Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria
Doctoral Program In Information Technology

Techniques and Tools for Efficient, QoS-Driven Warehouse-Scale Computing

Doctoral Dissertation of:
Davide Basilio Bartolini

Supervisor:
Prof. Marco D. Santambrogio

Tutor:
Prof. Donatella Sciuto

The Chair of the Doctoral Program:
Prof. Carlo E. Fiorini

2015 — XXVII Cycle

Tarapìa tapiòco!

Prematurata la supercazzola, o scherziamo?

Acknowledgements

All the work presented in this dissertation originated from some kind of collaboration. Therefore, I need and want to acknowledge several people without whom this document would not exist.

AutoPro The research project that finally led to AutoPro went through many stages from random ramblings about autonomic computing to its final form. This process involved several *submit — reject — rewrite — resubmit* iterations to finally get the paper published in the current form. The continuous discussion and collaboration with **Filippo Sironi** (a.k.a. Pippo) were vital to carry on with this process to the end; this work would not exist without his great help. Important contributions to this project also came from **Martina Maggio**, our trustworthy control theory person.

Rubik I worked on Rubik during my visit at MIT during the first half of 2014. While, in some way, I ended up playing an important role in the project, I believe that the amount of things I learned from this collaboration far exceeds my actual contribution. The main ideas that shape Rubik materialized, much in a Socratic way, during (often animated) discussions with **Harshad Kasture** (a.k.a. Harsh), who also implemented the missing parts in zsim. The constant supervision of Prof. **Daniel Sanchez** guided the two of us (often through more animated discussions) in reworking things until the whole picture came

together. **Nathan Beckmann** contributed with interesting perspectives (microeconomics, to name one) and helped a lot from the office next-door in the rush for the first *submit — reject* iteration (hopefully, we will not need as many as for AutoPro).

My sincerest thanks to the people I mentioned, to all the others who contributed to these two projects, but whom I am now forgetting due to my goldfish-like memory, and to those I worked with (particularly, all the people of the NECST Lab) on the many other projects that have nothing to do with the topic of this dissertation.

Getting a PhD is not (only) about research While this dissertation marks the end of my life as a PhD student, it can't come close to summing up all the experiences I went through in the last three years. A large thank you goes to my PhD supervisor, Prof. **Marco D. Santambrogio**, for allowing me to spend one year out of three visiting top universities in the United States; I think that this is a privilege not granted to many. Thanks also to my PhD tutor, Prof. **Donatella Sciuto**, who always supported me whenever the need arose. The two 6-month experiences at Berkeley (2013) and MIT (2014) mean a lot to me, beyond work and research. Another thanks is due to Prof. **John Kubiatowicz** (a.k.a. Kubi) and to Prof. **Daniel Sanchez**, for hosting me at these two prestigious institutions as a visiting student. During these two semesters abroad, I met many great people with whom I did some really incredible things: climbing in Yosemite, the Gunks and many other beautiful places; camping in Camp 4; swimming in Lake Tahoe; ice-climbing on Mount Washington (a.k.a. Mount Wash'N'Tan); doing some extremely serious bushwhacking up Katahdin; . . . . Unfortunately, I can't name everyone; just thanks to all.

Last but not least Infine, un grande grazie ai miei genitori, **Silvana** e **Giuseppe**, per il loro costante supporto sotto ogni forma, e a **Valentina**, per tutto quello che abbiamo vissuto e vivremo ancora insieme. Senza il vostro affetto, i momenti difficili di questi tre anni sarebbero probabilmente stati insormontabili. Grazie.

Abstract

Warehouse-scale computing, which is supported by datacenters, emerged in the last decade as a fundamental enabling technology for pervasive phenomena such as the Web 2.0, big data, and cloud computing. Despite being assembled from commodity components (servers, interconnects, ...), these datacenters opened the way to a new paradigm for mainstream computing; as researchers work on understanding this new paradigm, two important themes emerge in a new way compared to traditional systems. A major concern for datacenter operators is their efficiency and cost-effectiveness, which are crucial to supporting the growth in the services and value coming from big data and cloud computing. Additionally, public cloud computing presents further challenges for both datacenter operators and users. A major issue for users that want to bring their workloads to the cloud to take advantage of utility computing is that performance on virtualized resources is hard to understand and often unpredictable. For this reason, using public clouds for applications that need to provide a required quality of service (QoS) level is not straightforward and often leads to increased inefficiency due to conservative resource allocations.

There is a tension between these two issues (efficiency and QoS), as techniques to improve efficiency (e.g., virtualization, power management, colocation, ...) impact performance, often unpredictably. This dissertation attacks both sides of this tension and proposes novel tech-

niques and tools to help solve it, towards future efficient QoS-driven warehouse-scale computing.

First, we analyze a well-known model for the total cost of ownership (TCO) of a datacenter and find that, as things stand today, opportunities to further reduce TCO, and allow datacenters to scale further, mostly lie in improvements in the efficiency of IT equipment, particularly the efficiency of servers. On this basis, there are three main opportunities to improve efficiency: increasing server utilization, reducing static power consumption, reduce dynamic power consumption. The challenge is being able to target these opportunities without hurting QoS. We show that traditional mechanisms and policies to pursue these goals are not well-suited for datacenters: colocating applications causes inefficiency and performance degradation due to contention on shared resources; deep sleep states impose high transition latencies and flush shared state, impairing performance; traditional controllers for dynamic voltage and frequency scaling (DVFS) reduce active power, but can heavily impact performance, because they are oblivious to the peculiarities of datacenter applications.

Then, we analyze metrics to quantify the performance of datacenter applications and define their QoS. Throughput is a general metric to quantify rate of progress or load, but it is not enough to capture the performance of *latency-critical* applications, such as user-facing services, which need to provide performance guarantees on the end-to-end latency of each request. Latency-critical applications are particularly interesting, because they define an operating context that is peculiar to datacenters; we analyze the behavior of five latency-critical applications, studying how latency is affected by different operating conditions. One important consequence of defining QoS with application-level metrics is that traditional systems that optimize for aggregated, low-level metrics cannot provide this type of QoS guarantees.

The main contribution of this dissertation is proposing novel approaches to the problem of achieving QoS enforcement in an efficient way in two complementary scenarios. We analyze these two scenarios and propose two methodologies and practical systems (AutoPro and Rubik) that solve this problem:

- AutoPro tackles on the problem of providing predictable performance with automated resource allocation in public infrastructure-as-a-service (IaaS) cloud computing. AutoPro provides a practical solution based on a control-theoretical background for systems running compute-bound, throughput oriented applications. With AutoPro, we focus on current hardware and propose a solution that is directly deployable on modern datacenters with no hardware changes.

- Rubik analyzes datacenters running latency-critical applications, along with other batch work and tackles the problem of reducing the TCO while maintaining QoS guarantees on the tail latency, thus improving efficiency. Rubik provides a solution based on a runtime system and few key hardware changes, mainly to provide partitioning of the memory hierarchy; this solution could be implemented with negligible overhead on next-generation servers.

Both AutoPro and Rubik demonstrate the importance of three principles that we suggest as guidelines for the development of next-generation computer architecture and operating systems for datacenters:

- Availability through the hardware/software stack of application-level information is key for effective control.

- Control systems used to tune system-level knobs need to be founded on solid theoretical bases (e.g., AutoPro uses control theory, Rubik uses statistics and control theory); ad-hoc empirical controllers do not generalize well and often fail due to unpredictable pathological cases.

- In order to support the dynamic execution context of datacenters, as opposed to the static runtime of traditional clusters, control systems need to operate at a high frequency; coarse-grained adaptation cannot adapt to quick changes and imposes overly conservative guardbands, leaving much on the table.

Completely solving the problems of providing QoS and operating datacenters efficiently remains an open research problem, and different techniques and approaches are needed depending on the specific

context (application types, public versus public clouds, criticality of the QoS requirements, ...). This dissertation analyzes these problems and provides two practical solutions for two somewhat complementary scenarios.

Sommario

Il *warehouse-scale computing* (letteralmente, computazione a scala di magazzino) è emerso, nell'ultimo decennio, come una fondamentale tecnologia di supporto per fenomeni pervasivi come *Web 2.0*, *big data* e *cloud computing*. I *datacenter*, che forniscono l'architettura fisica per il *warehouse-scale computing*, sono stati realizzati con componenti (server, interconnessioni, ...) *presi in prestito* dal mercato di largo consumo e già disponibili da tempo. Nonostante ciò, questi datacenter hanno aperto la strada ad un nuovo paradigma di computazione che i ricercatori stanno cercando di analizzare. In particolare, due temi emergono in maniera nuova rispetto a sistemi di computazione tradizionali. Un tema fondamentale per chi gestisce un datacenter è la sua efficienza, che si rivela fondamentale per supportare la crescita nei servizi offerti e nel valore ricavato, ad esempio tramite offerte come il cloud computing e dallo sfruttamento dell'enorme mole di dati a disposizione (big data). Inoltre, considerando i servizi di cloud computing pubblico, si presentano ulteriori sfide sia per chi gestisce l'infrastruttura che per gli utenti. In questo contesto, una preoccupazione fondamentale per gli utenti che vorrebbero spostare le proprie applicazioni su un servizio di cloud computing, sfruttando così i vantaggi in termini di elasticità dei costi, è che le prestazioni delle applicazioni su risorse *virtualizzate* sono più complesse da analizzare e spesso impredicibili. Per questa ragione, utilizzre un servizio di cloud computing per applicazioni che necessi-

tano di un certo livello di qualità del servizio—o, in inglese, quality of service (QoS)—è complesso e spesso rende il servizio meno efficiente a causa di allocazione troppo conservativa delle risorse.

C'è una tensione tra queste due problematiche (efficienza e QoS), poiché le tecniche che possono migliorare l'efficienza (ad esempio, virtualizzazione, gestione energetica, colocamento, ...) hanno un impatto sulle prestazioni, spesso in modo impredicibile. Questa dissertazione attacca entrambi i lati di questa tensione e propone nuove tecniche e strumenti per tentare di risolverla.

Come prima cosa, analizziamo un noto modello per il costo complessivo—in inglese, total cost of ownership (TCO)—di un datacenter e mostriamo che, per lo stato delle cose odierno, le opportunità di migliorare l'efficienza dei datacenter si trovano soprattutto nel migliorare l'efficienza dei singoli componenti l'infrastruttura, in particolare quella dei singoli server. Sulla base di questa osservazione, identifichiamo tre principali opportunità per migliorare l'efficienza: aumentare l'utilizzo dei server, ridurre il consumo di potenza statico, ridurre il consumo di potenza dinamico. La sfida è riuscire a sfruttare queste opportunità senza peggiorare la qualità del servizio offerto. Mostriamo che meccanismi e politiche tradizionali non sono adatti per raggiungere questi obiettivi nei datacenter: colocare applicazioni cause inefficienze e degrado delle prestazioni a causa di contesa su risorse condivise; gli stati di risparmio energetico *profondi* richiedono lunghe latenze di transizione e ripristino dello stato, degradando le prestazioni; i controllori tradizionali per il controllo di frequenza e voltaggio—in inglese, dynamic voltage and frequency scaling (DVFS)—sono in grado di ridurre la potenza attiva, ma possono causare un forte degrado prestazionali, poiché non tengono conto delle caratteristiche peculiari delle applicazioni.

In seguito, analizziamo delle metriche per quantificare le prestazioni delle applicazioni eseguite nei datacenter e per definirne la qualità del servizio (QoS). Il throughput è una metrica generica per quantificare la *velocità* di una applicazione o il carico, ma non è sufficiente per descrivere appieno le prestazioni di applicazioni sensibili alla latenza—in inglese, *latency-critical*. Queste applicazioni richiedono garanzie prestazionali sulla latenza complessiva di ogni richiesta; un esempio è un servizio di ricerca in un grande database (o su internet). Le applicazio-

ni latency-critical sono particolarmente interessanti, poiché definiscono un contesto operativo tipico dei datacenter; qui analizziamo il comportamento di cinque diverse applicazioni di questo tipo, studiando come diverse condizioni operative influenzino la latenza di servizio delle richieste. Una importante conseguenza che si ha nel definire la QoS con metriche di livello applicativo (ad esempio, throughput o latenza) è che tecniche note che ottimizzano sulla base di metriche aggregate e di basso livello non sono adeguate per garantire le prestazioni desiderate.

Il contributo principale di questa dissertazione è la proposta di nuovi approcci al problema di fornire la QoS desiderata in modo efficiente, guardando a due scenari complementari. Analizziamo questi due scenari e proponiamo due metodologie e due sistemi reali (AutoPro e Rubik) che risolvono questo problema:

- AutoPro si occupa del problema di garantire performance predicibili attraverso l'allocazione automatica delle risorse in un servizio di cloud computing pubblico di tipo *infrastructure-as-a-service (IaaS)*. AutoPro rappresenta una soluzione pratica basata su teoria del controllo per sistemi che eseguono applicazioni *compute-bound* e *throughput-oriented*. Con AutoPro, ci focalizziamo su sistemi attuali e proponiamo una soluzione che è direttamente realizzabile su datacenter moderni, senza alcun cambiamento hardware.

- Con Rubik analizziamo datacenter che eseguono applicazioni latency-critical insieme ad altre applicazioni *batch*, ovvero senza forti requisiti prestazionali. Ci occupiamo del problema di migliorare l'efficienza del servizio riducendo i costi senza disattendere i requisiti di QoS stipulati sulle applicazioni latency-critical. Rubik è una soluzione basata su un sistema software e alcune specifiche modifiche hardware che, principalmente, permettono il supporto al partizionamento della gerarchia di memoria; questa soluzione può essere implementata con costi trascurabili su server di prossima generazione.

Sia AutoPro che Rubik dimostrano l'importanza di tre principi che suggeriamo come linee guida per lo sviluppo delle architetture e dei sistemi operativi per le prossime generazioni di datacenter:

- La disponibilità, ad ogni livello del sistema, di informazioni di livello applicativo è fondamentale per un controllo efficiente.
- I sistemi di controllo utilizzati per regolare parametri di sistema devono avere una solida base teorica (ad esempio, AutoPro usa la teoria del controllo e Rubik usa analisi statistica e teoria del controllo). Controllori ad-hoc basati su euristiche non generalizzano bene e spesso falliscono a causa di casi particolari patologici che sono difficili da individuare in sistemi di questa complessità.
- Per supportare l'ambiente di esecuzione molto dinamico dei datacenter, rispetto all'ambiente statico tipico dei cluster, i sistemi di controllo devono operare ad alta frequenza. Adattare i parametri ad grana grossa non permette di adeguarsi a cambiamenti rapidi e impone di essere molto conservativi nelle allocazioni, riducendo notevolmente l'efficienza.

Risolvere completamente i problemi di garantire qualità del servizio e di operare i datacenter in modo efficiente rimane un problema di ricerca aperto e diverse tecniche ed approcci sono necessari in specifici contesti (a seconda del tipo di applicazioni, criticità dei servizi, ...). Questa dissertazione analizza questi problemi e propone due soluzioni pratiche per due scenari complementari.

Contents

List of Figures

List of Figures

List of Tables

CHAPTER 1

Introduction and Background

During the past few years, mainstream computing has been shifting from personal computers (PCs) to mobile devices and cluster of computers. These two trends seemingly go towards opposite ends of the computing devices spectrum, which spans from small and constrained embedded systems to large-scale installations with massive compute and storage resources, but they are really deeply connected. The link that connects them is the Internet that, by becoming faster and more available across the population, allows to *outsource* computation from mobile devices through *web services* that run remotely on clusters of computers [7]. These remote resources are known in different "flavors", according to their specificities: *datacenters*, *server farms*, or *warehouse-scale computers* [7]; in this dissertation, we will use *datacenter* as a generic term for the whole category.

More specifically, the rising importance of datacenters is due to three factors [91]:

- Datacenters can make size- and power-constrained mobile computing devices such as smartphones or tablets much more interesting and appealing, by increasing their compute and storage capacity through *web-services* such as web search and social networking.

- The storage of huge amounts of data coming from clients (the so-called *big data*) in datacenters enables to extract, through accurate analysis, valuable information (e.g., customer preferences and attitudes).

- Massive datacenters are a key enabling factor to offering *utility computing* through public cloud computing services such as Amazon Web Services (AWS), Apple iCloud, Google AppEngine, and Microsoft Azure.

In a sense, the growth of datacenters is both driven by and a key enabler for the Web 2.0, big data, and cloud computing.

A major concern for datacenter operators is their efficiency and cost-effectiveness, which are crucial to supporting the growth in the services and value coming from big data and cloud computing. Improving datacenter efficiency is a major challenge that researchers have been increasingly addressing in recent years and remains an open research problem.

Additionally, public cloud computing presents further challenges for both datacenter operators and users. A major issue for users that want to bring their workloads to the cloud to take advantage of utility computing is that performance on virtualized resources is hard to understand and often unpredictable [102]. For this reason, using public clouds for applications that need to provide a required quality of service (QoS) level is not straightforward and often leads to increased inefficiency due to conservative resource allocations [40].

There is a tension between these two issues (efficiency and QoS), as techniques to improve efficiency (e.g., virtualization, power management, colocation, ...) impact performance, often unpredictably. This dissertation attacks both sides of this tension and proposes novel techniques and tools to help solve it, towards future efficient QoS-driven warehouse-scale computing.

The remaining of this introduction presents background material useful to contextualize and motivate the main contributions (summarized in Section 1.5):

- Section 1.1 briefly looks at opportunities to improve datacenter efficiency based on a cost analysis;
- Section 1.2 analyzes, through some experiments on datacenter servers, possible techniques to improve server efficiency;
- Section 1.3 analyzes application-level QoS metrics for datacenter applications;
- Section 1.4 introduces the problem of efficiently attaining QoS.

Additional background material specific to subsequent chapters is presented therein.

1.1 Improving Datacenter Efficiency, One Server at a Time

Operating datacenters efficiently is an important concern for both economic and sustainability reasons: datacenters already draw tens of megawatts of power [9], reportedly about 1.5 to 2% of all global electricity in 2011, growing at a rate of 12% a year [43]. Further scaling up datacenter capabilities to support the explosion of virtual information would lead to unbearable costs and power requirements at the current efficiency levels.

From an economic standpoint, a concise metric commonly used to quantify datacenter-related expenses is the total cost of ownership (TCO). Since providing a detailed introduction to how to compute and evaluate TCO is out of the scope of this thesis, here we just propose a brief overview; Leverich [77] provides a more detailed discussion. In brief, TCO captures all the expenses related to owning and operating a datacenter; it is computed as the sum of capital expenses and operating expenses. Breaking down the TCO into its components is a good way to identify the major contributors to costs, that need be acted upon to improve efficiency.

Hamilton [47] proposed a simple and illustrative model to estimate TCO for a datacenter and break it down into components. This model

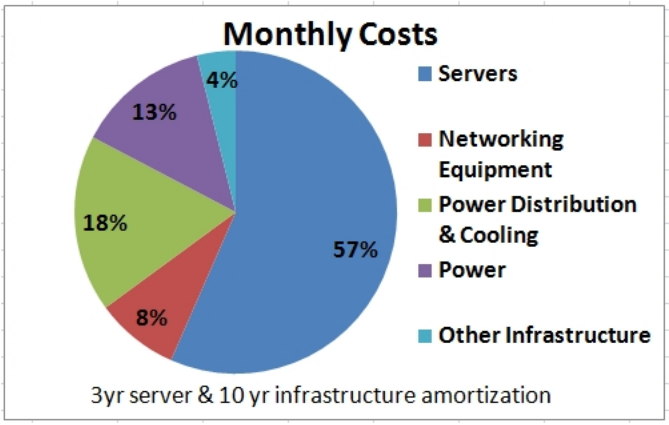


Figure 1.1: *Analysis of monthly costs to own and operate a datacenter, according to Hamilton [47].*

is based on the assumption the servers have a 3-year lifetime and the power usage effectiveness (PUE) of the datacenter (i.e., the ratio between total power draw—including cooling, power delivery losses, etc.—and power drawn by actual IT equipment) is 1.45. While these assumptions might be conservative (e.g., state-of-the-art datacenters have PUE under 1.2), the model is still representative. Figure 1.1 (borrowed from Hamilton [47]) shows a breakdown, based on this model, of the TCO in five components due to servers, networking equipment, power distribution and cooling, power supply, and other infrastructure. Expenses due to servers alone make up for almost 60% of the TCO in Hamilton's model. Moreover, the decade-long optimization process that brought the PUE of recent state-of-the-art datacenters very close to 1 (as low as 1.06 for Facebook's Prineville datacenter [98]) exploited most of the opportunities to improve efficiency at the facility level. For this reason, opportunities to further reduce TCO, and allow datacenters to scale further, mostly lie in improvements in the efficiency of IT equipment, particularly the efficiency of servers [77].

Improving datacenter servers efficiency is challenging and it requires considering the whole hardware/software stack. A key consideration to make when trying to reduce the fraction of TCO due to servers is that cost reductions must not hurt performance beyond the required QoS. Ideally, efficiency should come from costs reductions that do not

hurt performance at all; however, most techniques to reduce costs (e.g., power management, colocation) inevitably introduce some overhead on performance. Moreover, classic techniques such as sleep states or dynamic voltage and frequency scaling (DVFS) are not well tuned to the characteristics of datacenter workloads. For these reasons, efficiency and QoS considerations need to go side by side; the main contributions of this thesis, described in Chapters 2 and 3, tackle both QoS and efficiency.

1.2 Techniques to Improve Efficiency

We see three main opportunities to improve the efficiency of datacenter servers [77]:

- increase server utilization by consolidating workloads or growing workloads on existing hardware.
- reducing static power consumption (i.e., power drawn regardless of load) in each server; and
- reduce per-server dynamic power consumption (i.e., power drawn to do useful work);

The rest of this section analyzes these three opportunities. Section 1.2.1 deals with workload consolidation; Section 1.2.2 describes sleep states, which are the most common way to reduce static power when servers are idle; Section 1.2.3 deals with dynamic voltage and frequency scaling (DVFS), which is the most common mechanism available on modern servers to reduce active power and improve energy efficiency.

1.2.1 Workload Consolidation

Consolidating (or colocating) workloads has the potential of reducing the number of servers needed to run a given service, with obvious efficiency gains. However, the multicore processors that equip current servers were not designed to efficiently handle multiprogrammed workloads and colocation can adversely affect performance due to scheduling artifacts or contention on shared resources, most notably, the last level cache (LLC) [66, 92].

Researchers tackled these issues with systems that provide QoS when using shared memory resources. On the one hand, software-only systems [27, 28, 92, 137] detect interference empirically, and throttle or migrate batch applications that cause too much degradation. These schemes work with current multicores, but they react to interference instead of preventing it. On the other hand, relatively simple changes in the hardware allows to to explicitly partition shared memory resources (e.g., cache capacity and memory bandwidth) [13, 20, 45, 60, 69, 71, 88, 101, 105, 109, 115]; clearly, these mechanisms need policies to dynamically size partitions, to avoid the pitfall of low utilization due to over-provisioning of partition sizes to maintain performance [22, 33, 46, 57, 79, 100, 109, 123, 141]. These schemes explicitly manage shared resource partitioning to allow colocation and guarantee some kind of QoS to the colocate workloads.

Indeed, colocation is a valuable idea to improve the efficiency of datacenters; **we exploit colocation in both of the main contributions of this thesis**. With Rubik (Chapter 3), we take advantage of research in shared resource partitioning and we assume that the shared portion of the memory system (most notably, the LLC) is partitioned in hardware, to avoid interference between colocated applications. This assumption is particularly important in this case, since Rubik deals with latency-critical applications (see Section 1.3), the performance of which is particularly sensitive to contention and interference. While this assumption is reasonable looking at future hardware, servers currently deployed in datacenters do not implement hardware partitioning schemes. For this reason, AutoPro (Chapter 2), that looks at implementing a Performance-as-a-Service (PeaaS) model on current servers, does not assume hardware partitioning Instead, AutoPro can take advantage of software-only techniques to classify compatible workloads and it is robust to interference that may still occur.

1.2.2 Sleep States

Starting from the observation that datacenters are reported to often operate at low load [9, 91], a straightforward idea to improve energy efficiency of systems operating at low load is minimizing idle system

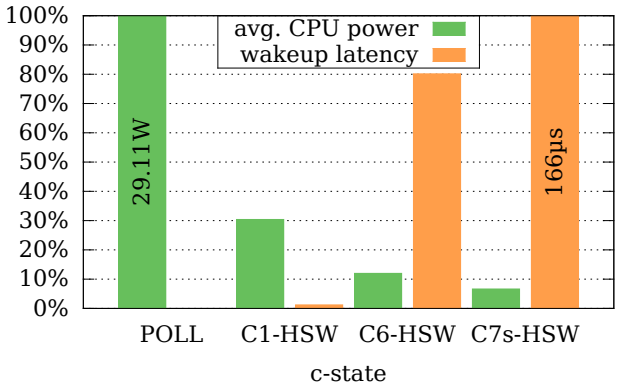


Figure 1.2: *Power draw and wakeup latencies of different C-states on a Haswell processor*

power. Research in this direction developed mechanisms and policies to enable various components to operate in *low power states* when idle.

Modern CPUs, for example, support c-states, which save power during idle periods by clock- or power- gating parts of the chip. Deeper c-states (i.e., those that *put to sleep* larger portions of the chip) save more instantaneous power, but require longer transition times and may impact performance due to loss of microarchitectural state (e.g., LLC state). For instance, Figure 1.2 shows the latency and average CPU power measured on a quad-core Intel processor based on the Haswell microarchitecture. The power data were gathered by reading the appropriate model specific register (MSR) reporting the energy consumed by the core portion of the processor (i.e., by the four cores, excluding the LLC) through the RAPL interface. After reading the counter, all the four cores were left idle for 10 seconds and the energy counter was probed again; the figure reports the difference between the two measurements. This methodology allows the processor to exploit both core- and package-level c-states; therefore, power savings are an upper bound to what is achievable. The wakeup latency data reported are taken from what the processor exposes to the OS; Linux exposes this information through the `sysfs`. The figure is consistent with the c-states terminology used in Linux for Intel processors (the -HSW suffix stands for Haswell):

- POLL represent a busy idle loop: when the CPU goes idle (i.e., upon executing the MWAIT instruction), sleep mode is effectively disabled and the cores "keep spinning"; power draw is not reduced compared to when cores are active (often called C0-state) and there is no wakeup latency.
- C1-HSW is the shallowest sleep state; it stops the CPU main internal clock via software, but the bus interface and APIC keep running at full speed.
- C6-HSW, besides stopping the CPU clock, also flushes inner-level caches (i.e., L1 and L2) to a dedicated SRAM, saves the cores architectural state to a dedicated SRAM, and reduces the CPU internal voltage to $0\,V$.
- C7-HSW additionally flushes and powers down the LLC (i.e., the shared L3 cache).

There is also a C3 c-state in-between C1 and C6, which has latency/power characteristics quite similar to those of C6.

As Figure 1.2 shows, deeper c-states save more power but impose a higher wakeup latency to return to normal operation (i.e., C0). For this reason, while deeper c-states will bring higher power savings, switching to them is increasingly more expensive in terms of wakeup latency. Moreover, the deepest c-state (i.e., C7), flushes the shared LLC, imposing additional misses to DRAM for the cached data that were flushed. For this reason, deep sleep states are beneficial only with relatively long idle periods which, in datacenters, might occur or not depending on the type of running applications (see Section 1.3 for more details on datacenter application types). Particularly interesting is the case of latency-critical applications which, despite often operating at low average utilization [25, 66], have very brief (but frequent) idle periods, making deep sleep states ineffective.

Sleep States and Latency-Critical Applications

The overheads deriving from wakeup latency and loss of state make deep c-states ineffective for servers running latency-critical tasks. As an example, Figure 1.3 shows the tail-latency degradation and the

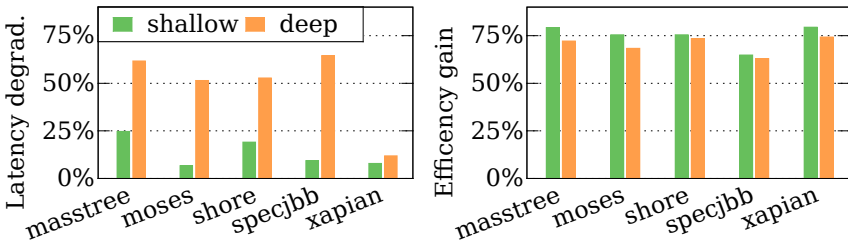


Figure 1.3: *95th percentile tail latency degradation and efficiency (requests per second over average power draw) gain over no sleep states when using shallow (C6-HSW) or deep (C7-HSW) C-states on a Haswell processor for five latency-critical applications.*

energy efficiency gain when using c-states of different depth for five latency-critical applications running at circa 20 % utilization on the same server equipped with a quad-core Intel processor we used for Figure 1.2. These applications were first used by Kasture and Sanchez [66] as a varied benchmark set for latency-critical applications and are the same we use to evaluate Rubik (they are briefly described in Chapter 3, Table 3.3). As for Figure 1.2, we run only one application on one core, while the other three are left idle, in order to trigger package-level c-states and evaluate a "best-case" scenario in terms of power savings. We compute energy efficiency as throughput (i.e., requests per second) over average power to serve a fixed number of requests.

