roughly the size of a Google web search job and the latter of a DNS look-up job (cf. Table 3.5). Unless otherwise specified, jobs are assumed to be CPU-bound (and thus service time scales linearly in frequency).

The average wake-up latencies for the system to return to the active $C0_{(a)}S0_{(a)}$ state from various low-power states are set as follows. To wake-up from $C1S0_{(i)}$ we



(a) DNS-like: $\rho = 0.1$



(b) Google-like: $\rho = 0.1$

Figure 3.1: Mean response time $\mathbb{E}[R]$ and average power $\mathbb{E}[P]$ trade-off.

set the average latency to 10 $\mu s$, from $C3S0_{(i)}$ we set it to 100 $\mu s$, from $C6S0_{(i)}$ we set it to 1 $ms$, and from C6S3 we set it to 1 $s$. These values fall in the ranges specified in Table 3.4. *We observe that other choices from the range specified do not greatly change the following engineering lessons.*

We begin our engineering lesson by introducing Figure 3.1. In each sub-figure we plot average power consumption $\mathbb{E}[P]$ as a function of two parameters. The first parameter is the mean response time $\mathbb{E}[R]$ normalized by the average service time $\frac{1}{\mu}$. We normalize the mean response time in order to compare different workloads (with different average job sizes) directly. The second parameter is the DVFS frequency setting $f$. Selected choices of $f$ are indicated by the hash marks on each curve. As $f$ is changed the effective service rate $\mu f$ varies proportionally. The left end of each curve is $f = 1$, corresponding to the fastest processing, and hence smallest response time but most power. The right end of the curve is set by the smallest frequency under which the system is stable, roughly $f = \rho$, where $\rho$ is the utilization. The hash marks are spaced uniformly in increments of 0.05.

We also evaluate systems based on Atom processors with the power numbers from [4]. While simulation results will not be shown, we will discuss our observations.

1**) There exists an optimal joint choice of frequency setting and low-power state.** Figure 3.1 plots the DNS-like and Google-like workloads results for $\rho = 0.1$. We observe that there is an optimal joint choice of frequency setting and low-power state that yields the minimum power consumption. Without a constraint on the mean response time, the optimal choice corresponds to the bottom of the lowest bowl. For example, in Figure 3.1-(a) the globally optimum policy uses C6S3 and $f = 0.42$ and runs at an average power of 70 *W*. Setting $f$ too high leads to worsening efficiency since power increases cubically in frequency. But, setting it too small means that each job takes longer to complete, thereby reducing the possibility of entering a low-power state. The optimal choice – the policy at the bottom of the bowl – strikes a balance between these two effects. Policies such as "race-to-halt" [3] wherein the server runs at maximum frequency $f = 1$ until the queue empties and then immediately enters a single low-power state can consume 50% more power. Such a policy corresponds to the leftmost tip of each curve.

Due to small processor power and relatively large platform power, for Atom processors running DNS-like jobs at low utilizations, it is better to run fast and enter low-power state immediately after the job queue empties.

2**) At low utilization, the best low-power state depends on the mean response time budget.** In Figures 3.1 the left-hand side of the plots corresponds to a tight requirement on normalized mean response time. As the constraint is loosened different choices of policies and operating frequency settings become optimal. For example, in Figure 3.1-(a) under the tightest constraint (when the normalized mean response time $\mu\mathbb{E}[R]$ is required to be in the range $[1, 2]$) policies using $C6S0_{(i)}$ are optimal; under a mid-range constraint policies using $C0_{(i)}S0_{(i)}$ are the best; and under the loosest constraint policies using C6S3 prove to be the best. The intuition is as follows.

Consider the range of normalized mean response time for which $C6S0_{(i)}$ is the best in Figure 3.1-(a). This is the best choice for the tightest response time

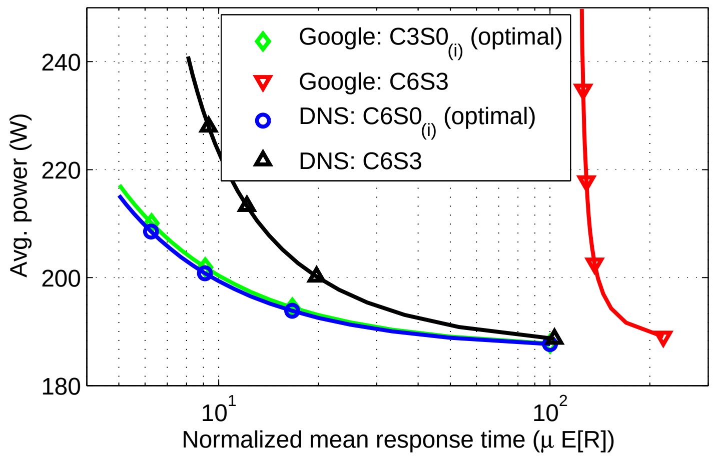


Figure 3.2: Optimal low-power states for Google and DNS-like workload under high utilization.

requirements because setting the frequency high results in faster job completion, thereby increasing the opportunities to enter a more aggressive power saving mode such as C6. On the other hand, under a looser constraint on response time the frequency can be lowered yielding the cubic decrease in power consumption. In this situation the expected duration of an empty queue is small and thus it becomes too costly to enter states like C6 that have high wake-up latencies. Thus $C0_{(i)}S0_{(i)}$ outperforms $C6S0_{(i)}$ in this situation. This observation is also valid for systems of Atom processors running Google-like workload.

3**) The best power state depends on the job size.** Figure 3.2 shows the optimal low-power states among other possible ones (not all shown) for servers under high utilization. When heavily utilized the server rarely exits the active operating state $C0_{(a)}S0_{(a)}$. This reduces opportunities to realize power savings by transitioning to a sleep state, thus most power savings must come from DVFS.

However the job size also plays a crucial rule in the optimal low-power state. For the DNS-like workload, policies using $C6S0_{(i)}$ dominate since the wake-up latency of $C6S0_{(i)}$ is negligible compared to average job size (194 $ms$). But, for Google-like workload the relatively small job size is sensitive to large wake-up latencies, thus policies using $C3S0_{(i)}$ becomes optimal. For the same reason, very aggressive sleep states such as C6S3 should not be used for small size jobs or should be used only during extremely long idle period, "guarded" by workload prediction techniques [8]. Similar observations can also be made in Atom-based systems.

4**) The delay $\tau_i$ to enter a low-power state should be jointly determined with frequency.** When a server becomes idle, it may wait some amount of time before entering a low-power state to avoid unnecessary wake-up costs [39]. Of course, for some low-power states with very small wake-up latencies, there is no need to delay the entrance as the wake-up incurs negligible penalty. However, for other low-power states with heavy wake-up penalties, it is not immediately clear how long the system should wait.

Figures 3.3 considers this situation and show what happens if we delay the entrance to state C6S3 by various amounts. In "$C0_{(i)}S0_{(i)} \to C6S3$", the server first enters $C0_{(i)}S0_{(i)}$ immediately ($\tau_1 = 0$) whenever the job queue empties and will enter C6S3 if it idles for some $\tau_2$ seconds (recall the definition of $\tau_i$ in Section 3.2). We compare policies using delayed C6S3 with ones using immediate C6S3 and immediate $C0_{(i)}S0_{(i)}$. We observe that the delay parameter $\tau_2$ interpolates between the immediate C6S3 and $C0_{(i)}S0_{(i)}$ curves: setting $\tau_2 = 0$ reduces to immediate C6S3 and setting $\tau_2 = \infty$ reduces to immediate $C0_{(i)}S0_{(i)}$. From the plots we

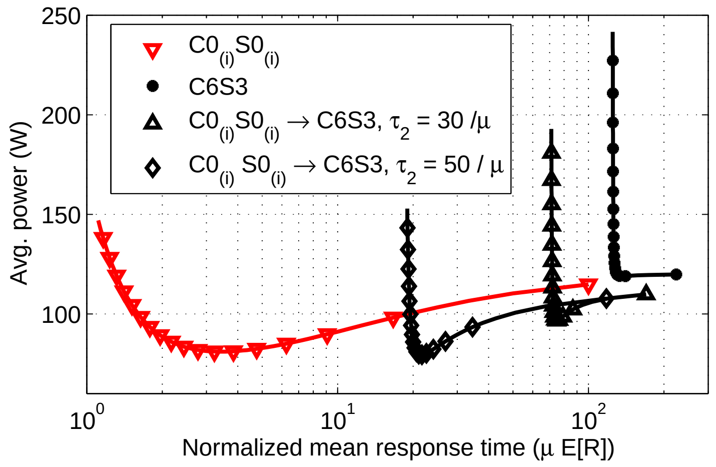


Figure 3.3: Entering the second low-power state after delays for Google-like workload.

observe that by delaying C6S3, more power savings can be made at mild mean response time budget (e.g. consider $\mu\mathbb{E}[R] = 20$). Essentially, what we observe is that there is an optimal combination between frequency and entrance delay that minimizes the power under a certain mean response time constraint.

5**) Sequential power throttle-back is conservative.** It is tempting to concatenate all low-power states i.e., building a system with large number of low-power states and then letting the system enter those states in sequence to derive the most benefit from each. However from our intensive simulation on entering $C0_{(i)}S0_{(i)}$, $C1S0_{(i)}$, $C3S0_{(i)}$, $C6S0_{(i)}$ and C6S3 in sequence, we discover that such policies are not often efficient. The reason is that at high utilization the system rarely enters the last state. At low utilization it is a waste of power not to go to the optimal state immediately. Nevertheless, such sequential power reducing policies can be useful when the arrival statistics are unknown.

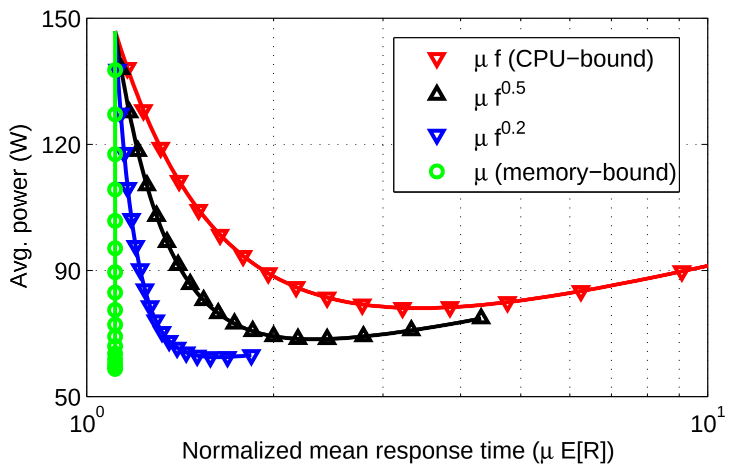


Figure 3.4: Comparison between different CPU usages for DNS-like workload.

6**) Service time dependency on CPU frequency matters.** Now we discuss what happens when the workload is *not* CPU-bound. Recall that for CPU-bound jobs the service rate $\mu$ scales linearly with frequency $f$. For less CPU-bound jobs, the service rate scales sub-linearly and in the extreme case (memory-bound) service rate is insensitive to frequency (as job completion time is dominated by memory access time rather than processing time). In Figure 3.4 we plot a DNS-like workload at low utilization for service rate varying as different functions of clock frequency. It can be observed that the optimal choice of frequency depends on the scaling. For memory-bound jobs, the optimal speed is the lowest speed.

### 3.3.3 Analytic results

In fact, under the Poisson job arrival process and exponential service time distribution we can derive in closed-form the average power $\mathbb{E}[P]$ and mean response time $\mathbb{E}[R]$ for the system model described in Section 3.2. The results obtained from the closed-form expressions match those presented in Figure 3.1. Note that constructing Figure 3.1 using closed-form expression does not involve simulations described in Section 3.3.1. The closed-form solution can also be derived when the service time follows general distributions, i.e., not limited to exponential distribution [35]. However for both general arrival process and service time distribution (which SleepScale assumes, as will be discussed next), no closed-form solution exists to the best of our knowledge. We include these theoretical results in the Appendix A for reference.

## 3.4 SleepScale design

In this section we present SleepScale, a runtime power management algorithm. It consists of a policy manager and a runtime predictor. First, in Section 3.4.1 we detail the policy manager that selects the optimum policy as a function of workload statistics. Second, in Section 3.4.2 we describe our runtime predictor of the statistical characteristics of the workload such as utilization, and arrival rate and service time distributions. This allows SleepScale to select the best policy in an online manner.

### 3.4.1 Policy manager

In this section we describe the policy manager. The manager takes a statistical description of the current workload as input and determines the best policy.

**Policy characterization and selection.**

The policy manager bases its determination of the best policy on the empirically observed distributions of recent arrivals and service times, collected from the server at runtime. The observed recent arrivals and service times can be arbitrary statistics, not limited to Poisson process and exponential distributions. Given the collected statistics it characterizes the power-performance trade-off of each low-power state at a range of frequency settings. It does this by simulating the queuing process as described in Algorithm 1. The optimal policy is the policy that minimizes power consumption while meeting a target QoS constraint. As noted in Section 3.3.1, simulating a single policy takes only 6 $ms$. Considering the finite number of frequencies and low-power states, the overhead of simulating all policies is thus negligible compared to the policy updating period (which will be measured in minutes, as discussed later).

Our QoS constraint is determined by a baseline system. Our baseline is motivated by the fact that data centers are often provisioned to meet a QoS target for some peak demand, often specified in an SLA. To meet SLA commitments during periods of peak demand, the data center should be running full out, i.e., at maximum frequency $f = 1$ without using a low-power state. In contrast, at lower loads there is slack in meeting the QoS which can be exploited to reduce operating costs (e.g., power) as much as possible.

We parameterize the target peak demand through a peak design utilization $\rho_b$. To understand the baseline QoS, consider Figure 3.5. This figure plots the power-delay trade-off for the Google-like workload running with low-power state

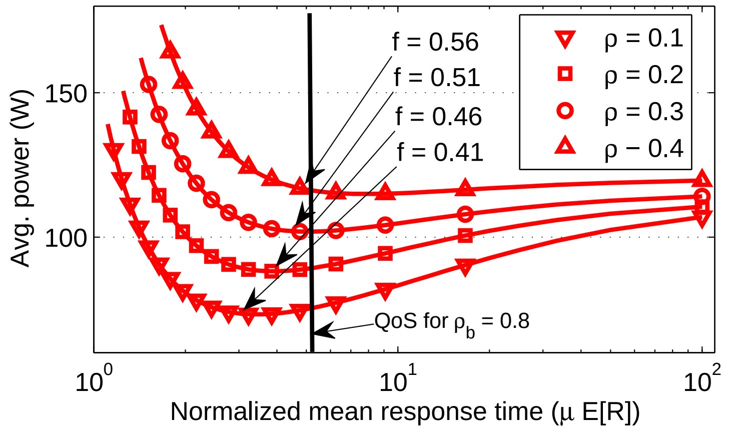
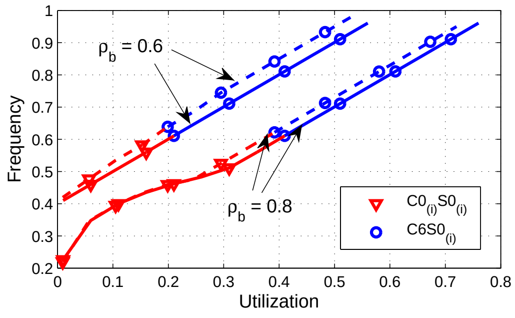


Figure 3.5: Average power/performance trade-off for Google-like workload.

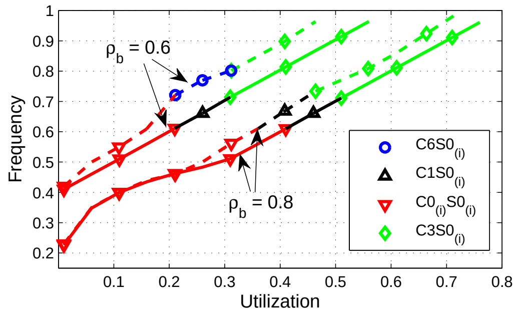
$C0_{(i)}S0_{(i)}$ at different frequency settings and under different utilizations $\rho < \rho_b$. In the plot the baseline QoS throughput constraint is indicated by the vertical bar. This vertical bar indicates that the allowable normalized response time is 5 when $\rho_b = 0.8$, calculated (under the idealized model) as $\mu\mathbb{E}[R] = \frac{1}{(1-\rho_b)} = \frac{1}{1-0.8} = 5$. In general as the utilization $\rho$ is increased from $\rho = 0$ to $\rho = \rho_b$ the curves shift up, meaning that a higher frequency setting is required to maintain the QoS throughput constraint. For instance, for $\rho = 0.4$, which is strictly less than $\rho_b = 0.8$, to minimize average power one must set $f = 0.56$ and the system will operate exactly at the requisite QoS. However, for even lower utilizations, e.g, $\rho = 0.1$, one operates at the lowest average power by setting $f = 0.41$. This is the global minimum for this utilization and the normalized mean response time achieved is about 3, which exceeds the QoS requirement. Thus, one can sometimes exceed the baseline QoS while minimizing power consumption.

| Workload | Inter-arrival Mean | Inter-arrival $C_v$ | Service Mean | Service $C_v$ |
|---|---|---|---|---|
| DNS | 1.1 s | 1.1 | 194 ms | 1.0 |
| Mail | 206 ms | 1.9 | 92 ms | 3.6 |
| Google | 319 μs | 1.2 | 4.2 ms | 1.1 |

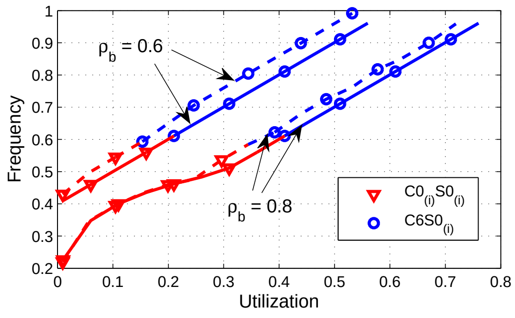
Table 3.5: Different workload types from [5] (partial list).