In the figure, the shallow and deep sleep states are, respectively, the C6-HSW and C7s-HSW c-states, described above; the main difference is that the deeper c-state flushes and powers down the LLC, further reducing idle power but also imposing higher wakeup latency and additional cache misses. In fact, the shallow sleep state dramatically reduces idle power, ensuring significant efficiency gains (up to 80% against disabling c-states) at the price of small tail-latency degradation. Instead, the deep sleep state causes major latency degradation and does not further improve energy efficiency. This counter-intuitive result is due to the large "inertia" associated with the LLC [66]: whenever the CPU wakes up from sleep, applications need to re-build their working set in the LLC, causing significant performance and energy overheads that counterbalance the power savings during the brief idle

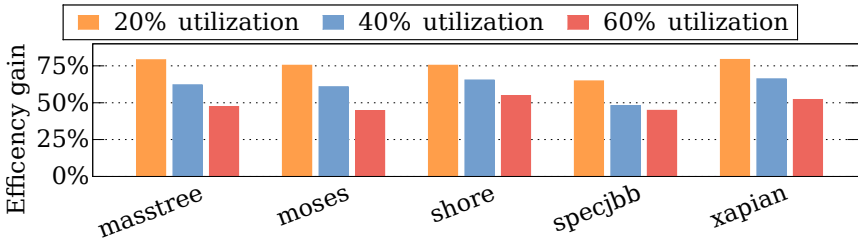


Figure 1.4: *Efficiency (requests per second over average power draw) gain over no sleep states when using a shallow (C6-HSW) C-state on a Haswell processor for five different latency-critical applications at three utilization levels.*

periods. Instead, private caches (L1/L2) have a relatively small inertia and using shallower sleep state results in negligible overheads.

These results rule out deep sleep states to improve efficiency when running latency-critical applications, event at low utilization and indicate that shallow sleep states can be beneficial in that case. However, as research allows to increase datacenter utilization while maintaining QoS (e.g., through safe workload colocation, as described in Section 1.2.1), idle periods will decrease further, reducing the opportunity of sleep states to improve efficiency. As an example, Figure 1.4 show the efficiency gain of our five latency-critical application as utilization increases to 40 % and 60 % (the 20 % case is the same shown in Figure 1.2); as expected, the gains decrease circa linearly. These results indicate that sleep states, while useful in some cases, are not the most relevant tool to improve the efficiency of datacenter servers. Increasing utilization and managing active power are more important objectives. **For this reason, both AutoPro and Rubik aim at keeping utilization high, and Rubik exploits DVFS to manage active power so as to maximize efficiency.**

Sleep States for Other Components

Other system components such as DRAM, disks, NICs and fans are far less energy proportional than the CPU [6, 9] and system-level sleep states are an active area of research. For example, Powernap [95] proposes using components such as self-refreshing DRAMs and solid-state disks (SSD) to support full system low power modes with short transition times; Dreamweaver [94] builds on Powernap, adding a co-

processor that determines when the system should enter deep sleep states. These techniques require idle periods to be relatively long in order to be effective; this requirements makes them incompatible with latency-critical applications, which typically have very brief but frequent idle periods [85]. Another example is Knightshift [134], which proposes a tightly coupled, low power compute node that powers down the CPU and takes over its functionality during periods of low utilization. Knightshift has the same incompatibility with latency-critical tasks as Powernap and Dreamweaver, since moving to and from the low power state incur relatively large transition time due to the transfer of large amounts of state.

1.2.3 Dynamic voltage and frequency scaling

Dynamic voltage and frequency scaling (DVFS) is the most widespread knob to trade off performance and power draw in modern processors. Scaling down frequency f and voltage V through DVFS leads to power P decreasing as $P \propto CV^2 f$, with (at most) linear decrease in performance. As voltage is scaled linearly with frequency in CMOS technology, this property leads to cubic power reductions at the price of (sub) linear performance loss when scaling down frequency.

The actual performance loss mostly depends on how the running application uses memory, since the drop in frequency only influences performance of the fraction of time spent on in-core computation and not the fraction of time due to memory accesses. For instance, if the application is totally compute-bound (i.e., there are no stalls on memory accesses in the processor pipeline), then performance decreases linearly with the drop in frequency. Instead, if the application is mostly memory-bound, performance will not be affected much, since the actual fraction of time spent in the core (i.e., when the pipeline is not stalled on memory accesses) is a small part of the total runtime.

While voltage still scales linearly with frequency in modern processors[1], the actual voltage scaling range has been reducing due to the breakdown of Dennard scaling [31] in recent CMOS generations. For instance, on a 2007 Intel Pentium M the minimum voltage was $\approx 35\%$ lower than the nominal voltage, while this reduction is $\approx 20\%$ on Sandy

[1] e.g., $V \approx 1.169 \cdot 10^{-4} f + 0.634$ on an Intel Haswell processor.

Bridge and still $< 30\%$ on Haswell. This limited voltage range reduces the impact of scaling down DVFS on power draw on recent processors: we empirically found that $P \propto f^2$ on an Intel Haswell processor.

The diminished direct impact of DVFS on power draw, however, does not kill this knob as a useful tool for energy efficiency. The key observation is that, since memory accesses are still much slower than the core frequency, whenever a workload is memory bound scaling down DVFS has negligible impact (i.e., strongly sublinear) impact on performance [61]. Therefore, the most energy-efficient DVFS setting (also called a p-state) depends on the running application [34]. For instance, Figure 1.5 reports the CPU energy per instruction of three applications from the SPEC CPU 2006 benchmark suite running on a quad-core Intel processor at each p-state, from 800 MHz to *Turbo mode*. The three applications have different characteristics, with `bzip2` being the most compute-bound[2] and *libquantum* being the most memory-bound. According to these characteristics, the most energy-efficient frequency changes for each application. Moreover, the most efficient p-state changes at runtime, as an application goes through more or less memory-bound phases [61].

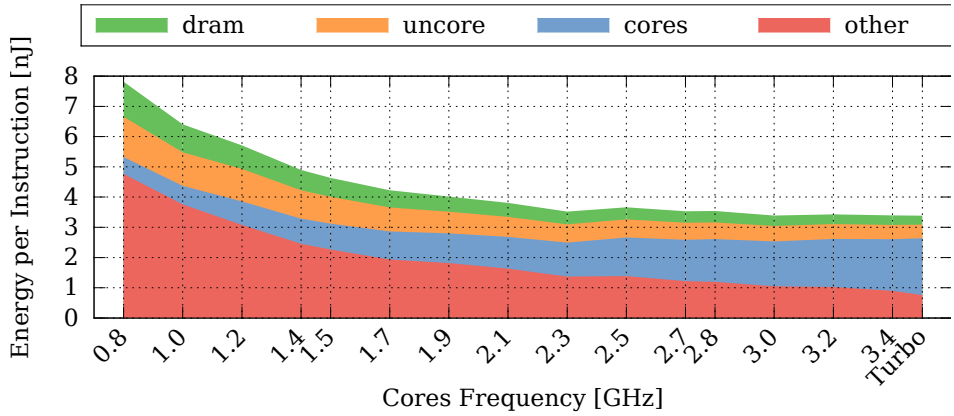
For these reasons, Rubik (Chapter 3) continuously profiles, through performance counters, the running applications, building CPI stacks [34] in order to identify memory-bound phases and find the most efficient p-state.

1.3 Datacenter Applications and QoS

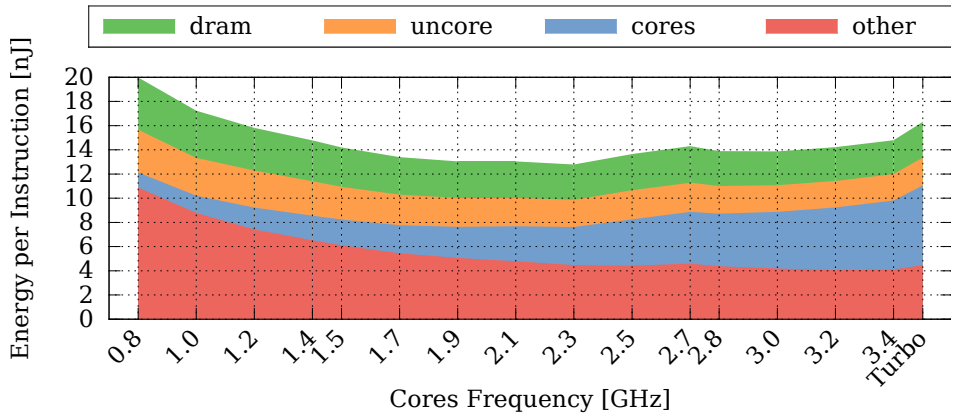
Measuring and evaluating the quality of service (QoS) of the running applications is particularly important in datacenters, since revenue for the service provider largely depends on the offered QoS. For instance, if the latency of queries to a web search engine is often too high, users will move to a different search engine. As another example, often QoS terms are specified as service-level agreements (SLAs), or service-level objectives (SLOs) in the contract for public cloud computing services.

An important difference with respect to other large-scale computing installations, such as high-performance computing (HPC) clusters, is

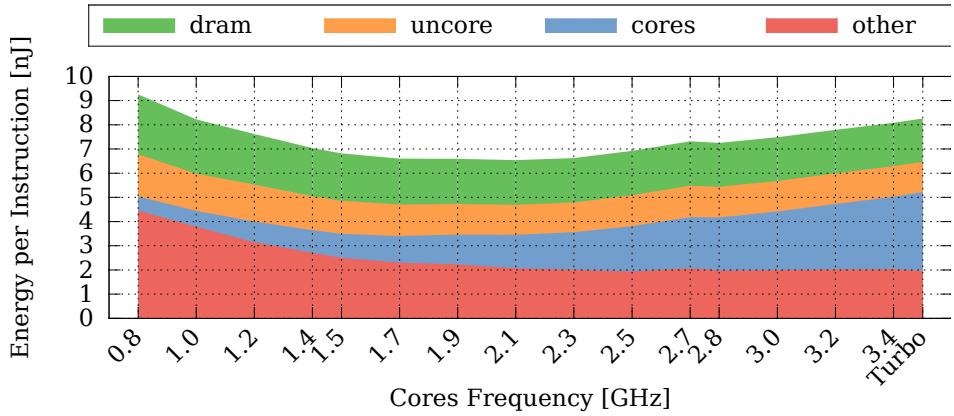
[2] We run on in-memory inputs.



(a) *bzip2*



(b) *mcf*



(c) *libquantum*

Figure 1.5: *Energy efficiency at different frequencies for three applications on a quad-core Intel processor based on the Haswell architecture.*

that, *in datacenters, performance needs be evaluated from a user perspective, with high-level metrics that capture the perceived performance*. Low-level metrics based on hardware performance counters, such as instructions per cycle (IPC), are not well-suited for this task. Moreover, performance and needs to satisfy some objective, not necessarily being maximized. There exist two major general high-level metrics to assess the performance of applications deployed to a datacenter: throughput and latency.

1.3.1 Performance as Throughput

Throughput is an aggregate metric that can assess the overall performance of the application. For instance, the performance of a video-encoding task is well-represented by the rate at which it encodes frames (i.e., *frames per second*).

Throughput applies to most applications and it can be computed at different levels. For instance, the throughput of a mapreduce job could be monitored as the number of (map or reduce) tasks completed per second, or as the amount of data processed per second; at the same time, each task could be monitored looking at the rate of completion of each basic operation. The job-level measurement (i.e., tasks per second) gives an overall indication of progress, while the task-level measurement could be used to check the progress of the single tasks and identify stragglers [2].

Throughput is also important for request-based applications such as on-line transaction processing (OLTP). In this case, the rate at which requests are served (i.e., requests per second) gives an indication of overall performance and can be compared to the load (i.e., the rate of request submission) to asses the health of the service.

AutoPro (Chapter 2) uses throughput as the reference performance metric to automatically allocate a contended resource (i.e., CPU bandwidth) so as to meet user-specified performance SLOs in a public cloud computing scenario.

1.3.2 Latency-critical Applications

While throughput is a general metric, user-facing, request-based applications often need to provide performance guarantees on the end-to-end

latency of each request (or, more commonly, statistical guarantees on the distribution of latencies), which throughput does not capture.

In order to serve each request, user-facing services (e.g., web search) access very large data sets distributed in a tree-like structure: front-end nodes accept requests and dispatch them to leaf nodes. Each leaf node analyzes a portion of the data and returns a partial result to the front-end node, which aggregates the partial results to serve the request. Therefore, the user-perceived latency of each request depends on the response time of the *slowest* leaf node, which determines the end-to-end quality of service (QoS). Previous research [120] showed that a server-side delay of the order of 100 ms can reduce the QoS to the point of significantly impairing revenues; for this reason, leaf nodes need to respect strict bounds on *tail-latencies* [25] (95^{th} or 99^{th} percentile latency). Depending on the specific structure of each service, the bound on tail-latency at the leaf nodes can be very tight, as low as 100s of microseconds in some cases [85].

The request latency can be factored into four terms:

- the service time, i.e., the time it takes to compute the response to the request once since when it starts being processed;
- the queuing time, i.e., the time the request spends in the queue, waiting to be processed;
- network stack overhead, i.e., the time the request spends being dispatched in the server network stack;
- network delays, i.e., the time the request spends being transfered from the client to the server plus the time the response spends in transit back to the client.

Using the same approach proposed by Kasture and Sanchez [66], we focus on the first two terms (service time and queuing time); network stack overheads were tackled in recent research through user-level networking [65], while dealing with network delays is out of the scope of this work.

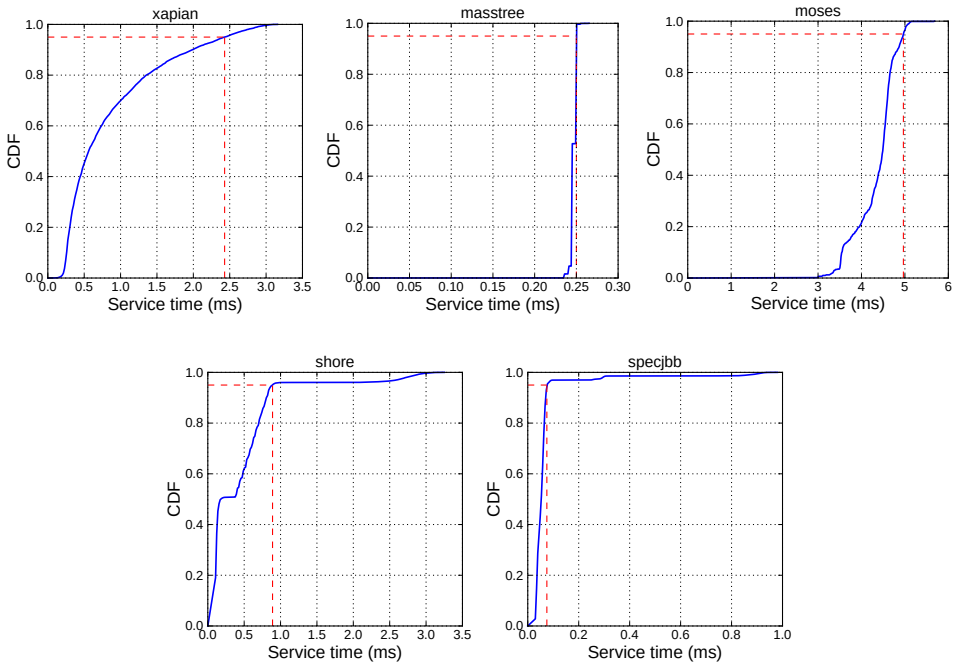


Figure 1.6: *Cumulative distribution functions (CDFs) of request service time (no queueing delay) for five latency-critical applications. Dashed lines indicate the 95th percentile service times (courtesy of Kasture and Sanchez [66]).*

Service Time Distributions

Figure 1.6 shows the cumulative distribution function (CDF) of the service time of five latency-critical applications (they are briefly described in Chapter 3, Table 3.3), when simulated with the zsim [116] multicore simulator, modeling a processor with six out-of-order cores validated against a real Intel Westmere system running at a fixed frequency of 2 GHz [66]. The applications are simulated, rather than run on real hardware, to take advantage of a partitioned LLC, which is not available on current commodity processors. The five applications present a varied set of service time distributions: `masstree` and `moses` have near-constant service times, with a very short tail, while `xapian`, `shore-mt`, and `specjbb` present multi-modal or long-tailed distributions. This variability in service time, which is intrinsic in the different compute requirements of the single requests poses a first challenge to

maintaining a desired QoS level (i.e., a desired tail latency) for these applications.

Impact of Queuing Time on Latency

While the distribution of service times largely determines the end-to-end latency at low load, when load (i.e., the arrival rate of requests) increases, queuing time becomes critical in determining the latency distribution.

To illustrate this effect, Figure 1.7 presents an analysis of the initial part of the traces of the simulated five latency-critical applications at high load. For most applications, queuing time determines most of the latency. Long queues can have two causes:

- the arrival of "trains" of requests; or
- the arrival of one or more requests with particularly long service times (i.e., those in the tail of the distributions in Figure 1.6).

The first effect is particularly evident for `masstree`, while the second can be identified in `specjbb` and `xapian`.

Impact of Load on Latency

Intuitively, as load increases more requests will have to be queued up waiting for a previous request to be served, causing increased queuing times and, therefore, increased latency. Again, we analyze this effect on the five latency-critical applications; Figure 1.8 reports the mean and 95th percentile latencies for the five latency-critical applications with increasing load. Since service times do not change, the increase both mean and tail latency is due to increased queuing times. Three observations can be made on Figure 1.8 [66]:

1. tail latency is far from the mean;
2. increased load imposes a critical overhead on tail latency;
3. tail latency degrades superlinearly.

These observations lead to the conclusion that latency-critical applications subject to tail-latency QoS requirements inherently need to run at low utilization to meet their QoS, since increasing load has a dramatic

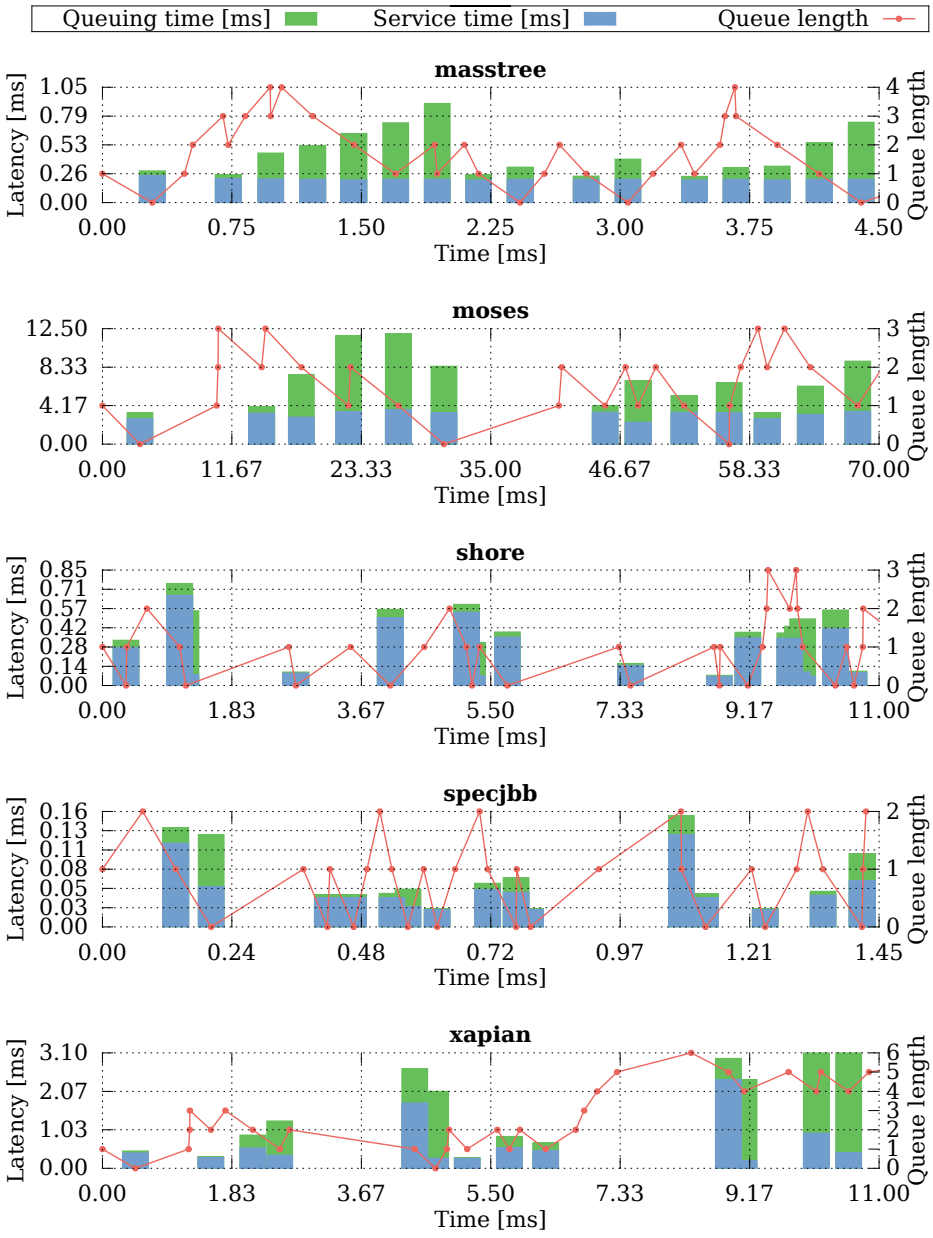


Figure 1.7: *Latency, broken down into service and queuing time, and queue size of the initial portion of the simulation traces of the five latency-critical applications. When a new request arrives, it is added to the end of the queue. When the first request in the queue is served, the queue size is decreased by one and a bar on the plot indicates the request latency.*

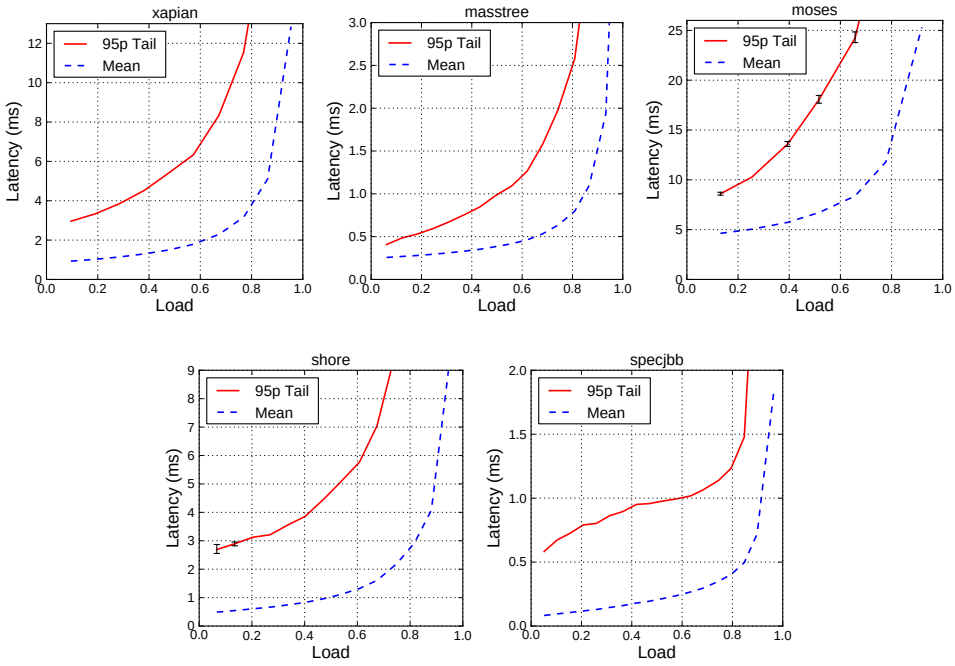


Figure 1.8: *Load-latency diagrams for mean (dashed blue) and 95th percentile tail (solid red) latencies for the five latency-critical applications (courtesy of Kasture and Sanchez [66]).*

impact on latency due to queuing delay. Previous research [66] showed that carefully colocating batch work with latency-critical applications, by taking care of the issues with scheduling and memory hierarchy partitioning, can increase utilization, improving the efficiency of datacenters running latency-critical applications. With Rubik (Chapter 3) we show that application-aware DVFS can be used to increase the utilization of latency-critical workload itself, by speeding up cores when needed (e.g., when a large queue builds up) to "flatten" the load-latency curves of Figure 1.8.

1.4 Efficiently Attaining Quality of Service

The QoS and efficiency of services and applications running in a datacenter ultimately depends on management and allocation of resources (e.g., cores, memory bandwidth, . . .) and "knobs" (e.g., DVFS) that affect performance and power draw. Manually managing these resources and knobs to tune performance in an efficient way is already hard on a single server; the scale of datacenters makes it a daunting task. This problem opens a great opportunity for control systems able to automatically manage the datacenter, creating a *goal-oriented*, automated framework to efficiently meet QoS constraints.

Both the main contributions of this dissertation regard building systems of this sort. While these systems use different control techniques and target different scenarios, they share one important property, that is key to achieving efficiency: *availability of application-level information throughout the computing stack*. In order to efficiently manage resource and knobs to meet QoS objectives on application-level metrics (as described in Section 1.3), control systems for datacenters direly need to be aware of application-level information such as current throughput, request latency, or service time. Traditional systems that optimize for aggregated, low-level metrics (one outstanding example is the turbo mode controller available in recent processors [84]) cannot achieve this goal. **We believe that this property (availability of application-level information throughout the computing stack) should guide the development of computer architecture and operating systems for datacenters**.

1.5 Contributions and Guiding Principles

To sustainably support the shift of mainstream computing from personal computers to mobile devices and datacenters, several challenges need be addressed throughout the computing stack, from hardware to applications, through architecture and operating system. Addressing all of these challenges goes beyond the scope of one dissertation; this dissertation focuses on rethinking and improving key aspects at the system level (i.e., the computer architecture and the operating system) for datacenters. We identify and tackle two important goals that can be approached by working at the system level: efficiency of datacenter operation and quality of service (QoS) enforcement. Jointly achieving QoS enforcement and efficiency is a difficult problem, as naïve solutions tend to trade-off one for the other; for instance, one could easily guarantee QoS by over-committing resources in an inefficient way.

The main contribution of this dissertation is proposing novel approaches to the problem of achieving QoS enforcement in an efficient way in two complementary scenarios. The following two chapters analyze these two scenarios and propose two methodologies and practical systems that solve this problem:

- **AutoPro** [12] (Chapter 2) tackles on the problem of **providing predictable performance with automated resource allocation** in public infrastructure-as-a-service (IaaS) cloud computing. **AutoPro provides a practical solution based on a control-theoretical background** for systems running **compute-bound, throughput oriented applications**. Chapter 2 **focuses on current hardware** and proposes a solution that is directly deployable on modern datacenters with no hardware changes.

- **Rubik** [67] (Chapter 3) analyzes datacenters running **latency-critical applications**, along with other batch work and tackles the problem of **reducing the TCO while maintaining QoS guarantees on the tail latency**, thus improving efficiency. Rubik provides a **solution based on a runtime system and few key hardware changes**, mainly to provide partitioning of the memory hierarchy; this solution could be implemented with negligible overhead on next-generation servers.

Both AutoPro and Rubik demonstrate the importance of three principles that this dissertation suggests as guidelines for the development of next-generation computer architecture and operating systems for datacenters:

- Availability through the hardware/software stack of **application-level information is key** for effective control (see Section 1.4).
- Control systems used to tune system-level knobs need to be founded on **solid theoretical bases** (e.g., AutoPro uses control theory, Rubik uses statistics and control theory); ad-hoc empirical controllers do not generalize well and often fail due to unpredictable pathological cases.
- In order to support the dynamic execution context of datacenters, as opposed to the static runtime of traditional clusters, **control systems need to operate at a high frequency**; coarse-grained adaptation cannot adapt to quick changes and imposes overly conservative guardbands, leaving much on the table.

CHAPTER 2

AutoPro[1]

Ideally, the pay-as-you-go model of Infrastructure as a Service (IaaS) clouds should enable users to rent just enough resources (e.g., CPU or memory bandwidth) to fulfill their service level objectives (SLOs). Achieving this goal is hard on current IaaS offers, which require users to explicitly specify the amount of resources to reserve; this requirement is non-trivial for users, because estimating the amount of resources needed to attain application-level SLOs is often complex, especially when resources are virtualized and the service provider colocates virtual machines (VMs) on host nodes. For this reason, users who deploy VMs subject to SLOs are usually prone to over-provisioning resources, thus resulting in inflated business costs.

This chapter tackles this issue with *AutoPro*: a runtime system that enhances IaaS clouds with automated and fine-grained resource provisioning based on performance SLOs. Our main contribution with *AutoPro* is filling the gap between application-level performance SLOs

[1] This chapter was adapted from the research paper "Automated Fine-Grained CPU Provisioning for Virtual Machines" [12].

and allocation of a contended resource, without requiring explicit reservations from users. In this chapter, we focus on CPU bandwidth allocation to throughput-driven, compute-intensive multithreaded applications colocated on a multicore processor; we show that a theoretically sound, yet simple, control strategy can enable automated fine-grained allocation of this contended resource, without the need of offline profiling. Additionally, *AutoPro* helps service providers optimize infrastructure utilization by provisioning idle resources to best-effort workloads, so as to maximize node-level utilization.

Our extensive experimental evaluation confirms that *AutoPro* is able to automatically determine and enforce allocations to meet performance SLOs, while maximizing node-level utilization by supporting batch workloads on a best-effort basis.

2.1 Introduction

Infrastructure as a Service (IaaS) clouds promise to enable business flexibility with a pay-as-you-go model for computation. Within this model, the interest of users is minimizing business costs for executing a given workload with the desired performance, while the interest of vendors is optimizing infrastructure utilization, so as to minimize the total cost of ownership (TCO), without breaking service level objectives (SLOs). Current virtualization infrastructures lack tools to easily fulfill these interests: users need to manually determine, for each virtual machine (VM), the amount of resources to rent; providers have to be conservative when consolidating workloads on multicore-powered host nodes to reduce TCO, since colocation can lead to unexpected performance degradation [36, 42, 92].

For these reasons, a system able to automatically size and enforce allocations of a contended resource based on user-defined, application-level performance requirements would be a valuable tool to help more easily meet users' and provider's interests. Designing such a system is an interesting and challenging problem. The main contribution of this chapter is presenting the design and evaluating the benefits of *AutoPro*: a runtime system we developed to serve as such tool.

Recent research [42, 66, 92] on workload consolidation provides techniques to estimate and mitigate performance degradation for consoli-

dated workloads, enabling safe sharing of host nodes. Other contributions [103, 108, 125] describe systems that automate resource allocation based on SLOs, but work at a coarse scale and granularity (see Section 2.5). In contrast, with *AutoPro*, we explore the possibility of enhancing IaaS clouds with automated, fast, and fine-grained resource allocation to meet users' SLOs, while allowing providers to safely share host nodes among VMs to maximize node-level utilization and reduce TCO. Section 2.2 motivates, through a case study, the advantages that *AutoPro* can bring to IaaS infrastructures.

AutoPro leverages feedback control to automate resource allocation, filling the gap between application-level performance SLOs and allocation of a contended resource, thus dispensing users from the need to explicitly file resource reservations. Moreover, *AutoPro* automatically allocates unclaimed resources to batch workloads on a best-effort basis, enabling providers to maximize node-level utilization. Section 2.3 discusses our design of *AutoPro* and its sub-systems, presenting the methodology that allows *AutoPro* to achieve these goals.

The main strength of *AutoPro* is its ability to take fast decisions and perform precise resource allocations without requiring to profile VMs in advance. This strength comes from the use of a theoretically sound, yet simple, control strategy. *AutoPro* uses controllers based on a resource-performance model that binds VM performance, measured in a metric meaningful to the user, and resource allocation. While the appropriate performance metric (e.g., throughput, latency, turnaround time) depends on the characteristics of the application wrapped in each VM, users do not need to worry about modeling: system developers can determine models that apply to a whole class of applications. Section 2.4.2 validates a resource-performance model for VMs that expose runtime throughput measurements (e.g., requests/s for a web server) and Section 2.4.3 evaluates *AutoPro* on managing such VMs. We are extending *AutoPro* to support other performance metrics.

In general, VM performance depends on the supply of different resources (e.g., compute, I/O bandwidth); however, in most cases, a single resource is the bottleneck at any given time. Contention on a single bottleneck resource will arise in many non-trivial colocation cases, since finding colocation mixes where the performance of different

VMs is bound to complementary resources is often not possible. For these reasons, automating allocation of a contended resource to meet performance SLOs is an interesting problem; *AutoPro* copes with it through a resource-performance model that considers the bottleneck resource. To concretely evaluate our approach, this chapter focuses on throughput-driven, compute-intensive multithreaded applications wrapped inside VMs colocated on a multicore processor, often found in datacenter host nodes. In this scenario, the compute bandwidth is the bottleneck resource we consider; extending *AutoPro* to deal with shifting bottleneck resources and different performance metrics is an orthogonal issue we are investigating (see Section 2.6).

We evaluate *AutoPro* running representative multithreaded applications that stress contention on compute bandwidth. Our extensive experimental evaluation (see Section 2.4) shows that *AutoPro* meets its goals of attaining SLOs with a lower error than the best static allocation and maximizing node-level utilization.

2.2 Case Study and Overview

In order to provide some context and motivation and to show what advantages *AutoPro* can bring to IaaS infrastructures, we present a case study where two users, u_1 and u_2 deploy a VM to a public IaaS cloud. Each VM wraps a compute-intensive multithreaded application subject to the service-level objective (SLO) of maintaining a required performance level, set on an application-specific throughput measurement. In this scenario, users want to achieve the respective SLO with the minimum amount of resources, to minimize business cost, while the cloud provider wants to optimize the datacenter utilization to minimize the total cost of ownership [78].

As a concrete example of this scenario, we colocate two VMs, respectively executing the *swaptions* and *x264* multithreaded applications from the PARSEC 2.1 benchmark suite [14], on a host node with a hexa-core processor (see Section 2.4.1 for details on its configuration). We set SLOs on application-specific performance metrics (i.e., swaptions/s and frames/s, respectively) such that both VMs can attain their SLOs within the host node capacity. We analyze a set of provisioning scenarios that the service provider might choose; to do

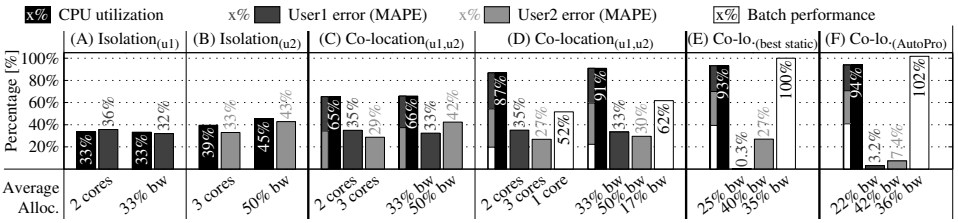


Figure 2.1: *Host node CPU utilization, mean average percentage error (MAPE) on the performance of VMs by $User_1$ and $User_2$, and performance of the batch VM in different scenarios (A-F). VMs run in isolation (A, B) or colocated (C-F); we evaluate static allocation of whole cores or CPU bandwidth (bw) (A-E) based on user requests of 2 and 3 vCPUs (A-D) or on an oracle (E) and dynamic allocation with* AutoPro *(F). The thin bars in scenarios C-E show the per-VM CPU utilization breakdown, using the same shading as error / performance.*

so, we leverage the integration of the KVM hypervisor [72] with the Linux kernel's resource containers (control groups, or *cgroups*) [4, 97] to allocate the CPU.[2] The service provider can choose space partitioning or time multiplexing to map the multithreaded applications that run in users' VMs onto the multicore processor of our host node. To evaluate both strategies, we configure all VMs with six virtual CPUs (vCPUs) and allocate the equivalent of n physical CPUs to a VM by either packing its vCPUs onto n cores or by using all cores and capping its CPU bandwidth to $n/6$. Figure 2.1 reports the performance and node-level CPU utilization of the two VMs in six different provisioning scenarios, marked (A) through (F).[3]

Scenarios (A) through (D) model current public IaaS clouds, which require users to explicitly rent an integer number of vCPUs. For these scenarios, we pick demands of 2 and 3 vCPUs for u_1 and u_2, respectively. These allocations are, for each VM, the integer number of vCPUs that more closely meets the SLO, i.e., that minimizes the mean average percentage error (MAPE), our accuracy metric. Intuitively, MAPE gives a measure of how far each VM is, on average, from meeting its performance SLO across its execution; Equation (2.6), in Section 2.4.2, reports a formal definition.

[2] Using *cgroups* for resource partitioning makes all our discussion directly applicable to traditional multiprocessing, where applications are not wrapped in VMs, as *cgroups* support regular Linux tasks.

[3] The results in Figure 2.1 are the 95th percentile over 30 runs for each scenario.

Scenario (E) evaluates the best (i.e., with lowest MAPE) static (i.e., fixed throughout VM execution) allocation, that we determined through offline profiling.

Finally, we deploy *AutoPro* on our host node to manage our case study; Scenario (F) reports these results, showing the advantages over current IaaS management policies.

2.2.1 Analysis of the Case Study

Figure 2.1 (A) and (B) analyze the baseline scenario in which the provider serves u_1 and u_2 on dedicated CPUs, backing the 2 and 3 vCPUs of u_1 and u_2 respectively with 2 and 3 cores (partitioning), or 33 % and 50 % CPU bandwidth (multiplexing). Using dedicated CPUs (i.e., single-socket host nodes or a multi-socket host node) avoids issues due to contention on shared resources (e.g., the last-level cache), but leads to under-utilization of the host nodes/sockets (CPU utilization is between 33 % and 45 %). Low node-level utilization is generally inefficient for the service provider, as it negatively impacts the TCO due to poor energy proportionality of current host nodes [95]. Moreover, the provisioning configuration of this scenario is also inefficient for users, who are far from precisely meeting their SLOs (MAPEs range from 32 % to 43 %). The performance traces show that both VMs actually outperform the respective SLOs, meaning that users are using, and paying for, more resources than they would really need; this situation is common, as IaaS users tend to over-provision resources [40]. Notice that the vCPUs-to-CPU mapping strategy variably influences CPU utilization and performance for the two VMs. For u_2's VM, which runs *x264*, the MAPE increases when using 6 cores at $\approx$ 50 % CPU bandwidth, than when packed on 3 reserved cores. This higher error is actually due to the VM running faster when using more cores (note that node-level utilization also increases). Instead, u_1's VM, which runs *swaptions*, runs slightly faster when partitioning cores. Section 2.3.3 elaborates further on partitioning versus multiplexing.

To increase node-level utilization, the provider can colocate the two VMs onto one host node/socket; Figure 2.1 (C) covers this scenario. Co-locating VMs improves node-level utilization (reaching $\approx$ 65 %),

as now the two VMs share hardware resources;[4] therefore, VMs' colocation benefits the provider. However, colocation can unpredictably impair the performance of users' VMs due to contention over shared resources (e.g., last-level cache (LLC), memory bandwidth). While this effect is negligible in this specific scenario (MAPEs are very close to the isolation scenarios and we verify that both VMs still outperform their SLO), because these VMs make for a good colocation pair, more noticeable performance interference arises in other cases, as we will discuss. In general, building good colocation mixes to limit performance interference is a challenging problem that is orthogonal to the scope of this chapter [42, 92].

To further increase node-level utilization, the provider can employ the leftover CPU bandwidth to run additional batch applications (e.g., maintenance jobs) on a best-effort basis; Figure 2.1 (D) covers this scenario. To simulate a batch workload, we run the *psearchy* benchmark [16], set to index the Linux kernel 3.9 source tree, in an additional VM.[5] We provision this batch VM with the compute capacity left unused by the two SLO-bound VMs: either 1 core or 17 % of the CPU bandwidth. Co-locating the batch VM bumps node-level utilization up to ≈ 90 % and turns out to reduce the MAPE for u_2's VM. This apparently positive effect is actually due a reduction of the performance of u_2's VM, due to the sensitivity of *x264* to contention over the LLC [23] caused by cache-intensive co-runners, like *psearchy*. Therefore, this effect is not beneficial to u_2, whose VM is still far from matching the SLO and also gets reduced performance while having to pay for the same amount of resources as in the previous scenario.

In all the scenarios we explored so far, users are far from precisely meeting SLOs and they face increased costs due to over-provisioning, since they rent more resources than meeting their SLO would require. We will explore and combine two ways of improvement: (1) automated and (2) finer-grained resource estimates and allocations. If the IaaS cloud could automatically determine the exact resource needs based on the SLOs and allocate resources on a fine-grained scale, the provider

[4] CPU utilization in scenario (C) is less than the sum of the CPU utilizations in scenarios (A) and (B) because, according to data from performance counters, in this case VMs colocation partly hides memory access latencies.

[5] Similarly, the SPECvirt_sc2013 benchmark [130] periodically runs file compression tasks in an otherwise idle VM to simulate batch workloads.

could take the burden of estimating resource needs from users. To evaluate the benefits of this scenario, we derive (through offline profiling) that the best static allocations (i.e., the ones that yield the lowest MAPEs on the performance SLO of both user-submitted VMs) are 1.5 vCPUs for u_1 and 2.4 vCPUs for u_2. Since we need to allocate fractions of a core, CPU partitioning is too coarse-grained for this case; therefore, we allocate vCPUs by capping the CPU bandwidth. Figure 2.1 (E) shows that, in this case u_1's VM (i.e., *swaptions*) almost perfectly matches its SLO (with 0.3 % error), while the lowest MAPE for u_2's VM (i.e., *x264*) is 27 %. The higher error for *x264* is due to this application showing varying execution phases (as Figures 2.3 and 2.11 further illustrate); due this characteristic, no static allocation will be able to closely track the SLO. This scenario is the best we explored so far, as both users are closer to meeting their SLOs, while renting fewer resources, and node-level utilization is kept high; moreover, the CPU bandwidth freed by using just enough resources to minimize the MAPEs goes to increase by ≈ 60 % the performance of the batch workload, benefiting the provider.

With *AutoPro*, we show that it is even possible to improve over scenario (E) by leveraging a resource-performance model, runtime performance measurements, and dynamic fine-grained resource allocation. Figure 2.1 (F) reports the results we obtain when deploying *AutoPro* to manage our case study: both VMs closely match the respective SLO (MAPEs are 3.2 % and 7.4 %, respectively) and both node-level utilization and performance of the batch VM are maximized. *AutoPro* reduces the error for *x264* with respect to the scenario (E) thanks to a fast dynamic allocation that, in contrast to static allocation, can react to varying execution phases; moreover, *AutoPro* keeps the error for *swaptions* low. Section 2.4.3 picks one of the runs of this scenario and provides further insights. Section 2.3 explains how we designed and implemented *AutoPro* to achieve such results.

Notice that, while the performance interference between u_2's and the batch VM impairs the colocation efficiency, *AutoPro* is robust to contention on unmanaged shared resources (as we discuss in Section 2.4.3). By exploiting performance feedback, *AutoPro* can realize a soft-partitioning of unmanaged resources by adjusting the allocation of

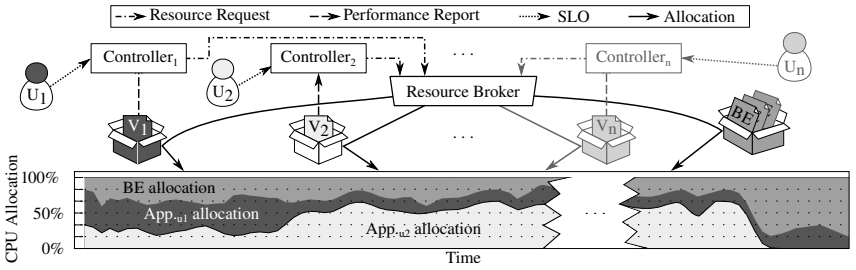


Figure 2.2: AutoPro *uses per-VM (V_i) controllers to estimate the resource needs to meet SLOs defined by users (U_i). A resource broker aggregates demands and determine allocations, fair-sharing leftover capacity among batch best-effort (BE) VMs.*

the bottleneck resource [37, 127]. Nonetheless, limiting the contention on unmanaged resources is still valuable, since this way of attacking degradation is inefficient and can complicate the billing schema (see Section 2.6).

2.3 Automated Fine-grained Provisioning

The idea at the base of *AutoPro* is extending IaaS clouds with fine-grained automated resource allocation, in order to provide users with a simple interface for specifying SLOs on understandable performance metrics. Figure 2.2 depicts the architecture of *AutoPro*. A resource container enforces allocations to each VM; a dedicated controller estimates, based on performance reports, the amount of resources each SLO-bound VM needs to meet its SLO; a broker collects resource demands and determines allocations, giving leftover resources to an additional resource container that fairly provisions batch VMs on a best-effort basis. The controllers that estimate the resource need for each VM leverage an adaptive resource-performance model that captures the relationship between performance measurements and allocation of the contended resource; Section 2.3.2 elaborates on the defined *AutoPro*'s control schema.

AutoPro deals with resource allocation and not with admission control or workload placement, which are related but orthogonal issues. We rely on the presence of an admission control system that chooses VM placement across the datacenter [19, 121] and we assume that

the hypervisor supports live migration [62] in case a node becomes overloaded. While both VM placement and migration present open questions for research, *AutoPro* focuses on a different challenging and interesting problem: automating allocation of a contended resource within a single node, based on application-level performance SLOs.

The innovative contribution of this chapter is showing how it is possible to extend IaaS clouds towards a *Performance-as-a-Service* model [11] that benefits both users, with SLO-based automated provisioning, and providers, with support for system-level management of idle resources. Thanks to fast and precise allocation decisions, *AutoPro* can adapt to time-varying workloads, automating the allocation of a contended resource to meet SLOs and maximize node-level utilization.

2.3.1 Performance Metrics and Measurements

AutoPro requires each SLO-bound VM to make periodic performance reports available to its controller, in order to leverage its resource-performance models, as proposed in previous works [10, 52, 108, 125, 127, 128, 140]. Any performance metric meaningful to the user can be used for these reports and to express SLOs; for instance, a web server can use throughput (e.g., requests/s for a web server) or latency (i.e., response time).

While this chapter focuses on throughput as the reference performance metric, *AutoPro* can be extended to support other performance metrics, provided that the appropriate resource-performance models are made available. Notice that application developers do not have to worry about resource-performance models: they simply need to provide performance measurements according to one or more supported performance metrics.

Application-specific high-level performance metric requires support from applications themselves. For this reason, we developed the *throughput library* (*libthroughput*) to instrument applications so as to provide feedback and ease users' task of specifying SLOs. Similarly to previous proposals [52, 127], *libthroughput* exports a simple API for applications and controllers.

Listing 1 Code snippet of a controller employing *libthroughput*.

```
throughput_t *monitor = throughput_init("Antani");
wait_application_start();
while (...) {
        sleep(...);
        slo = throughput_get_slo(monitor);
        g = throughput_get_global(monitor);
        r = control(g, slo);
}
throughput_destroy(monitor);
```

Listing 1 exemplifies the structure of a controller. With `throughput_init()`, the controller initializes a monitoring structure identified by a key (e.g., *Antani*) and backed by shared memory. After the application starts, the controller periodically queries the application's SLO and global throughput. At the end, `throughput_destroy()` cleans up the shared memory.

Listing 2 Code snippet of an application instrumented with *libthroughput*.

```
throughput_t *monitor = throughput_attach("Antani");
throughput_set_slo(monitor, slo);
while (...) {
        ... // complete one unit of work
        throughput_signal(monitor);
}
throughput_detach(monitor);
```

Listing 2 exemplifies the structure of an application. Applications attaches a monitoring structure identified by a key and already initialized by the controller through `throughput_attach()`, specify their SLOs by means of `throughput_set_slo()`, and signal progresses by calling `throughput_signal()` upon completion of a unit of work. When applications exhaust units of work they call `throughput_detach()` and exit.

To make sure we evaluate the design and implementation of *AutoPro* on a varied set of multithreaded workloads, we use several applications from the PARSEC 2.1 benchmark suite [14] (see Section 2.4). While these benchmarks were not designed to capture all the characteristics of typical cloud workloads, colocating PARSEC applications does create contention on compute bandwidth, thus stressing the problem that this chapter addresses. Since PARSEC applications do not natively

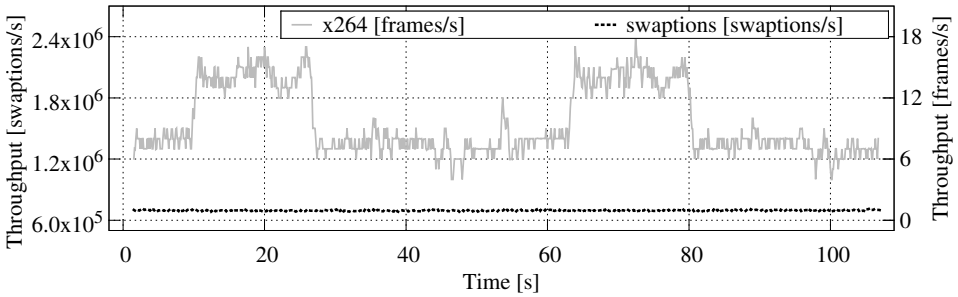


Figure 2.3: *Throughput of 6-threaded* swaptions *[swaptions/s] and* x264 *[frames/s], colocated on our hexa-core host node with static CPU bandwidth allocations.*

report performance at runtime, we instrument a subset of the suite[6] to report throughput through *libthroughput*. The hypervisor accesses VM performance measurements as in previous work [108, 125]. In real deployments, performance reports may be obtained from application logs or monitoring infrastructures.

The controllers we use to determine resource allocations (see Section 2.3.2) work on throughput measurements computed as a moving average on a sliding window. To give an example on how this metric can characterize the performance of PARSEC applications, Figure 2.3 shows the traces of two VMs with radically different behavior: with a constant resource allocation, *swaptions* has a stable performance throughout its execution, while *x264* shows input-dependent[7] oscillations [10, 118, 128]. Measuring throughput over an adequate sliding window allows the controllers to promptly catch transitions between execution phases and adjust resource allocations as needed (see Section 2.4.3).

2.3.2 Estimating Resource Needs

AutoPro periodically evaluates throughput measurements against the SLO and updates resource allocations to keep the performance of each VM in-line with its SLO. We devise a control schema that splits this process onto two levels:

[6] We pick applications that allow a simple and meaningful definition of throughput measurements.

[7] Here we encode the PARSEC native input twice.

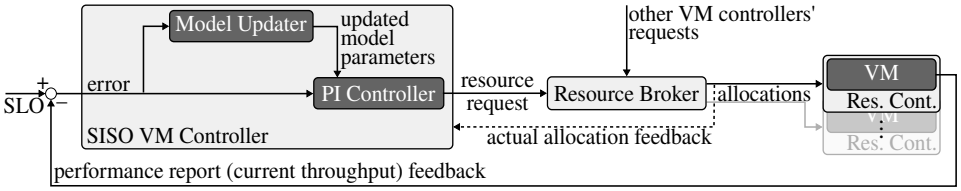


Figure 2.4: *Overall architecture of* AutoPro*'s control schema.*

1. estimating resource demands independently for each SLO-bound VM;

2. aggregating demands and determining allocations.

Figure 2.4 illustrates this schema, focusing on one VM. At each control step, (1) a dedicated *VM Controller* files a resource request, determined through an adaptive proportional-integral (PI) controller [3]; and (2) a resource broker (RB) aggregates these requests, determines the actual allocations based on system-level policies, and enforces them through the resource containers.

VM Controllers

VM Controllers (see the left-hand part of Figure 2.4) take care of continuously estimating the CPU bandwidth each VM needs to match its SLO.

The core of the *VM Controller* is a resource-performance model that binds the VM performance over a given time window (i.e., its throughput over that time window) to resource allocation. Equation (2.1) formalizes this resource-performance model, where $t_w(k)$ and $t_w(k+1)$ are the average throughput measured at control steps k and $(k+1)$ on a window of w seconds, and $r(k)$ is the CPU bandwidth the VM is provisioned with during the time quantum $(k, k+1)$.

$$t_w(k+1) = a \cdot t_w(k) + b \cdot r(k) \tag{2.1}$$

The terms a and b are parameters that characterize the workload and the scalability of the VM; for the model to be accurate, these two parameters must adapt to the characteristics of each VM. We use a Recursive Least Squares (RLS) filter (see the *Model Updater* in Figure 2.4) to continuously update the a and b parameters based on ob-

served throughput and allocated resources. Through this adaptive approach [3], we avoid the need of profiling VMs in advance and support VMs with varying load. Section 2.4.2 evaluates the accuracy of this model and of the online estimation.

In order to estimate resource requests based on the model in Equation (2.1), we leverage formal control theory to synthesize a PI controller: we derive an expression of the resource request $r(k)$ at step k as a function of the resource assignment $r(k-1)$ granted during the quantum $(k-1, k)$, the performance error $e(k) = \bar{t} - t_w(k)$, and the parameters a and b. First, in Equation (2.2), we determine the transfer function $\mathcal{P}(z)$ by applying the $\mathcal{Z}$-transform to Equation (2.1).

$$\begin{aligned} z \cdot \mathcal{T}_w(z) &= a \cdot \mathcal{T}_w(z) + b \cdot \mathcal{R}(z) \\ \mathcal{P}(z) = \frac{\mathcal{T}_w(z)}{\mathcal{R}(z)} &= \frac{b}{z-a} \end{aligned} \tag{2.2}$$

$\mathcal{T}_w(z)$ and $\mathcal{R}(z)$ are the $\mathcal{Z}$-transforms of the throughput measurement $t_w(k)$ and VM-owned resources $r(k)$, respectively.

We leverage formal control theory to synthesize the PI controller by constraining the transfer function of the feedback loop [49]; Equation (2.3) shows this operation

$$\mathcal{L}(z) = \frac{\mathcal{C}(z)\mathcal{P}(z)}{1 + \mathcal{C}(z)\mathcal{P}(z)} \triangleq \frac{1-p}{z-p}, \quad p \in (0,1) \tag{2.3}$$

$\mathcal{L}(z)$ and $\mathcal{C}(z)$ indicate the transfer functions of the feedback loop and the PI controller, respectively. We set $\mathcal{L}(z)$ to a first-order transfer function with one pole in p. The constraint $p \in (-1, 1)$ ensures that the closed loop is asymptotically stable; furthermore, $p \in (0, 1)$ guarantees convergence without overshooting and oscillations [49]. Within this interval, the parameter allows to set a trade-off between fast response and safety: smaller values of p will result in faster responses. We find that the actual value of p in the $(0, 1)$ range does not have considerable impact on the behavior of our controllers; we ascribe this fact to the nonlinearities that our linear resource-performance model discards. We empirically find $p = 0.1$ to be a good choice and use this value in all our experiments.

To actually find an expression for $r(k)$, we start by deriving, in Equation (2.4), an expression of $\mathcal{C}(z)$ by combining Equations (2.2)

and (2.3); we impose the transfer function of the PI controller to be the ratio between VM-owned resources $\mathcal{R}(z)$ and performance error $\mathcal{E}(z)$.

$$\mathcal{C}(z) = \frac{(1-p)\cdot(z-a)}{b\cdot(z-1)} \triangleq \frac{\mathcal{R}(z)}{\mathcal{E}(z)} \tag{2.4}$$

Elaborating further and applying the $\mathcal{Z}$-antitransform and a time shift we finally get, in Equation (2.5) the desired expression for $r(k)$.

$$r(k) = r(k-1) + \frac{1-p}{b}\cdot e(k) - a\cdot\frac{1-p}{b}\cdot e(k-1) \tag{2.5}$$

The PI controller in each *VM Controller* uses Equation (2.5) to estimate the resource request that should let the VM match its SLO in the next control period.