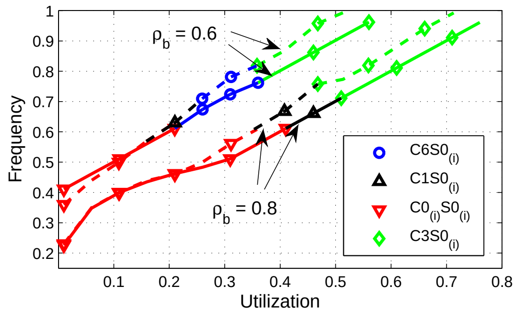
(a) DNS-like with $\mathbb{E}[R]$ constraint



(b) Google-like with $\mathbb{E}[R]$ constraint



(c) DNS-like with $\Pr(R \geqslant d)$ constraint



(d) Google-like with $\Pr(R \geqslant d)$ constraint

Figure 3.6: Policy selection for DNS and Google-like workloads. Each curve plots the optimal pairing of frequency setting and low-power states as a function of utilization $\rho$, and parameterized by QoS constraint $\rho_p$. The different markers (see legend) indicate which low-power state is optimal at each utilization. The solid lines represent what an idealized model computes and dashed lines represent the results using real-world workload statistics.

**Results and observations.**

We now present the results of our policy characterization according to the workload statistics used by the BigHouse [5] simulator. The BigHouse simulator stores statistics of inter-arrival and service times accumulated from long-term observation of live traffic traces for various real-world workloads. Table 3.5 lists summary statistics (inter-arrival and service time mean and coefficient of variation ($C_v$)) of three workloads in the BigHouse simulator.

Figure 3.6 shows the optimum policy as a function of workload, utilization, and QoS constraint ($\rho_b$). As an example, Figure 3.6-(a) plots the optimum policy for the DNS-like workload. Operating frequency is indicated on the vertical axis, utilization on the horizontal axis, and the optimum choice of low-power state by the hash marks is indicated in the legend. Depending on the utilization two different policies become optimal. At low utilization $C0_{(i)}S0_{(i)}$ is optimal and at high utilization $C6S0_{(i)}$ is optimal. Two pairs of curves are plotted. The upper two curves are plotted for a baseline QoS constraint set by $\rho_b = 0.6$, the bottom two for $\rho_b = 0.8$. Note that the constraint under $\rho_b = 0.6$ is *tighter* than the one under 0.8. In each pair of curves one is dashed, meaning it is the policy choice based on the statistics of the BigHouse simulator for that particular workload. The other curve is solid, meaning that it is the optimum policy choice computed by the model considered in Section 3.3 (i.e., Poisson job arrivals and exponential service times) with the same mean inter-arrival and service time as its paired BigHouse curve.

The plots in the second row of Figure 3.6 are defined analogously except that the QoS is measured in terms of the 95th percentile response time, rather than in terms of normalized mean response time. Each sub-plot in the second row shares the same workload as the plot directly above. We summarize key observations below.

1**) There is no "one-size-fits-all" policy.** Different workloads and different utilizations require different policies. Almost all low-power states are useful for some set of operating conditions. Thus, relying on a single low-power state when designing power-efficient architectures can be a poor choice.

2**) The idealized model is sometimes good, but often one needs to use more realistic models.** Recall that the model of Section 3.3 is limited to idealized Poisson job arrivals and exponential service times. These are analytically tractable distributions and when they closely match the actual workload statistics policies based on those results can perform almost as well as the policies based on actual statistics.

We also note that the discrepancy between the policy results based on the idealized model and the BigHouse model is different for two performance constraints; cf. Figures 3.6-(c) and -(d). This is due to the fact that while the mean response time is only concerned with the mean, the 95th percentile response time constraint is concerned with the tail behavior of the response time distribution, which depends critically on the variation in job size and inter-arrival time.

3**) Often the idealized model computes the best choice of low-power state, but not the frequency setting.** Often for a given utilization the computed optimal low-power state is the same but the frequency setting computed by the idealized model is lower than the one computed by BigHouse. If there is a way to adjust the frequency in runtime, one can rely simply on the idealized model without simulation to compute the optimal policy. We leave this as a part of our future work.

4**) The bump in low utilization region indicates that the policy is exceeding its QoS constraint.** At low utilization, the optimal frequency curve for $C0_{(i)}S0_{(i)}$ often has a concave shape. The consistency of this shape can be explained by referring back to Figure 3.5. As was observed earlier, at low utilizations the QoS constraint can be exceeded while reaching the global minimum power (optimized across all frequencies). In terms of Figure 3.5 this means that the systems is operating strictly to the left of the vertical bar. Since the same model underlies the policy optimization based on the idealized and BigHouse models, the global power minimum will be the same for both. For this reason the BigHouse and idealized curves can overlap at low enough utilizations.

As the utilization increases, the optimal frequency setting also increases to keep the power at the global minimum. However beyond some utilization level (roughly $\rho = 0.3$ in Figure 3.5) the lowest power will no longer continue to meet the target performance constraint. At that level the frequency needs to increase more quickly to continue to meet the constraint. Above that utilization levels the $C0_{(i)}S0_{(i)}$ curves in Figure 3.6 transition into their respective linear regimes. Since the $\rho_b = 0.6$ constraint is *tighter* than the $\rho_b = 0.8$ constraint and so in curves with arrow "$\rho_b = 0.6$" in Figure 3.6 we see no evidence of the bump.

### 3.4.2 Runtime predictor

The runtime predictor works epoch by epoch, predicting for the current epoch two important aspects of the workload based on its history: 1) inter-arrival time and

**Algorithm 2** Pseudo-code for utilization predictor

```
1: Initialize history depth $hist$.
2: Initialize a weight vector of size $p = hist$: $\mathbf{v} = \{v(1), \ldots v(p)\}$. Set each of the $p$
   entries to $1/p$.
3: while prediction for $\rho(t)$ do
4:   {// Predict the utilization at time $t$, $\rho'(t)$ using LMS:}
5:   Predict $\rho'(t) = \min[\sum_{i=1}^{p} v(i)\rho(t-i),\ 1]$ from the past $p$ utilization values.
6:   Compute $error = |\rho(t) - \rho'(t)|$.
7:   Update weight $\mathbf{v}$ based on $error$ and $\rho(t-1 : t-p)$.
8:   if $error$ is larger than some adaptive threshold then
9:     {// CUSUM test:}
10:    Reset $p = 1$ and set $v(1) = sum(\mathbf{v})$.
11:  else
12:    Grow $p$, $p = \min(p = p+1,\ hist)$ and set $v(i) = sum(\mathbf{v})/p$ for all $1 \leqslant i \leqslant p$.
13:  end if
14:  $t = t + 1$
15: end while
```

service time statistics and 2) utilization. Each epoch is a $T$ minutes long ($T \geqslant 1$) time period. The prediction is fed into the policy manager. The policy is updated at the beginning of each epoch and is held constant throughout the epoch.

**Distribution prediction.**

The inter-arrival time and service time distributions are predicted based upon jobs events logged in previous epochs. The logs we collect detail the arrival and service times of each jobs. These logged statistics are scaled by the predicted utilization (to be discussed later) and fed into the policy manager. Using Algorithm 1 the policy manager then computes the predicted power consumption and QoS of each candidate policy and then selects the policy to use. Note that since we have already obtained the workload log, generating jobs by sampling the distribution (step 1 in

Algorithm 1) is not needed. Logging all job events is not necessary either: average behavior from the past several epochs will suffice.

Implicitly the predictor predicts the inter-arrival time and service time distributions based on the past epochs. The motivation for working with the logged arrival and service times is that constructing, maintaining and updating a fine-grained distribution histogram and simulation via sampling is expensive in both time and space. Thus we find it is effective to use logs from previous epochs without explicitly building a distribution. As shown in Section 3.3.1, it takes only 6.3 $ms$ for evaluating one single policy. The policy manager only needs to determine the best policy *once* every epoch. Since, e.g., in the results of Section 3.5 epochs will be in minutes-length and the determination of the best policy takes less than 1 $s$ to compute, the computational overhead is negligible.

**Utilization prediction.**

The distribution predictor predicts the statistics based on the recent epochs. We further enhance such prediction by a fine-grained, minute-by-minute utilization prediction. The workload log gathered in Section 3.4.2 used to simulate the policies is adjusted based on the predicted utilization of the upcoming epoch (the first minute of the epoch, by default): the empirical inter-arrival times between jobs are scaled to match the upcoming predicted utilization.

There is a large body of literature on utilization prediction; much is based on pattern matching [40] across days. As an illustration, in Figure 3.7 we plot several days worth of minute-granularity utilization traces from academic departmental servers.

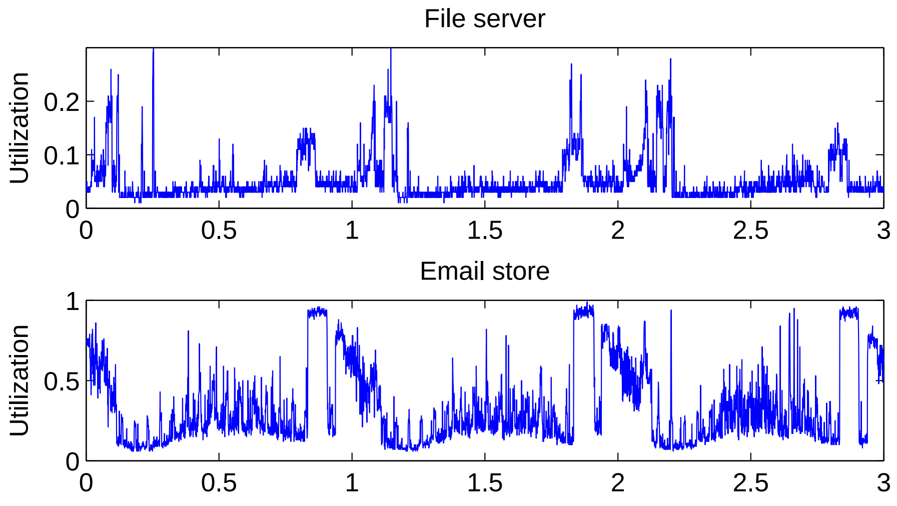


Figure 3.7: Utilization traces plotted across 3 days for different services. Both start at 12 AM of a day. Email store is the host for student, faculty and staff email storage. File server is the host for student files [7, 8].

We observe a periodic daily pattern to the utilization. The abrupt surges observed towards the end of each day in the email store workload, are due to maintenance and back-up services. In contrast to the earlier work based on pattern matching, we study fine-grained minute-by-minute fluctuations in workload behavior. Our approach lets the policy manager react at the processor level to real-time fluctuations in the workload. Our examination will reveal some fundamental aspects of what constitutes good prediction and how the predictor should interact with the selected policy.

In SleepScale we implement three different utilization predictors: a naive-previous predictor, a least-mean-square adaptive filter (LMS) [41], and an LMS filter in conjunction with a cumulative sum change point detection (LMS+CUSUM) [42].

The naive-previous predictor simply uses the utilization in the last minute of the past T-minute epoch as the prediction for the current epoch. This predictor is best suited to track sudden changes in utilization, however it does not effectively predict the stationary behavior of the workload.

The LMS adaptive filter predicts the utilization based on a weighted combination of the utilizations observed over the past $p$ minutes. The weights are updated every minute based on the prediction error. The LMS adaptive filter outperforms the moving average predictor (which would take the average utilization over the past $p$ minutes) because the weight for each of the past $p$ minutes is chosen adaptively, rather than being fixed to a constant $1/p$. However, like the moving average predictor the LMS filter smoothes the data, it does not track abrupt changes well.

As an intermediary between naive-previous predictor and LMS filter, LMS+CUSUM does both tracking and stationary behavior prediction. The pseudo-code of this LMS+CUSUM is given in the Algorithm 2 box. When the CUSUM algorithm detects an abrupt change, the look-back period $p$ in the LMS is reset to 1 (cf. line 10). This resetting drops the smoothing effect of LMS and allows the filter to track the change better. As long as no further abrupt change is detected, $p$ grows until some maximum value is reached (cf. line 12).

**Dynamic frequency over-provisioning.**

The utilization predicted for the first minute of the upcoming T-minute epoch is used to scale the workload log for the entire epoch. The larger T is, the less likely the prediction will be a good one for the entire T-minute epoch. If the predictor overestimates the utilization realized in the epoch, jobs will be processed faster and the queue will tend to empty. However, if the predictor underestimates the realized utilization, the queue will back up, and large delays may result, delays that can propagate into subsequent planning epochs. To control for this in SleepScale we implement the following over-provisioning mechanism.

At the beginning of every T-minute epoch SleepScale computes the average delay incurred by the jobs that were completed in the epoch just past. If the average delay is *below* the delay in the baseline system with utilization $\rho_b$, i.e., if it is less than $\frac{1}{(1-\rho_b)\mu}$, then the frequency determined by the policy manager is further increased by a factor of $\alpha$. At first glance this strategy may appear counter-intuitive since one might think it is more natural to over-provision when the past delay has been above (rather than below) the average. However, we observe that such over-provisioning works best as a "guard band" to buffer against sudden increases in utilization. We illustrate the benefits and the costs of over-provisioning in the next section.

## 3.5 Results

To evaluate SleepScale we combine the real-life daily utilization traces discussed in Figure 3.7 with the BigHouse. We first generate sequences of jobs by sampling the inter-arrival time and service time cumulative distribution functions (CDF) from BigHouse [5]. In systems that serve only a single type of job, the service time distribution is stationary. What varies with utilization is the distribution of inter-arrival times. In our simulated workload traces we then scale the inter-arrival time between generated jobs to match the time-varying utilization of Figure 3.7.

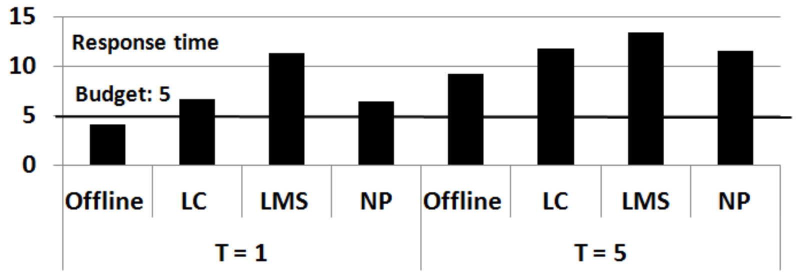


Figure 3.8: Average response time under different predictors and policy update intervals. No over-provisioning is used (i.e., $\alpha = 0$).

SleepScale uses the job stream as the causal input.

### 3.5.1 Under real-world utilization

We first consider a DNS-like server following the email store utilization trace of Figure 3.7. Across the day the utilization trace covers a large range: from 0.1 to 0.9. We evaluate SleepScale over the period extending from 2 AM to 8 PM as from 8 PM to 2 AM everyday back-up and maintenance operations are scheduled. We select the baseline system to be $\rho_b = 0.8$ resulting in a normalized mean response time budget of $1/(1 - \rho_b) = 5$.

1**) Utilization predictors and policy update interval.** In Figure 3.8 we study the performance of different predictors: LMS+CUSUM (LC), LMS-only (LMS), naive-previous (NP) and offline (Offline) as well as the policy update periods (T). The offline predictor is a genie-aided predictor where the true utilizations are assumed to be known non-causally in advance. The naive-previous predictor simply uses the utilization in the last minute of the past T-minute epoch as the prediction for the duration of the next T-minute epoch. The LMS+CUSUM and LMS-only predictors are designed with history length $p = 10$. The average response time is measured when running SleepScale with no over-provisioning ($\alpha = 0$).

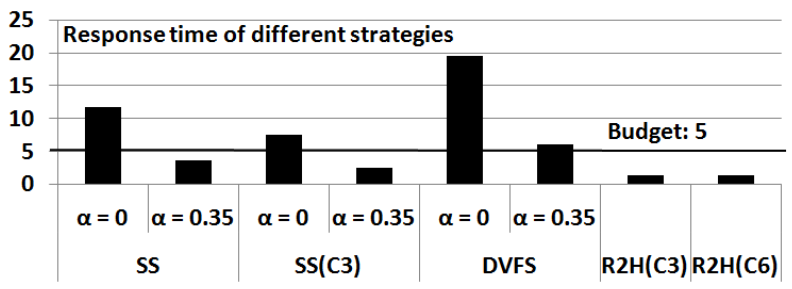
We note that the more often SleepScale updates its policy (i.e., the smaller the $T$), the smaller the response time. Since SleepScale selects the best policy based on the predicted utilization, updating the policy fast often helps to mitigate the prediction error. We also note that the LMS+CUSUM predictor outperforms the LMS-only predictor since the former can track sudden changes in utilization. However, in some cases this tracking can be detrimental: if a big surge is preceded by a short dip, the tracking predictor may under-estimate the surge since it over-reacts to the dip. For most of the utilization traces that we have, we notice that the accuracy of the naive-previous predictor is often comparable to that of the LMS+CUSUM predictor. The accuracy of these predictors can be further improved by considering the correlation (i.e., repeated daily patterns) across past days.

Finally we note that in Figure 3.8 the average response time exceeds the allowed budget in all cases when a utilization predictor is used. As mentioned in Section 3.4.2, if the predictor underestimates the realized utilization, the queue will back up, and large delays may result, delays that can propagate into subsequent planning epochs. Thus, we next set the over-provisioning factor $\alpha = 0.35$ and compare the results to the other power management strategies. Recall from Section 3.4.2 that this means the policy manager will increase the frequency by 35% if the delay in the past $T$-minute epoch is within budget.

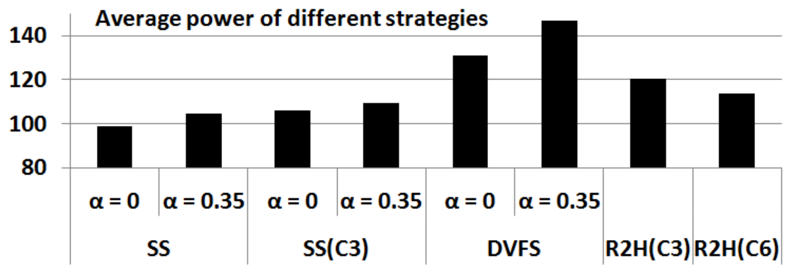
2**) Comparing SleepScale with other strategies.** In Figure 3.9 we compare SleepScale (SS) with other power control strategies, including a SleepScale method that uses only low-power state $C3S0_{(i)}$ (SS(C3)), a DVFS-only strategy that only uses DVFS and no low-power state (DVFS) and race-to-halt mechanisms using $C3S0_{(i)}$ and $C6S0_{(i)}$ (R2H(C3) and R2H(C6) respectively). Both R2H(C3) and R2H(C6) correspond to the strategy that always operates at the maximum frequency setting ($f = 1$) and transitions into a low power state ($C3S0_{(i)}$ or $C6S0_{(i)}$) immediately upon the queue emptying. All strategies use the LMS+CUSUM predictor with the history length $p = 10$ and are updated every $T = 5$ minutes. We make a number of observations.

SleepScale, when equipped with the frequency over-provisioning, achieves the best power efficiency of all strategies while maintaining the response time within budget. Among others, using DVFS only wastes power as the server is not allowed to enter any low-power state when idling. The race-to-halt mechanism also consumes more power than SleepScale as it sets frequency to the maximum. When SleepScale is set to use only $C3S0_{(i)}$ it also consumes more power as this low-power state is a sub-optimal one. *Our results clearly demonstrate the importance of joint optimization of speed scaling and sleep state selection.*

The properly set over-provisioning reduces response time at the cost of a slight increase in power. This is due to the fact that the extra capacity over-provisioning allocates during periods of low utilization allows the server to accommodate unpredictable surges in utilization. This proves to be essential to meet the mean response time budget. Also running slightly faster does not cost too much power as the sever can enter low-power state sooner. (This is not true for DVFS-only strategy as it has no low-power state to use.)



(a) Response time comparison



(b) Average power comparison

Figure 3.9: Comparing SleepScale with other power control strategies. All strategies are running with LMS+CUSUM predictor with $p = 10$ and are updated every $T = 5$ minutes.

Finally we comment on the large response time in the DVFS-only strategy. To minimize power under a given response time budget, the optimal solution for the DVFS-only strategy is to set the frequency low enough that the response time just meets the budget. This however consumes all the performance budget thus even a slight utilization miss-prediction can result in large queuing delay. However for SleepScale, as we demonstrate in Figure 3.5, does not always consume the entire performance budget.

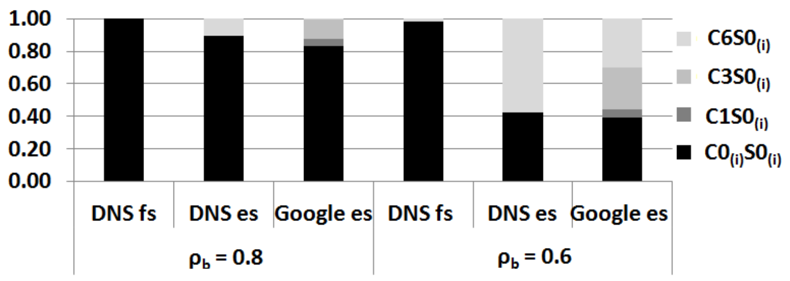


Figure 3.10: Distribution of optimal low-power states selected by SleepScale. LMS+CUSUM predictor is used with history length $p = 10$, update interval $T = 5$ and over-provisioning $\alpha = 0.35$.

## 3.5.2 Distribution of low-power states

In Figure 3.10 we present the distributions of the low-power states selected by SleepScale for different workloads, baselines, and utilizations. We report results for the file server (fs) and email store (es) utilizations. We run both DNS and Google-like services with $\rho_b = 0.6$ and $\rho_b = 0.8$.

At low average utilization and when there are few time-varying fluctuations (which is the case for file server), a single low-power state often suffices. For highly time-varying utilization traces (such as email store), multiple low-power states are used: $C0_{(i)}S0_{(i)}$ and $C6S0_{(i)}$. Tightening the performance constraint further will lead to deeper sleep states being used more often, as the fast processing required to meet the budget creates more opportunities for entering aggressive power-saving states like $C6S0_{(i)}$. These observations all match our analyses in Section 3.3.

## 3.6 Conclusion

In this chapter we present SleepScale, a power management tool to manage active low-power modes and idle low-power states for processors. SleepScale uses queuing-based simulations to determine the optimal configuration for sleep states and operating frequency in modern processors. The optimum such policy can be determined at runtime, via online prediction for workload statistics and utilization. We characterize the performance of SleepScale on data center workload traces. We evaluate SleepScale realizing significant power savings relative to some conventional power management strategies while meeting the same QoS constraints.

# Chapter 4

# FastCap

## 4.1 Introduction

In Chapter 3 we introduced SleepScale, an active low-power mode and idle low-power state manager for processors. Besides processors, other system components such as memory and disks are also equipped with active low-power modes. To develop a method for optimizing low-power settings across different system components, we present FastCap, an optimization framework and search algorithm for performance-aware full-system power capping while promoting fairness across applications. FastCap jointly operates the active low-power modes in processors and memory.

It is important to coordinate the operation of active low-power modes in different system components. A lack of coordination hampers a system's ability to maximize performance under a full-system power cap. To see an example, suppose that the applications are mostly memory-bound, and just changed behavior, causing the system power consumption to decrease substantially below the power budget. In this situation, the CPU power manager (which does not understand memory power and assumes that it will stay the same regardless of the cores' frequencies) might decide that it could improve performance by increasing the core voltage/frequency and bringing the system power very close to the budget. The near-budget power consumption would prevent the (independent) memory power manager from increasing the memory frequency. Adhering to the power budget in this way would produce more performance degradation than necessary, since the applications would have benefited more from a memory frequency increase than core frequency increase(s). The situation would have been better, if the memory power manager had run before the CPU power manager. However, in this case, a similar problem would have occurred for CPU-bound applications.

Coordination is especially important when maximizing performance under a server power cap for three reasons: (1) exceeding the server power budget for too long may cause temperatures to rise or circuit breakers to trip; (2) it may be necessary to purchase more expensive cooling or power supply infrastructures to achieve the desired application performance; and (3) even when the power capping decisions are made at a coarser grain (e.g., rack-wise), individual servers must respect their assigned power budgets.

FastCap efficiently selects voltage/frequency configurations that maximize a many-core system's performance, while respecting a user-provided power budget. Importantly, FastCap also enforces fairness across applications, so its performance maximization is intended to benefit all applications equally instead of seeking only the highest possible instruction throughput. FastCap has very low time complexity (linear in the number of cores), despite the combinatorial number of possible power mode configurations.

To devise FastCap, we first develop a queuing model that effectively captures the workload dynamics in a many-core system (Section 4.2.1). Based on the queuing model, we formulate a non-linear optimization framework for maximizing the performance under a given power budget (Section 4.2.2). To solve the optimization problem, we make a key observation that core frequencies can be determined optimally in linear time for a given memory frequency. We develop the FastCap algorithm (Algorithm 3) and implement it to operate online. The operating system runs the algorithm periodically (once per time quantum, by default), and feeds a few performance counters as inputs to it (Section 4.2.3).

We highlight two aspects of FastCap: (1) it does OS-based full-system power capping, as the performance- and fairness-aware joint selection of CPU and memory DVFS modes is too complex for hardware to do; and (2) it enforces caps at a relatively fine per-quantum (e.g., several milliseconds) grain, as rapid control may be required depending on the part of the power supply infrastructure (e.g., server power supply, blade chassis power supplies, power delivery unit, circuit breaker) that has been oversubscribed and its time constants. Moreover, capping power efficiently at a fine granularity is more challenging than doing so at a coarse one. Nevertheless, FastCap assumes that the server hardware is responsible for countering power spikes at even shorter granularities, if this is necessary.

To evaluate FastCap, we simulate it for a server running different types of workloads (Section 4.3). (A real implementation is not possible mainly as FastCap applies memory DVFS, which has recently been proposed in [12, 25] and is not yet readily available in commercial servers.) Our results show that FastCap maintains the overall system power under the budget while maximizing the performance of each application. Our results also show that FastCap produces better application performance and fairness than many state-of-the-art policies (even after they are extended to use memory DVFS as well), because of its ability to fairly allocate the power budget and avoid performance outliers. Finally, our results demonstrate that FastCap behaves well in many scenarios, including different processor architectures (in-order vs. out-of-order execution), memory architectures (single vs. multiple memory controllers), numbers of cores, and power budgets.

## 4.2 FastCap Design

### 4.2.1 System model

FastCap models a system with $N$ (in-order) cores, $B$ memory banks, and a common memory bus for data transfers. (We also study out-of-order cores in Section 4.3.2.) Denote by $\mathcal{N}$ the set of cores. We assume each core runs one *application* and we name the collection of $N$ applications as a *workload*. We use a closed-network queuing model, as depicted in Figure 4.1.

**Many-core performance.** Every core periodically issues memory access requests (resulting from last-level cache misses and writebacks) independently of the other cores. Though the following description focuses on cache misses for simplicity, FastCap also models writebacks as occupying their target memory banks and

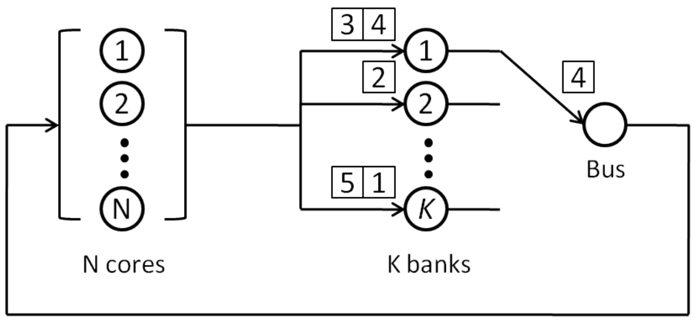


Figure 4.1: FastCap's queuing model and the "transfer blocking" property. Memory bank 1 receives requests from cores 4 and 3. The requested data for core 4 has been fetched and is being transferred on the memory bus. At the same time, bank 1 is blocked from processing the request from core 3 until the last request is successfully transferred to core 4.

the memory bus. In addition, FastCap assumes that writebacks happen in the background, off the critical performance path of the cores.

After issuing a request, the core waits for the memory subsystem to fetch and return the requested cache line before executing future instructions. We denote by $z_i, i \in \mathcal{N}$ the average time core $i$ takes to generate a new request after the previous request completes (i.e., data for the previous request is sent back to core $i$, see Figure 4.2). The term $z_i$ is often called the *think time* in the literature on closed queuing networks [35]. Further, to model core DVFS, we assume each core can be voltage and frequency scaled independently of the other cores, as in [43, 44]. This translates to a scaled think time: denote by $\overline{z_i}$ the minimum think time achievable at the maximum core frequency. Thus, the ratio $\overline{z_i}/z_i \in [0, 1]$ is the frequency scaling factor: setting frequency to the maximum yields $z_i = \overline{z_i}$. The minimum think time depends on the application running on the corresponding core and may change over time. FastCap takes the minimum think time $\overline{z_i}$ as an input. Determining the frequency for core $i$ is equivalent to determining the think time $z_i$. We assume there are $F$ frequency levels for each core.

We assume the shared last-level cache (L2) sits in a separate voltage domain that does not scale with core frequencies. According to our detailed simulations, changing core frequencies does not significantly change the per-core cache miss rate. Thus, for simplicity, we model the average L2 *cache time* $c_i$ for each core $i$ as independent of the core frequency.

**Memory performance.** Each of the $B$ memory banks serves requests that arrive within its address range. After serving one request, the retrieved data is sent back to the corresponding core through the common bus that is shared by all memory banks. The bus is used in a first-come-first-serve manner: any request that is ready to leave a bank must queue behind all other requests in other banks that finish earlier before it can acquire the bus. Furthermore, each memory bank cannot process the next enqueued request until its current request is transferred to the appropriate core (cf. Figure 4.1). In queuing-theoretic terminology, this memory subsystem exhibits a "transfer blocking" property [45, 46]. In Figure 4.1, we illustrate the transfer blocking property via an example.

An important performance metric for the memory subsystem is the *mean response time*, which is the average amount of time a request spends in the memory (cf. Figure. 4.2). To the best of our knowledge, no closed-form expression exists for the mean response time in a queuing system with the transfer blocking property. Instead of deriving an explicit form for the mean response time, FastCap uses the following approximation.

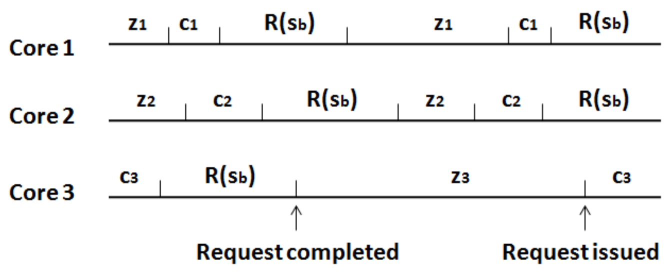


Figure 4.2: An example workload dynamics with $N = 3$ cores. Variables $z_i$ and $c_i$ are the think time and cache time for core $i$, respectively. $R(s_b)$ is the response time of the memory. $z_i$, $c_i$ and $R(s_b)$ are all average values. The sum $R(s_b) + c_i + z_i$ is the total time for one memory access of core $i$.

When a request arrives at a bank, let $Q$ be the expected number of requests enqueued at the bank (including the newly arrived request). When the request has been processed and is ready to be sent back to the requesting core, let $U$ be the expected number of enqueued requests waiting for the bus, including the departing request itself. Denote by $s_m$ the average memory access time at each bank. Denote by $s_b$ the bus transfer time. FastCap approximates the mean response time of the memory subsystem as:

$$R(s_b) \approx Q(s_m + Us_b). \tag{4.1}$$

A previous study [13] has found this equation to be a good approximation to the response time of the memory subsystem.

The memory DVFS method is based on MemScale [25], which dynamically adjusts memory controller, bus, and dual in-line memory module (DIMM) frequencies. Although these memory subsystem frequencies are adjusted together, we simplify the discussion by focusing on adjusting only the bus frequency. This translates to a scaled bus transfer time. Denote by $\overline{s_b}$ the minimum bus transfer time at the maximum bus frequency – the ratio $\overline{s_b}/s_b \in [0, 1]$ is the bus frequency scaling factor. We assume the bus frequency can take $M$ values. In the FastCap algorithm, the minimum bus transfer time $\overline{s_b}$ is used as an input, and determining a frequency for the memory is equivalent to determining the transfer time $s_b$.

**Power models.** Using our detailed simulator (see Section 5.3), we study the power consumption of cores and the main memory serving different workloads.

We model the power drawn by core $i$ as,

$$P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} + P_{i,static}, \tag{4.2}$$

where $P_i$ is the maximum voltage/frequency-dependent power consumed by the core, $\alpha_i$ is some exponent typically between 2 and 3, and $P_{i,static}$ is the static (voltage/frequency-independent) power the core consumes at all times. At runtime, FastCap periodically recomputes $P_i$ and $\alpha_i$ by using power estimates for core $i$ running at different frequencies, and solving the instances of Equation 4.2 for these parameters. We note that many prior papers e.g. [47, 48] used simple models (e.g., assuming the power is always linearly dependent on the frequency) that do not account well for different workload characteristics.

We model the memory power as

$$P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_{m,static}, \tag{4.3}$$

where $P_m$ is the maximum memory power. In practice, we observe that the exponent

$\beta$ is close to 1. This is because we only scale the frequency and not the voltage of the memory bus and DIMMs. The memory also consumes some static power $P_{m,static}$ that does not vary with the memory frequency. At runtime, FastCap periodically recomputes $P_m$ and $\beta$ by using power estimates for the memory running at different frequencies, and solving the instances of Equation 4.3 for the parameters.

We include all the sources of power consumption that do not vary with either core or memory frequencies into a single term $P_s$. This term includes the static power of all cores $\sum_i P_{i,static}$, the memory's static power $P_{m,static}$, the memory controller's static power, the L2 cache power, and the power consumed by other system components, such as disks and network interfaces.

To study the accuracy of our power model under dynamically changing workloads, we simulate both CPU- and memory-bound jobs and find that the modeling error is less than 10%.

*We include all the sources of power consumption that do not vary with either core or memory frequencies into a single term* $P_s$. This term includes the static power of all cores $\sum_i P_{i,static}$, the memory's static power $P_{m,static}$, the memory controller's static power, the L2 cache power, and the power consumed by other system components, such as disks and network interface cards.

**Model discussion.** By making $z_i$ represent the time between two consecutive *blocking* memory accesses, FastCap's model can easily adapt to out-of-order cores with multiple outstanding misses per core; assuming non-blocking accesses are off the critical path, just as cache writebacks. We discuss our out-of-order implementation in Section 4.3.2. FastCap can also easily adapt to multiple controllers by considering different response times for different controllers. In this scenario, the probability of each core using each controller (i.e., the access pattern) has to be considered. We defer the discussion of multiple controllers to Section 4.3.2.

### 4.2.2 Optimization and algorithm

FastCap's goal is to maximize the performance of applications under a full-system power budget. Importantly, FastCap seeks to fairly allocate the budget across the cores (applications) and memory, so that all applications degrade by the same fraction of their maximum performance as a result of the less-than-peak power. Thus, FastCap seeks to prevent "performance outliers", i.e. applications that get degraded much more than others.

Based on the queuing model (cf. Figure 4.2), we use the time interval between two memory accesses (we call it *turn-around time*, i.e., $z_i + c_i + R(s_b)$) as the performance metric. Since a certain number of instructions is executed during a given think time $z_i$, the shorter the turn-around time is, the higher the instruction throughput and thus the better the performance. Based on this metric, we propose the following optimization for FastCap.

$$\text{Maximize} \quad D \tag{4.4}$$

$$\text{subject to} \quad \frac{z_i + c_i + R(s_b)}{\overline{z_i} + c_i + R(\overline{s_b})} \leqslant 1/D \quad \forall\, i \in \mathcal{N} \tag{4.5}$$

$$\sum_i P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} + P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_s \leqslant B\overline{P} \tag{4.6}$$

$$s_b \geqslant \overline{s_b}, \quad z_i \geqslant \overline{z_i}, \quad s_b, z_i \in \mathbb{R} \quad \forall\, i \in \mathcal{N} \tag{4.7}$$

The optimization is over $z_i$ and $s_b$. The objective is to maximize the performance (or to minimize the performance degradation $1/D$ as much as possible). Constraint (4.5) specifies that each core's average turn-around time can only be at most $1/D \geqslant 1$ of the minimum average turn-around time for that core. (Recall that a higher turn-around time means lower performance.) To guarantee fairness, we apply the same upper-bound $1/D$ for all cores with respect to their best possible performance (highest core and memory frequencies). Constraint (4.6) specifies that the total power consumption (core power plus memory power plus system background power) should be no higher than the power budget. The budget is expressed as the peak full-system power $\overline{P}$ multiplied by a given budget fraction $0 < B \leqslant 1$. The constraints (4.7) specify the range of each variable. Since the objective function and each constraint are convex, the optimization problem is convex.

Note that the optimization problem is constrained by the overall system budget. However, it can be extended to capture per-processor power budgets by adding a constraint similar to constraint (4.6) for each processor.