We implemented the *VM Controller* in C++, using the *Armadillo* library [117]. The simplicity of our model allows a fast implementation, which enables *AutoPro* to run with an aggressive control period to react to quick workload changes: our implementation takes less than 10 µs to update the parameter models and compute the next allocations in all our experiments on our reference machine (see Section 2.4.1 for details on its configuration). This very small overhead allows us to use faster control periods (see Section 2.4.1) than previous works [108, 125], achieving finer-grained control (as we discuss in Section 2.5).

Controllers Stability

The stability theorem of formal control theory for linear, time-invariant systems states that a system represented by a transfer function $\mathcal{G}(z)$ is stable if and only if the poles (i.e., the roots of the dominator polynomial) of $\mathcal{G}(z)$ are within the unit circle in the complex coordinate plane [49]. Our choice of $p = 0.1$ provides such guarantee, since $z = p$ is the only pole of the feedback loop transfer function, as Equation (2.3) shows.

The pole-elision method we use to synthesize our controllers adds the requirement that the elided poles (in our case, the only pole a, as Equation (2.2) shows) must not be unstable. To abide to this condition, we verify that our model updater always returns parameters estimates such that $|a| \in (0, 1)$.

Whenever VMs expose a steady behavior (i.e., within a phase), the RLS filter we use as our model updater will make the parameters a and b converge to their "real" values and, thanks to the stability properties we just discussed, the VM controllers will drive performance towards SLOs. When the workload of a VM varies over time, switching its resource-performance behavior (i.e., going through different phases), the model updater will take some control steps to converge to the new parameter values, causing transient oscillations of the allocation on phase boundaries.

Resource Brokerage and Allocation

At each control step, the resource broker (RB) collects all the resource requests from the *VM Controllers* and determines the actual allocations. This aggregation step enforces global constraints, such as CPU capacity, and applies global policies, e.g., how to use unclaimed resources.

We consider two constraints on allocations: a global maximum (max) and a per-VM minimum (min). The max bound enforces CPU maximum capacity, while the min bound avoids allocating too-small CPU fractions, which can cause imprecise enforcement at the resource container level.

A resource request sum exceeding the max bound indicates that the CPU capacity is not enough to let all SLO-bound VMs deliver the respective required throughput. If this case occurs, the RB can make one of two decisions:

1. migrate one or more VMs to a less-loaded node;
2. fairly distribute the available capacity to VMs proportionally to their original requests.

While migrating VMs might be necessary to respond to chronic resource shortage, this operation has non-negligible cost and should be avoided when resource scarcity is due to short heavier phases or control glitches because, in these cases, proportionally distributing the available capacity would at most cause a transient drop in performance. In this chapter, we only evaluate cases where resource scarcity happens only temporarily and do not consider VM migration, which represents

an extreme way out to make up for poor VM placements by the admission control system.

When the sum of resource requests is below the *max* bound, the RB chooses how to use the unclaimed resources. Since this chapter considers the objective of maximizing node-level utilization, *AutoPro* employs unclaimed resources to execute additional batch VMs on a best-effort basis.

In general, the RB could implement other policies to pursue different objectives; for instance, idling unused resources and trigger low-power states to reduce power draw, or redistributing unclaimed resources to SLO-bound VMs, based on a priority schema, to support different QoS levels [103].

2.3.3 Containers and Resource Provisioning

To enforce allocations, *AutoPro* relies on the ability of the hypervisor to partition resources and allocate them to VMs through a resource container mechanism [4]. We base our implementation on the Kernel Virtual Machine (kvm) hypervisor [72], which uses Linux *control groups* (*cgroups*) [97] for resource partitioning. The vCPUs of a VM appear as processes to the hypervisor, which groups them into the same *cgroup*, which allows to set caps on resource usage. We exploit *cgroup* hierarchies to create an additional *cgroup* that contains all the batch VMs, which fairly share the resources unclaimed by SLO-bound VMs.

Linux *cgroups* offer a few *subsystems* to allocate different types of resources (e.g., cores, CPU bandwidth, block I/O bandwidth). Since this chapter focuses on managing VMs that run compute-intensive multithreaded tasks, we use the *cpuset* and *cpu* [132] subsystems to allocate fractions of the CPU. These two subsystems control CPU partitioning and multiplexing: the *cpuset* subsystem allows to pin threads (in our case, vCPUs) in a *cgroup* to a subset of the available cores, while the *cpu* subsystem allows to set a cap on the CPU bandwidth each *cgroup* can use [132].[8]

AutoPro can support both CPU partitioning and multiplexing; we evaluated both mechanisms on managing VMs executing multithreaded applications from the PARSEC 2.1 benchmark suite colocated on our

[8]The Xen hypervisor [5] offers similar facilities [135].

hexa-core host node (configuration details are in Section 2.4). We verify that these applications scale well to six threads [118] and configure VMs with six vCPUs. To allocate the host CPU, we either pack vCPUs on a subset of the host cores or cap the host CPU bandwidth [132].

Applications with poorer scalability or host nodes with a higher core count might lead to scalability issues, which were tackled in recent research [118]. While these issues would impair the efficiency of our system, this problem is orthogonal to the scope of this chapter; Section 2.6 sketches a solution based on partitioning the multicore in smaller time-multiplexed islands.

We found that *AutoPro* obtains similar results, in terms of accuracy in enforcing SLOs, utilization, and batch VM performance, with both allocation mechanisms (i.e., dynamic partitioning or multiplexing). While the overall results are similar, CPU multiplexing allows more stable allocations, thanks to its finer granularity (partitioning forces minimum allocation granularity to 1 core) and we base our evaluation (Section 2.4) on this mechanism.

2.4 Evaluation

We evaluate *AutoPro* in two steps: Section 2.4.2 validates our resource-performance model against both single- and dual-VM workloads; Section 2.4.3 evaluates *AutoPro* on managing workloads made of two SLO-bound VMs and a batch VM.

2.4.1 Platform, System, and Applications

Our host node is a Dell Precision T3500 workstation with an Intel Xeon W3690 hexa-core processor, clocked at 3.46 GHz and equipped with 12 MB of shared LLC, and 12 GB of main memory, distributed on 3 channels and clocked at 1066 MHz. We disable the Enhanced Intel SpeedStep and Turbo Boost Technologies to bind the processor frequency to its nominal value. Due to the lack of cache partitioning mechanisms, we cannot directly address (i.e., eliminate) conflicts on the shared LLC; in our evaluation, we show that *AutoPro* is robust to performance interference due to this limitation. We disable the Intel Hyper-Threading Technology (i.e., simultaneous multithreading),

which could give performance benefits for some workloads, but would introduce contention on core-private caches, thus possibly exacerbating the performance interference problem, which we already show *AutoPro* can cope with.

We use Debian GNU/Linux 7.0 with kernel version 3.9 and configure the *cpu* subsystem of *cgroups* to enforce CPU bandwidth caps on a period of 50 ms. We configure *AutoPro* to sample VM throughput on a sliding window of 750 ms and update the resource-performance models every 50 ms, and allocate resources every 750 ms. We empirically determine these parameters to yield precise resource allocation and responsiveness on our benchmarks: shorter periods lead to imprecise resource allocations with the *cpu* subsystem of *cgroups*, while longer periods lead to unresponsiveness to execution phases.

To evaluate *AutoPro* on a varied set of multithreaded applications, we use *blackscholes*, *bodytrack*, *canneal*, *dedup*, *ferret*, *swaptions*, and *x264*, from the PARSEC 2.1 benchmark suite [14], as our workloads for SLO-bound VMs. We configure each VM with 6 vCPUs and each application to run with 6 threads except for *dedup* and *ferret*, which use 6 threads per pipeline-stage. We synchronize the execution of the regions of interest through the *hooks* provided by the benchmark suite. We use the *native* inputs for all applications except for *swaptions*, which evaluates 96 swap options, and *x264*, which encodes its native input video twice.

We use the indexing part of the *psearchy* (i.e., *pedsort*) benchmark [16] to simulate a batch workload not bound to a SLO. We use *psearchy* to index the Linux kernel 3.9 source tree using 6 threads, each using at most 1 GB for its hash table. Just as for the PARSEC applications (see Section 2.3.1), we instrumented *psearchy* to report throughput defined, in this case, as the number of jobs per hour per thread.

2.4.2 Resource-Performance Model Evaluation

We validate the accuracy of our resource-performance model, defined in Equation (2.1), which binds resource allocations $r(k)$ (i.e., a percentage of the CPU bandwidth) to VM throughput $t_w(k)$ (i.e., moving averages on a sliding window w of 750 ms). Each VM runs one of the PARSEC

applications we selected and we evaluate both solo runs (single-VM) and workloads with two colocated VMs (dual-VM).

At each control step k (i.e., every 750 ms), we randomly vary the resource allocation $r(k)$ of each VM and we log the current throughput $t_w(k)$ and the corresponding resource allocation $r(k-1)$; for the dual-VM workloads, we randomly partition the host CPU capacity among the two VMs. We execute each workload 30 times to achieve statistical relevance.

We use the mean absolute percentage error (MAPE) metric, computed as in Equation (2.6) to evaluate the model accuracy.

$$\text{MAPE} = \frac{1}{K}\sum_{k=0}^{K}\left|\frac{\widehat{t_w}(k) - t_w(k)}{t_w(k)}\right| \tag{2.6}$$

K is the number of control steps, $t_w(k)$ is the actual throughput sampled at step k, and $\widehat{t_w}(k)$ is the throughput estimate according to our model, defined in Equation (2.1).

AutoPro uses a *Model Updater* (see Section 2.3.2) to compute the model parameters a and b online. Besides evaluating the accuracy of our model through the MAPE metric, we also validate this filter by comparing it against the optimal offline parameters estimated with the least squares algorithm.

Prediction Error on Single-VM Workloads

To evaluate our model in absence of contention on shared resources, we first consider solo runs of VMs executing each of the PARSEC applications. Figure 2.5 reports the MAPE metric for our model when using both online and offline parameter estimates. These results show a low prediction error (within 5 %) for five out of seven applications and indicate that updating the model parameters online with the *Model Updater* improves accuracy for all the workloads compared to offline estimates of the optimal static parameters. Our results compare favorably with previous work leveraging a similar resource-performance model [123] (see Section 2.5).

The prediction error on the solo runs using online estimation is close to 1 % for four out of seven workloads; we found that the higher error on *bodytrack* ($\approx$ 8 %), *ferret* ($\approx$ 5 %), and *x264* ($\approx$ 6 %) is due

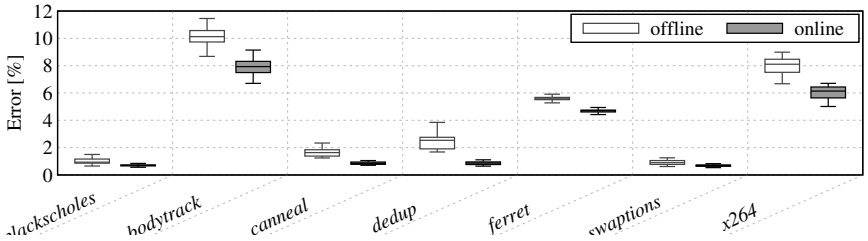


Figure 2.5: *MAPEs on solo runs, using offline (with the least squares algorithm) or online (with the* Model Updater*) estimation. Whiskers depict the minimum and maximum errors; boxes depict 25th, 50th, and 75th percentiles.*

to specific characteristics of these applications; the remainder of this section reports our observations.

The only meaningful throughput metric for *bodytrack* (i.e., frames/s), leads to low measurements: 4 frames/s on average with peaks of 8 frames/s with the highest resource allocation. These low measurements imply that *bodytrack* only completes 3 to 6 frames in the 750 ms sliding window we measure throughput on; this dynamic leads to sizable variations (up to 33 % on average) due to sampling even when the actual throughput varies only slightly. This jitter impairs the accuracy of our model. While using a longer sliding window and allocation period improves accuracy on *bodytrack*, it would reduce responsiveness on applications with varying phases. We found that 750 ms is a good trade-off to support all our benchmarks; application-specific optimizations could further improve accuracy.

x264 shares the low-throughput issue with *bodytrack*, albeit with lesser impact (measurements can be as low as 6 frames/s). Moreover, it shows input-dependent execution phases (see Figure 2.3), which introduce variations even with a stable resource allocation. For this reason, the *Model Updater* needs to adapt to the changing workload, by updating the model parameters, and its prediction error increases during the transition to a different execution phase. We could tune the *forgetting factor* of the *Model Updater* to address this issue; however, a more aggressive forgetting factor would make the *Model Updater* more subject to occasional noise.

ferret exploits pipeline parallelism and it runs with 24 threads: 6 threads for each of the 4 pipeline stages. This is the result of a ques-

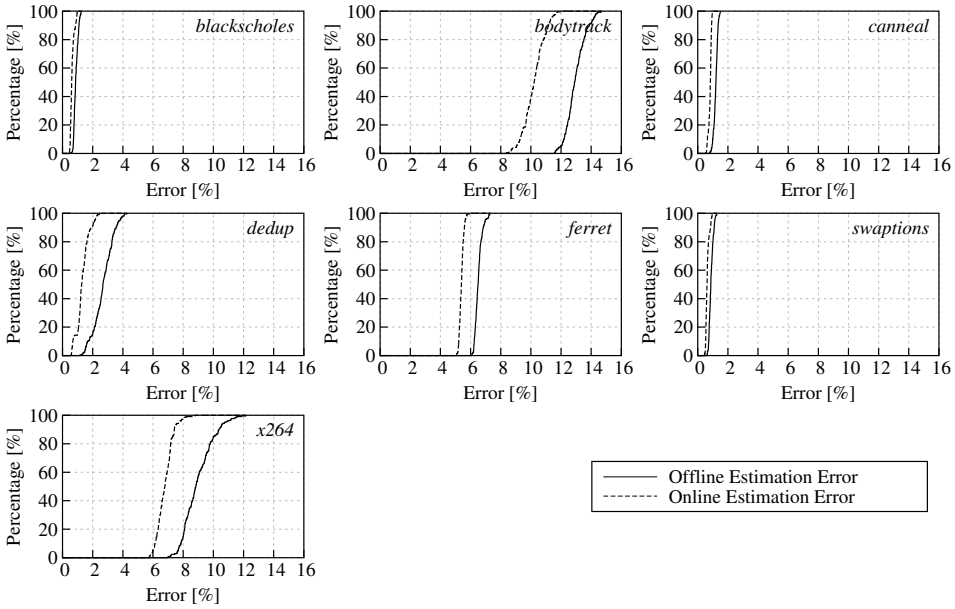


Figure 2.6: *Each plot shows the cumulative distribution function (CDF) of the MAPE for the VM in the label, considering all workloads where it is present (i.e.,* 7×30*). We report results using both offline (least squares) and online (*Model Updater*) estimation.*

tionable design: the application is simple from a developer standpoint, however, it can also be highly inefficient due to load imbalance and I/O bottlenecks [104]. Re-designing the applications following Navarro et al. [104] suggestions would greatly improve the predictability of the application and reduce the prediction error of our resource-performance model.

Prediction Error on Multi-VM Workloads

Since *AutoPro* aims at supporting workload consolidation to maximize node-level utilization, we evaluate the accuracy of our model on dual-VM workloads. Figure 2.6 reports that the prediction error remains low, indicating that our model is robust against performance degradation due to contention on unmanaged resources. These results follow very similar trends to the single-VM workloads (compare to Figure 2.5): the maximum error is below 1 % for three applications (*blackscholes*, *canneal*, *swaptions*) and overall below 12 %, when the

Model Updater is in place. The same considerations of Section 2.4.2 hold, to a greater extent as a result of the contention on unmanaged resources, for the higher error observed with *bodytrack*, *ferret*, and *x264*.

dedup and *x264* are the two applications whose values of MAPE increase the most when shifting from isolation to consolidation; we ascribe this behavior to the highest sensitivity to the contention on unmanaged resources [23].

The low values of the MAPE metric for both single- and multi-VM workloads indicate that our resource-performance model accurately predicts the behavior of the compute-intensive multithreaded applications we employed in these experiments. In addition, the *Model Updater* improves the model accuracy over offline estimates, besides dispensing from the need of profiling applications. The *Model Updater* also makes the model more resilient to the negative effects resulting from the contention on unmanaged resources (e.g., see *x264* in Figure 2.6).

2.4.3 Runtime System Evaluation

To evaluate *AutoPro*, we use a strategy similar to the validation of the resource-performance model. First, Section 2.4.3 shows that *AutoPro* can automatically allocate the right amount of CPU bandwidth to SLO-bound VMs running solo on our host node (see Section 2.4.1 for configuration details). Second, Section 2.4.3 shows that *AutoPro* can both automate CPU provisioning and maximize node-level utilization when managing two SLO-bound VMs colocated with a batch VM on our host node. In addition, Section 2.4.3 analyzes the dynamic behavior of *AutoPro* in one of the experiments of the motivational case study. The results of these experiments shows that:

1. *AutoPro* is a practical tool for automating the allocation of a contended resource (specifically, CPU bandwidth) in a datacenter host node based on performance requirements;

2. *AutoPro* allows the maximization of node-level utilization and it is robust to the performance degradation resulting from the contention on unmanaged resources.

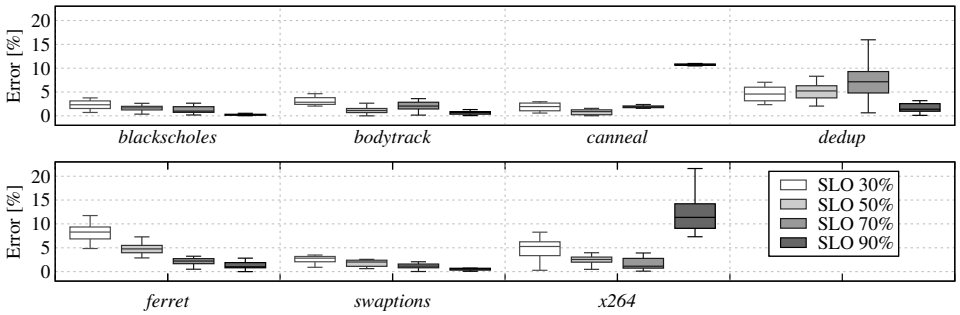


Figure 2.7: *MAPEs on SLO enforcement for single-VMs runs. SLOs range from* 30 % *to* 90 % *of each application's peak performance. Whiskers indicate the minimum and maximum errors; boxes indicate 25th, 50th, and 75th percentiles.*

Each SLO-bound VM executes one of the PARSEC applications we selected, while the batch VM executes *psearchy* (see Section 2.4.1). We set SLOs as a fraction of the average throughput each application achieves in a solo run on our host without any caps on resource usage (from now on, we refer to this value as the application's *reference throughput*).

To evaluate the ability of *AutoPro* to enforce SLOs, we use the same metric we used in our model evaluation, i.e., the mean average percentage error (MAPE). Referring to Equation (2.6), in this case $\widehat{t_w}(k)$ is the actual throughput measurement and $t_w(k)$ is the desired throughput, i.e., the SLO.

To evaluate the ability of *AutoPro* to maximize node-level utilization, we consider how close the average host node CPU utilization during each experiment is to 100 %.

Managing Single-VM Workloads

We evaluate the effectiveness of *AutoPro* in allocating just enough resources to single-VM workloads bound to an SLO ranging from 30 % to 90 % of each PARSEC application's reference throughput, with steps of 20 %; Figure 2.7 shows these results. Most of the errors are (often considerably) below 5 %; we analyze the cases of higher error.

The results for *x264*, *dedup*, and *ferret* present a sensibly higher variance; in these cases, the same application-specific considerations reported in Sections 2.4.2 and 2.4.2 hold.

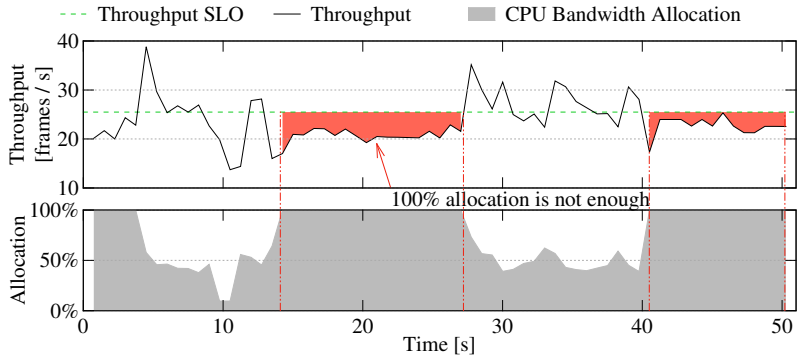


Figure 2.8: *Excerpt from a run of* x264 *bound to a SLO of* 90 % *its average solo throughput. Allocating* 100 % *CPU bandwidth is still not enough to match the SLO during heavier phases.*

Different considerations hold for the higher errors of *canneal* and *x264* when setting the SLO to 90 % of the respective reference throughput. These applications present considerable throughput variations, even with a fixed resource allocation, due to changes in their workload. *canneal* implements simulated annealing and, due to the structure of this algorithm, it performs more frequent exchange operations at the beginning of its execution (i.e., at "high temperature"), thus showing a constantly decreasing throughput, which is measured in exchanges/s. *x264* presents input-dependent execution phases triggered by the different structure of the frames in the incoming video stream [10, 118, 128]. For this reason, these two applications present heavier execution phases for which they cannot sustain 90 % of their reference throughput across the whole execution, even with all the available resources; Figure 2.8 exemplifies this issue for *x264*. The red-shaded areas highlight heavier execution phases during which the VM does not achieve the SLO even though *AutoPro* consistently allocates 100 % CPU bandwidth. This analysis shows that, in these cases, *AutoPro* still works properly; simply, the SLO is not attainable during some execution phases on our host server. In this situation, the provider might simply terminate the VM and report execution failure due to wrong settings or migrate the VM to a more powerful node. While this chapter does not directly deal with VM migration, we are working on a smart VM migration system

able to tell chronic from transient resource shortage and, in the latter case, decide to migrate the VM.

Managing Multi-VM Workloads

To evaluate *AutoPro* in a more comprehensive scenario, we use it to allocate resources to two SLO-bound VMs, running PARSEC applications, while maximizing node-level utilization by provisioning unused CPU bandwidth to a batch VM, running *psearchy*, on a best-effort basis. We run every possible colocation of the seven PARSEC applications, repeating each experiment 30 times and, for each VM, we analyze all the runs that comprise it, using the mean average percentage error (MAPE) metric to evaluate the ability of *AutoPro* to enforce SLOs and evaluating the average host node CPU utilization. For each experiment, we set the SLO of one VM to $x = \{30, 40, 50\}\%$ its reference throughput and the SLO of the colocated VM to $(80 - x)\%$ its reference throughput. Given these SLOs, there is no guarantee that the host node capacity will be enough to provision both VMs with the right amount of CPU bandwidth. Hence, some of the workloads may never have values of MAPEs equal to 0, even if the prediction error for those specific workloads is 0.

Figure 2.9 reports, on these experiments, the distributions of the MAPE for each application executed in a SLO-bound VM. These results show that *AutoPro* meets the three SLO levels with a small error margin (always within 5 % and lower in many cases) for four of the seven PARSEC applications (namely, *blackscholes*, *bodytrack*, *dedup*, and *swaptions*); in this respect, *AutoPro* is competitive with SLO-violation results reported in prior work [125] (see Section 2.5 for additional considerations).

We ascribe the higher (but still below 10 % at the 90th percentile in all cases) error reported for *ferret*, *canneal*, and *x264* to the same issues we discussed earlier in this section. In addition, for what regards *canneal* and *x264*, the effect illustrated in Figure 2.8 has sizable effects at SLO levels lower than 90 %. The reason is that now there are two colocated SLO-bound VMs that contend on the available host node CPU capacity, leading to higher possibility that heavier execution phases lead to resource scarcity. The same possible solution to

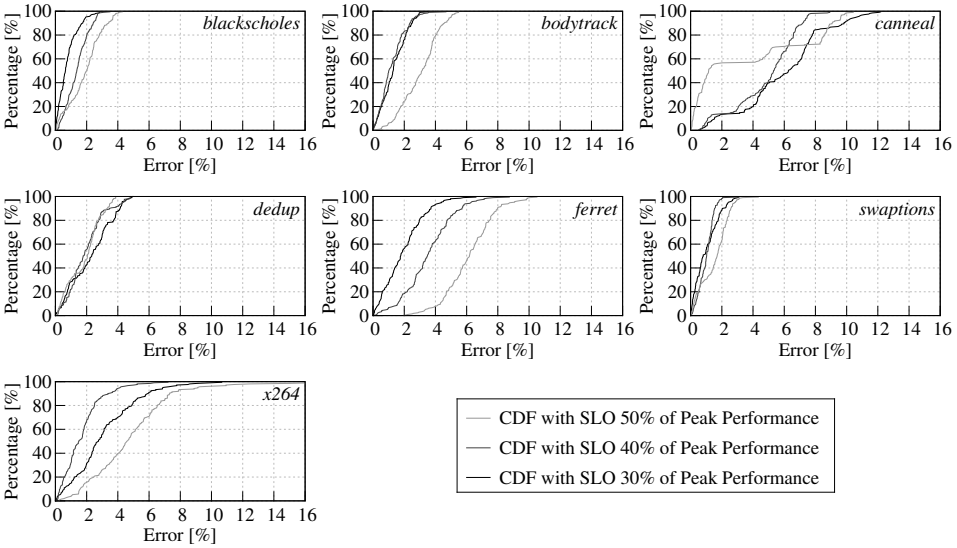


Figure 2.9: *Each plot shows the CDF of the MAPE for the VM in the label, considering all workloads where it is present (i.e.,* 7×30*). In each experiment, one VM has a SLO of* $x = \{30, 40, 50\}\%$ *its reference throughput, as indicated in the legend, while the colocated VM has a SLO of* $(80 - x)\%$ *its reference throughput. All workloads comprise a batch VM treated in a best-effort fashion.*

this issue discussed for the single-VM case (see Section 2.4.3) apply here.

Figure 2.10 reports the CDF of both global and per-VM host node CPU utilization for all the experiments with three colocated VMs (i.e., two SLO-bound and one batch). The plots show that the utilization of the host node CPU is always close to 100 % in all cases for all SLO levels. This result confirms that *AutoPro* can effectively maximize node-level utilization by allocating unused resources to batch workloads on a best-effort basis. The per-VM utilization CDFs show that *AutoPro* enacts consistent allocations across the different experiments. The variations in these values are due to adjustments *AutoPro* makes to respect SLOs despite the workload-dependent negative effects of contention on unmanaged resources.

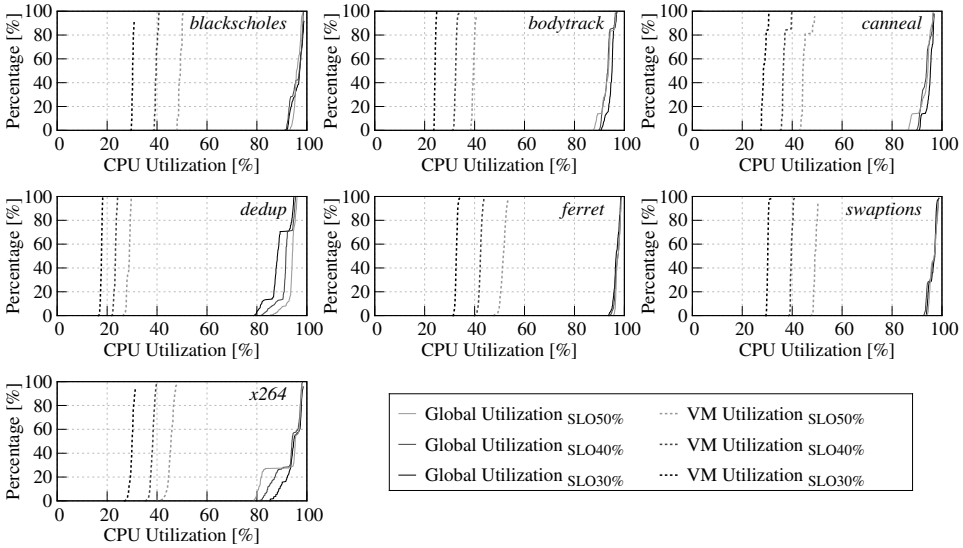


Figure 2.10: *Each plot shows the CDF of the global and per-VM host node CPU utilization registered in the experiments reported in Figure 2.9.*

Dynamic Behavior

We conclude our evaluation by analyzing more in detail scenarios (E) and (F) of our case study (see Section 2.2.1 and Figure 2.1). Figure 2.11 picks the 95th percentile execution of the multi-VM workload composed by *swaptions*, *x264*, and *psearchy* and shows how VM performance and CPU bandwidth allocations vary throughout the experiment. *AutoPro* achieves near optimal allocation for the VM running *swaptions*, which has stable performance, and tracks much better (with respect to the best static allocation) the SLO for the VM running *x264* by provisioning (at high frequency) the CPU bandwidth allocation. Notice that allocating 100 % of the CPU bandwidth does not imply a host node CPU utilization of 100 % due to practical issues (e.g., trapping, context switching, synchronization).

2.5 Related Work

The problem of building virtualization infrastructures able to support auto-scaling of applications and maximize node-level utilization is the natural evolution following the rise of public IaaS and Platform as a

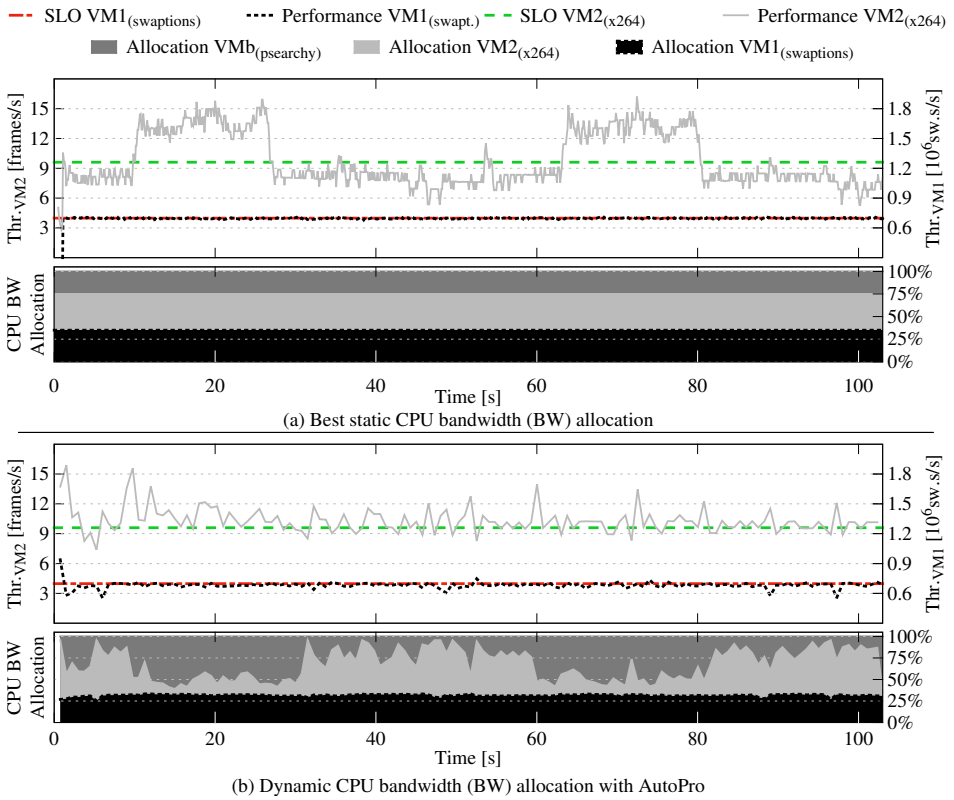


Figure 2.11: *Performance (throughput) and resource (CPU bandwidth) allocation traces for two SLO-bound VMs (VM1 and VM2), running* swaptions *and* x264*, respectively, colocated with a batch VM (VMb), running* psearchy*. The values of MAPE on the SLOs for the two VMs are, respectively,* $\epsilon_1 = 0.3\,\%$ *and* $\epsilon_2 = 27.4\,\%$ *with static allocation and* $\epsilon_1 = 3.2\,\%$ *and* $\epsilon_2 = 7.4\,\%$ *with* AutoPro*.*

Service (PaaS) clouds. This chapter addresses this problem by starting with a clean slate, "reversing" the classic resource allocation paradigm: with *AutoPro*, users specify an application-specific performance metric to state a SLO (e.g., requests/s for web server) and leave the task of determining resource needs to the virtualization infrastructure. In this section we survey, to the best of our knowledge, related works; where possible, we provide qualitative comparisons with *AutoPro* and highlight both strengths and limitations of our work.

Padala et al. [108] proposed *AutoControl*, an auto-scaling solution combining a model estimator and a set of application and node controllers. Similarly to *AutoPro*'s *Model Updater* (see Section 2.3.2), *Au-*

toControl's model estimator captures the relationship between application performance and resource allocations (i.e., the resource-performance model). *AutoControl* employs a set of multi-input multi-output (MIMO) application controllers demanding node controllers to allocate CPU and I/O bandwidth, whereas *AutoPro* exploits single-input single-output (SISO) control targeting the bottleneck resource.

Shen et al. [125] proposed *CloudScale*, an auto-scaling solution to allocate CPU bandwidth and memory, migrate VMs, and save energy through dynamic voltage and frequency scaling (DVFS). *CloudScale* leverages a system called *PRESS* [41] to handle resource demand prediction and prediction error. *PRESS* and *AutoPro*'s *Model Updater* carry on similar duties, as they both predict resource demands. *CloudScale* couples *PRESS* with ad-hoc heuristics to decide resource allocations; instead, *AutoPro* uses *VM Controllers* derived according to control theory from a resource-performance model, coordinated by the resource broker.

We acknowledge that our control strategy is currently limited in the number of resources it handles (the same holds for *CloudScale*, which handles resources disjointly). However, we believe that *AutoPro* can be extended to handle multiple resources by adding an additional layer to identify the bottleneck resource based on utilization, without incurring in the overhead of solving a minimization problem of exponential complexity (as it happens with *AutoControl*).

We argue that our evaluation is more compelling than the one proposed by Padala et al. [108] and Shen et al. [125] for the following reasons. First, we always report the error distributions of statistically relevant experimental results, allowing a clear analysis of these data. *AutoControl* and *CloudScale* only report single runs or average results, leaving much to be guessed. Second, we validate our resource-performance model with random resource allocations both in isolation and consolidation, while *AutoControl* only uses model-driven resource allocations in isolation and the heuristics used in *CloudScale* do not allow a similar validation. Finally, we collect experimental results with a much higher number of multi-VM workloads and SLOs and show that *AutoPro* achieves the additional goal of maximizing resource utilization. Furthermore, our control strategy works at a much higher

frequency (i.e., control period $< 1\,\mathrm{s}$ instead of $> 10\,\mathrm{s}$), giving *AutoPro* the ability of catching short-term trends that go unobserved with *AutoControl* and *CloudScale*. Short-term trends may become more and more important in the future, as predicted by other recent research [107].

Nguyen et al. [106] proposed AGILE, a control system that aims at improving the elasticity of distributed services by automatically scaling the number of VMs in response to load variations, so as to respect SLOs. AGILE uses a wavelets-based model to predict medium-term resource demand based on *resource pressure* (i.e., the ratio of the total resource demand to the total resource allocation) and past SLO violations. While AGILE shares with *AutoPro* the approach of using a model to predict resource demand, AGILE's model would not be applicable in *AutoPro*'s context, where there is no definition of resource pressure, since our workloads use up all their share of the contended resource. Moreover, AGILE adapts by spawning additional VMs, while *AutoPro* adjusts resource allocations of single VMs.

Nathuji et al. [103] proposed *Q-Clouds*, a cloud infrastructure embodying a novel billing solution for public IaaS clouds and a runtime system that aims at mitigating the negative effects of contention on shared hardware resources; the design of *Q-Clouds* resembles that of *AutoControl*. To exploit unused resources, *Q-Clouds* allows the users to state additional SLOs (dubbed *Q-States*) and it proportionally redistributes leftover capacity. Instead, *AutoPro* supports the execution of batch VMs on a best-effort basis and it is robust against the performance degradation these colocated VMs may introduce. *AutoPro*'s resource broker could support an equivalent of *Q-states* by redistributing (part of) the leftover capacity to SLO-bound VMs based on additional SLO levels.

Kocoloski et al. [73] proposed a dual stack virtualization infrastructure to support the consolidation of general-purpose applications and high-performance computing (HPC) applications. They use two hypervisors: KVM for general-purpose applications and *Palacios* for HPC applications. Their approach to achieve isolation is trivial, consisting in partitioning a dual-socket host node between the two hypervisors: KVM manages a non-uniform memory access (NUMA) domain (i.e.,

a multicore with its cache hierarchy, prefetchers, memory controller, and memory banks), while *Palacios* manages another NUMA domain. Instead, *AutoPro* is much finer-grained, as it considers multi-VM workloads sharing a single multicore (i.e., NUMA domain).

ControlWare [140], *METE* [123], and *PTRADE* [52] served as inspiration for sub-systems of *AutoPro*. These systems tackle environments that differ from virtualization infrastructures, making a full-system comparison with *AutoPro* unfeasible; instead, we outline quantitative comparisons between sub-systems. The error distributions for our resource-performance model (see Figure 2.5) display *maximum* values of MAPE in line with the *average* MAPEs of *METE*. The error distributions for *AutoPro* (see Figure 2.7) across a variety of applications with SLO 50 % of the reference throughput are in line with the average error of *PTRADE* on the same applications; in addition, we evaluate a wider range of SLOs.

2.6 Discussion and Future Work

This section expands the discussion to highlight the strengths and limitations of *AutoPro* and point at directions for future work.

2.6.1 Multiple Resources and Shifting Bottlenecks

While the current implementation of *AutoPro* provisions CPU bandwidth and cores and manages compute-intensive multithreaded applications, the architecture we described is general enough to support applications with bottleneck resources other than the CPU; the lone requirement is to use an appropriate resource-performance model.

Our current implementation of *AutoPro* is not designed to manage deployments where different resources alternate as the bottleneck (e.g., a multi-tier web application split into a front-end web/application server and a back-end database management system can alternate compute- and I/O-bound execution phases [108]). In this case, *AutoPro* could leverage different resource-performance models, *VM Controllers* and *Resource Brokers* for the different resources. To support this scheme, we plan to borrow ideas from *Dominant Resource Fair-*

ness [39], and extend the current implementation of the *Resource Broker*.

Some resources are tightly coupled on current multicores (e.g., CPU bandwidth and shared LLC usage) and do not lend themselves well to separate allocation. *AutoPro* already deals with this situation by soft-partitioning the unmanaged resources through adjustments of the bottleneck resource allocation [37, 127]. Though not very efficient, this is a feasible solution to deal with those resources that cannot be easily partitioned on commodity architectures (e.g., shared caches, memory bandwidth).

The obvious (and, in general, more efficient) alternative to soft partitioning is to actually partition and explicitly allocate other resources. While we already discussed possible ways to extend *AutoPro* towards managing multiple resources, we need practical partitioning mechanisms as the enabling technology. For instance, while commodity systems do not expose direct methods to partition the LLC, we are working on shared LLC partitioning by means of page coloring [82]. A preliminary evaluation of *Rainbow* [122], our *colored* page allocator extension for KVM, looks promising and could improve the efficiency of *AutoPro* by eliminating contention on the LLC. Other researchers have recently demonstrated that, with some limitations, it is possible to partition and allocate memory bandwidth [139]. We are planning to build on this research to extend *AutoPro*.

2.6.2 Different Performance Metrics

In this chapter, we present and evaluate a resource-performance model for throughput-driven applications; this model is applicable to any application that allows expressing a performance service-level objective (SLO) as a throughput measurement; the seven application we use in our evaluation show a varied sample of such applications. While throughput is an important metric, it does not fully capture the performance concerns of some applications; particularly, request-serving applications may be bound to a SLO on tail latency [25].

While the overall infrastructure of *AutoPro* is applicable to different performance metrics, supporting SLOs that are not expressed as a throughput target requires a different resource-performance model and,

possibly, a different structure for the VM controllers. In particular, since non-linearity is stronger in the resource allocation to tail latency relationship than in the resource allocation to throughput relationship, our linear model would probably not be accurate if used to deal with latency-driven applications. Moreover, the monitoring system needs to be adapted to report the latency at a given percentile instead of the average throughput over a time window. We are currently investigating these issues, which we believe are an important direction for future work.

An additional challenge that needs to be addressed to extend *AutoPro* to deal with latency-driven application lies at the resource-provisioning level. In our evaluation, we use a 750 ms control period and do not go faster due to the increasing allocation inaccuracy of the *cpu* subsystem of *cgroups* with decreasing control periods. While this control period is already much finer grained than previous works (see Section 2.5) and it is fast enough to respond to execution phases of our throughput-driven applications, it would probably be too slow to provide the desired quality of service to applications bound to SLOs of tens of milliseconds (or less) on tail latency [78].

2.6.3 Scalability to Manycores and Large-Scale Installations

Most of the PARSEC applications show near-optimal scalability up to the core count of our reference host node [118]. Our plan to deal with the scalability issues that applications might have with higher core counts (i.e., on large symmetric multiprocessing systems or manycores) is to provision CPU bandwidth and cores at the same time. *AutoPro* could partition the available cores into islands (i.e., groups of cores), cluster applications with similar scalability (i.e., speedup over number of cores), assign each cluster to an island of the appropriate size, and allocate CPU bandwidth within each island.

We believe that the architecture of *AutoPro* can be extended to manage large-scale installations and distributed applications, such as those leveraging the MapReduce computation model [26]. Our efficient user-space implementation of the Application Heartbeats API [51, 127] already supports distributed applications: tasks belonging to the same distributed applications share a multicast address and keep application-

specific throughput measurements up to date through asynchronous messages.

2.6.4 Billing Scheme

A possible concern about *AutoPro* is its robustness to malicious users that make their VMs report falsely low performance with the goal of fooling the resource-performance model and get additional resources. While this concern is legitimate, *AutoPro* is immune to such attack if the billing scheme used by the provider is based on resource consumption [1]. Using such a billing scheme with *AutoPro* arises two issues, both of which are practically solvable:

1. it is hard for users to make a good cost estimate, as they do not explicitly rent resources;
2. responding to performance degradation through soft-partitioning (see Section 2.6.1) increases resource usage, possibly leading to unfair billing.

To solve the first issue, the provider could ask users to state an upper bound to the resources they are willing to pay for and define an agreement on how to manage violations of this bound. To deal with the second issue, the provider could employ techniques to limit performance degradation [42, 92] and, when this is not enough, to estimate the extent of the performance degradation [17, 32] and scale the bill accordingly.

2.7 Concluding Remarks

Our experimental campaign (Section 2.4) shows with statistically relevant data that *AutoPro* achieves its two major goals:

1. extending IaaS clouds with automated fine-grained resource allocation, asking users to just state the performance level they require for their VMs;
2. enabling the cloud provider to safely share hardware resources among the VMs, allowing to optimize datacenter utilization (thus reducing the total cost of ownership) by maximizing node-level utilization.

While the current implementation of *AutoPro* leaves few open issues that we are still investigating, the discussion of Section 2.6 outlines concrete solutions to fill these gaps.

We believe that *AutoPro*, by allowing to safely share hardware resources and by operating at very fine scale (i.e., with sub-second control period and fine allocation granularity) is well suited for supporting better management of modern dynamic deployments; we hope that this proof of concept will help to improve the efficiency of our datacenters.

CHAPTER 3

Rubik[1]

Latency-critical workloads (e.g., web search), common in datacenters, require stable tail (e.g., 95^{th} percentile) latencies of a few milliseconds. Due to the difficulties in managing such strict performance constraints with traditional techniques, these workloads cause low datacenter utilization, wasting billions of dollars in energy and equipment. Recently, researchers worked on achieving high utilization by using memory hierarchy partitioning to safely colocate batch applications (e.g., MapReduce) with latency-critical applications. While this approach successfully increases utilization, it uses conservative power management to maintain tail latency and cannot use prior datacenter power management schemes, which improve energy efficiency at low utilization.

This chapter proposes Rubik, a power management scheme for systems with partitioned caches and memories. Rubik relies on a novel, analytical DVFS scheme that adjusts frequencies at sub-millisecond

[1]The work presented in this chapter was done in collaboration with Harshad Kasture, Daniel Sanchez, and Nathan Beckmann, while the author was visiting Prof. Daniel Sanchez's group at MIT CSAIL. This chapter was adapted from a paper that, at the time of writing, is under submission to ISCA 2015 [67].

granularity. This technique improves the efficiency of latency-critical workloads without degrading their tail latency and allows more aggressive colocation of latency-critical and batch workloads than memory partitioning alone. Rubik policies build on this scheme to maximize either server-level or full datacenter efficiency. Rubik reduces both datacenter power and number of servers by 40% vs. a datacenter that segregates latency-critical and batch work.

3.1 Introduction

User-facing, latency-critical applications are common in current datacenters and pose new challenges for system designers, mainly due to one peculiar characteristic: *tail latency*, not average latency, determines the performance of these applications; for example, web search leaf nodes must provide 99^{th} percentile latencies of a few milliseconds [25]. Since latency (particularly, tail latency) rapidly increases with load (as the data presented in Figure 1.8, Section 1.3 exemplify), the servers running latency-critical workloads are kept lightly loaded to maintain tail latency. Low load causes poor datacenter utilization, typically 5–30% [6, 28, 95], wasting billions of dollars in equipment and, since systems are not energy proportional, terawatt-hours of energy annually [6].

Ideally, datacenter efficiency would be improved by running *batch applications* (e.g., MapReduce) during idle periods, taking advantage of the fact that batch applications only desire high throughput, and can yield to latency-critical applications as needed to maintain tail latency. Unfortunately, colocating batch and latency-critical applications is not possible in current multicore servers, due to uncontrolled sharing of memory system resources, such as cache capacity and memory bandwidth. Uncontrolled sharing exposes the applications to the *inertia* [66] of shared resources, i.e., the transient performance degradation resulting from interference on a shared resource. For example, since the last-level cache (LLC) can take tens of milliseconds to fill, if a batch application steals LLC capacity from a latency-critical application during an idle period, it can result in QoS violations for a latency-critical application as its data is brought back into the cache. Consequently, recent work has proposed hardware techniques to parti-

tion and share these resources in a controlled way to safely mix batch and latency-critical applications [66, 79, 109, 123].

While such schemes improve datacenter utilization, they ignore another critical shared resource: power. Prior work has proposed power management schemes to improve average performance or energy efficiency in highly utilized systems [38, 53, 110, 112], but these schemes introduce variability that significantly hurts tail latency [64, 85]. Within the datacenter, other prior work [84, 85, 94, 95, 134] improves the energy proportionality of current, low-utilization systems. However, with hardware support for colocation, provisioning fewer highly utilized systems is more efficient. Unfortunately, designing datacenter systems for low utilization is inherently inefficient: fixed costs dominate idle power (e.g., power supply, fans, DRAM refresh and LLC leakage), and deep sleep modes that rely on power-gating most components, such as deep C-states [48], have high transition latencies and often flush the LLC, which causes unacceptably high tail latencies [66] (see Section 1.2.2). Additionally, even if we could design energy-proportional systems [95], low utilization uses hardware inefficiently. Moreover, prior datacenter power management schemes adapt slowly and conservatively to maintain tail latency [85]. Adapting at low frequency fine-grained management, which is especially useful on forthcoming systems with on-chip regulators [18, 70] that allow dynamic voltage and frequency scaling (DVFS) at sub-µs latencies.

Prior work has been limited by the low performance of DVFS on current commodity systems, which can take milliseconds to change frequency. Such delays preclude using fine-grain power management to guarantee QoS. Instead, forthcoming systems support much faster frequency changes, taking as little as 1 µs to change core frequency (Section 3.2). Since cores constitute a major component of total system power, this capability allows for fine-grain adaptation of core frequency to meet QoS requirements while improving efficiency.

We propose *Rubik*, a power management scheme for datacenters featuring future multicores with partitionable memory systems and fine-grained DVFS. Rubik runs mixes of latency-critical and batch applications *within a single server* at high utilization, high energy efficiency, and maintains strict tail latency guarantees. Rubik relies on a

novel, analytical DVFS control scheme that uses request-level statistics to scale voltage and frequency at a sub-millisecond scale, orders of magnitude faster than the state-of-the-art [85], without violating tail latency. Besides improving efficiency, this technique allows more aggressive sharing of hardware resources among applications than prior colocation schemes [27, 66, 92, 137]. We use this technique to design three power management schemes that progressively build on each other:

- RubikLC (Section 3.4) reduces power without degrading tail latency on mixes of latency-critical applications.
- RubikSC (Section 3.5) extends RubikLC to allow mixes of batch and latency-critical applications, optimizing batch throughput-per-watt without degrading tail latency.
- RubikDC (Section 3.6) extends RubikSC to the datacenter, optimizing *total datacenter power* while meeting global tail latency and batch throughput requirements.

Rubik uses an empirically validated power model (Section 3.7) to make full-system efficiency tradeoffs. Rubik requires minimal hardware support and imposes negligible software management overheads. We evaluate Rubik using microarchitectural simulation of latency-critical and batch workloads, and show that it vastly improves energy efficiency (Section 3.9):

- RubikLC reduces active core power by up to 40%, and approaches the efficiency of a dynamic oracle scheme that minimizes power while maintaining tail latency;
- RubikSC improves batch throughput-per-watt by 30%; and
- RubikDC uses 40% less power and fewer servers than a datacenter that does not colocate latency-critical and batch work.

3.2 Background and Related Work

Large-scale online services (e.g., web search) have strict performance requirements that, with traditional management techniques, limit utilization and energy efficiency. Data is spread among many nodes, and

hundreds to thousands of nodes collaborate in serving each request, so the few slowest responses determine overall latency. Thus, single-node latencies must be small, tightly distributed, and uniform across the datacenter. Responsive online services must achieve end-to-end latencies of about 100 ms to provide an acceptable quality of service to the users [25, 120]. This end-to-end requirement translates into individual nodes having to provide *tail latencies* (e.g., 95^{th} or 99^{th} percentile latency) in the millisecond range [25]. Moreover, since tail latency increases quickly with load, nodes run at low utilization [66, 85] to avoid large queuing delays and handle traffic spikes gracefully.

These strict requirements preclude conventional colocation and dynamic power management techniques, as both cause large performance variability and hurt tail latency. We now discuss prior work that attacks these problems.

3.2.1 Improving Utilization Through Colocation

Recent work has proposed software and hardware techniques to provide *quality of service* (QoS) on shared memory resources. These techniques enable safe colocation, but need stable frequencies to maintain tail latency, and are incompatible with current dynamic power management schemes.

On the one hand, software-only resource managers perform QoS-aware scheduling in datacenters [27, 28, 92, 137]. They detect interference empirically, and throttle or migrate batch applications that cause too much degradation. These schemes work with current multicores, but they react to interference instead of preventing it. Lacking hardware partitioning, they cannot provide strict guarantees on tail latency and colocate conservatively, leaving memory resources underutilized.

On the other hand, prior work has proposed hardware to explicitly partition shared memory resources (e.g., cache capacity and memory bandwidth) [13, 20, 45, 60, 69, 71, 88, 101, 105, 109, 115], and policies to dynamically size partitions that guarantee performance [22, 33, 46, 57, 79, 100, 109, 123, 141]. These schemes only guarantee long-term average performance (e.g., IPC or QPS over a few seconds), instead of short-term performance (e.g., bounding the degradation of a 1 ms request), and cannot guarantee tail latency.

Inertia

Most dynamic partitioning schemes cannot guarantee tail latency because they reconfigure infrequently and assume instant transitions between steady states, *ignoring transient behavior* [66]. For example, cache partitioning policies [109, 123] periodically reallocate space among cores. When an application is assigned extra space, it takes tens of milliseconds to fill it and reap the benefits of a larger allocation. This *performance inertia* cannot be ignored in latency-critical applications. Intuitively, inertia depends on the amount of state: multi-megabyte last-level caches induce large inertia, while memory bandwidth has little state and thus little inertia.

A simple way to avoid the harmful effects of inertia is to statically size partitions. Alternatively, Ubik [66] observes that cache inertia is highly predictable, and leverages this to *predict* transients and dynamically partition the cache to eliminate their effect on tail latency. This approach increases cache utilization over static partitioning and improves batch performance without degrading tail latency.

3.2.2 Improving Efficiency Through Power Management

Dynamic power management techniques reduce idle power through clock- and power-gating, and lower active power through voltage and frequency scaling (DVFS). Prior work has comprehensively studied how to manage core sleep states and DVFS to improve efficiency in multicores running multithreaded [21, 53, 56] and multiprogrammed [80, 119] batch applications, and in interactive systems [38, 86, 136, 138]. However, recent work has shown that these techniques are inadequate for latency-critical applications [64, 84, 85, 96], because they ignore millisecond-level inertia, thus hurting tail latency. Recent work proposed techniques that respect tail latency [83–85] on datacenter servers at low utilization and without colocation. While these techniques are useful when low utilization is unavoidable, Rubik uses colocation to achieve high utilization, leading to greater advantages in terms of efficiency.

Considering the inertia of different power-saving techniques reveals which ones are compatible with latency-critical applications. *Shallow sleep states* clock- or power-gate portions of the core and have short

wakeup latencies, but yield limited power savings (see Figure 1.2, Section 1.3). By contrast, *deep sleep states* power-gate large portions of the chip, but have long wakeup latencies and flush significant microarchitectural state (e.g., the last-level cache, which takes tens of milliseconds to warm up [48, 66]). Early work in datacenter power management focused on using coordinated deep sleep modes with reduced transition times in all system components [94, 95, 134]. However, Meisner et al. [96] show that latency-critical applications cannot use even these sophisticated techniques, because they run frequent, short requests and cannot tolerate batching. Kanev et al. [64] show that, in conventional multicores, sleep states barely help Google's applications, and can severely degrade tail latency. Instead, both papers advocate using techniques that reduce active power.

DVFS is the most attractive approach to reduce active power: reducing frequencies allows lower voltages, yielding superlinear power savings. However, off-chip regulators can take tens to hundreds of microseconds to adjust voltage [70, 96]. Fortunately, recent on-chip switching regulators [18, 70] have sub-µs delays (e.g., 100–500 ns on Haswell [18]), allowing for extremely fast per-core DVFS. DVFS techniques often focus on core DVFS, though recent work has studied DVFS [29, 30] and low-power active modes [24, 87, 133] for main memory. At high utilization, cores consume most of the power, so we focus on core DVFS and leave extending Rubik to other resources to future work.

Prior DVFS policies run at the firmware or operating system (OS) level. Firmware-level policies (e.g., Intel's Turbo Boost [48, 84]) rely on algorithms that measure and predict future activity. This approach is inadequate for latency-critical applications, because it is oblivious of performance requirements. For example, Kanev et al. [64] report Turbo Boost to cause extreme tail latency variability. While current OS-level policies (e.g., *cpufreq governors* in Linux) rely on similar algorithms, they can more easily be made aware of performance bounds. Therefore, we focus on OS-level, software-managed DVFS.

3.2.3 Using DVFS with Latency-Critical Applications

In designing Rubik, we found a key challenge was predicting how frequency changes affect tail latency. This prediction is hard because tail latency depends on the complex interplay of arrival, queuing, and service time distributions. Prior work has avoided this problem in two ways.

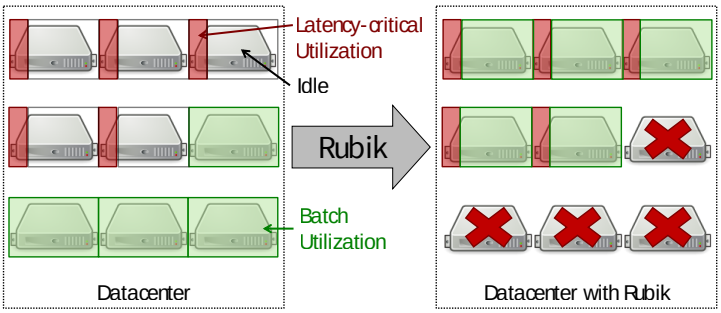
First, schemes that optimize for responsiveness in embedded systems, such as PACE [86, 136] and Grace [138], try to satisfy each request by a given deadline, and do not consider queuing time. Ignoring queuing works well in interactive systems that run one task at a time, but is not applicable to the datacenter, where queuing is significant and unavoidable.