FastCap solves the optimization problem for $z_i$ and $s_b$, and then sets each core (memory) frequency to the value that, after normalized to the maximum frequency, is the closest to $\overline{z_i}/z_i$ ($\overline{s_b}/s_b$). For the cores and memory controller, a change in frequency may entail a change in voltage as well. Thus, the power consumed by each core and memory is always dynamically adjusted based on the applications' performance needs. The coupling of the objective (4.4) and constraint (4.5) seeks to minimize the performance degradation of the application that is furthest away from its best possible performance. Since each core has its own minimum turn-around time and the same upper-bound proportion is applied to all cores, we ensure fairness among them and mitigate the performance outlier problem.

The optimization problem can be solved quickly using numerical solvers such as CPLEX. However, it can be solved substantially faster using the following observations.

**Theorem 4.1.** *Suppose the solution $D^*$, $s_b^*$ and $z_i^*, i \in \mathcal{N}$ are the optimal solution to the optimization problem. Then, inequalities (4.5) and (4.6) must be equalities.*

We defer the proof to the Appendix B.

Theorem 4.1 suggests that the optimal solution must consume the entire power budget and each core must operate at $1/D$ times of its corresponding target. With constraints (4.5) and (4.6) as equalities, the optimal think time $z_i$ can be solved in linear time $O(N)$ for a given bus time $s_b$. This is because $z_i$ can be written as

$$z_i = \frac{\overline{z_i} + c_i + R(\overline{s_b})}{D} - c_i - R(s_b). \tag{4.8}$$

We then substitute equation (4.8) into constraint (4.6), and solve for $D$ using the equality condition for constraint (4.6). Then, all optimal $z_i$ can be computed in linear time using equation (4.8).

We can then exhaustively search through $M$ possible values for $s_b$ to find the globally optimal solution. However, since the optimization problem is convex, we only need to find a local optimal. Since we can find an optimal solution for each bus transfer time $s_b$, we can simply perform a binary search across all $M$ possible

**Algorithm 3** FastCap $O(N \log M)$ algorithm

```
1: Inputs: $\{P_i\}$, $\{\alpha_i\}$, $P_m$, $\beta$, $P_s$, $\{\overline{z_i}\}$, $\overline{s_b}$, $Q$, $U$, $s_m$, $B$, $\overline{P}$ and an ordered array of $M$
   candidate values for $s_b$.
2: Outputs: $\{z_i\}$ and $s_b$
3: Let $\ell := 0$ and $r := M - 1$.
4: while $\ell \neq r$ do
5:   $m := (\ell + r)/2$.
6:   Solve the optimal $D$ for the $m^{th}$ $s_b$ value.
7:   Solve the optimal $D$ for the $(m \pm 1)^{th}$ $s_b$ values. Let the optimal $D$ be denoted
     as $D^+$ and $D^-$ respectively.
8:   if $D < D^+$ then
9:     $\ell := m$
10:  else if $D^- > D$ then
11:    $r := m$
12:  else
13:    break
14:  end if
15: end while
16: Set each core (memory) frequency to the closest frequency to $\overline{z_i}/z_i$ ($\overline{s_b}/s_b$) after
    normalization.
```

values for $s_b$ to find the local optimal. This results in the $O(N \log M)$ algorithm shown in Algorithm 3.

### 4.2.3 Implementation

**Operation.**

FastCap splits time into fixed-size epochs of $L$ milliseconds each. It collects performance counters from each core 300 $\mu$s into each epoch, and uses them as inputs to the frequency selection algorithm. We call this 300 $\mu$s the *profiling phase* and empirically we find its length enough to capture the latest application behaviors. During the profiling phase, the applications execute normally.

Given the inputs, the OS runs the FastCap algorithm and depending on the outcome may transition to new core and/or memory voltage/frequencies for the remainder of the epoch. During a core's frequency transition, the core does not execute instructions. To adjust the memory frequency, all memory accesses are temporarily halted, and PLLs and DLLs are re-synchronized. The core and memory transition overheads are negligible (tens of microseconds) compared to the epoch length (in milliseconds).

**Collecting input parameters.** Several key FastCap parameters, such as $P_i$, $\alpha_i$, $P_m$, $\beta$, the minimum think time $\overline{z_i}$, and queue sizes $Q$ and $U$ come directly or indirectly from performance counters. Now, we detail how we obtain the inputs to the algorithm from the counters. To compute $\overline{z_i}$, we use the following counters: 1) $TPI_i$, *Time Per Instruction* for core $i$ during profiling, 2) $TIC_i$, *Total Instructions Executed* during profiling, and 3) $TLM_i$, *Total Last-level Cache Misses* (or number of memory accesses) during profiling. The ratio between $TIC_i$ and $TLM_i$ is the average number of instructions executed between two memory accesses. We then set $\overline{z_i}$ as:

$$TPI_i \times \frac{TIC_i}{TLM_i}, \tag{4.9}$$

then scale it by the ratio between the maximum frequency and the frequency used during profiling.

To obtain $P_i$, $\alpha_i$, $P_m$, and $\beta$, FastCap keeps data about the last three frequencies it has seen, and periodically recomputes these parameters. To obtain $Q$ and $U$, we use the performance counters that log the average queue sizes at each memory bank and bus. We obtain $Q$ by taking the average queue size across all banks. We obtain $U$ directly from the corresponding counter. To obtain $s_m$, we take the average memory access time at each bank during the profiling phase. The minimum bus transfer time $\overline{s_b}$ is a constant and, since each request takes a fixed number of cycles to be transferred on the bus (the exact number depends on the bus frequency), we simply divide the number of cycles by the maximum memory frequency to obtain $\overline{s_b}$. All background power draws (independent of core/memory frequencies or workload) are measured and/or estimated statically.

### 4.2.4 Hardware and software costs

FastCap requires no architectural or software support beyond that in [13]. Specifically, core DVFS is widely available in commodity hardware, although today one may see fewer voltage domains than cores. Research has shown this is likely to change soon [43] [44]. Existing DIMMs support multiple frequencies and can switch among them by transitioning to powerdown or self-refresh states [49], although this capability is typically not used by current servers. Integrated CMOS memory controllers can leverage existing DVFS technology. One needed change is for the memory controller to have separate voltage and frequency control from other processor components. In recent Intel architectures, this would require separating shared cache and memory controller voltage control. In terms of software, the OS must periodically invoke FastCap and collect several performance counters.

| Name | MPKI | WPKI | Applications ($\times$N/4 each) | | | |
|---|---|---|---|---|---|---|
| ILP1 | 0.37 | 0.06 | vortex | gcc | sixtrack | mesa |
| ILP2 | 0.16 | 0.03 | perlbmk | crafty | gzip | eon |
| ILP3 | 0.27 | 0.07 | sixtrack | mesa | perlbmk | crafty |
| ILP4 | 0.25 | 0.04 | vortex | gcc | gzip | eon |
| MID1 | 1.76 | 0.74 | ammp | gap | wupwise | vpr |
| MID2 | 2.61 | 0.89 | astar | parser | twolf | facerec |
| MID3 | 1.00 | 0.60 | apsi | bzip2 | ammp | gap |
| MID4 | 2.13 | 0.90 | wupwise | vpr | astar | parser |
| MEM1 | 18.22 | 7.92 | swim | applu | galgel | equake |
| MEM2 | 7.75 | 2.53 | art | milc | mgrid | fma3d |
| MEM3 | 7.93 | 2.55 | fma3d | mgrid | galgel | equake |
| MEM4 | 15.07 | 7.31 | swim | applu | sphinx3 | lucas |
| MIX1 | 2.93 | 2.56 | applu | hmmer | gap | gzip |
| MIX2 | 2.55 | 0.80 | milc | gobmk | facerec | perlbmk |
| MIX3 | 2.34 | 0.39 | equake | ammp | sjeng | crafty |
| MIX4 | 3.62 | 1.20 | swim | ammp | twolf | sixtrack |

Table 4.1: Workload descriptions.

## 4.3 Evaluation

In this section, we evaluate FastCap and compare it against several other DVFS-based power capping approaches.

### 4.3.1 Methodology

**Workloads.** Table 4.1 describes the workloads we use. We construct the workloads by combining applications from the SPEC 2000 and SPEC 2006 suites. We use workloads exhibiting a range of compute and memory behavior, and group them into the same mixes as [25, 50]. The workload classes are: memory-intensive (MEM), compute-intensive (ILP), compute-memory balanced (MID), and mixed (MIX, one or two applications from each other class). The rightmost column of Table 4.1 lists the application composition of each workload; we execute N/4 copies of each application to occupy all N cores.

We run the best 100M-instruction simulation point for each application (selected using Simpoints 3.0 [51]). A workload terminates when its slowest application has run 100M instructions. Table 4.1 lists the L2 misses per kilo-instruction (MPKI) and writebacks per kilo-instruction (WPKI) for $N = 16$. In terms of the workloads' running times, the memory-intensive workloads tend to run more slowly than the CPU-intensive ones.

**Simulation infrastructure.** Our evaluation uses a two-step simulation methodology. In the first step, we use M5 [52] to collect memory access traces (consisting of L1 cache misses and writebacks), and per-core activity counter traces. In the second step, we feed the memory traces into our detailed last-level cache/memory simulator of an N-core system with a shared L2 cache, an on-chip memory controller, multiple memory channels, and DRAM devices. We also feed core activity traces, along with the runtime statistics from the L2 module, into McPAT [53] to dynamically estimate the CPU power. Overall, our infrastructure simulates in detail the aspects of cores, caches, memory controller, and memory devices that are relevant to our study, including memory device power and timing, and row buffer management.

Table 4.2 lists our default simulation settings. We simulate in-order cores with the Alpha ISA. Each core is allowed one outstanding L2 miss at a time. In Sec-

| Feature | | Value |
|---|---|---|
| CPU cores | | N in-order, single thread, 4GHz<br>Single IALU IMul FpALU FpMulDiv |
| L1 I/D cache (per core) | | 32KB, 4-way, 1 CPU cycle hit |
| L2 cache (shared) | | 16MB, N-way, 30 CPU cycle hit |
| Cache block size | | 64 bytes |
| Memory configuration | | 4 DDR3 channels for 16 and 32 cores<br>8 DDR3 channels for 64 cores<br>8 2GB ECC DIMMs |
| Time | tRCD, tRP, tCL | 15 ns, 15 ns, 15 ns |
| | tFAW | 20 cycles |
| | tRTP | 5 cycles |
| | tRAS | 28 cycles |
| | tRRD | 4 cycles |
| | Refresh period | 64 ms |
| Current | Row buffer | 250 (read) mA, 250 (write) mA |
| | Precharge | 120 mA |
| | Active standby | 67 mA |
| | Active pwrdown | 45 mA |
| | Precharge standby | 70 mA |
| | Precharge pwrdown | 45 mA |
| | Refresh | 240 mA |

Table 4.2: Main system settings.

tion 4.3.2, we simulate an optimistic out-of-order design as well. Table 4.2 also details the memory subsystem we simulate: 4 DDR3 channels for 16 and 32 cores and 8 DDR3 channels for 64 cores, each of which populated with two registered, dual-ranked DIMMs with 18 DRAM chips each. Each DIMM also has a PLL device and 8 banks. Timing and power parameters are taken from Micron datasheets for 800 MHz devices [54].

Our simulated memory controller exploits bank interleaving and uses closed-page row buffer management (a bank is kept open after an access only if another access for the same bank is already pending), which outperforms open-page policies for multi-core CPUs [55]. Memory read requests (cache misses) are scheduled using FCFS, with reads given priority over writebacks until the writeback queue is half-full. More sophisticated memory scheduling is unnecessary for our single-issue workloads, as opportunities to increase bank hit rate via scheduling are rare. In fact, such improvements are orthogonal to our studies. In Section 4.3.2, we also simulate multiple memory controllers.

We assume per-core DVFS, with 10 equally-spaced frequencies in the range 2.2-4.0 GHz. We assume a voltage range matching Intel's Sandybridge, from 0.65 V to 1.2 V, with voltage and frequency scaling proportionally, which matches the behavior we measured on an i7 CPU. We assume uncore components, such as the shared L2 cache, are always clocked at the nominal frequency and voltage.

As in [25], we scale memory controller frequency and voltage, but only frequency for the memory bus and DRAM chips. The on-chip 4-channel memory controller has the same voltage range as the cores, and its frequency is always double that of the memory bus. We assume that the memory bus and DRAM chips may be frequency-scaled from 800 MHz to 200 MHz, with steps of 66 MHz. We determine power at each frequency using Micron's calculator [54]. Transitions between bus frequencies are assumed to take 512 memory cycles plus 28 ns, which accounts for a DRAM state transition to fast-exit precharge powerdown and DLL re-locking [25, 56]. Some components' power draws also vary with utilization. Specifically, register and memory controller power scale linearly with utilization, whereas PLL power scales only with frequency and voltage. As a function of utilization, the

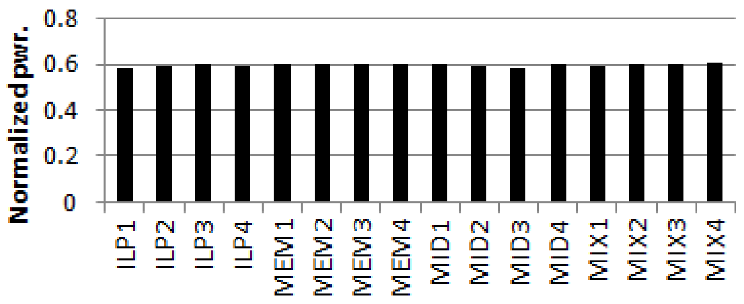


Figure 4.3: FastCap average power consumption normalized to the peak power. Power budget is 60% of the peak.

PLL/register power ranges from 0.1 W to 0.5 W [25, 57], whereas the memory controller power ranges from 4.5 W to 15 W.

We model the power for the non-CPU, non-memory system components as a fixed 10 W. Under our baseline assumptions, at maximum frequencies, the CPU accounts for roughly 60%, the memory subsystem 30%, and other components 10% of system power.

### 4.3.2 Results

We first run all workloads under the maximum frequencies to observe the peak power the system ever consumed. We observe the peak power $\overline{P}$ to be 60 Watts for 4 cores, 120 Watts for 16 cores, 210 Watts for 32 cores, and 375 Watts for 64 cores. We present results for a 16-core system in which FastCap is called every $L = 5$ $ms$. (The 5 $ms$ epoch length matches a common OS time quantum.)

**Power consumption.** We first evaluate FastCap under a 60% power budget fraction, i.e. $B$ in equation (4.6) equals 60%. Figure 4.3 shows the average power spent

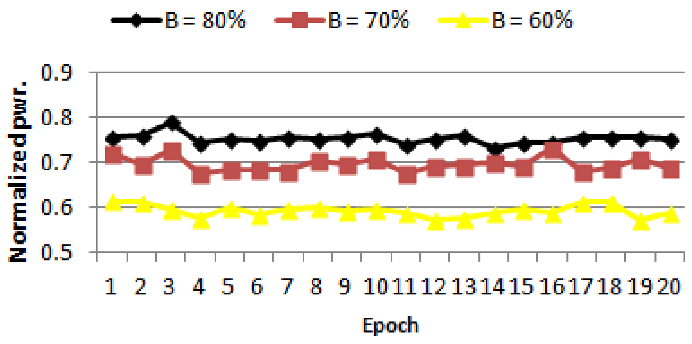


Figure 4.4: Normalized average power draw when running MEM3, as a function of time and power budget.

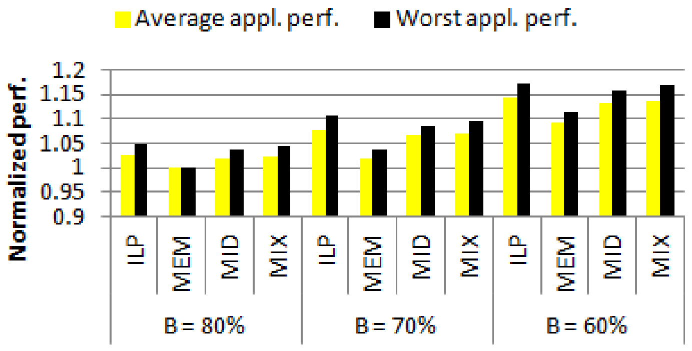


Figure 4.5: Average and worst application performance for each workload class and three power budgets.

by FastCap running each workload on the 16-core system. FastCap successfully maintains overall system power to be just under 60% of the peak power.

Figure 4.4 shows the FastCap behavior for 3 power budgets (as a fraction of the full-system peak power) for the MEM3 workload, as a function of epoch number. The figure shows that FastCap corrects budget violations very quickly (within 10ms), regardless of the budget. Note that MEM3 exhibits per-epoch average powers somewhat lower than the cap for $B = 80\%$. This is because memory-bound workloads do not consume 80% of the peak power, even when running at the maximum core and memory frequencies.

**Application performance.** Recall that, under tight power budgets, FastCap seeks to achieve similar (fractional) performance losses compared to using maximum frequencies for all applications. So, where we discuss a *performance loss* below, we are referring to the performance degradation (compared to a run at maximum frequency) due to power capping, and *not* to the absolute performance.

Figure 4.5 shows the average and worst application performance (in cycles per instruction or CPI) normalized to the baseline system (maximum core and memory frequencies) for all ILP, MEM, MID and MIX workloads. The higher the bar, the worse the performance is compared to the baseline. For each workload class, we compute the average and worst application performance across all applications in workloads of the class. For example, the ILP average performance is the average CPI of all applications in ILP1, ILP2, ILP3 and ILP4, whereas the worst performance is the highest CPI among all applications in these workloads. In the figure, values above 1 represent the percentage application performance loss.

Figure 4.5 shows that the worst application performance differs only slightly from the average performance. This result shows that FastCap is fair in its (performance-aware) allocation of the power budget to applications. The figure also shows that the performance of memory-bound workloads (MEM) tends to degrade less than that of CPU-bound workloads (ILP) under the same power budget. This is because the MEM workloads usually consume less full-system power than their ILP counterparts. Thus, for the same power budget, the MEM workloads require smaller frequency reductions, and thus exhibit smaller fractional performance losses.

**Core/memory frequencies.** FastCap smartly adjusts the core and memory frequencies based on the application needs. For instances, in the CPU-bound workload ILP1, the cores run at high frequency (around 3.5 GHz) while the memory runs at low frequency (around 200 MHz). In the memory-bound workload MEM1, the cores run at low frequency (around 3.0 GHz) while the memory runs at high frequency (around 800 MHz). In workload MIX4, which consists of both CPU- and memory-bound applications, memory frequencies are in the middle of the range (around 500 MHz). The exact frequencies FastCap selects vary in epochs depending on the workload dynamics in each epoch.