Second, PEGASUS [85] takes a feedback-based approach. PEGASUS periodically measures tail latency, adjusts frequency, and iterates to keep the tail within a given target. Relying exclusively on feedback is simple, but it is unresponsive to traffic bursts, since reliably measuring tail latency takes time. Thus, purely feedback-based approaches can adapt to the long-term characteristics of the request stream, but cannot respond to short-term variability. Instead, Rubik employs a combination of short-term, analytical adaptation and long-term, feedback-based adaptation. These complementary techniques let Rubik handle request bursts without degrading tail latency.

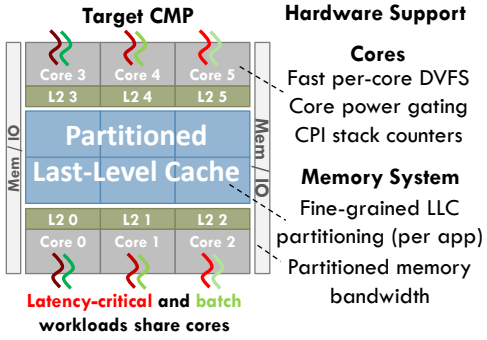
Finally, prior latency-critical DVFS schemes target current, low-utilization datacenters, where systems are not power-constrained, and play within the limited power management interfaces of current hardware, which limits their reconfiguration frequency. By contrast, power is a limited resource in highly utilized systems, so Rubik manages it as a scarce resource and at fine granularity.

3.3 Rubik Overview

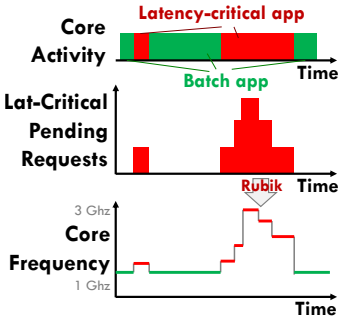
In current datacenters, latency-critical applications are segregated from batch applications to avoid interference. Although latency-critical nodes run at low utilization [25], they use relatively high frequencies to meet their latency targets, wasting additional power. In contrast, as Figure 3.1a illustrates, Rubik colocates latency-critical and batch



(a) *Rubik in the datacenter*



(b) *System architecture*



(c) *Maintaining the tail with DVFS*

Figure 3.1: *Rubik uses fine-grained DVFS to colocate batch and latency-critical applications without degrading tail latency. (a) By increasing server utilization and reducing power, Rubik improves datacenter efficiency and reduces provisioned servers. (b) Rubik requires modest hardware extensions over commodity systems. (c) Rubik adjusts core frequency on each request arrival and completion to enforce the tail latency bound.*

applications, increasing node utilization, and uses fine-grain DVFS to lower core frequency whenever possible, reducing power without degrading latency. Rubik improves the efficiency of each node within the datacenter and requires fewer machines to perform the same amount of work.

As shown in Figure 3.1b, Rubik partitions shared memory system resources (LLC capacity [66] and memory bandwidth [101]) among latency-critical and batch applications. In contrast to prior colocation schemes, which dedicate cores to latency-critical applications [66, 92], in Rubik batch and latency-critical threads share cores. Latency-critical threads always have priority over batch threads. To support

sharing cores, Rubik partitions the LLC *by application* (rather than by core [46, 57, 66, 79, 100, 109, 123]) to protect the working set of latency-critical applications. Rubik supports as many latency-critical applications as can fit in the LLC; we use one per core in our evaluation. Rubik uses Vantage [115] cache partitioning to maintain high associativity and support a large number of partitions cheaply.

Since Rubik time-multiplexes latency-critical and batch applications, core microarchitectural state is also a shared resource. Fortunately, this state is small enough that DVFS can compensate for its inertia. For example, private caches (L1s and L2) can be refilled from a "warm" LLC in microseconds, instead of the milliseconds needed to refill the LLC from main memory. Other state (branch predictors, registers, TLBs) has similarly low inertia. Thus *given a warm LLC partition*, judicious DVFS can maintain tail latency.

Rubik adjusts core frequency when requests arrive or complete (Figure 3.1c) to maintain tail latency efficiently (RubikLC, Section 3.4). When all requests in the queue have been served, Rubik yields the core back to batch applications, setting the frequency that optimizes throughput-per-watt for the node (RubikSC, Section 3.5) or datacenter (RubikDC, Section 3.6).

3.4 RubikLC: Fast DVFS for Latency-Critical Applications

Rubik Latency-Critical (RubikLC) manages core frequencies at fine granularity to minimize power on latency-critical applications. RubikLC does not address colocation with batch applications or power caps—we will extend Rubik to tackle this issue in Section 3.5.

RubikLC must adjust frequency without violating the tail latency bound. As discussed in Section 3.2.3, the main difficulty in achieving this lies in computing the appropriate frequency in the presence of queued requests. PEGASUS [85] uses a feedback-based approach to sidestep this issue. Because reliably measuring tail latency takes time, this is slow, and PEGASUS only adjusts frequencies every 5 seconds. In contrast, RubikLC adjusts frequencies at the sub-millisecond level when requests arrive or complete, exploiting short-term workload variability to set the right frequency. This capability is made possible

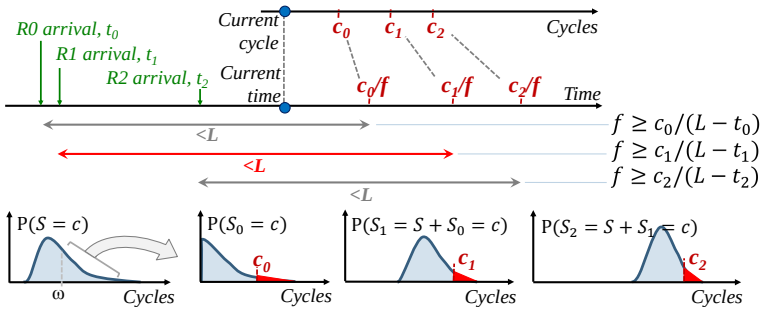


Figure 3.2: *RubikLC example with three requests and no memory-bound cycles: cycles/seconds timelines, frequency constraints to meet the tail latency (L), and probability distributions of service cycles (S), cycles to serve the running request (S_0) and the queued requests (S_1 and S_2), used to compute constraints.*

by exploiting statistical information about the queued and previously served requests.

RubikLC combines two forms of control: fast, fine-grained analytical modeling; and slower, coarse-grain feedback control for fine-tuning. RubikLC's analytical model uses the *current system state* (service time distribution and arrival times of current requests) to find the lowest frequency that meets tail latency. By operating in this way, RubikLC adapts to the arrival process implicitly, by immediately reacting to state changes. While, in theory, modeling the arrival process of the requests would allow a more precise model, it would also make the problem intractable in an analytical form. Since Rubik needs to run online, the model makes conservative approximations, so it achieves a tail latency 2–15% lower than the bound. A feedback controller then observes these differences and tunes the model to converge to the bound. Thanks to this approach, Rubik approaches the performance of a dynamic oracle with perfect future knowledge (Section 3.9.1). We first focus on the analytical aspects of RubikLC, and then show an efficient implementation.

3.4.1 Fast Analytical Frequency Control

Figure 3.2 shows a concrete example of RubikLC's operation. In the example represented in the figure, the application has received three requests, R0, R1, and R2, which arrived at time t_0, t_1, and t_2. The

application is currently processing R0, and has spent ω cycles doing so; R1 and R2 are queued. For this example, assume that requests are not memory-bound, so performance scales linearly with frequency. Figure 3.2 shows that RubikLC spans two parallel timelines, measured in time (i.e., seconds) and core cycles. The tail latency bound is given in time, but requests complete after some number of core cycles. Core frequency (f) connects the two. Our goal is to set f to meet the tail with minimal power.

RubikLC achieves this goal by finding the lowest f that satisfies the tail latency bound L for the current requests. This bound is specified as a percentile; in this example, 95% of requests must be served by $L = 2\,\text{ms}$. RubikLC exploits the probabilistic nature of the problem as follows. First, RubikLC treats the completion *cycle* for each request Ri as a random variable, S_i, with probability distribution $\text{P}[S_i = c]$. Figure 3.2 shows these distributions for R0, R1, and R2 (we discuss how to compute them later). Second, RubikLC finds the tail completion cycle of each request, c_i (the 95^{th} percentile of each $\text{P}[S_i = c]$), shown in red in Figure 3.2. The timelines in Figure 3.2 shows how frequency scaling maps each c_i in cycles to c_i/f in time. Request Ri has already spent t_i time in the system, and the 95^{th} percentile will be served by time c_i/f. Satisfying the tail bound for request Ri requires $t_i + c_i/f \leq L$, so to satisfy *all* current requests:

$$f \geq \max_{i=0\ldots N} \frac{c_i}{L - t_i} \tag{3.1}$$

In this example, request R1 has the most stringent constraint—the longest time between t_1 and c_1/f—and sets the frequency. RubikLC computes f from Equation (3.1) each time a request arrives or completes, quickly adapting to changing conditions.

Notice that the last request in the queue often does not set the frequency, for two main reasons. First, later requests arrived more recently (e.g., $t_N \approx 0$), so they have more headroom than earlier requests (e.g., $t_0 \approx L$). Second, the completion time of queued requests often becomes more tightly distributed the longer the queue length, which shortens the tail of their distributions (e.g., compare $\text{P}[S_1 = c]$ and $\text{P}[S_2 = c]$ in Figure 3.2).

Computing the distributions

Rubik computes the completion cycle distributions by assuming the work for each request is drawn independently from a single distribution, $\mathrm{P}[S = c]$. S gives how many cycles it takes to process one request, not including queueing time. Independence is a reasonable assumption for datacenter nodes, because each node serves requests from many users, and caches serve repeated requests before they reach leaf nodes, reducing temporal correlations [8].

The completion time distribution of the current request, $\mathrm{P}[S_0 = c]$, is computed from S using conditional probability. From above, R0 has been processing for ω cycles and has not yet completed. Thus its completion time is distributed as:

$$\mathrm{P}[S_0 = c] = \mathrm{P}[S = c + \omega | S > \omega] = \frac{\mathrm{P}[S = c + \omega]}{\mathrm{P}[S > \omega]}$$

This scales and shifts $\mathrm{P}[S = c]$ at ω, as shown in Figure 3.2.

From S_0 and S, we can derive the completion time distributions of all queued requests. In order to service the i^{th} request, we must first service the $i - 1$ requests preceding it. Hence, $S_1 = S_0 + S$, and $S_2 = S_1 + S = S_0 + S + S$, and in general $S_i = S_0 + \sum_{j=1}^{i} S$. Using independence, each S_i is distributed according to the *convolution* $(*)$ of S and S_{i-1} [44]. Thus if $\mathrm{P}_X(x) = \mathrm{P}[X = x]$:

$$\mathrm{P}_{S_i} = \mathrm{P}_{S_{i-1}} * \mathrm{P}_S = \mathrm{P}_{S_0} * \overbrace{\mathrm{P}_S * \ldots * \mathrm{P}_S}^{i \text{ times}}$$

Core DVFS and memory

Core frequency does not affect the time spent due to stalls on LLC and main memory accesses, limiting the impact of core DVFS. Prior work has shown that using additional performance counters, one can produce CPI stacks that separate compute and memory-bound cycles, even for complex cores that overlap multiple memory accesses [35]. Prior work has also applied CPI stacks to perform memory-aware DVFS with throughput applications [34, 68, 99, 113]. RubikLC extends this to avoid tail latency violations: it profiles the probability distributions of per-request *compute cycles*, $\mathrm{P}[C = c]$, and *memory-bound times*, $\mathrm{P}[M = t]$.

Work per request (in cycles) is the sum of these two random variables at the current frequency: $S = C + Mf$. Computing the lowest acceptable frequency exactly would require considering the joint distribution of C and M. RubikLC, instead, makes the conservative approximation that the tail of S is no better than the combination of the tails of C and M (triangle inequality). So for each request, RubikLC computes the tails of each distribution C_i and M_i, following the procedure for S_i as discussed above. This procedure yields tail compute cycles c_i and tail memory times m_i until completion of request Ri. The m_i values are a fixed cost that DVFS cannot compensate for, so Equation (3.1) becomes:

$$f \geq \max_{i=0\ldots N} \frac{c_i}{L - (t_i + m_i)} \tag{3.2}$$

3.4.2 RubikLC Implementation

In this section, we give details about the salient aspects of the implementation of RubikLC, which enable us to exploit the statistical model described in Section 3.4.1.

Target tail tables

Computing the c_i and m_i percentiles from scratch on each frequency adjustment would be very expensive, but fortunately they can be precomputed. Periodically, the runtime updates the service cycle and time distributions, performs the convolutions, and fills in the c_i and m_i values in a *target tail tables*. Each row has the c_i and m_i values for selected quantiles of the service time distribution (we use octiles in our implementation). On each request arrival and completion, RubikLC picks the appropriate row and computes the minimum f. Thanks to this approach, computing each constraint requires few instructions (Equation (3.2)), so updates take negligible time and Rubik can take advantage of this statistical analysis to take decisions online.

Large queues

While, in theory, the number of queued requests is unbounded, in practice we rarely observe more than 4–10 queued requests in the applications we evaluate. Still large queues could build up with lax tail

bounds; to support these rare cases with our method based on convolution (as explained in the previous paragraph), we would need to keep long target tail tables, up to large queue size. Fortunately, by Lyapunov's Central Limit Theorem [15], at large i, $\mathrm{P}[S_i = c]$ converges to a Gaussian distribution with mean $\mathrm{E}[S_0] + i \cdot \mathrm{E}[S]$ and variance of $\mathrm{var}[S]$. By precomputing the tail value for the zero-centered Gaussian with variance of $\mathrm{var}[S]$, each c_i and m_i for large i can be computed by adding the mean. We use this formulation for $i \geq 16$, avoiding long tables.

Overhead

Every 100 ms in our implementation, RubikLC updates the core and memory-bound service distributions and uses them to compute the target tail tables. We use 128-bucket distributions, and use FFTs to accelerate convolutions: we transform the source distributions and perform successive convolutions in the frequency domain (where they are point-wise multiplications). Each update of the target tail tables takes 0.2 ms, resulting in a 0.2% overhead.

Feedback-based fine-tuning

RubikLC as described so far will satisfy the desired tail latency provided the available frequencies allow it to. However, since its estimates are conservative, it may waste power by lowering tail latency unnecessarily. To improve accuracy and efficiency, we use a simple proportional-integral feedback controller [131] that observes the difference between the measured and predicted tail latencies over a rolling 1-second window, and adjusts RubikLC's internal tail latency target. This controller performs minor adjustments, as the analytical model is typically only a few percentage points away.

3.5 RubikSC: DVFS for Efficient Colocation

Rubik Single-Computer (RubikSC) extends RubikLC to colocate batch and latency-critical applications. As discussed in Section 3.3 (see Figure 3.1), batch applications share cores with latency-critical applications and run when latency-critical applications have no work to

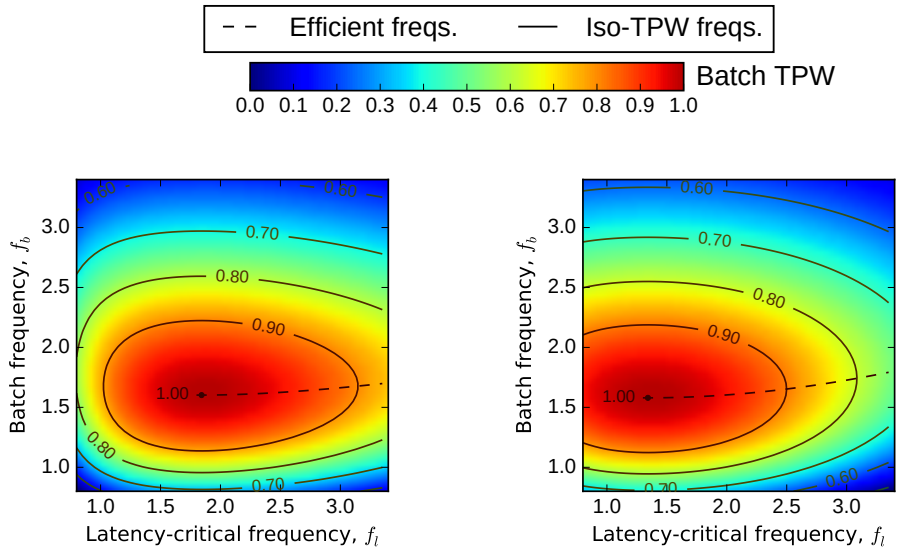
do. RubikSC chooses core frequencies to maximize the system's batch throughput-per-watt (TPW), while staying within chip thermal design power (TDP) and maintaining tail latencies.

Effect of core sharing on optimal frequencies

In RubikLC, cores sit idle when latency-critical applications run out of work, so using the lowest acceptable frequency minimizes power. However, when cores are shared with batch applications, running the latency-critical applications faster than needed can be beneficial, because spending power to accelerate the latency-critical application frees time for the batch application. In this way, RubikSC can improve batch TPW by, for example, executing the same batch work at a lower frequency.

RubikSC's challenge is to navigate these tradeoffs and find the optimal operating frequencies. The two main tradeoffs are how memory-bound applications are, and how often the latency-critical application needs to run. Figure 3.3 illustrates the first tradeoff by showing contour plots of batch throughput-per-watt (normalized within each system) vs. latency-critical (x-axis) and batch (y-axis) frequencies in two scenarios. Compute-bound applications (Figure 3.3a) respond well to DVFS, so the optimal operating point lies at relatively high frequencies. In contrast, the throughput of memory-bound applications (Figure 3.3b) is not affected by DVFS as much, so the optimal operating point shifts to lower frequencies. In this example, the latency-critical application is memory-bound, so the optimal latency-critical frequency is lower, while the optimal batch frequency is almost the same as in Figure 3.3a.

Higher latency-critical core utilization shifts the optimal operating point to higher frequencies. To see why, suppose 90% of cycles are spent servicing requests. In this case, a small boost in latency-critical frequency that reduces this to 80% *doubles* batch throughput. Since so little time is available for batch work, it is also sensible to run at high batch frequencies to amortize latency-critical power. Thus, batch and latency-critical frequencies increase with latency-critical load. However, higher frequencies are not a perfect solution—in absolute terms, TPW is lower under high load than low load.



(a) *Both compute-bound*

(b) *Latency-critical memory-bound*

Figure 3.3: *Batch throughput-per-watt (normalized) as a function of batch and latency-critical frequencies. The most efficient frequencies change with system configuration. If tail latency bounds force high latency-critical frequencies, the most efficient batch frequency lies on the dashed line.*

Using the optimal frequency pair is not always possible, since RubikLC may require a higher latency-critical frequency to meet the tail latency bound. In that case, dashed lines in Figure 3.3 show the most efficient batch frequency when the tail bound forces a higher latency-critical frequency. Take, for instance, Figure 3.3b, where $f_l \approx 1.3\,\text{GHz}$ and $f_b \approx 1.6\,\text{GHz}$ maximize batch TPW. If the latency-critical application must run at $f_l \geq 2.4\,\text{GHz}$, the dashed line at 2.4 GHz tells us that $f_b \approx 1.7\,\text{GHz}$ maximizes TPW under that constraint.

Analytical formulation

Suppose one batch and one latency-critical application share a core, running at frequencies f_l and f_b. The latency-critical application is active for a fraction u of the time, and the batch application is active for the remainder, $1 - u$. While active, the batch application gets throughput T. If the batch and latency-critical applications have active power P_l and P_l, then batch TPW is:

$$TPW = \frac{(1-u)\cdot T}{u \cdot P_l + (1-u)\cdot P_b} \tag{3.3}$$

Frequency scaling affects u, T, and powers P_l and P_b. If a fraction m of cycles stall on memory at nominal frequency f_0, DVFS reduces execution time at frequency f by a factor $\tau(f) = m + \frac{f_0}{f} \cdot (1-m)$. Latency-critical utilization u is simply the nominal utilization u_0 scaled by $\tau(f_l)$. u_0 is easily computed from the mean arrival rate of requests and profiled service times. Batch throughput T is nominal throughput scaled by speedup, $1/\tau(f_b)$. Powers P_l and P_b are cubic polynomials on frequency that can be obtained by using each application's performance counters in Rubik's power model (Section 3.8).

Both the Utilization u and the throughput T scale sublinearly with frequency, as they depend on the ratio of core-bound and memory-bound cycles, while P_l and P_b both scale superlinearly with frequency, due to voltage and frequency scaling. Thus, Equation (3.3) is concave, and optimization is trivial using hill climbing.

Implementation

RubikSC requires small modifications over RubikLC. RubikSC profiles the core- and memory-bound cycles for each application, as well as the

performance counters needed to drive the power model. Periodically (every 100 ms), RubikSC uses hill climbing to find the frequency pairs that maximize TPW, and uses them throughout the next interval. Batch applications run at this frequency, while latency-critical applications run at the maximum of the TPW-optimum frequency f_l and the frequency that maintains tail latency.

Enforcing power limits

Latency-critical applications sometimes need to briefly exceed the nominal frequency to meet tail latency bounds. This may exceed the chip's thermal design power (TDP) when all cores are active. If needed, RubikSC throttles cores to stay within the TDP as follows:

- first, it throttles batch applications, starting with memory-bound ones (as they lose less throughput);
- second, if necessary, it deschedules active batch applications;
- finally, as a last resort, throttles all active latency-critical applications equally to gracefully degrade tail latencies.

This last resort is exceedingly rare in our experiments (Section 3.9.2).

3.6 RubikDC: DVFS for the Datacenter

Rubik Datacenter (RubikDC) extends RubikSC to optimize datacenter efficiency. RubikSC maximizes the throughput-per-watt of each individual server, but it may not yield optimal datacenter efficiency. The discrepancy arises because RubikSC does not account for datacenter-level throughput needs: RubikSC is optimal for the datacenter only when the datacenter's throughput requirements happen to match the aggregate throughput achieved by RubikSC.

In some cases, RubikSC is too aggressive. Consider a datacenter similar to Figure 3.1a. Suppose most applications are latency-critical, and there is little batch work to be done. In this scenario, servers have ample spare cycles to provide the desired batch throughput. RubikSC maximizes TPW assuming that there is always a supply of batch work to keep the servers busy. This assumption, however, does not hold

in this case and optimizing without a throughput target yields a frequency higher than necessary: batch work uses only a fraction of cycles and leaves the machine idle for the remainder. Instead, a lower batch frequency would save power while achieving the same throughput.

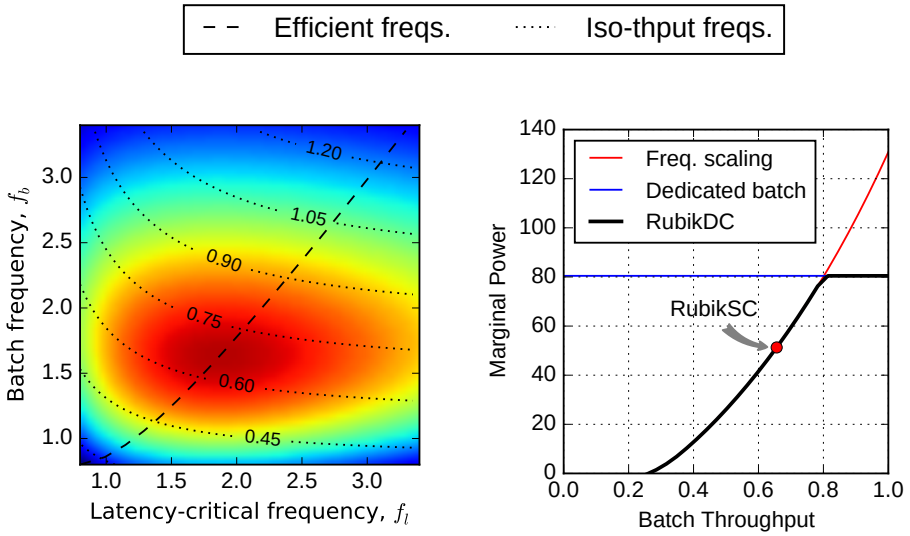
In other cases, RubikSC is too conservative. Suppose that batch work is plentiful, and only a few applications are latency-critical. In this scenario, Rubik can colocate just a fraction of the batch work on latency-critical machines, while the rest must run on dedicated batch machines. Because systems are not energy proportional, it is overall more efficient to exceed RubikSC's optimal operating point, colocate more batch work per machine than RubikSC does, and thereby reduce the number of dedicated batch machines. RubikDC sacrifices the TPW of individual colocated machines, but optimizes *datacenter TPW*. Colocated machines do not run at their optimal TPW point, but the full datacenter does.

In general, RubikDC improves datacenter efficiency by considering global batch throughput needs and:

- lowering frequencies when latency-critical work dominates, and
- raising frequencies when batch work dominates.

Figure 3.4a illustrates how RubikDC chooses frequencies to match the desired batch throughput. Colors show TPW, borrowed from Figure 3.3a. Dotted lines are *iso-throughput curves*: all frequency pairs along a single dotted curve provide the same throughput (normalized to a dedicated batch machine). For a given throughput target, RubikDC chooses the frequencies along the appropriate iso-throughput curve that maximize TPW. The dashed line traces out RubikDC's decisions at different throughput targets. As in RubikSC, latency-critical frequency may be constrained to maintain tail latency.

RubikDC will continue to scale frequencies along the dashed line until it is more efficient to provision dedicated batch machines. Figure 3.4b illustrates this, showing the *marginal power* required to supply additional batch throughput. To the left of the knee (0.8), RubikDC scales frequencies; to the right, RubikDC provisions dedicated batch machines. Figure 3.4 gives a microeconomic [89] formulation of Ru-



(a) *Frequency scaling of Figure 3.3a*

(b) *Decisions vs. desired throughput*

Figure 3.4: *RubikDC sets frequencies to meet a given batch throughput at minimal power. (a) The dashed line traces efficient frequencies at different throughputs. (b) RubikDC scales frequency until dedicated batch machines are more efficient.*

bikDC. Colocation is efficient because static power is a sunk cost on latency-critical machines.

Figure 3.4a is a consumer optimization problem: TPW is the utility function, dotted lines are the budget constraints, and the dashed line is the income-consumption curve. Figure 3.4b is the supply curve of batch throughput. Our discussion assumes perfectly inelastic demand for batch throughput, but Figure 3.4b suggests how to extend this to settings where demand varies with cost.

3.6.1 RubikDC Case Studies

To further exemplify how RubikDC's global formulation improves over RubikSC, which only looks at optimizing TPW on each single server in isolation, we discuss two case studies.

As a first example, consider a segregated datacenter with 100 machines split between 75 latency-critical and 25 batch. This gives RubikDC a throughput target of $^{25}/_{75} = {}^{1}/_{3}$ per colocated machine (normalized to a dedicated batch machine). In Figure 3.4b, ⅓ occurs before

the knee, meaning that all batch work is provided by the spare cycles of latency-critical machines. Since ⅓ lies below RubikSC's operating point, RubikDC will scale *down* batch frequency to lower power. By eliminating batch machines, RubikDC reduces the number of required servers by 25%.

As a second example, consider a segregated datacenter evenly split between 50 latency-critical and 50 batch. This case requires a normalized throughput target of $^{50}/_{50} = 1$ per colocated machine. In Figure 3.4b, 1 lies beyond the knee, so RubikDC will do the following:

1. First, RubikDC will scale *up* frequencies above RubikSC's operating point until the marginal power of frequency scaling equals that of dedicated batch machines. This occurs at colocated throughput of 0.8 in Figure 3.4b.

2. Second, RubikDC will provision dedicated batch machines to supply the remaining batch throughput.

In this case, the colocated machines supply normalized batch throughput of 50 machines $\times$ 0.8 = 40 batch machines, requiring only 10 dedicated machines and reducing the number of required servers by 40%.

3.7 Power Modeling

Rubik needs a model to predict the power draw of an entire server at different frequencies and utilizations, in order to determine the most efficient operating point.

A power model can describe a system at different levels of abstraction. On the one hand, frameworks such as McPAT [81] focus on modeling at the circuit and technology levels, to yield a detailed power breakdown of the on-chip components. Such approach is useful for architects to evaluate new solutions, but it is extremely fine-grained and incurs significant overhead, which makes it inadequate to estimate full-system power online. On the other hand, models at an higher-level of abstraction [111] can estimate the power draw of whole components (e.g., CPU, DRAM), or of the entire system, without requiring detailed circuit-level information. We use this second approach to model full-system power with good accuracy and low overhead and we validate this model against representative datacenter hardware.