**FastCap compared with others policies.** We now compare FastCap against other power capping policies. *All policies are capable of controlling the power consumption to around the budget*, so we focus mostly on their performance implications. We first compare against policies that do *not* use memory DVFS.

“CPU-only” sets the core frequencies using the FastCap algorithm for every epoch, but keeps the memory frequency fixed at the maximum value. The comparison to CPU-only isolates the impact of being able also to manage memory subsystem power using DVFS. All prior power capping policies suffer from the lack of this additional capability.

“Freq-Par” is a control-theoretic policy from [47]. In Freq-Par, the core power is adjusted in every epoch based on a linear feedback control loop; each core receives a frequency allocation that is based on its power efficiency. Freq-Par uses a linear power-frequency model to correct the average core power from epoch to epoch. We again keep the memory frequency fixed at the maximum value.

Figure 4.6 shows the performance comparison between FastCap and these poli-

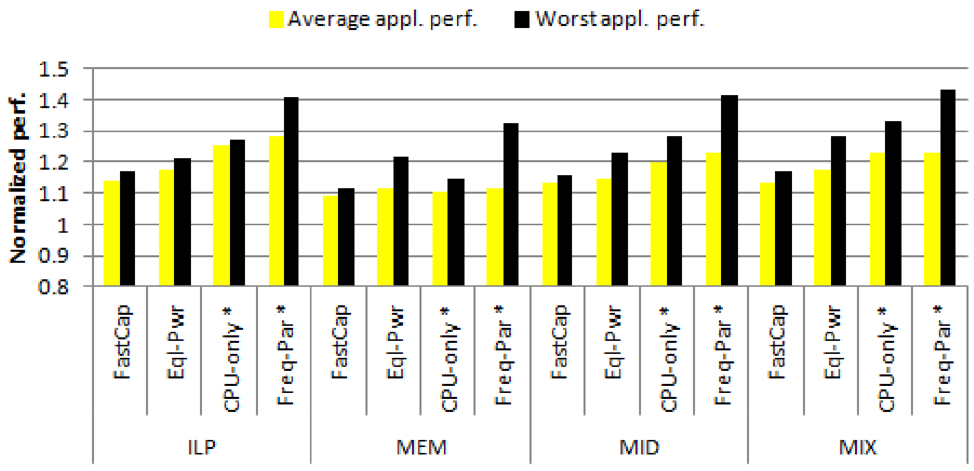


Figure 4.6: FastCap compared with CPU-only*, Freq-Par* and Eql-Pwr in normalized average/worst application performance. "*" indicates fixed memory frequency. Power budget = 60%.

cies on a 16-core system. FastCap performs at least as well as CPU-only in both average and worst application performance, showing that the ability to manage memory power is highly beneficial. Setting memory frequency at the maximum causes the cores to run slower for CPU-bound applications, in order to respect the power budget. This leads to severe performance degradation in some cases. For the MEM workloads, FastCap and CPU-only perform almost the same, as the memory subsystem can often be at its maximum frequency in FastCap to minimize performance loss within the power budget. Still, it is often beneficial to change the power balance between cores and memory, as workloads change phases. FastCap is the only policy that has the ability to do so.

The comparison against Freq-Par is more striking. FastCap (and CPU-only) performs substantially better than Freq-Par in both average and worst application performance. In fact, Freq-Par shows significant gaps between these types of performance, showing that it does not allocate power fairly across applications (inefficient cores receive less of the overall power budget). Moreover, Freq-Par's linear power-frequency model can be inaccurate and causes the feedback control to over-correct and under-correct often. This leads to severe power oscillation, although the long-term average is guaranteed by the control stability. For example, the power oscillates between 53% and 65% under Freq-Par for MIX3.

Next, we study policies that use DVFS for *both* cores and the memory subsystem. These policies are inspired by prior works, but we add FastCap's ability to manage memory power to them:

"Eql-Pwr" assigns an equal share of the overall power budget to all cores, as proposed in [58]. We implement it as a variant of FastCap: for each memory frequency, we compute the power share for each core by subtracting the memory power (and the background power) from the full-system power budget and dividing the result by $N$. Then, we set each core's frequency as high as possible without violating the per-core budget. For each epoch, we search through all $M$ memory frequencies, and use the solution that yields the best $D$ in equation (4.4).

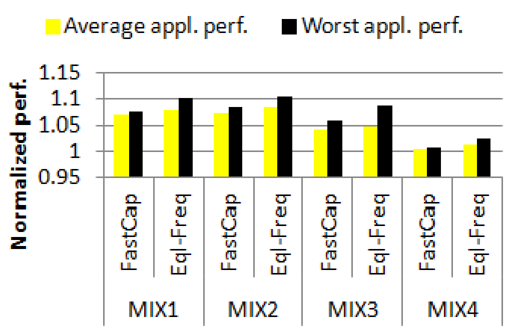
"Eql-Freq" assigns the same frequency to all cores, as proposed in [17]. Again, we implement it as a variant of FastCap: for each epoch, we search through all $M$ and $F$ frequencies to determine the pair that yields the highest $D$ in equation (4.4).

"MaxBIPS" was proposed in [27]. Its goal is to maximize the total number of executed instructions in each epoch, i.e. to maximize the throughput. To solve the optimization, [27] exhaustively searches through all core frequency settings. We implement this search to evaluate all possible combinations of *core and memory* frequencies within the power budget.

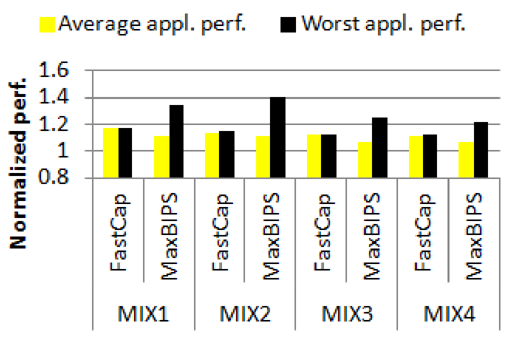
Eql-Pwr ignores the heterogeneity in the applications' power profiles. By splitting the core power budget equally, some applications receive too much budget and even running at the maximum frequency cannot fully consume it. Meanwhile, some power-hungry applications do not receive enough budget thus result in performance loss. This is most obvious in workloads with a mixture of CPU-bound and memory-bound applications (e.g., MIX4). As a result, we observe in Figure 4.6 that Eql-Pwr's worst application performance loss is often much higher than FastCap's.

Eql-Freq also ignores application heterogeneity. In Eql-Freq, having all core frequencies locked together means that some applications may be forced to run slowly, because raising all frequencies to the next level may violate the power budget. This is a more serious problem when the workload consists of a mixture of CPU- and memory-bound applications on a large number of cores. To see this, Figure 4.7a plots the normalized average and worst application performance for FastCap and Eql-Freq, when running the MIX workloads on a 64-core system. (We choose to show results on a 64-core system to magnify the comparison.) The figure shows that Eql-Freq is more conservative than FastCap and often cannot fully harvest the power budget to improve performance.

Finally, besides its use of exhaustive search, the main problem of MaxBIPS is that it completely disregards fairness across applications. Figure 4.7b compares the normalized average and worst application performance for the MIX workloads under a 60% budget. Because of the high overhead of MaxBIPS, the figure shows results for only 4-core systems. The figure shows that FastCap is slightly inferior in



(a) Normalized FastCap and Eql-Freq perf.



(b) Normalized FastCap and MaxBIPS perf.

Figure 4.7: FastCap compared with EqlFreq and maxBIPS.

average application performance, as MaxBIPS always seeks the highest possible instruction throughput. However, FastCap achieves significantly better worst application performance and fairness. To maximize the overall throughput, MaxBIPS may favor applications that are more power-efficient, i.e. have higher throughput at a low power cost. This reduces the power allocated to other applications and the outlier problem occurs. This is particularly true for workloads that consists of a mixture of CPU and memory-bound applications.

**Impact of number of cores.** We now study the impact of the number of cores on FastCap's ability to limit power draw (Figure 4.8) and to provide fair application performance (Figure 4.9).

Figure 4.8 depicts pairs of bars for each workload class on systems with 16, 32, and 64 cores, under a 60% power budget. The bar on the right of each pair is the maximum average power of any epoch of any application of the same class normalized to the peak power, whereas the bar on the left is the normalized average power *for the workload with the maximum average power.* Comparing these bars determines whether FastCap is capable of respecting the budget even when there are a few epochs with slightly higher average power. The figure clearly shows that FastCap is able to do so (all average power bars are at or slightly below 60%), even though increasing the number of cores does increase the maximum average power slightly. This effect is noticeable for workloads that have CPU-bound applications on 64 cores. In addition, note that the MEM workloads do not reach the maximum budget on 64 cores, as these workloads do not consume the power budget on this large system even when they run at maximum frequencies.

Figure 4.9 also shows pairs of bars for each workload class under the same assumptions. This time, the bar on the right of each pair is the normalized worst performance among all applications in a class, and the bar on the left is the normalized average performance of all applications in the class. The figure shows that FastCap is very successful at allocating power fairly across applications, regardless of the number of cores; the worst application performance is always only slightly worse than the average performance. The figure also shows that application performance losses decrease as we increase the number of cores, especially for MEM workloads. (Recall that we are comparing performance *losses* due to power capping; absolute performance cannot be compared because the baselines are different.) The reason is that MEM workloads on the larger numbers of cores are bottlenecked by the memory subsystem even when they execute at maximum frequencies.

**Epoch length and algorithm overhead.** By default, FastCap runs at the end of every OS time quantum (5 $ms$ in our experiments so far in the paper). The overhead of FastCap scales linearly with the number of cores. Specifically, we run the FastCap algorithm for 100$k$ times and collect the average time of each execution.

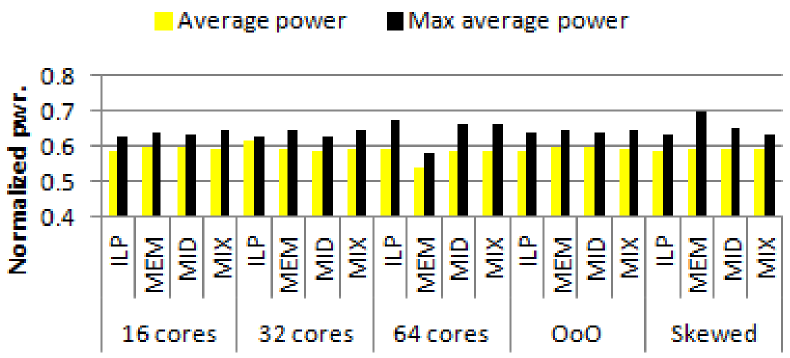


Figure 4.8: Normalized FastCap average power and maximum average power in many configurations. Power budget = 60%.

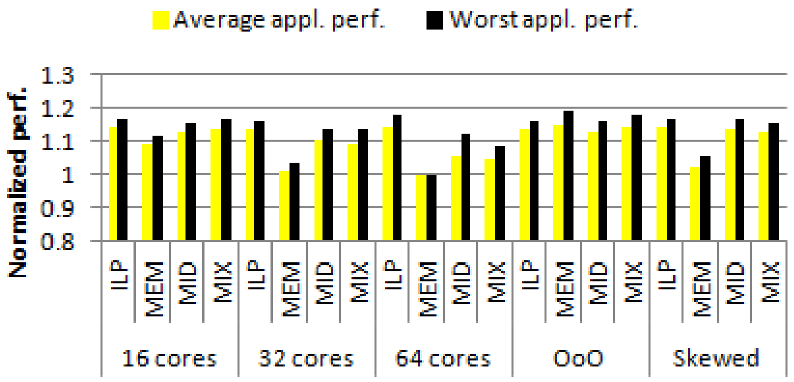


Figure 4.9: Normalized FastCap average and worst application performance in many configurations. Power budget = 60%.

The average time is 33.5 µs for 16 cores, 64.9 µs for 32 cores and 133.5 µs for 64 cores. For a 5 ms epoch length, these overheads are 0.7%, 1.3%, and 2.7% of the epoch lengths, respectively. If these levels of overhead are unacceptable, FastCap can execute at a coarser granularity. Using our simulator, we studied epoch lengths of 10 ms and 20 ms. We find that these epoch lengths do not affect FastCap's ability to control average power and performance for the applications and workloads we consider.

**Out-of-order (OoO) execution.** Our results so far have assumed in-order cores.

However, FastCap can be easily extended to handle the OoO executions. In OoO, the core may be able to continue executing the next instruction after issuing a memory access without stalling. In FastCap's terminology, the think time thus becomes the interval between two core stalls (not between two main memory accesses). The workload becomes more CPU-bound.

Unfortunately, our trace-based methodology does not allow detailed OoO modeling. However, we can approximate the latency hiding and additional memory pressure of OoO. Specifically, we simulate idealized OoO executions by assuming a large instruction window (128 entries) and disregarding instruction dependencies within the window. This models an upper-bound on the memory-level parallelism (and has no impact on instruction-level parallelism, since we still simulate a single-issue pipeline).

Figure 4.8 shows four pairs of bars for the OoO executions of the workload classes on 16 cores and under a 60% power budget. The results can be compared to the bars for 16 cores on the left side of the figure. This comparison shows that FastCap is equally successful at limiting the power draw to the budget, regardless of the processor execution mode.

Similarly, Figure 4.9 shows four pairs of bars for OoO executions on 16 cores, under a 60% budget. These performance loss results can also be compared to those for 16 cores on the left of this figure. The comparison shows that workloads with memory-bound applications tend to exhibit higher performance losses in OoO execution mode. The reason is that the performance of these applications improves significantly at maximum frequencies, as a result of OoO; both cores and memory become more highly utilized. When FastCap imposes a lower-than-peak budget, frequencies must be reduced and performance suffers more significantly. Directly comparing frequencies across the execution modes, we find that memory-bound workloads tend to exhibit higher core frequencies and lower memory frequency under OoO than under in-order execution. This result is not surprising since the memory can become slower in OoO without affecting performance because of the large instruction window. Most importantly, FastCap is still able to provide fairness in power allocation in OoO, as the performance losses are roughly evenly distributed across all applications.

**Multiple memory controllers.** So far we have assumed a single memory controller. In many-core systems, there may be multiple memory controllers, each handling a subset of the memory channels. For FastCap to support multiple memory controllers (operating at the same frequency), we use the existing performance counters to keep track of the average queue sizes $Q$ and $U$ of each memory controller. Thus, different memory controllers can have different response times (cf. equation (4.1)). We also keep track of the probability of each core's requests going through each memory controller. In this approach, the response time $R$ in equation (4.5) becomes a weighted average across all memory controllers and different cores experience different response times.

To study the impact of multiple memory controllers, we simulate four controllers in our 16-core system. In addition, we simulate two memory interleaving schemes: one in which the memory accesses are uniformly distributed across memory controllers, and one in which the distribution is highly skewed. In the uniformly distributed case, all memory controllers have roughly the same response time and all cores see the same response time. In the skewed distribution, some memory controllers are overloaded.

Figure 4.8 shows four pairs of bars for the skewed distribution on 16 cores, under a 60% budget. Compare these results to the 16-core data on the left side of the figure. The skewed distribution causes higher maximum power in the MEM workloads. Still, FastCap is able to keep the average performance for the workload with this maximum power slightly below the 60% budget.

Again, Figure 4.9 shows four pairs of the skewed distribution on 16 cores, under a 60% budget. We can compare these performance losses to the 16-core data on the left of the figure. The comparison shows that FastCap provides fair application performance even under multiple controllers with highly skewed access distributions.

Finally, FastCap uses the same frequency for all memory controllers. However, it may be more desirable to have different controllers running at different frequencies. This would introduce extra algorithmic complexity and we leave it as future work.

## 4.4 Conclusion

In this chapter we presented FastCap, an optimization framework and algorithm for system-wide power capping while promoting fairness across applications. FastCap uses both CPU and memory DVFS. The FastCap algorithm solves the optimization online and its complexity is the lowest among some of the state-of-the-arts. Our evaluation showed that FastCap caps power draw effectively, while producing better application performance and fairness than many sophisticated CPU power capping methods, even after they are extended to use memory DVFS as well.

# Chapter 5

# FastEnergy

## 5.1 Introduction

In Chapter 4 we introduced FastCap, a fast and effective mechanism to coordinate active low-power modes in multi-core systems for power capping. In this chapter we introduce FastEnergy, an optimization framework and search algorithm for energy conservation developed based on the queuing model discussed in Section 4.2.1. The goal for FastEnergy is to conserve as much energy as possible under a user-defined performance degradation bound.

Similar to the scenario in power capping, a lack of coordination in active low-power modes may leave the system unable to properly manage energy consumption while limiting performance degradation. In fact, Deng *et al.* [13] showed that the management of CPU and memory active low-power modes must be coordinated for stability and better energy savings. Since the core power modes affect the traffic seen by the memory subsystem and the memory power modes affect how fast cache misses are serviced, a lack of coordination may leave the system unable to properly manage energy consumption while limiting performance degradation.

In [13], the authors developed a optimization technique, CoScale, to manage the speed settings of multiple cores jointly with that of the memory subsystem to minimize energy consumption while meeting user-defined performance loss bounds. To do this, CoScale relies on execution profiling of core and memory access performance in a sequence of time epochs. In each epoch, CoScale uses performance counters and analytic models of core/memory performance and power to assess opportunities for per-core DVFS, DVFS of the on-chip MC, and DFS of memory channels and DRAM devices. CoScale explores the space of per-core and memory frequency settings (voltages are set as a function of the selected frequencies) in a greedy manner. It repeatedly computes the marginal energy and performance cost (or benefit) of altering each component's (or set of components') power mode by one step. CoScale then greedily selects the best next frequency combination and iterates until a (local) minimum is attained. CoScale is implemented in the OS, so an epoch typically corresponds to an OS time quantum (e.g., 5 ms).

Although CoScale is effective at conserving energy and meeting performance bounds, its search heuristic has a high time complexity of $O(M + FN^2)$, where $M$ is the number of memory frequencies, $F$ is the number of possible core frequencies, and $N$ is the number of cores. The *quadratic* dependence on the number of cores is problematic, as core counts may increase at the speed of Moore's law. In contrast, FastEnergy introduces an algorithm that is *linear* in the number of cores.