Previous research proposed a variety of high-level power models, focusing on different aspects. Wei et al. [54] use an event-driven model based on performance counters to estimate the core power in a Power7 processor. Shen et al. [124] focus on per-task power budgeting based on a simple linear model of dynamic power. Koukos et al. [75] propose a processor power model that leverages an estimate of effective switching capacitance based on retired instructions [129]; they validate this model against an Intel Sandy Bridge processor. CoScale [30] uses a power model based on performance counters in the processor and DRAM, but does not provide a validation against real hardware. Our model shares the basic approach with these previous proposals but, instead of only focusing on the processor [54, 75] or on dynamic power [124], it estimates full-system power. We build our model based on modern servers featuring multicore processors of the Intel Core family, which are widely used in datacenters, and we validate its power predictions for two such servers of different generations.

3.7.1 Full system power

We identify three main components to estimate the full-system power P_{sys} (Equation (3.4)): processor power, DRAM power, and the power drawn by the other system components.

$$P_{\text{sys}} = P_{\text{proc}} + P_{\text{DRAM}} + P_{\text{other}} \tag{3.4}$$

Since we focus on steady-state operation of the leaf nodes in a datacenter, our benchmarks only stress the processor and the main memory; for this reason, we focus on accurately modeling the first two terms of Equation (3.4) and we model P_{other} as a constant. While this choice slightly reduces the overall accuracy of our model, when validated on real data, it does not impair its accuracy in estimating energy savings, which we achieve through DVFS of the processor. Future work might specialize P_{other} into additional terms to consider applications that stress dynamic power from other components (e.g., the disk for an I/O intensive application) or techniques that focus on different mechanisms.

To develop our models for processor and DRAM power, we take a mixed analytical / empirical approach. We use known relationships

that bind power to switching frequency for the CMOS technology as a baseline model, which we refine based on architectural considerations and accounting for utilization. We only use information easily available, through performance counters and model-specific registers (MSRs), on the commodity servers we target; this choice allows us to validate the model against real hardware and makes it directly deployable.

3.7.2 Processor power

To accurately model the processor power P_{proc}, we need to consider the different power domains the processor is divided into. For each domain, the two main power components are due to leakage power, which is proportional to the voltage V, and dynamic switching power, which is proportional to $V^2 \cdot f$, where f is the frequency of the domain. Equation (3.5) formalizes this general model for a processor with N power domains; the parameters i_i and c_i are estimates of the average leakage current and switching capacitance.

$$P_{\text{proc}} = \sum_{l=1}^{N} (P_{\text{leak } l} + P_{\text{dyn } l}) = \sum_{l=1}^{N} (i_l \cdot V_l + c_l \cdot V_l^2 \cdot f_l) \qquad (3.5)$$

We adapt this model to the multicore architecture we consider, which is general enough to model both the Intel chips we use to validate our model and the multicore we use in our simulations to evaluate Rubik. Our reference architecture has M homogeneous cores, each in a separate power domain (i.e., our model supports per-core DVFS [54]) and the uncore (i.e., on a first-order approximation, the LLC) on a separate power domain. Equation (3.6) formalizes this model: each core contributes its own leakage and dynamic power (terms in the summation), as does the uncore (terms with subscript u).

$$P_{\text{proc}} = \sum_{l=1}^{M} (i \cdot V_l + c \cdot V_l^2 \cdot f_l) + i_{\text{u}} \cdot V_{\text{u}} + c_{\text{u}} \cdot V_{\text{u}}^2 \cdot f_{\text{u}} \qquad (3.6)$$

This model has two inputs sets: (1) voltages and frequencies are variables measured online; (2) i and c (which are the same for all the homogeneous cores), and i_u and c_u are parameters that we train to capture the characteristics of the processor.

Since modern processors increasingly use different types of transistors for different subsystems and aggressively clock- and power-gate

unused parts, we split c into the sum of four components (see Equation (3.7)): a constant term c_0, representing transistors that switch independently of utilization, and three terms, respectively depending on the retired instructions per cycle, memory hits to on-core caches (i.e., L1 or L2), and off core memory accesses (i.e., LLC hits and misses).

$$c = c_0 + \text{IPC} \cdot c_1 + \text{LOAD}_{\text{L1/L2}} \cdot c_2 + \text{ACC}_{\text{LLC}} \cdot c_3 \tag{3.7}$$

Adding more terms (e.g., one depending on floating point operations) does not significantly increase the accuracy of the model for our target platforms and applications.

Similarly, Equation (3.8) accounts for the utilization of the uncore by breaking c_u (see Equation (3.6)) into two terms, respectively proportional to the number of LLC hits and misses per cycle.

$$c_u = c_{u1} \cdot \text{HIT}_{\text{LLC}} + c_{u2} \cdot \text{MISS}_{\text{LLC}} \tag{3.8}$$

The combination of Equation (3.6) with Equations 3.7 and 3.8 formalizes our processor power model. All the variables (e.g., IPC, HIT_{LLC}) are measured on a per-cycle basis (e.g., IPC is instructions per cycle, HIT_{LLC} is LLC hits per cycle), at the frequency of the power domain they refer to.

3.7.3 DRAM Power

While we can use performance counters for deriving several statistics to refine the processor power model, current hardware does not expose as much information about DRAM. The only meaningful measurement correlated with the DRAM power draw P_{DRAM} is the frequency of DRAM accesses, which corresponds to the frequency of LLC misses. Equation (3.9) is our DRAM power model, based on a constant term that models the power draw due to DRAM refresh and on a term proportional to DRAM accesses per second.

$$P_{\text{DRAM}} = P_{\text{refresh}} + \text{ACC}_{\text{DRAM}} \cdot d \tag{3.9}$$

While future hardware implementing additional performance counters for main memory [30] will enable improvements of this model, we can only rely on information available to a runtime system on current hardware. Moreover, since we focus on DVFS of the processor, and not of

DRAM, it is critical for Rubik to have a very accurate processor power model, while we can tolerate some inaccuracy on the other components.

3.7.4 Implementation and Model Training

Rubik use our power model to predict power draw at different configurations in order to solve their optimization problem. Additionally, we included our power model in zsim, the simulator we use for our evaluation. The validation in Section 3.7.5 shows that this model is representative of current servers that may be found in datacenters and thus it indicates that the results of our simulations are realistic, within a small error margin. Notice, however, that neither our optimization techniques nor zsim rely on this particular power model: building a more accurate model is an orthogonal improvement that would directly further benefit both the efficacy of Rubik and the accuracy of the simulator.

The equations of our power model are based on measurements (i.e., the uppercase terms in the right side of Equations 3.7, 3.8, and 3.9) and parameters to be estimated (i.e., $P_{\text{other}}, i, c_0, \ldots c_3, i_u, c_{u1}, c_{u2}, P_{\text{refresh}}$, and d). These parameters need to be estimated once for each machine configuration based on training data and can then be used in the model to yield predictions. We use the non-linear least square algorithm to estimate the model parameters based on training data gathered from two different servers (one with Haswell and one with Sandy Bridge Intel processors). These processors report, through MSRs, the frequency and voltage of the cores, which are all in the same power domain, but not of the uncore; we assume that the frequency and voltage of the uncore (i.e., f_u and V_u) are fixed to the nominal processor frequency. Section 3.7.5 provides a validation of our model on these two servers.

3.7.5 Power Model Validation

We train our power model to predict power draw of two modern servers equipped with Intel processors: (1) a Supermicro SuperServer 1027R-WRF, with an Intel Xeon E5 2640 clocked at 2.50 GHz, and 64GB of DDR3 buffered DRAM, and (2) a Supermicro SuperServer 5018D-MTF, with an Intel Xeon E3 1240 clocked at 2.50 GHz, and 64GB of DDR3 unbuffered DRAM. We refer to these two servers with the

name of their processor microarchitecture: (1) Sandy Bridge and (2) Haswell, respectively.

We collect power measurements for the core, uncore, and DRAM components through the RAPL interface exposed by these processors and the other measurements our power model demands through performance counters. In addition, we use a WattsUp Pro power meter to measure full-system power draw. Each trace contains 20000 measurements sampled every 25 ms[2]. The workload comprises random mixes of SPEC CPU2006 applications and we vary the core frequency to a new random value (including Turbo mode) every 4 seconds. Notice that the Sandy Bridge server has a dual-socket motherboard; in this case, we only run applications on the first socket; we leave the second socket idle at a fixed frequency, with c-states disabled, and we add its power consumption, as reported by the RAPL interface, as an additional measured term in the model expression for the full-system power (see Equation (3.4)). We use one trace from each server to learn the model parameters and we evaluate the accuracy of the model on a different trace.

We use our power model for two purposes: 1. providing online information to Rubik, 2. evaluating aggregate power savings. To evaluate the first use, we compute the fine-grained absolute prediction error on the system power and the three components of Equation (3.4) on each 25 ms sample of the test sets; Figure 3.5 reports the cumulative distribution function (CDF) of the error, relative to the system power measured with the WattsUp meter. The average error on full system power for the two servers is 1.7 %, or 3.05 W, for the Sandy Bridge server and 5.1 %, or 2.8 W, for the Haswell server. While, in both cases, most of the error is due to P_{other}, this error is still reasonably small, 1.7 % and 4.5 % for the two servers, respectively. These results mostly derive from noise in the data, due to the lower resolution of the WattsUp meter, which forced us to interpolate the data in order to use it with the other measurements.

To evaluate the second use of the power model, we compute the prediction error over each whole test set (i.e., over several seconds of

[2]The WattsUp sampling period is limited to 1 s; we interpolate and align the data to match the other measurements.

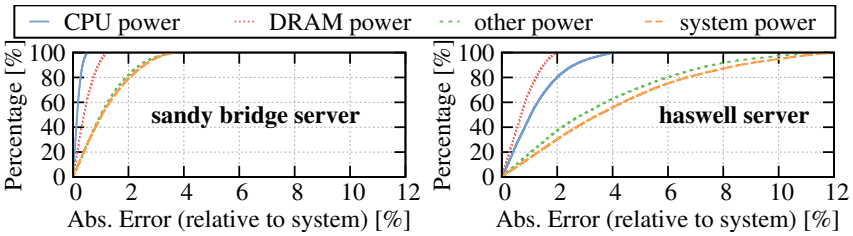


Figure 3.5: *Cumulative distributions of the error, relative to full system power, on fine-grained prediction (25 ms samples) of CPU, DRAM, other components, and full system power draw.*

	P_{CPU}	P_{DRAM}	P_{other}	P_{sys}
Sandy Bridge	0.04 %	0.30 %	0.01 %	0.03 %
Haswell	0.35 %	0.03 %	0.01 %	0.14 %

Table 3.1: *Long-term absolute error of our power model.*

runtime); Table 3.1 reports these results, showing that our model is extremely accurate on long-term prediction.

3.8 Rubik Evaluation Methodology

In order to evaluate Rubik, we simulate next-generation servers with a partitioned memory hierarchy and we use a set of five representative latency-critical applications, borrowed from Kasture and Sanchez [66].

Simulated system

We use and extend zsim [116] to perform microarchitectural simulation of a 6-core system with parameters shown in Table 3.2. This configuration is representative of modern high-performance servers [76, 126], with the additional assumption that the memory system is statically partitioned, with 1 MB of LLC space per application and 8.6 GB/s (1 channel) per core. The system supports per-core DVFS and sleep states modeled after Haswell [18, 48].

Latency-critical workloads

We use five diverse latency-critical applications, borrowed from [66]: `xapian`, a web search engine configured as a leaf node [25, 59]; `masstree`,

Cores	6 x86-64 cores, detailed Westmere-like OOO [116]
L1 caches	32 KB, 4-way set-associative, split D/I, 1-cycle latency
L2 caches	256 KB private per-core, 16-way set-associative, inclusive, 7-cycle latency
L3 cache	6 banks, 12 MBs total, 4-way 52-candidate zcache [114], 20 cycles, inclusive, Vantage [115] partitioning
Coherence protocol	MESI protocol, 64-byte lines, in-cache directory, no silent drops, TSO
Memory	48 GB, 6 DDR3-1066-CL7 channels, partitioned [101], 8.6 GB/s per core
Power	2.4 GHz nominal frequency; Haswell-like FIVR [18] per-core DVFS: 0.8–3.4 GHz frequency range, in 200 MHz steps; core sleep with L1s & L2 flushed to LLC (Haswell C3 [55]); 65 W chip TDP

Table 3.2: *Configuration of the simulated 6-core CMP.*

a high-performance key-value store [90]; `moses`, a statistical machine translation system configured to perform real-time translation (e.g., as in Google Translate) [74]; `shore`, an online transaction processing database running TPC-C [63]; and `specjbb`, a Java real-time middleware benchmark. Table 3.3 shows their input sets and simulated requests. Section 1.3.2 further characterize the diverse workload of the five applications.

To measure the tail latency of our five latency-critical applications under simulation, we integrate server and client under the same process. The client then produces a realistic stream of requests with exponentially distributed interarrival times at a configurable rate (i.e., a Markov input process, which is common in datacenter workloads [93, 96]). Client overheads are negligible (∼150 ns/request). This faithfully captures all compute overheads.

Workload	Configuration	Requests
`xapian`	English Wikipedia, zipfian query popularity	6000
`masstree`	mycsb-a (50% GETs, 50% PUTs), 1.1GB table	9000
`moses`	opensubtitles.org corpora, phrase-based mode	900
`shore`	TPC-C, 10 warehouses	7500
`specjbb`	1 warehouse	37500

Table 3.3: *Configuration and number of requests for latency-critical applications.*

Batch workloads

We use mixes of SPEC CPU2006 workloads as batch applications, executed in a similar manner to prior work [58, 66, 109]. Each batch application is fast-forwarded 5 B instructions, executes for 400 M instructions, and is restarted until the latency-critical applications in the mix finish executing their target number of requests.

Metrics

We define tail latency as 95^{th} percentile latency, which is typical [85]; other percentiles are possible. When running batch and latency-critical mixes, we report the datacenter power and number of machines needed to satisfy different combinations of latency-critical and batch work (Section 3.9.2).

To ensure stable results, we perform enough runs per experiment to achieve 95% confidence intervals below 1%.

3.9 Evaluation

We first characterize RubikLC without colocation, focusing on its impact on latency-critical applications. Then, we characterize RubikSC and RubikDC at the datacenter level. Finally, we present focused sensitivity experiments and analysis of Rubik.

3.9.1 RubikLC Evaluation

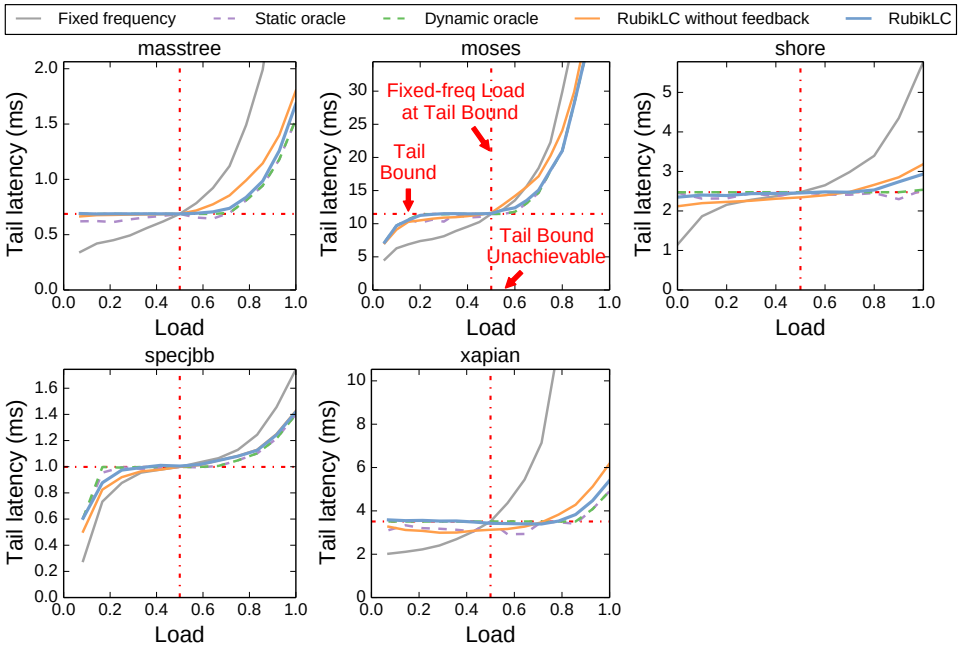
We first characterize RubikLC using trace-driven experiments. We capture per-request arrival times, core cycles, memory-bound times,

and performance counters in zsim, and replay the trace under different schemes. This setup allows us to compare RubikLC against two oracular schemes. First, the *static oracle* chooses the lowest static frequency that satisfies tail latency for the whole trace, up to the maximum frequency. This static oracle is an upper bound on the efficiency of feedback-based controllers such as PEGASUS [85]; real controllers need to be more conservative to maintain tail latency (e.g., using guardbands [85]). Second, the *dynamic oracle* finds the frequency schedule that minimizes power while staying within the tail latency.

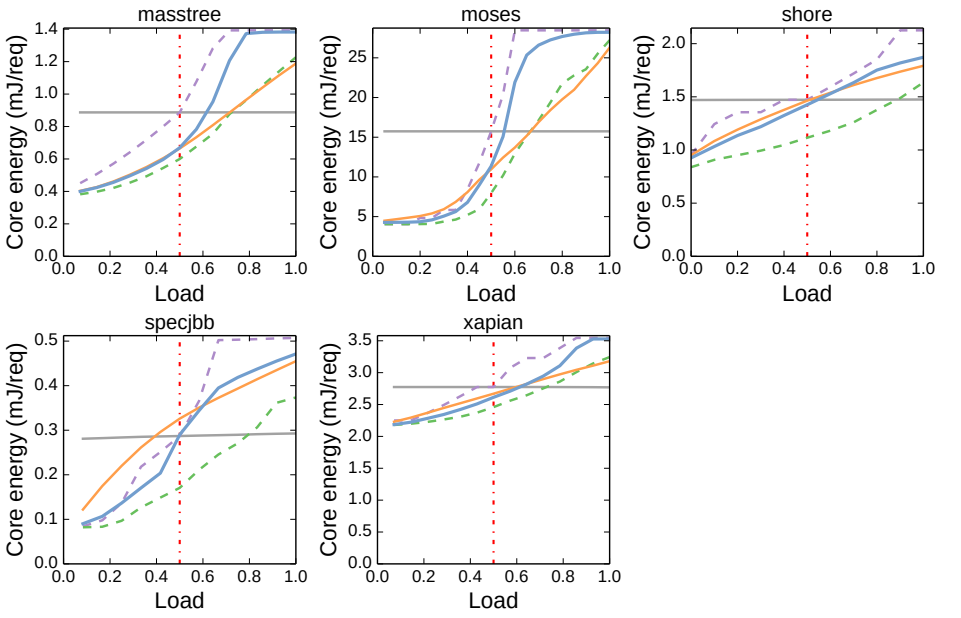
This oracle first computes, for each request, the lowest frequency that makes *all* requests satisfy the latency bound; when the bound is not achievable with the available frequencies, it chooses the highest frequency. Then, it progressively lowers the frequencies until 5% of the requests are above the tail bound (if it is achievable) or further optimizations would increase the 95^{th} percentile latency. During this operation, the oracle chooses when to reduce frequencies by prioritizing those reductions that affect tail latency the least and save most power. The dynamic oracle gives a lower bound on power (the only way to consume less power is to further degrade tail latency).

Figure 3.6 shows the tail latency and average core energy per request as a function of load in five different cases: with fixed frequency, when using the static or the dynamic oracle, when using RubikLC without the fine-grained feedback loop controller and when using the full version of RubikLC as described in Section 3.4. A load of 100% corresponds to the maximum request rate at nominal frequency (2.4 GHz). Each plot characterizes an application, and each line shows a single scheme. The fixed-frequency results run at nominal frequency, while the oracles and RubikLC can use all available frequencies (0.8–3.4 GHz). We show RubikLC without and with the feedback controller. We use the tail latency of the fixed-frequency scheme at 50% load as the target for all other schemes.

Focusing on Figure 3.6a, the fixed-frequency results show that tail latency is highly sensitive to load. By contrast, both oracles lower frequencies to match the tail bound, producing a flat tail latency curve until at least 50% load in all applications. The region where load is high enough that no scheme can meet the tail bound is shaded red in each



(a) *Load-latency diagrams, showing 95^{th} percentile tail latencies.*



(b) *Load-energy diagrams, showing core energy per request (includes dynamic and static for pipeline, L1s, and L2, when core is active).*

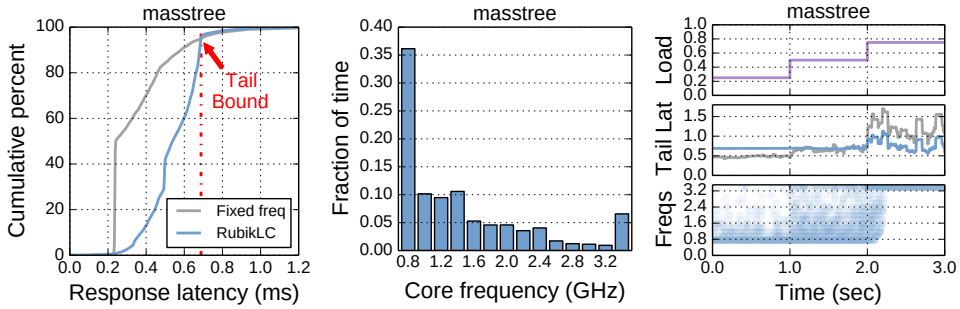
Figure 3.6: *Tail latencies and core energy per request for each latency-critical applications studied under a fixed frequency, static and dynamic oracles, and RubikLC without and with the feedback loop. The tail latency at fixed-frequency under 50% load is the tail latency bound for all other schemes. In the shaded areas, load is high enough that no scheme can meet the tail bound.*

graph. RubikLC without feedback closely tracks the desired behavior: with loads below 50%, it achieves a near-flat tail, but its conservative approximations produce a lower tail than needed in `shore`, `specjbb`, and `xapian`; at high loads (shaded region), tail latency is slightly above the minimum achievable tail, as set by the oracles. RubikLC's feedback controller fixes these deviations, matching the latency curves of the oracles. This shows that the analytical controller is accurate, and only needs minor corrections.

Figure 3.6b shows the core energy per request of each scheme. To simplify the discussion, we report active core energy only (pipeline, L1s, and L2); adaptive schemes also reduce idle energy when running without batch applications. At a fixed frequency, active energy per request does not change with load. Below 50% load, adaptive schemes reduce frequencies often and lower energy; above 50% load, these schemes often use higher frequencies and more energy to keep the tail latency as close to the target as possible. Comparing the static and dynamic oracles reveals the benefit of short-term adaptation: across the range, the dynamic oracle saves significantly more energy than the static oracle, as much as 2× (`moses`). At 50% load, the dynamic oracle often saves 20–45% of energy. RubikLC outperforms the static oracle and reaps most of the benefits of the dynamic oracle, especially in applications with tightly clustered service times (`masstree`, `moses`). With more variable service times (`shore`, `specjbb`), RubikLC saves less power than the dynamic oracle because it lacks future knowledge and must guard against long requests. Note that RubikLC without feedback often consumes less energy than with feedback when the tail bound cannot be met (shaded region); this is expected, and happens because the feedback controller increases frequencies to try to match the tail bound.

RubikLC on `masstree`

Figure 3.7 characterizes RubikLC in more detail by comparing it against Fixed-frequency on one application, `masstree`. In particular, Figure 3.7 shows that RubikLC saves power without degrading tail latency and achieves more stable tail latencies, even under sudden load changes. Figure 3.7a compares the response latency cumulative dis-



(a) *Response latency CDF* **(b)** *Frequency histogram* **(c)** *Sudden load changes*

Figure 3.7: *RubikLC on* `masstree`*: (a) RubikLC serves requests later without degrading tail latency, (b) uses low frequencies often to save power, and (c) handles sudden load changes well.*

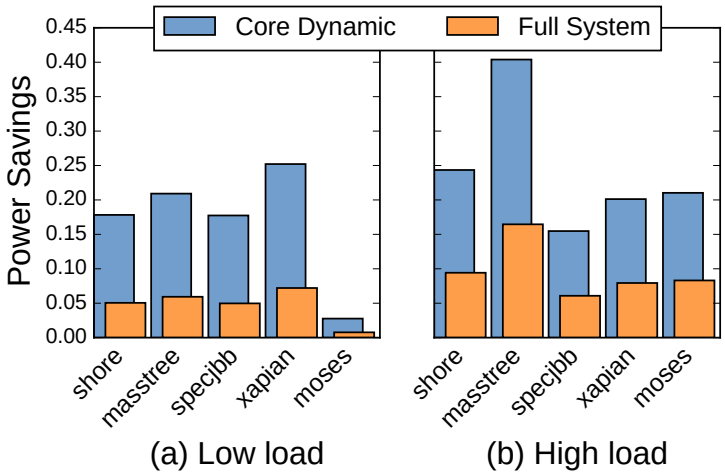


Figure 3.8: *Core and total power savings of RubikLC when running one latency-critical application per core.*

tribution functions (CDFs) for RubikLC and Fixed-frequency at 50% load. RubikLC shifts the low part of the CDF to the right (delaying responses), but above the tail bound, it matches Fixed-frequency's CDF. Figure 3.7b shows RubikLC's frequency histogram at 50% load: most time is spent at low frequencies, saving significant power. Finally, Figure 3.7c shows the effect of sudden load changes over a 3-second span. The top graph shows load over time, which grows in steps, spanning 25%, 50%, and 75% loads. The middle graph shows the 95^{th} percentile latency over a rolling 200 ms window for RubikLC and Fixed-frequency. At 25% and 50% loads, where the tail bound can be met (Figure 3.6a), RubikLC has a flat tail over time and perfectly matches the tail bound. Its tail is more stable than Fixed-frequency, which has variations over time. At 75%, where the tail bound cannot be met (see Figure 3.6a), RubikLC provides a more stable tail latency. The bottom graph shows the frequencies RubikLC uses over time to achieve this behavior. On both steps, RubikLC's analytical controller starts using higher frequencies *immediately*. As a result, RubikLC maintains a flat tail in the 25–50% step. Additionally, on the 50–75% step, the feedback controller sees the tail bound is not being met, and forces RubikLC to always run at maximum frequency. This effect takes place over 200 ms (2.0–2.2 s interval).

Latency-critical mixes

Figure 3.8 shows RubikLC's core and full-system power savings over Fixed-frequency when running six copies of the same latency-critical application, one per core. Each application is run at both low load (20%, left) and high load (60%, right). The tail bound of each application and load is the tail latency that Fixed-frequency achieves. These detailed simulation results match the trace-driven characterization: RubikLC reduces core power by 3–40% (avg 21%). RubikLC also respects the tail bound in all cases (not shown), staying within 95%–100% of the tail bound. However, full-system power savings are moderate, 1–16%, due to low utilization. Thus, RubikLC improves latency-critical mix efficiency, but substantially reducing full-system power also requires colocation.

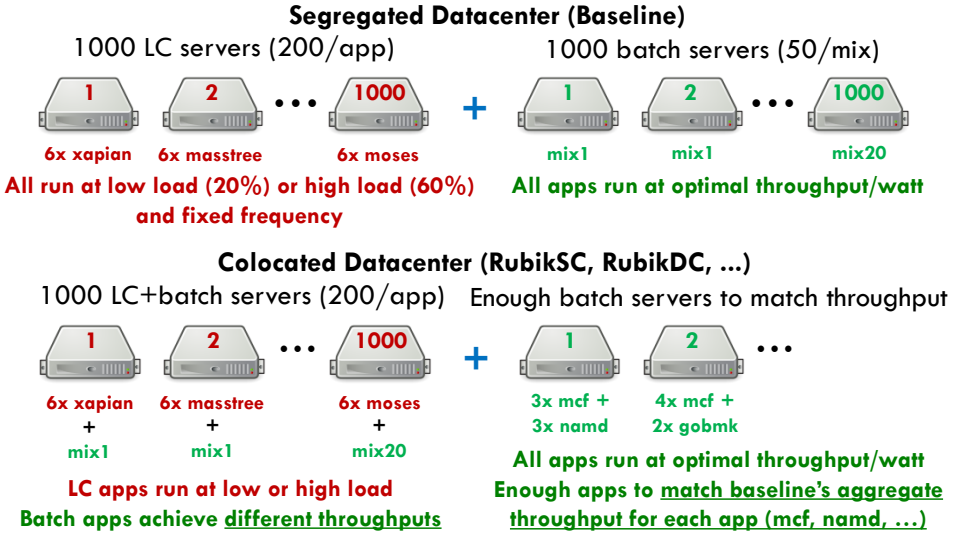


Figure 3.9: *Experimental setup used to compare schemes.*

3.9.2 RubikSC and RubikDC Evaluation

Experimental setup

To evaluate different latency-critical/batch colocation schemes on an equal footing, we first consider a baseline datacenter that segregates batch and latency-critical applications. As shown in Figure 3.9, this datacenter has 1000 machines that run the 5 latency-critical applications, with 200 machines dedicated to each application. Each latency-critical machine runs 6 copies of the application at nominal frequency, each at either low (20%) or high (60%) load, as in Section 3.9.1. The datacenter we model also has 1000 machines running batch work. We produce 20 mixes of six randomly chosen SPEC CPU2006 applications, and dedicate 50 machines to each mix. Each batch application runs at its optimal throughput per watt Because all servers use a partitioned memory system, the optimal frequency for each application does not depend on the applications it is colocated with (applications in dedicated batch servers do not run above nominal frequency to stay within TDP).