## 5.2 FastEnergy Design

### 5.2.1 Optimization and algorithm

Based on the queuing model (cf. Figure 4.2), we design FastEnergy to minimize the average energy required for all cores to complete one memory request each, which is

$$\sum_i P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} z_i + \left[P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_s\right] \frac{\sum_i (R(s_b) + z_i + c_i)}{N}, \tag{5.1}$$

subject to the per-core delay constraints:

$$R(s_b) + z_i + c_i \leqslant T_i \quad \forall\, i \in \mathcal{N}. \tag{5.2}$$

The parameter $T_i$ constrains the maximum allowable performance degradation, and specifies the performance requirement for each core in terms of the average time to complete one memory access (cf. Figure 4.2).

The optimization variables are $z_i$ and $s_b$, which must satisfy

$$\overline{s_b} \leqslant s_b, \quad \overline{z_i} \leqslant z_i \quad \forall\, i \in \mathcal{N}. \tag{5.3}$$

The first term in equation (5.1),

$$\sum_i P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} z_i$$

corresponds to the total energy consumed by the cores. The second term,

$$\left[P_m\left(\frac{\overline{s_b}}{s_b}\right)^{\beta}+P_s\right]\frac{\sum_i(R(s_b)+z_i+c_i)}{N}$$

corresponds to the energy consumed by the memory and all other power components that do not depend on CPU/memory frequencies. The second term is averaged over the number of cores $N$ because it is the background power seen by all the cores. Because of the term $P_m z_i(\overline{s_b}/s_b)^{\beta}$ the objective function (5.1) is non-convex in $z_i$ and $s_b$. For out-of-order cores, equation (5.1) can be interpreted as the average energy needed for all cores to complete one *blocking* memory request each.

To derive FastEnergy's algorithm, we make the following key observation in Theorem 5.1.

**Theorem 5.1.** *For each given bus transfer time $s_b$, the optimization problem (5.1)-(5.3) is conditionally convex in the $z_i$, and the optimal think time $z_i$ can be determined in linear time $O(N)$ and in closed-form. Specifically, each optimal $z_i$ can be one of only three values $z_i'$, $\widehat{z_i}$ and $\overline{z_i}$ (the minimum think time) where*

$$z_i' = T_i - c_i - R(s_b),$$
$$\widehat{z_i} = \left(\frac{(\alpha_i-1)NP_i\overline{z_i}^{\alpha_i}}{P_s+P_m(\overline{s_b}/s_b)^{\beta}}\right)^{\frac{1}{\alpha_i}}.$$

Intuitively, since the bus transfer time is experienced by all $N$ cores, it is natural first to determine its value and then, conditioned on the bus transfer time, determine the think time for each core. We defer the proof of Theorem 5.1 to the Appendix C

Motivated by Theorem 5.1, FastEnergy implements the following algorithm. For each bus transfer time $s_b$, it computes the think times $z_i$ for all cores in linear time. Then, it evaluates the objective function (5.1) using the computed $z_i$ and $s_b$. After exhausting all the $M$ possible values for $s_b$, it returns the outputs with the minimum objective value. The algorithm complexity is $O(NM)$, and we outline the pseudo-code in Algorithm 4.

The optimality of the Algorithm 4 is shown in Theorem 5.2.

**Theorem 5.2.** *Algorithm 4 computes the global optimal solution for the non-convex optimization problem (5.1)-(5.3).*

We defer the proof of Theorem 5.2 and the derivation of Algorithm 4 to the Appendix. Note that the optimization problem (5.1)-(5.3) itself is non-convex (it only becomes convex for a given bus transfer time). However, as the bus transfer time can only take $M$ discrete values, we are able to exhaustively search for the global optimal solution.

## 5.2.2 Implementation

**Operation**

We operate FastEnergy the same way as we operate FastCap in Section 4.2.3. FastEnergy splits time into fixed-size epochs of $L$ milliseconds each. It collects performance counters from each core 300 $\mu$s into each epoch, and uses them as inputs to the frequency selection algorithm. The optimization parameters, such as

**Algorithm 4** FastEnergy $O(NM)$ Algorithm

```
1: **Inputs**: $\{P_i\}$, $\{\alpha_i\}$, $P_m$, $\beta$, $P_s$, $\{\overline{z_i}\}$, $\overline{s_b}$, $Q$, $U$, $s_m$, $B$, $\overline{P}$ and an ordered array of $M$
   candidate values for $s_b$.
2: **Outputs**: $\{z_i\}$ and $s_b$
3: Initialize $s_b = \overline{s_b}$ and $z_i = \overline{z_i}$, for all $i \in \mathcal{N}$.
4: Compute performance constraint $T_i$, for all $i \in \mathcal{N}$.
5: **for** each $s_b$ in an increasing order in $\mathcal{M}$ **do**
6:   **for** each core $i \in \mathcal{N}$ **do**
7:     Compute $\widehat{z_i} = \left(\frac{(\alpha_i-1)NP_i\overline{z_i}^{\alpha_i}}{Ps+Pm(\overline{s_b}/s_b)^{\beta}}\right)^{\frac{1}{\alpha_i}}$.
8:     Compute $z_i' = T_i - c_i - R(s_b)$.
9:     **if** $\overline{z_i} > z_i'$ **then**
10:      **return** $s_b$ and $z_i$, for all $i \in \mathcal{N}$.
11:    **else if** $\widehat{z_i} < z_i'$ **and** $\widehat{z_i} > \overline{z_i}$ **then**
12:      $z_i = \widehat{z_i}$.
13:    **else if** $\overline{z_i} < z_i'$ **and** $\frac{(\alpha_i-1)P_i\overline{z_i}^{\alpha_i}}{z_i'^{\alpha_i}} - \frac{Ps+Pm(\overline{s_b}/s_b)^{\beta}}{N} > 0$ **then**
14:      $z_i = z_i'$.
15:    **else if** $\overline{z_i} < z_i'$ **and** $\frac{Ps+Pm(\overline{s_b}/s_b)^{\beta}}{N} - (\alpha_i - 1)P_i > 0$ **then**
16:      $z_i = \overline{z_i}$.
17:    **else**
18:      $z_i = \overline{z_i}$.
19:    **end if**
20:  **end for**
21:  Evaluate equation (5.1) and compare with the best objective value. Update if
     the best objective value is larger.
22: **end for**
23: **return** $s_b$ and $z_i$ for all $i \in \mathcal{N}$.
```

$P_i$, $\alpha_i$, $P_m$, $\beta$, the minimum think time $\overline{z_i}$, and queue sizes Q and U are computed the same way as in FastCap.

**Performance management** We now discuss how we set the performance constraint $T_i$. This constraint is recalculated at the beginning of each epoch. Similar to the approach proposed in [59], our policy is based on the notion of performance slack. The performance slack is the difference between the execution time of a (fast) baseline implementation and the execution time targeted by FastEnergy. The baseline implementation does not use energy management. Rather, it keeps the frequencies of all cores and of the memory at the maximum. The amount of slack controls the tradeoff between performance and energy consumption. In contrast to the prior works [59], we use the number of memory accesses to manage performance.

To understand how we do so, recall that $R(s_b) + z_i + c_i$ is the average time between two consecutive memory accesses by core $i$. Thus, $L/(R(s_b) + z_i + c_i)$ is the average number of accesses issued by core $i$ during an $L$ $ms$ epoch. Let $R(\overline{s_b})$ denote the average memory access time under the maximum memory frequency, i.e., $R(\overline{s_b}) = Q(s_m + U\overline{s_b})$ cf. (4.1), then $L/(R(\overline{s_b}) + \overline{z_i} + c_i)$ is the average number of accesses in the baseline system. FastEnergy's performance target is to complete a fraction of at least $1 - \gamma$ of the memory accesses in the baseline system, where $0 \leqslant \gamma < 1$. The parameter $\gamma$ can be interpreted as the maximum allowed fractional performance slowdown/degradation. When $\gamma > 0$, each epoch adds some amount of performance slack.

To set the performance constraints (and compute $T_i$), at the beginning of every epoch and for every core, FastEnergy computes

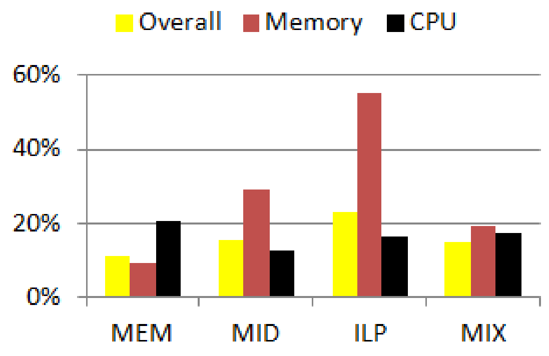
$$(1-\gamma)\left(\frac{L}{R(\overline{s_b})+\overline{z_i}+c_i}\right)+X_i, \tag{5.4}$$

where the value $X_i$ measures the difference between the target performance and the actual FastEnergy execution in number of memory accesses *so far*. A positive $X_i$ means FastEnergy is falling behind, and thus needs to perform more memory accesses in the next epoch. A negative $X_i$ means FastEnergy is running faster than the target, and the execution can be slowed down if doing so would save energy. Equation (5.4) provides the desired bound for $L/(R(s_b)+z_i+c_i)$. Rearranging the terms, we arrive at our performance constraint $T_i$:

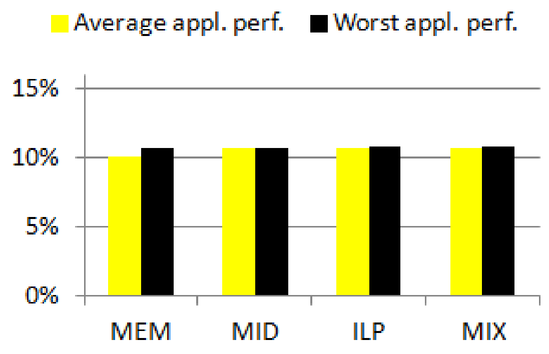
$$R(s_b)+z_i+c_i \leqslant \frac{L}{(1-\gamma)\left(\frac{L}{R(\overline{s_b})+\overline{z_i}+c_i}\right)+X_i} =: T_i \quad \forall i. \tag{5.5}$$

It is possible that equation (5.5) cannot be satisfied, meaning that one or more cores cannot catch up in a single epoch. In such a situation, FastEnergy runs those cores at the maximum frequency. With the slack added in the next epochs, the equation will eventually be satisfied again.

FastEnergy requires the same features and has the same costs as FastCap. Specifically, FastEnergy requires no architectural or software support beyond that in [13].



(a) Energy savings in percentage



(b) Performance degradation, maximum allowed: 10%.

Figure 5.1: FastEnergy energy and performance on $N = 16$ cores.

## 5.3 Evaluation

To evaluate FastEnergy, we use the same evaluation methodology as for FastCap in Section 4.3. We now summarize some key results.

**FastEnergy energy and performance.** In Figure 5.1a, we plot the FastEnergy energy savings (full-system, CPU and memory) as a percentage of the baseline execution of our workloads. Figure 5.1b shows the performance degradation compared to the baseline, where the maximum allowed degradation is $\gamma = 0.1$ (cf. (5.4)), i.e. FastEnergy is allowed to slow applications down by at most 10% compared to the baseline.

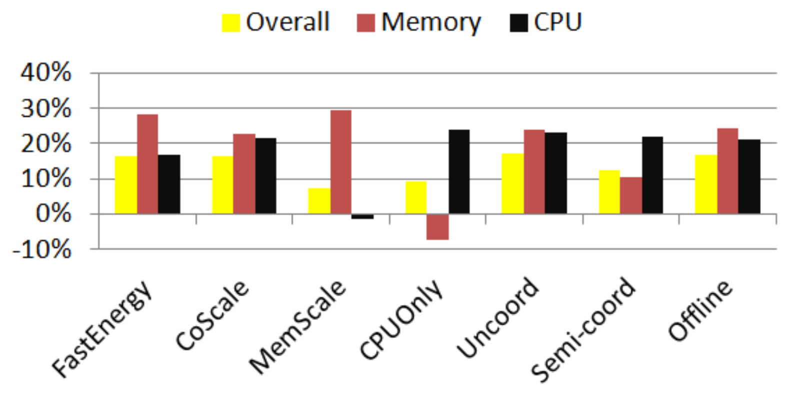
As one would expect, Figure 5.1a shows that FastEnergy obtains more energy savings from the memory subsystem in CPU-intensive workloads (long think times $z_i$), and more savings from the cores in memory-intensive workloads (short think times $z_i$). The full-system energy savings averaged across all the workloads is 16%.

In Figure 5.1b, we note that FastEnergy maintains the maximum performance degradation of any application in a workload very close to 10%, the maximum allowed slowdown. For some workloads, both the average and worst case degradations are lower than 10%. The reason is that, for those workloads, running slower would have *increased* energy consumption (the small savings from the lower dynamic energy would have been outweighed by the static energy).

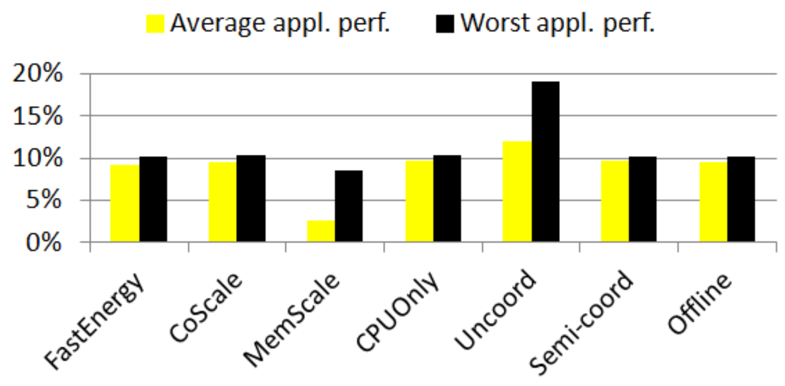
**FastEnergy compared to other methods.** In Figure 5.2, we compare FastEnergy with other energy management methods, again for $N = 16$ cores with 10% maximum allowed performance degradation ($\gamma = 0.1$). The energy savings in the figure are the average savings over all workloads.

We have already described CoScale. MemScale represents the scenario in which the system uses only memory subsystem DVFS. CPUOnly represents the scenario with CPU DVFS only. To be optimistic about this alternative, we assume that it considers all possible combinations of core frequencies and selects the best. In both MemScale and CPUOnly, the performance-aware energy management policy assumes that the behavior of the components that are not being managed will stay the same in the next epoch as in the profiling phase. Uncoordinated (or Uncoord for short) applies both MemScale and CPU DVFS, but in a completely independent fashion. In determining the performance slack available to it, the CPU power manager assumes that the memory subsystem will remain at the same frequency as in the previous epoch, and that it has accumulated no performance degradation; the memory power manager makes the same assumptions about the cores. Hence, each manager believes that it alone influences the slack in each epoch, which is not the case. Semi-coordinated (or simply Semi-coord) increases the level of coordination slightly by allowing the CPU and memory power managers to share the same overall slack, i.e. each manager is aware of the past performance degradation produced by the other. However, each manager still tries to consume the entire slack independently in each epoch (i.e., the two managers account for one another's past actions, but do not coordinate their estimate of future performance). Finally, Offline relies on a perfect offline performance trace for every epoch, and then selects the best frequency for each epoch by considering all possible core and memory frequency settings. As the number of possible settings is exponential, Offline is impractical and is studied simply as an upper bound on how well FastEnergy can do. However, Offline is not necessarily optimal, since it uses the same epoch-by-epoch greedy decision-making as CoScale (i.e., a hypothetical oracle might choose to accumulate slack in order to spend it in later epochs).

We observe that FastEnergy's energy savings and performance degradation are comparable to those of CoScale. The MemScale method only saves memory subsystem energy, as it does not optimize the cores' frequencies. CPU-only does the opposite, only saving energy in the CPU without adapting memory's frequency. Although Uncoordinated can save substantial energy, it is unable to keep performance within the maximum allowed degradation. In some cases, the performance degradation reaches 19%, nearly twice the maximum allowed. Semi-coordinated keeps the worst performance degradation within the bound, because both the CPU and memory frequency managers share the same performance estimate. However, because of frequent oscillations and settling at sub-optimal solutions, Semi-coordinated



(a) Energy savings in percentage

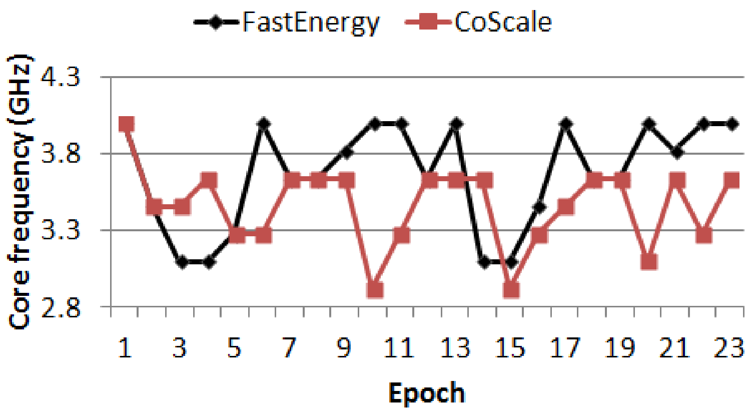


(b) Performance degradation, maximum allowed: 10%

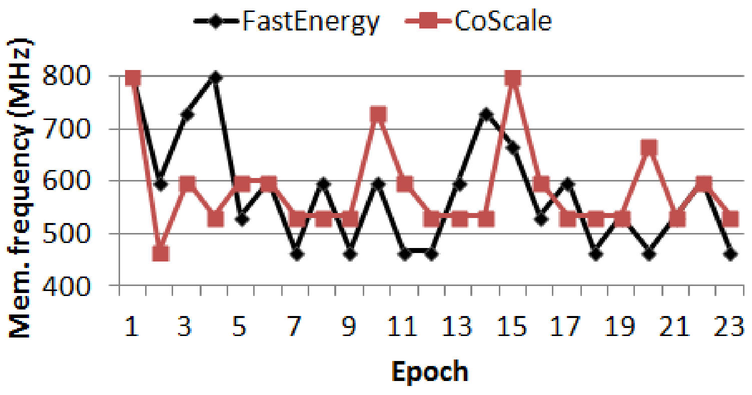
Figure 5.2: FastEnergy compared with other methods, N = 16.

consumes more energy than FastEnergy. The comparison to Offline shows that both FastEnergy and CoScale behave extremely well in terms of energy savings and performance.

**Dynamic behavior.** In Figure 5.3, we compare the optimal frequencies selected by CoScale and FastEnergy. We plot the frequencies of the core running application applu in MIX2 and the frequencies of the memory over time.



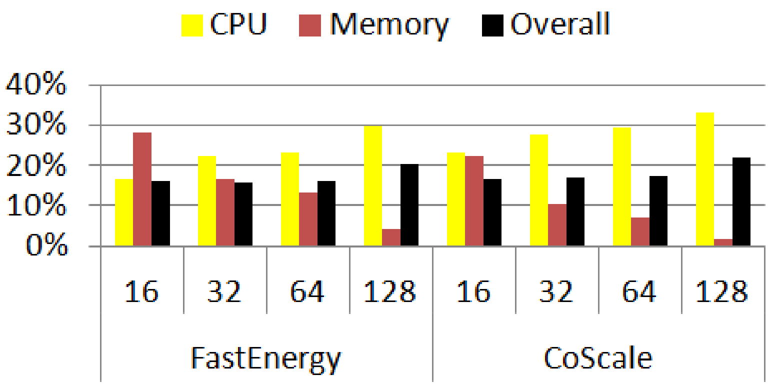
(a) Core frequencies



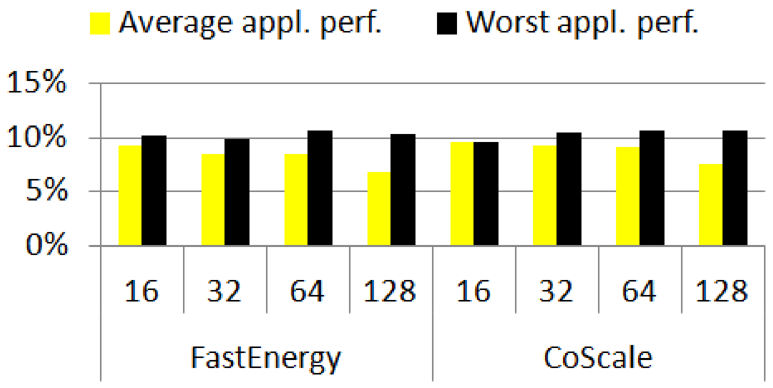
(b) Memory frequencies

Figure 5.3: Frequencies selected while running MIX2. The core frequency plots the core that runs application applu.

We note that FastEnergy selects different frequencies than CoScale for both the memory and the core. However, the full-system energy savings for FastEnergy running MIX2 is 14.7%, almost the same as the 14.6% for CoScale. This is because, although FastEnergy computes the *global* optimal solution for the optimization (5.1)-(5.2), the optimization formulation itself is only an approximation of the real



(a) Energy savings in percentage



(b) Performance degradation, maximum allowed: 10%

Figure 5.4: FastEnergy and CoScale with N = 16, 32, 64 and 128.

system. Algorithms that do not leverage FastEnergy's optimization framework may settle on a different solutions with comparable energy savings and performance results.

**Impact of the number of cores.** In Figure 5.4, we study the (average) energy savings and (average and worst-case) performance degradations of FastEnergy and CoScale running on N = 16, 32, 64, and 128 cores. Our results show that both FastEnergy and CoScale are able to maintain the worst-case performance almost exactly within the 10% bound. Their average energy savings are also very similar for all numbers of cores. Interestingly, we note that, as the number of cores increases, the energy savings from the memory subsystem decrease while the savings from cores increase. This is because, for large $N$, the memory is almost always busy (i.e., large $R$), leaving little opportunity for the memory to slow down.

**Algorithm overhead.** In Figure 5.5, we plot the algorithm overhead (i.e., how long it takes to search for the next frequency configuration) in $\mu s$ for FastEnergy and CoScale, as a function of $N$. Since FastEnergy's complexity is $O(MN)$, it scales linearly with the number of cores, whereas CoScale scales quadratically ($O(M + FN^2)$, where $F$ is the number of core frequencies). In the figure, we also include FastEnergy-H, a heuristic algorithm to further reduce the complexity of FastEnergy to $O(N \log M)$. FastEnergy-H is based on the observation that the memory frequencies can be traversed via a binary search for a near-optimal solution to the optimization problem (5.1)-(5.3).

As we simulate 5 $ms$ epochs in these results, the overhead of CoScale is more than 10% with 256 cores, which is clearly unacceptable. To amortize this overhead, we would need substantially longer epochs, which may encompass multiple workload behaviors and render the information collected during profiling (and the predictions made based on it) obsolete. In contrast, FastEnergy achieves the same energy and performance results as CoScale at a much lower overhead. FastEnergy-H performs even better than FastEnergy with almost the same energy and performance. Since $M = 10$ in our experiments, it is roughly $3\times$ faster across the spectrum. Compared

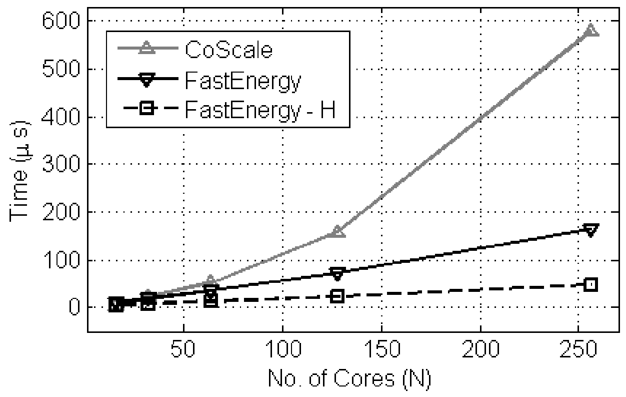
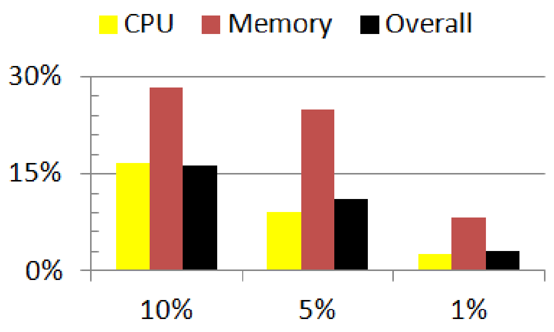


Figure 5.5: Algorithm overhead, as a function of core counts N.

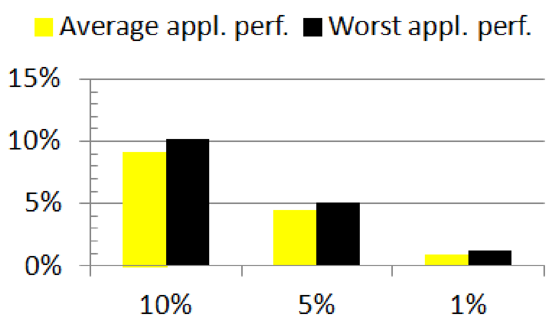
to CoScale, FastEnergy-H is roughly 10× faster. In terms of energy and performance, FastEnergy-H's average full system energy savings is 16.1% (vs. 16% for FastEnergy), and average performance degradation is 8.9% (vs. 9.2% for FastEnergy) for our workloads on 16 cores. (Since FastEnergy-H and FastEnergy only differ significantly in terms of algorithm overhead, we only present FastEnergy results in our other figures.) *These results demonstrate that FastEnergy and FastEnergy-H are practical for many-core systems, whereas CoScale is not.*

**Impact of the epoch length.** We also study the energy savings and performance degradation under different epoch lengths ($L = 5, 10, 15$, and $20\ ms$) on 64 cores. The longer the epochs, the less frequently the frequency selection algorithm executes. Running the algorithm less frequently may lead to worse performance for applications that have fast-changing dynamics.

Compared to CoScale, FastEnergy is slightly better at limiting the maximum (and average) performance degradations as the epoch length increases. For example, for $L = 20\ ms$, the max performance degradation is 10.9% for FastEnergy vs. 12.1%



(a) Energy savings in percentage



(b) Performance degradation

Figure 5.6: FastEnergy for different performance bounds, $N = 16$.

for CoScale. This is because the queuing and optimization framework used in FastEnergy model the average behavior of the system, making it less vulnerable to fast-changing workload dynamics. However, FastEnergy's average energy savings are slightly lower (16%) than those of CoScale (17.2%) for $L = 20$ $ms$.

**Impact of maximum allowed performance loss.** Thus far, we have assumed a performance bound of 10% ($\gamma = 0.1$). Figure 5.6 plots the average FastEnergy results for different performance bounds (10%, 5% and 1%) on 16 cores. FastEnergy adapts to different bounds while conserving energy. For example, with a bound of just 1%, FastEnergy conserves as much as 3.1% energy on average.

## 5.4 Conclusions

In this chapter we introduce FastEnergy, an optimization framework and search algorithm for energy conservation developed based on the queuing model discussed in Section 4.2.1. The goal for FastEnergy is to conserve as much energy as possible under a user-defined performance degradation bound. FastEnergy manages the frequencies of the cores and the memory subsystem in a coordinated manner to minimize the full-system energy under pre-defined performance constraints. Similar to FastCap, FastEnergy consists of a queuing model, an optimization framework, and an algorithm that is linear on the number of cores. As such, FastEnergy scales substantially better than the prior techniques.

# Chapter 6

# Related work

In this chapter we summarize related works. We start with a collection of works that focus on different low-power settings in CPU and memory. Then we discuss the works on coordinated power capping and energy conservation.

**CPU active low-power modes.** CPU active low-power modes such as DVFS have been widely used for computer systems to provide power savings by varying voltage and frequency [17, 18, 19]. It is most effective when CPU utilization is high as it provides substantial power savings, in exchange for a moderate loss in performance. Many researchers have studied the problem of finding the optimal DVFS settings. In [22] the problem of finding the optimal low-power mode was formulated as a stochastic dynamic problem and a numerical solution was derived. Considering power and job scheduling explicitly, Andrew *et al.* [60] studied speed scaling policies that minimize a weighted sum of the response time and energy use per job. For a soft real-time system, Bansal *et al.* [23] studied minimizing energy via speed scaling without violating task deadlines. Later, Wierman *et al.* [61] showed that a dynamic policy that adapts the server speed in proportion to queue length is robust to bursty traffic and mis-predictions of workload parameters.

**CPU idle low-power states.** CPU idle low-power states are most effective when CPU experiences long period of idleness. In [3] the authors proposed a method for eliminating idle power in servers by quickly transitioning between a high-performance active mode and a single idle low-power state. The development was based on queuing theory [35]. Recent advances in the $M/G/k$ queue with setup costs [62] extended the case to the multi-server scenario: the impact of data center size was studied in [39] and power allocation in server farms was studied in [63]. In a slightly different vein [64] took a stochastic optimization approach, optimizing time average system performance using optimization theory.

However, due to high wake-up penalty, many researchers have suggested using idle low-power states with extreme care. In [8] the authors warned of potential problems of using these low-power states and suggested "guarded" mechanisms to avoid negative power savings. In fact, most of existing works only limit to the usage of a single idle low-power state (e.g. [3, 39]).

**Memory active low-power modes.** Past work [24, 25, 26] has shown that active low-power modes are particularly useful for server workloads when the memory bus is underutilized for long periods. This provides ample opportunities for memory power management. To harness such opportunities, Deng *et al.* proposed MemScale, a technique that employs DVFS of the memory controller (MC), and dynamic frequency scaling (DFS) of the memory channels and DRAM devices [25]. David *et al.* [12] also studied memory power management via DVFS. In another related work [65], dynamic voltage scaling is applied to DRAM chips, which we do not consider.

**Memory idle low-power states.** The use of different idle low-power states for memory has been extensively studied, e.g. [20, 21, 66, 59, 67]. However, it has been shown [34] that in modern memory technology, idle low-power states are rarely effective as it requires a large chunk of data not being accessed for a long period of time – which is very unlikely in multi-core servers.

**Coordinated power capping.** Many prior works have proposed using a global controller to coordinate cores' DVFS subject to a CPU-wide power budget, e.g. [28, 27, 58]. Most of them formulate it as optimization problems. Isci *et al.* [27] used exhaustive search over pre-computed power and performance information about all possible power mode combinations. Their algorithm's time and storage space complexities grow exponentially with the number of cores. Teodorescu *et al.* [48] developed a linear programming method to find the best DVFS settings under power constraints. However, they assumed power is linearly dependent on the core frequency, which is often a poor approximation. Meng *et al.* [68] developed a greedy algorithm that starts with maximum speeds for all cores and repeatedly selects the neighboring lower global power mode with the best $\Delta_{power}/\Delta_{perf}$ ratio. The algorithm may traverse the entire space of power mode combinations. Winter *et al.* [69] improved this algorithm using a max-heap data structure and reduced the complexity to $O(FN \log N)$, where $F$ is the number of core frequencies and $N$ is the number of cores. They also developed a heuristic that runs in only $O(N \log N)$ time. Bergamaschi *et al.* [70] formulated a non-linear optimization and solved it via the interior-point method. The method usually takes many steps to converge and its average complexity is a high polynomial in the number of cores.

There are also past works which use control-theoretic approaches (e.g. [71, 47, 72]). Mishra *et al.* [71] studied power management in multi-core CPUs with voltage islands. They assumed that the power-frequency model (power consumption as a function of the cores' frequencies) is fixed for all islands, which may be inaccurate under changing workload dynamics. Ma *et al.* [47] used a method that stabilizes the power consumption by adjusting a frequency quota for all cores. In a similar vein, Chen *et al.* [72] used a control-theoretic method along with (idle) memory power management via rank activation/deactivation. Unfortunately, rank activation/deactivation is too slow for many applications [73]. Moreover, [47, 72] require a linear power-frequency model, which may cause under- and over-correction in the feedback control due to poor accuracy. This may lead to large power fluctuations, though the long-term average power is guaranteed to be under the budget.

Finally, Shen *et al.* [74] recently considered power capping in servers running interactive applications. They used model-based, per-request power accounting and CPU throttling (not DVFS) for requests that exceed their fair-share power allocation. There are also existing works that use machine learning [75, 76] methods. However extensive training and profiling are required for these type of approaches.

**Coordinated energy conservation.** CPU cores and the memory dominate the energy consumption of modern servers [6, 77, 26, 78] and in most earlier works, cores and memory are managed separately. The implicit assumption was that cores and memory behave independently of management actions. A few other works have considered coordinating CPU and memory power management for energy conservation subject to a performance bound. In [79, 66] the authors considered a single-core system with memory idle low-power states. However as discussed earlier, memory idle low-power states are rarely useful. Deng *et al.* [13] proposed CoScale which operates active low-power modes in both CPU and memory. In CoScale, the time complexity in searching the optimal DVFS setting remains high.

# Chapter 7

# Conclusion and Future Work

In this thesis, we state the problem of optimizing low-power settings in computer systems and developing efficient methods to select the optimal configurations at runtime. We suggest managing the settings in a coordinated manner to improve performance and efficiency. Based on novel queuing models and detailed performance analyses, we devise effective and low-complexity algorithms for power capping and energy/power conservation. Our results demonstrate the importance of coordinating low-power settings and the effectiveness of using queuing models for finding the optimal configuration.

SleepScale manages both the active low-power modes and idle low-power states in processors. We build a queuing simulator to model job executions in different DVFS and sleep state settings. Through extensive simulation, we verify that different DVFS choices should be used together with different sleep states. The optimal choice of the low-power setting depends on workload and hardware characteristics, and should be adjusted at runtime in response to workload changes. Our results show that SleepScale offers better power savings than some conventional methods while respecting the same QoS constraints.

To develop a method for optimizing low-power settings across system components, we develop FastCap, an optimization approach for system-wide power capping using both processor and memory active low-power modes. We first design a queuing model that effectively captures the workload dynamics in a many-core system. Based on the queuing model, we formulate an optimization problem and develop a fast and efficient algorithm. Our results show that FastCap maintains the overall system power under the budget while maximizing the performance of each application. Our results also show that FastCap produces better performance and fairness among applications than many state-of-the-art policies.

We also develop FastEnergy, an optimization framework and search algorithm for energy conservation by extending the queuing model designed for FastCap. Based on the queuing model, we formulate an optimization framework for minimizing the full-system energy under performance constraints. We derive a fast and efficient algorithm for solving the optimization problem. Our results show that FastEnergy produces significant energy savings across a range of core counts. In addition, our results show that FastEnergy consistently scales to larger core counts than the state-of-the-art designs.

## 7.1 Future Work

As the number of low-power settings continues to grow in modern computer systems, managing these settings correctly and effectively remains a challenging task. Looking forward, we propose the following future research directions.

**On algorithm overhead.** In SleepScale one needs to simulate all possible active modes and sleep states to identify an appropriate combination. Reducing the amount of simulations would be a significant improvement. Ideally, based on the simulation results of one setting, the policy manager can determine the next best setting to simulate thus avoid testing all the combinations. In FastCap and FastEnergy, although the algorithms have the lowest complexity among the state-of-the-arts, the benefits of FastEnergy-H (c.f. Figure 5.5) suggest that some heuristics could further reduce the complexity without sacrificing on performance and efficiency.

**On multi-server systems.** In this thesis, we use high-level queuing models and simulations to develop effective power control strategies for individual computing server. It would be interesting to extend SleepScale, FastCap and FastEnergy to consider server-level parallelisms. It is worth investigating how low-power settings can be optimized across servers with low overhead and whether centralized or distributed algorithms should be used.

**On unified optimization.** In this thesis, we consider the joint operation of active low-power modes and idle low-power states for processors in SleepScale, as well as active low-power modes in processors and memory in FastCap and FastEnergy. Besides CPU and memory, other system components such as disks also use low-power settings. Developing a unified and scalable framework for optimizing all low-power settings in a tandem still remains a challenging task.

# Appendix A

## Derivation of closed-form $\mathbb{E}[R]$ and $\mathbb{E}[P]$ in Section 3.3.3

The average power consumption for the single-server system with $n$ low-power states described in Section 3.3 is

$$\mathbb{E}[P] = \frac{1}{\lambda L}\left[\sum_{i=1}^{n-1} P_i(e^{-\lambda\tau_i} - e^{-\lambda\tau_{i+1}}) + P_n e^{-\lambda\tau_n}\right] + P_0\left(1 - \frac{e^{-\lambda\tau_1}}{\lambda L}\right)$$

where $L$ is defined as

$$L = \frac{\mu f + \mu f\lambda\left[\sum_{i=1}^{n-1} w_i(e^{-\lambda\tau_i} - e^{-\lambda\tau_{i+1}}) + w_n e^{-\lambda\tau_n}\right]}{\lambda(\mu f - \lambda)}.$$

This can be proved using busy period analysis and first principles, see [35]. The mean response time $\mathbb{E}[R]$ is

$$\mathbb{E}[R] = \frac{1}{\mu f - \lambda} + \frac{2\mathbb{E}[D] + \lambda\mathbb{E}[D^2]}{2(1 + \lambda\mathbb{E}[D])},$$

where

$$\mathbb{E}[D^{\alpha}] = \sum_{i=1}^{n-1} w_i^{\alpha}(e^{-\lambda\tau_i} - e^{-\lambda\tau_{i+1}}) + w_n^{\alpha}e^{-\lambda\tau_n}.$$

This can be derived using the result from [80] and some algebraic manipulations. Both $\mathbb{E}[R]$ and $\mathbb{E}[P]$ can be extended to the case where service time is not exponential.

The probability the response time $R$ exceeds deadline $d$ is

$$\Pr(R \geqslant d) = \frac{e^{-(\mu f - \lambda)d} - w_1(\mu f - \lambda)e^{-d/w_1}}{1 - w_1(\mu f - \lambda)}.$$

Note that when $d = 0$ the $\Pr(R \geqslant d) = 1$. When $d = \infty$ the $\Pr(R \geqslant d) = 0$. When $w_1 = 0$ the $\Pr(R \geqslant d) = e^{-(\mu f - \lambda)d}$. When $w_1 = \infty$ the $\Pr(R \geqslant d) = 1$. This result can be proved using a Laplace transform analysis of the response time $R$.

# Appendix B

## Proof of Theorem 4.1

We present the proof of Theorem 4.1 in Chapter 4.

*Proof.* We first show that constraint 4.6 must be an equality. Suppose otherwise, then we can always reduce the optimal bus speed $s_b^*$ such that the performance of each core is improved (because of the decrease in $R(s_b^*)$). As a result, we can achieve a better objective, larger than $D^*$. This leads to a contradiction. Thus, the power budget constraint must be an equality.

Now, we show that constraint 4.5 must also be an equality. Suppose otherwise, i.e. there exists a $j$ such that constraint 4.5 is strictly smaller than $1/D^*$. Then, we can increase $z_j^*$. The power budget saved from this core can be redistributed to other cores that have equalities in constraint 4.5. As a result, we can achieve an objective that is larger than $D^*$. This leads to a contradiction as well. $\square$

# Appendix C

## Proof of Theorem 5.1 and Theorem 5.2

**Proof of Theorem 5.1.** With $s_b$ fixed, the optimization problem reduces to minimizing

$$\sum_i \left( \frac{P_i (\overline{z_i})^{\alpha_i}}{z_i^{\alpha_i - 1}} \right) + \frac{1}{N} \sum_i \left[ P_s + P_m \left( \frac{\overline{s_b}}{s_b} \right)^{\beta} \right] z_i, \tag{C.1}$$

subject to

$$QUs_b + z_i \leqslant T_i' \quad \forall\, i \in \mathcal{N}, \tag{C.2}$$

where $T_i' = T_i - Qs_m - c_i$, and we have substituted $R(s_b)$ by equation (4.1). The variable constraints are:

$$\overline{s_b} \leqslant s_b, \quad \overline{z_i} \leqslant z_i, \quad \forall\, i \in \mathcal{N}. \tag{C.3}$$

Since each additive term in (C.1) is convex in $z_i$, (C.1) itself is convex in $z_i$. Also since constraints (C.2) and (C.3) are linear in $z_i$ (and thus convex), the optimization problem is convex in $z_i$ with each fixed $s_b$.

To solve for the optimal $z_i$, we rely on the Karush-Kuhn-Tucker (KKT) conditions that the optimal $z_i$ must satisfy. Since the optimization problem (C.1) under constraints (C.2)-(C.3) is convex, the $z_i$ satisfying the KKT conditions are the global optimal solution. The KKT conditions can be written as

$$\frac{P_s + P_m(\overline{s_b}/s_b)^{\beta}}{N} + \lambda_i - \gamma_i = \frac{(\alpha_i - 1)P_i(\overline{z_i})^{\alpha_i}}{z_i^{\alpha_i}} \quad \forall\, i \in \mathcal{N} \tag{C.4}$$

$$\lambda_i(T_i' - z_i - QUs_b) = 0 \quad \forall\, i \in \mathcal{N} \tag{C.5}$$

$$\gamma_i(z_i - \overline{z_i}) = 0 \quad \forall\, i \in \mathcal{N} \tag{C.6}$$

$$z_i + QUs_b \leqslant T_i' \quad \forall\, i \in \mathcal{N} \tag{C.7}$$

$$\lambda_i, \gamma_i \geqslant 0 \quad z_i \geqslant \overline{z_i} \quad \forall\, i \in \mathcal{N} \tag{C.8}$$

Based on the above KKT conditions, we make the following observations:

1) If $\lambda_i = \gamma_i = 0$, then $z_i = \widehat{z_i}$ provided that $\widehat{z_i}$ satisfies (C.7) and (C.8). This corresponds to line 11-12 in Algorithm 4.

2) If $\lambda_i > 0$ and $\gamma_i = 0$, then $z_i = T_i' - QUs_b = T_i - c_i - R(s_b)$, provided that this value satisfies (C.4) and (C.8). This corresponds to line 13-14 in Algorithm 4.

3) If $\lambda_i = 0$ and $\gamma_i > 0$, then $z_i = \overline{z_i}$, provided that $\overline{z_i}$ satisfies (C.4) and (C.7). This corresponds to line 15-16 in Algorithm 4.

4) If $\lambda_i > 0$ and $\gamma_i > 0$, then $z_i = \overline{z_i} = T_i - c_i - R(s_b)$. This corresponds to line 18 in Algorithm 4.

5) Finally, if $\overline{z_i} > T_i' - QUs_b$, then there is no solution since the optimization is infeasible. This corresponds to line 9-10 in Algorithm 4.

As a result, the optimal $z_i$ can only take one of the three values: $\overline{z_i}$, $z_i'$ and $\widehat{z_i}$ defined in Theorem 5.1. This completes the proof.

**Proof of Theorem 5.2** Note that Theorem 5.1 computes the global optimal $z_i$ for a given $s_b$. If we exhaust all the possible $s_b$ choices and compare the objective value, then we find a global optimal solution to the original optimization problem (5.1)-(5.3). This completes the proof of Theorem 5.2.

# Bibliography

[1] Intel, “Intel Xeon processor E5-1600/E5-2600/E5-4600 product families,” May 2012. [Online]. Available: http://tinyurl.com/d7ma5nf

[2] HP, Intel, Microsoft, Phoenix and Toshiba, “The ACPI specification,” Dec. 2011. [Online]. Available: http://www.acpi.info/

[3] D. Meisner, B. T. Gold, and T. F. Wenisch, “PowerNap: eliminating server idle power,” *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 205–216, Mar. 2009.

[4] M. Guevara, B. Lubin, and B. C. Lee, “Navigating heterogeneous processors with market mechanisms,” *IEEE International Symposium On High Performance Computer Architecture*, pp. 95–106, Feb. 2013.

[5] D. Meisner, J. Wu, and T. F. Wenisch, “BigHouse: a simulation infrastructure for data center systems,” *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, pp. 35–45, 2012.

[6] L. A. Barroso, J. Clidaras, and U. Holzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Morgan Claypool, 2013.

[7] D. Wong and M. Annavaram, "KnightShift: scaling the energy proportionality wall through server-level heterogeneity," *IEEE/ACM International Symposium on Microarchitecture*, pp. 119–130, 2012.

[8] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram, "A case for guarded power gating for multi-core processors," *IEEE International Symposium On High Performance Computer Architecture*, pp. 291–300, Feb. 2011.

[9] P. Thibodeau, "Data centers are the new polluters," 2014. [Online]. Available: http://tinyurl.com/ov7sx5b

[10] L. A. Barroso and U. Hölzle, "The Case for Energy-Proportional Computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, Dec. 2007.

[11] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM Computer Communication Review*, vol. 39, pp. 68–73, Dec. 2008.

[12] H. David, C. Fallin, E. Gorbatov, U. Hanebutte, and O. Mutlu, "Memory Power Management via Dynamic Voltage/Frequency Scaling," *Proceedings of the International Conference on Autonomic Computing*, pp. 31–40, 2011.

[13] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, “CoScale: Coordinating CPU and Memory DVFS in Server Systems,” *IEEE/ACM International Symposium on Microarchitecture*, pp. 143–154, 2012.

[14] E. V. Carrera, E. Pinheiro, and R. Bianchini, “Conserving Disk Energy in Network Servers,” *ACM International Conference on Supercomputing*, pp. 86–97, June 2003.

[15] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke, “DRPM: Dynamic Speed Control for Power Management in Server Class Disks,” *ACM/IEEE International Symposium on Computer Architecture*, pp. 169–181, June 2003.

[16] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu, “Energy Proportional Datacenter Networks,” *ACM/IEEE International Symposium on Computer Architecture*, pp. 338–347, June 2010.

[17] S. Herbert and D. Marculescu, “Analysis of Dynamic Voltage/Frequency Scaling in Chip-Multiprocessors,” *International Symposium on Low Power Electronics and Design*, pp. 38–43, 2007.

[18] S. Kaxiras and M. Martonosi, “Computer Architecture Techniques for Power-Efficiency,” *Synthesis Lectures on Computer Architecture*, 2009.

[19] D. C. Snowdon, S. Ruocco, and G. Heiser, “Power management and dynamic voltage scaling: Myths and facts,” *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, Sep. 2005.

[20] B. Diniz, D. Guedes, W. M. Jr, and R. Bianchini, “Limiting the Power Consumption of Main Memory,” *ACM/IEEE International Symposium on Computer Architecture*, pp. 290–301, 2007.

[21] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, “Power Aware Page Allocation,” *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 105–116, 2000.

[22] J. M. George and J. M. Harrison, “Dynamic control of a queue with adjustable service rate,” *Operation Research*, vol. 49, no. 5, pp. 720–731, 2001.

[23] N. Bansal, T. Kimbrel, and K. Pruhs, “Speed Scaling to Manage Energy and Temperature,” *Journal of the ACM*, vol. 54, no. 1, 2007.

[24] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, “MultiScale: Memory System DVFS with Multiple Memory Controllers,” *International Symposium on Low Power Electronics and Design*, pp. 297–301, 2012.

[25] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, “MemScale: Active Low-Power Modes for Main Memory,” *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 225–238, 2011.

[26] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, “Power management of online data-intensive services,” *ACM/IEEE International Symposium on Computer Architecture*, pp. 319–330, Jun. 2011.

[27] 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," *IEEE/ACM International Symposium on Microarchitecture*, pp. 347–358, 2006.

[28] P. Bose, A. Buyuktosunoglu, J. A. Darringer, M. S. Gupta, M. B. Healy, H. Jacobson, I. Nair, J. A. Rivers, J. Shin, A. Vega, and A. J. Weger, "Power Management of Multi-Core Chips: Challenges and Pitfalls," *Design, Automation and Test in Europe Conference and Exhibition*, pp. 977–982, 2012.

[29] N. Rasmussen, "Electrical efficiency measurement for data centers."

[30] Intel, "Intel xeon processor e3-1200 v3 product family," 2013.

[31] AMD, "ACP: The truth about power consumption starts here," 2009.

[32] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits*. Prentice Hall, 2003.

[33] Intel, "Desktop 4th generation Intel core processor family, desktop Intel Pentium processor family, and desktop Intel Celeron processor family," 2014.

[34] D. Qingyuan, "Active low-power modes for main memory," Ph.D. dissertation, Rutgers University, 2014.

[35] M. Harchol-Balter, *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[36] Seagate, "Desktop HDD ST500DM002," Dec. 2012. [Online]. Available: http://tinyurl.com/c358blv

[37] Intel, "Intel ethernet controller I350 datasheet," Mar. 2013. [Online]. Available: http://tinyurl.com/cc8mhvp

[38] ——, "Intel 80200 processor based on Intel XScale microarchitecture datasheet," Jan. 2013. [Online]. Available: http://tinyurl.com/l5xz77u

[39] A. Gandhi and M. Harchol-Balter, "How data center size impacts the effectiveness of dynamic power management," *Allerton Conference on Communication Control and Computing*, pp. 1164–1169, Sep. 2011.

[40] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," *Proceedings of the International Symposium on Workload Characterization*, pp. 171–180, 2007.

[41] F. Gustafsson, *Adaptive filtering and change detection*. Wiley, Sep. 2000.

[42] E. S. Page, "Continuous inspection schemes," *Biometrika Trust*, vol. 41, pp. 100–115, Jun. 1954.

[43] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators," *IEEE International Symposium On High Performance Computer Architecture*, Feb. 2008.

[44] G. Yan, Y. Li, Y. Han, X. Li, M. Guo, and X. Liang, "AgileRegulator: A Hybrid Voltage Regulator Scheme Redeeming Dark Silicon for Power Efficiency in

a Multicore Architecture," *IEEE International Symposium On High Performance Computer Architecture*, pp. 1–12, 2012.

[45] I. Akyildiz, "On the exact and approximate throughput analysis of closed queuing networks with blocking," *IEEE Transactions on Software Engineering*, vol. 14, no. 1, 1988.

[46] S. Balsamo, V. D. N. Persone, and R. Onvural, *Analysis of Queuing Networks with Blocking*. Springer, 2001.

[47] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable Power Control for Many-Core Architectures Running Multi-Threaded Applications," *ACM/IEEE International Symposium on Computer Architecture*, pp. 449–460, 2011.

[48] R. Teodorescu and J. Torrellas, "Variation-Aware Application Scheduling and Power Management for Chip Multiprocessors," *ACM/IEEE International Symposium on Computer Architecture*, pp. 363–374, 2008.

[49] JEDEC, "DDR3 SDRAM Standard," 2009.

[50] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu, "Decoupled DIMM: Building High-Bandwidth Memory System Using Low-Speed DRAM Devices," *ACM/IEEE International Symposium on Computer Architecture*, pp. 255–266, 2009.

[51] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherw ood, and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," *ACM SIGMETRICS*, pp. 318–319, 2003.

[52] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, G. Saidi, and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, Jul. 2006.

[53] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," *IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, 2009.

[54] Micron, "1Gb: x4, x8, x16 DDR3 SDRAM," 2006.

[55] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 219–130, 2010.

[56] JEDEC, "DDR3 SDRAM Standard," 2009.

[57] Intel, "Intel Xeon Processor 5600 Series," 2010.

[58] J. Sharkey, A. Buyuktosunoglu, and P. Bose, "Evaluating Design Tradeoffs in On-Chip Power Management for CMPs," *International Symposium on Low Power Electronics and Design*, pp. 44–49, 2007.

[59] X. Li, Z. Li, F. M. David, P. Zhou, Y. Zhou, S. V. Adve, and S. Kumar, "Performance-directed energy management for main memory and disks," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 346–380, 2004.

[60] L. L. H. Andrew, M. Lin, and A. Wierman, "Optimality, fairness and robustness in speed scaling designs," *ACM SIGMETRICS*, pp. 37–48, 2010.

[61] A. Wierman, L. L. H. Andrew, and A. Tang, "Power-aware speed scaling in processor sharing systems," *Powerformance Evaluation*, vol. 69, pp. 601–622, Dec. 2012.

[62] A. Gandhi, M. Harchol-Balter, and I. Adan, "Server farms with setup costs," *Performance Evaluation*, vol. 67, pp. 1123–1138, Nov. 2010.

[63] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy, "Optimal power allocation in server farms," *ACM SIGMETRICS*, pp. 157–168, Jun. 2009.

[64] M. J. Neely, "Lower power dynamic scheduling for computation systems," University of Southern California, Tech. Rep. arXiv:1112.2797, Dec. 2011.

[65] H.-W. Lee, K.-H. Kim, Y.-K. Choi, J.-H. Shon, N.-K. Park, K.-W. Kim, C. Kim, Y.-J. Choi, and B.-T. Chung, "A 1.6V 1.4 Gb/s/pin consumer DRAM with self-dynamic voltage-scaling technique in 4.4nm CMOS technology," *IEEE International Solid-State Circuits Conference*, Feb. 2011.

[66] X. Li, R. Gupta, S. Adve, and Y. Zhou, "Cross-component energy management: Joint adaptation of processor and memory," *ACM Transactions on Architecture and Code Optimization*, 2007.

[67] V. Pandey, W. Jiang, Y. Zhou, and R. Bianchini, "DMA-Aware Memory Energy Management," *IEEE International Symposium On High Performance Computer Architecture*, pp. 133–144, 2006.

[68] K. Meng, R. Joseph, R. P. Dick, and L. Shang, “Multi-Optimization Power Management for Chip Multiprocessors,” *International Conference on Parallel Architectures and Compilation Techniques*, pp. 177–186, 2008.

[69] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, “Scalable Thread Scheduling and Global Power Management for Heterogeneous Many-Core Architectures,” *International Conference on Parallel Architectures and Compilation Techniques*, pp. 29–40, 2010.

[70] R. Bergamaschi, G. Han, A. Buyuktosunoglu, H. Patel, and I. Nair, “Exploring Power Management in Multi-Core Systems,” *IEEE Design Automation Conference*, pp. 708–713, 2008.

[71] A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das, “CPM in CMPs: Coordinated Power Management in Chip Multiprocessors,” *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2010.

[72] M. Chen, X. Wang, and X. Li, “Coordinating Processor and Main Memory for Efficient Server Power Control,” *ACM International Conference on Supercomputing*, pp. 130–140, 2011.

[73] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, “Power Management of Online Data-Intensive Services,” *ACM/IEEE International Symposium on Computer Architecture*, Jun. 2011.

[74] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers," *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 65–76, 2013.

[75] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps," *IEEE/ACM International Symposium on Microarchitecture*, pp. 64–75, 2011.

[76] P. Petrica, A. M. Izraelevitz, and C. A. Shoemaker, "Flicker: A Dynamically Adaptive Architecture for Power Limited Multicore Systems," *ACM/IEEE International Symposium on Computer Architecture*, pp. 13–23, 2013.

[77] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller, "Energy Management for Commercial Servers," *IEEE Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[78] M. Ware, K. Rajamani, M. Floyd, B. Brock, J. Rubio, F. Rawson, and J. Carter, "Architecting for Power Management: The IBM POWER7 Approach," *IEEE International Symposium On High Performance Computer Architecture*, pp. 1–11, 2010.

[79] X. Fan, C. S. Ellis, and A. R. Lebeck, "The Synergy between Power-aware Memory Systems and Processor Voltage Scaling," *Proceedings of the International Conference on Power-Aware Computer Systems*, pp. 164–179, 2003.

[80] P. D. Welch, "On the generalized M/G/1 queuing process which the first customer of each busy period receives exceptional service," *Operation Research*, vol. 12, pp. 736–752, Sep. 1964.