We then evaluate each colocation scheme as shown in Figure 3.9. First, each latency-critical machine now also runs the mix from the corresponding batch machine of the segregated datacenter, becoming

a colocated machine. Mixes are interleaved so that each latency-critical application is co-scheduled with all batch mixes equally. Second, because each application in the batch mixes gets less throughput when colocated, we provision a variable number of batch-only servers and run additional copies of each batch application to match the throughput of the segregated datacenter for each batch application. Schemes that achieve higher batch throughputs on the colocated machines will need fewer batch machines, but may consume more power on colocated machines to do so.

This experiment is carefully designed to have three desirable properties:

1. it is *fixed-work* [50] (all schemes run matching batch and latency-critical work);
2. it allows comparing *end-to-end metrics* (tail latencies, datacenter power, and machines used);
3. by interleaving mixes, it exposes each latency-critical application to all batch applications.

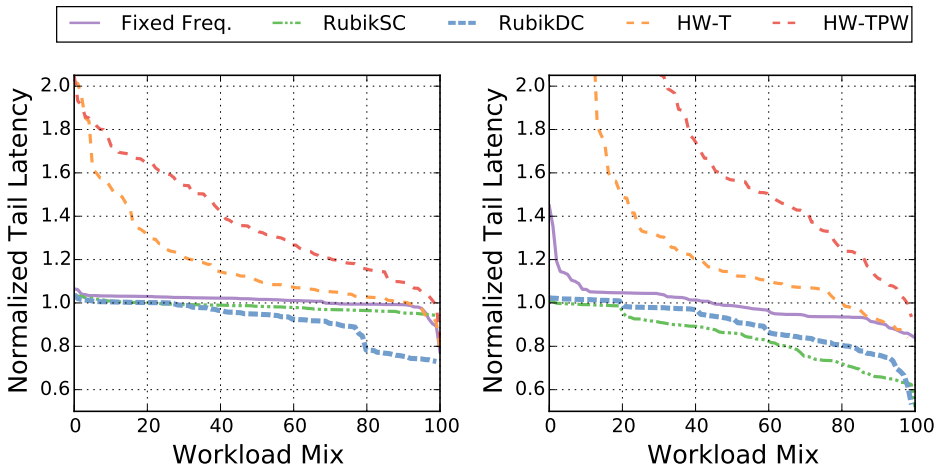
We do not claim this is the best approach to manage datacenters with latency-critical/batch mixes; it is just a controlled and fair way to compare schemes.

Schemes

We evaluate five colocation schemes: fixed-frequency, RubikSC, RubikDC, and two hardware-controlled schemes that perform coordinated per-core DVFS, *HW-T* and *HW-TPW*. HW-T sets frequencies to maximize aggregate system throughput (IPC) while staying below TDP; HW-TPW maximizes aggregate throughput per watt. These schemes adapt every 100 µs, and represent hardware-controlled DVFS schemes typical of modern chips (e.g., Turbo Boost [84, 112]) that are unaware of latency-critical app requirements. All schemes run with a partitioned memory system.

Tail latencies

Figure 3.10 shows the distribution of tail latencies achieved by different schemes. Each line represents a single scheme, and the x-axis



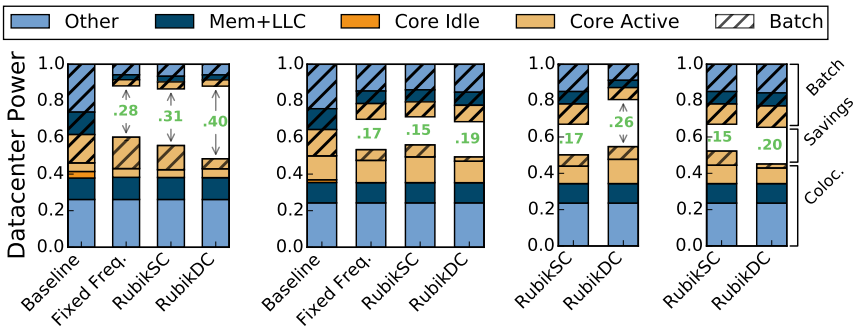
(a) *Low load* **(b)** *High load*

Figure 3.10: *Distributions of tail latency (lower is better) relative to the baseline datacenter, for the 5 latency-critical × 20 batch mixes simulated, split into low and high latency-critical loads.*

represents the $5 \times 20 = 100$ latency-critical/batch mixes in the 1000 colocated machines. For each scheme, mixes are sorted from highest to lowest tail latency, relative to the tail bound (lower is better), independently for each line.

In the low-load datacenter (Figure 3.10a), Fixed-frequency, RubikSC, and RubikDC meet the tail bounds in all mixes. RubikSC and RubikDC sometimes lower tail latency significantly, as they run latency-critical applications beyond their required frequency to improve efficiency. By contrast, HW-T and HW-TPW grossly violate tail latencies, by as much as 2×. This result shows that hardware-managed DVFS schemes introduce interference among applications and are not suitable for colocation.

In the high-load datacenter (Figure 3.10b) tail latency is more sensitive: HW-T and HW-TPW degrade tails by up to 8.2× and 3.2×, respectively. Even Fixed-frequency degrades tail latency for 40% of the mixes (by up to 42%) because sharing cores degrades performance. RubikSC and RubikDC maintain tail latency across all mixes, making up for core sharing by automatically using higher frequencies when needed.



(a) *Low load* **(b)** *High load* **(c)** *xapian* **(d)** *specjbb*

Figure 3.11: *Power breakdown by component of different schemes for colocated and batch machines, normalized to the baseline datacenter: (a) and (b) vary load; (c) and (d) are case studies at high load.*

Because HW-T and HW-TPW violate tail bounds in all cases, we do not consider them further (they are also less efficient than RubikDC).

Power breakdown

Figure 3.11a shows the breakdown of full datacenter power for each scheme in the low-load case. Power is normalized to the baseline datacenter's; note the y-axis goes up to 1.0. Each set of *two opposing vertical bars* shows the power breakdown of a single scheme: the bottom bar, starting from 0, shows total power for latency-critical (baseline) or colocated servers; the top bar, starting from 1, shows total power for dedicated batch servers. Both bars are normalized to the total baseline datacenter power, so the vertical gap between them shows the power savings of the scheme. Each bar is further broken down by component: core (active and idle power), memory system (including uncore and main memory), and other components (chipset, HDD, fans, etc.). Memory system and other power barely change with utilization. Power used for batch work is hatched: dedicated batch server power (top bar), and core power to run batch work in colocated machines (bottom bar).

Figure 3.11a shows that, in the baseline, latency-critical servers consume 47% of the power despite only being used at 20% load, and batch servers consume 53%. All other schemes use much lower batch power, because they colocate most batch work and use fewer batch servers.

Fixed-frequency runs colocated servers at relatively high frequencies, which is inefficient. RubikSC shows that optimizing for batch throughput per watt saves power on colocated servers compared to Fixed-frequency, at the expense of a moderate increase in batch servers and batch server power. Finally, RubikDC *achieves both the lowest colocated and batch server powers* in Figure 3.11a. Overall, RubikDC saves 40% of power; RubikSC saves 31%; and Fixed-frequency saves 28%.

Figure 3.11b shows the power for the high-load case. Savings are smaller due to the higher latency-critical utilization. RubikDC performs best, saving 19% of power; RubikSC saves 15%; and Fixed-frequency saves 17% but violates tail latencies (Figure 3.10b).

Figure 3.11c and Figure 3.11d show that RubikDC improves efficiency by being aware of global throughput requirements and application characteristics. Figure 3.11c shows the breakdown of RubikSC and RubikDC for a datacenter that only runs one latency-critical application, `xapian`, the most core-bound one, at high load; and a single batch mix (`xalanc`, `tonto`, `h264`, `lbm`, `wrf`, `gamess`). Figure 3.11d shows the same breakdown when only running `specjbb`, the most memory-bound latency-critical application, instead. With `xapian`, RubikDC colocates batch work more aggressively than RubikDC. With `specjbb`, RubikDC runs the colocated machines at lower frequencies, as `specjbb` sees no savings from high frequencies, and uses more batch machines. In comparison, RubikSC optimizes locally and is unaware of this tradeoff. In Figure 3.11a and Figure 3.11b, RubikDC exploits this tradeoff on a per-server basis to reduce total power.

Power limits

In the high-load case, Rubik throttles execution rarely to stay within the 65 W chip TDP, as described in Section 3.5. Considered together, RubikDC and RubikSC throttle batch applications 2.2% of the time, idle batch applications 0.05% of the time, and resort to throttling latency-critical applications less than 0.01% of the time. All throttling events are more rare in the low-load case. Overall, this shows that power caps do not necessarily hurt tail latency.

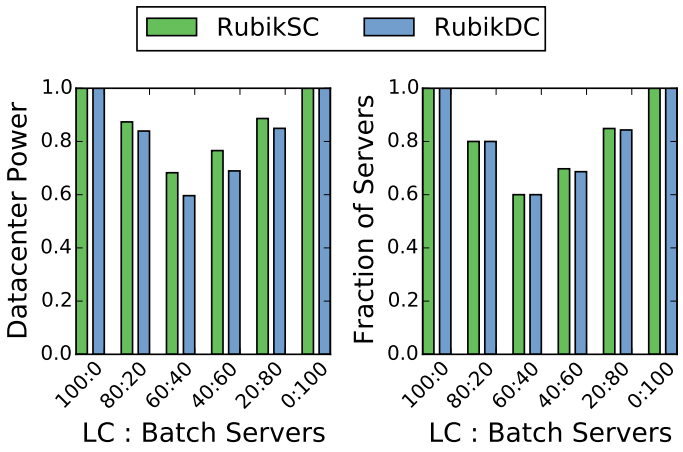


Figure 3.12: *Datacenter power and number of servers with RubikSC and RubikDC relative to the segregated baseline vs. the ratio of latency-critical to batch servers in the baseline.*

Lax deadlines and low utilization

To evaluate a scenario representative of online services running below peak load (e.g., due to diurnal variations [85]), we execute all latency-critical workloads at low load but use the tail latency bounds from the high-load case. RubikDC uses the extra slack to improve efficiency, saving 44% of datacenter power over the baseline.

3.9.3 Sensitivity to Amount of Batch Work

Figure 3.12 shows the total datacenter power (left) and servers (right) that RubikSC and RubikDC use compared to the baseline, as the ratio of latency-critical and batch servers of the baseline datacenter change. In the figure, 100:0 means all servers are latency-critical, 80:20 means 80 latency-critical servers for every 20 batch servers, and so on. We have studied 50:50 so far. Latency-critical workloads run at low load in this experiment. RubikDC adjusts its policy depending on the amount of batch work available to colocate, always using less power and slightly fewer machines than RubikSC. In the best case, at 60:40, RubikDC uses 60% of the baseline's power and servers, reducing energy and machines by 40%.

3.10 Conclusions

We have presented Rubik, a power management scheme for future multicores with partitioned memory systems. Rubik's analytical DVFS control improves latency-critical workload efficiency without degrading tail latency, and allows more aggressive colocation of latency-critical and batch workloads than memory partitioning alone (e.g., sharing cores). Rubik policies can maximize system efficiency or minimize total datacenter power. Compared to conventional datacenters, Rubik reduces both datacenter power and number of servers by 40%.

CHAPTER 4

Concluding Remarks

During the last decade, datacenters became a fundamental asset in supporting phenomena such as cloud computing and social media, which have a wide impact on our society. With warehouse-scale computing entering the *teenage decade* [7], we are at a point where we start to clearly understand the characteristics and requirements of this new computing paradigm, so that we can systematically tackle its issues. This dissertation, tackled two important issues in today's datacenters: providing (1) QoS guarantees and (2) efficiency; this final chapter concludes the dissertation with a summary highlighting the conclusions we can draw from this work.

4.1 Summary and Takeaways

Chapter 1

In Chapter 1, we provided an overview of the current landscape of warehouse scale computing, focusing on ways to improve its efficiency and on the quality of service (QoS) of different types of applications.

For what concerns efficiency, this analysis suggests that:

- techniques to reduce idle power (e.g., sleep states) are not well suited for datacenters, due to application characteristics and the high cost of static power;

- increasing the utilization and the efficiency of the single servers is the most valuable course of action to improve the efficiency of today's datacenters.

Focusing on applications, we noted that there are at least two important performance metrics: throughput and latency. These two metrics can be variably important for different applications.

Latency is crucial for user-facing services (such as web-search); to these applications, which we call *latency-critical*, latency is a measure of the user-perceived QoS. An important issue with the performance of latency-critical applications is that, in order to provide QoS guarantees, looking at average latency is not enough. Instead, we need to make sure that at least X % of the requests (commonly, 95 %, or 99 %, according to the criticality of the applications) are served by some deadline.

Throughput is a more general metric. Throughput can be used as a measure of progress in applications such as video encoders (i.e., *frames per second*) or map-reduce jobs. Moreover, it also applies to latency-critical applications (i.e., *requests per second*); in this case, throughput measures the *load* of the application and, often, higher load leads to increased latency due to the building up of request queues that introduce significant queuing delay. The prominence of queuing delay is an important difference between latency-critical applications in the datacenter space and traditional soft real-time systems.

Another peculiarity regarding the performance of datacenter applications is that it is crucial to quantify it in some high-level user-centric metric. Control systems in charge of guaranteeing QoS and tuning energy/performance tradeoffs throughout the hardware-software stack (e.g., resource allocators, or DVFS controllers) cannot rely on low-level metrics only, but need application-level information.

Chapter 2

Chapter 2 tackles the problem of determining the relationship between performance and allocated resources in public cloud computing when virtual machines (VMs) from different clients contend on a shared resource. This chapter focuses on throughput-based applications and consider the case of multithreaded, compute-bound workloads where CPU bandwidth is the contended resource.

Through a case study, we show that the common practice of having users rent a desired amount of resources on a coarse grain (in the case we analyze, CPU bandwidth, as a number of virtual CPUs) does not work well when VMs are associated with a service-level objective (SLO) that specifies a performance goal. There are two main issues:

1. In a virtualized environment, it is difficult for users to accurately estimate the performance their VMs will provide given a certain amount of allocated resources.

2. Some applications have varying performance with a fixed resource allocation due to load variations.

To solve these two issues in a way that is transparent to users and efficient for cloud providers, we propose AutoPro, a runtime system able to automatically determine and enforce allocations to meet performance SLOs, while maximizing node-level utilization by supporting batch workloads on a best-effort basis.

AutoPro observes application-level performance reports and leverages a simple resource-performance model to run a PI controller that automatically allocates, through resource containers, CPU bandwidth to VMs colocated on a multicore server. AutoPro allocates spare CPU cycles to batch workloads, thus allowing to maximize node-level utilization, one of the key ways to improve datacenter efficiency.

With AutoPro, we target current datacenters, where servers are equipped with multicore processors that have a shared memory hierarchy without hardware partitioning support. The advantage of this choice is that AutoPro could be readily deployed to today's servers The downside is that, if VMs contend on the shared memory resources (most commonly, on the last-level cache), the efficiency of the system will be reduced (i.e., cycles will be wasted on additional cache misses).

AutoPro cannot fix this issue, which requires cache partitioning support, but it is robust against the effects of contention: if possible, it will still meet SLOs by means of *soft-partitioning* through CPU bandwidth allocation.

Chapter 3

Chapter 3 tackles the problem of improving the efficiency of datacenters that run latency-critical applications, while providing strict bounds on their tail latency. In the light the observations of Chapter 1 on improving efficiency, we aim at optimizing the energy-efficiency, measured as throughput-per-watt (TPW), of the single servers through DVFS and at keeping utilization high, thus requiring fewer servers to provide the same service.

To reach these goals, we tackle two main issues:

1. Latency-critical applications inherently have low utilization due to their structure: increasing load impairs latency.

2. Traditional DVFS controllers are oblivious of application behavior and cannot improve TPW without hurting tail latency.

To reach our goals despite these issues, we build on recent research on multicore processors with a partitioned memory hierarchy and propose Rubik, a DVFS and colocation management system that improves the efficiency of latency-critical workloads without degrading their tail latency and allows to obtain high utilization.

Rubik sidesteps the chronic low-utilization of latency-critical applications by supporting colocation with batch applications. Latency-critical applications are prioritized by the scheduler and always preempt batch work whenever there are requests to serve. The partitioned memory hierarchy of the baseline architecture allows to avoid interference of batch on latency-critical applications.

Rubik exploits traditional performance counters and application-level information and applies statistical analyses on the arriving requests to dynamically determine, for each core, the most efficient frequency (in terms of TPW) that still allows to provide strict bounds on tail latency. Recent development on on-chip voltage regulators, that

enable quick DVFS, are a key enabling technology for Rubik, which adjusts frequencies at request-granularity. This fast-paced control is the only way to maintain tail latency bounds despite unpredictable load changes and to approach closely the efficiency of an idealized oracle.

Bibliography

[1] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. The Resource-as-a-Service (RaaS) Cloud. In *Proceedings of the 4th Workshop on Hot Topics in Cloud Computing*, HotCloud'12, Berkeley, CA, USA, 2012. USENIX Association. Referenced at page 57.

[2] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proc. of OSDI*, 2010. Referenced at page 14.

[3] Karl Johan Åström and Björn Wittenmark. *Adaptive Control*. Dover Publications, 2008. Referenced at pages 35 and 36.

[4] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association. Referenced at pages 27 and 39.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. doi: 10.1145/945445.945462. Referenced at page 39.

[6] L.A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12), 2007. Referenced at pages 10 and 60.

[7] Luiz Andre Barroso. Warehouse-scale computing: Entering the teenage decade. In *ISCA-39 Keynote*, 2011. Referenced at pages 1 and 101.

[8] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2), 2003. Referenced at page 71.

Bibliography

[9] Luiz Andrè Barroso, Jimmy Clidaras, and Urs Hölzle. *The Datacenter as a Computer*. Morgan & Claypool, 2 edition, 2013. Referenced at pages 3, 6, and 10.

[10] Davide B. Bartolini, Riccardo Cattaneo, Gianluca C. Durelli, Martina Maggio, Marco D. Santambrogio, and Filippo Sironi. The Autonomic Operating System Research Project: Achievements and Future Directions. In *Proceedings of the 50th Design Automation Conference*, DAC '13, pages 77:1–77:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488828. Referenced at pages 32, 34, and 47.

[11] Davide B. Bartolini, Filippo Sironi, Martina Maggio, Gianluca C. Durelli, Donatella Sciuto, and Marco D. Santambrogio. Towards a Performance-as-a-service Cloud. In *Proceedings of the 4th Symposium on Cloud Computing*, SoCC '13, pages 26:1–26:2, New York, NY, USA, 2013. ACM. doi: 10.1145/2523616.2525933. Referenced at page 32.

[12] Davide B. Bartolini, Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. Automated Fine-Grained CPU Provisioning for Virtual Machines. *ACM Transactions on Architecture and Code Optimization*, 2014. Referenced at pages 21 and 23.

[13] Nathan Beckmann and Daniel Sanchez. Jigsaw: scalable software-defined caches. In *Proc. of the 22nd Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2013. Referenced at pages 6 and 63.

[14] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011. Referenced at pages 26, 33, and 41.

[15] Patrick Billingsley. *Probability and measure*. John Wiley & Sons, 2008. Referenced at page 73.

[16] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. Referenced at pages 29 and 41.

[17] Alex D. Breslow, Ananta Tiwari, Martin Schulz, Laura Carrington, Lingjia Tang, and Jason Mars. Enabling Fair Pricing on HPC Systems with Node Sharing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 37:1–37:12, New York, NY, USA, 2013. ACM. doi: 10.1145/2503210.2503256. Referenced at page 57.

[18] Edward A Burton, Gerhard Schrom, Fabrice Paillet, Jonathan Douglas, William J Lambert, Kaladhar Radhakrishnan, and Michael J Hill. FIVR: Fully integrated voltage regulators on 4th generation Intel® Core SoCs. In *Proc. of the 29th annual IEEE Applied Power Electronics Conference and Exposition (APEC)*, 2014. Referenced at pages 61, 65, 86, and 87.

[19] Lydia Y. Chen, Danilo Ansaloni, Evgenia Smirni, Akira Yokokawa, and Walter Binder. Achieving Application-Centric Performance Targets via Consolidation on

Multicores: Myth or Reality? In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 37–48, New York, NY, USA, 2012. ACM. doi: 10.1145/2287076.2287083. Referenced at page 31.

[20] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. of the 37th Design Automation Conf.*, 2000. Referenced at pages 6 and 63.

[21] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *Proc. of the 44th annual IEEE/ACM intl. symp. on Microarchitecture*, 2011. Referenced at page 64.

[22] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. In *Proc. of the 50th Design Automation Conf.*, 2013. Referenced at pages 6 and 63.

[23] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency While Preserving Responsiveness. In *Proceedings of the 40th International Symposium on Computer Architecture*, ISCA '13, pages 308–319, New York, NY, USA, 2013. ACM. doi: 10.1145/2485922.2485949. Referenced at pages 29 and 45.

[24] Elliott Cooper-Balis and Bruce Jacob. Fine-grained activation for power reduction in DRAM. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010. Referenced at page 65.

[25] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2): 74–80, February 2013. ISSN 0001-0782. doi: 10.1145/2408776.2408794. Referenced at pages 8, 15, 55, 60, 63, 66, and 86.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Symposium on Opearting Systems Design and Implementation*, OSDI '04, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association. Referenced at page 56.

[27] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2013. Referenced at pages 6, 62, and 63.

[28] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM. doi: 10.1145/2541940.2541941. Referenced at pages 6, 60, and 63.

Bibliography

[29] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: Active Low-power Modes for Main Memory. In *Proc. of the 16th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2011. Referenced at page 65.

[30] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. CoScale: Coordinating CPU and Memory System DVFS in Server Systems. In *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture*, 2012. Referenced at pages 65, 81, and 83.

[31] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), 1974. Referenced at page 11.

[32] Tyler Dwyer, Alexandra Fedorova, Sergey Blagodurov, Mark Roth, Fabien Gaud, and Jian Pei. A Practical Method for Estimating Performance Degradation on Multicore Processors, and Its Application to HPC Workloads. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 83:1–83:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press. doi: 10.1109/SC.2012.11. Referenced at page 57.

[33] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proc. of the 15th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2010. Referenced at pages 6 and 63.

[34] Stijn Eyerman and Lieven Eeckhout. A counter architecture for online DVFS profitability estimation. *IEEE Trans. on Computers*, 59(11), 2010. Referenced at pages 12 and 71.

[35] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A performance counter architecture for computing accurate CPI components. In *Proc. of the 12th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2006. Referenced at page 71.

[36] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D. Bowers, and Michael M. Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of the 3rd Symposium on Cloud Computing*, SoCC '12, pages 20:1–20:14, New York, NY, USA, 2012. ACM. doi: 10.1145/2391229.2391249. Referenced at page 24.

[37] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society. doi: 10.1109/PACT.2007.40. Referenced at pages 31 and 55.

[38] Krisztián Flautner and Trevor Mudge. Vertigo: Automatic Performance-setting for Linux. In *Proc. of the 5th USENIX symp. on Operating Systems Design and Implementation*, 2002. Referenced at pages 61 and 64.

[39] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation*, NSDI'11, pages 323–336, Berkeley, CA, USA, 2011. USENIX Association. Referenced at page 55.

[40] Daniel Gmach, Jerry Rolia, and Ludmila Cherkasova. Selling T-shirts and Time Shares in the Cloud. In *Proceedings of the 12th International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '12, pages 539–546, Washington, DC, USA, 2012. IEEE Computer Society. doi: 10.1109/CCGrid.2012.68. Referenced at pages 2 and 28.

[41] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Proceedings of the 6th Internation Conference on Network and Services Management*, CNSM '10, pages 9–16, Laxenburg, Austria, Austria, 2010. International Federation for Information Processing. doi: 10.1109/CNSM.2010.5691343. Referenced at page 52.

[42] Sriram Govindan, Jie Liu, Aman Kansal, and Anand Sivasubramaniam. Cuanta: Quantifying Effects of Shared On-Chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2nd Symposium on Cloud Computing*, SoCC '11, pages 22:1–22:14, New York, NY, USA, 2011. ACM. doi: 10.1145/2038916.2038938. Referenced at pages 24, 29, and 57.

[43] Greenpeace. How dirty is your data? a look at the energy choices that power cloud computing. `http://www.greenpeace.org/international/Global/international/publications/climate/2011/Cool%20IT/dirty-data-report-greenpeace.pdf`, 2011. Referenced at page 3.

[44] Charles Miller Grinstead and James Laurie Snell. *Introduction to probability*. American Mathematical Soc., 1998. Referenced at page 71.

[45] Boris Grot, Joel Hestness, Stephen W Keckler, and Onur Mutlu. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proc. of the 38th Intl. Symp. on Computer Architecture*, 2011. Referenced at pages 6 and 63.

[46] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proc. of the 40th annual IEEE/ACM intl. symp. on Microarchitecture*, 2007. Referenced at pages 6, 63, and 68.

[47] J. Hamilton. Overall Data Center Costs, 2010. URL `http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx`. Referenced at pages XXVII, 3, and 4.

[48] Per Hammarlund, Alberto J. Martinez, Atiq A. Bajwa, David L. Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, Randy B. Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, Steve Gunther, Tom Piazza, and Ted Burton. Haswell: The Fourth-Generation Intel Core Processor. *IEEE Micro*, 34(2), 2014. Referenced at pages 61, 65, and 86.

Bibliography

[49] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004. Referenced at pages 36 and 37.

[50] Andrew Hilton, Neeraj Eswaran, and Amir Roth. FIESTA: A sample-balanced multi-program workload methodology. In *Proc. of the Workshop on Modeling, Benchmarking and Simulation (MoBS)*, 2009. Referenced at page 95.

[51] Henry Hoffmann, Jonathan Eastep, Marco D. Santambrogio, Jason E. Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 79–88, New York, NY, USA, 2010. ACM. doi: 10.1145/1809049.1809065. Referenced at page 56.

[52] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. A Generalized Software Framework for Accurate and Efficient Management of Performance Goals. In *Proceedings of the 11th International Conference on Embedded Software*, EMSOFT '13, pages 19:1–19:10, Piscataway, NJ, USA, 2013. IEEE Press. doi: 10.1109/EMSOFT.2013.6658597. Referenced at pages 32 and 54.

[53] Michael Huang, Jose Renau, Seung-Moon Yoo, and Josep Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *Proc. of the 33rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2000. Referenced at pages 61 and 64.

[54] Wei Huang, C. Lefurgy, W. Kuk, A. Buyuktosunoglu, M. Floyd, K. Rajamani, M. Allen-Ware, and B. Brock. Accurate Fine-Grained Processor Power Proxies. In *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture*, 2012. Referenced at pages 81 and 82.

[55] Intel. Intel Xeon Processor E3-1200 v3 Product Family Datasheet, 2014. Referenced at page 87.

[56] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006. Referenced at page 64.

[57] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *Proc. SIGMETRICS*, 2007. Referenced at pages 6, 63, and 68.

[58] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proc. of the 37th Intl. Symp. on Computer Architecture*, 2010. Referenced at page 88.

[59] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In

Proc. of the 37th Intl. Symp. on Computer Architecture, 2010. Referenced at page 86.

[60] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proc. of the 49th Design Automation Conf.*, 2012. Referenced at pages 6 and 63.

[61] Alexandra Jimborean, Konstantinos Koukos, Vasileios Spiliopoulos, David Black-Schaffer, and Stefanos Kaxiras. Fix the Code. Don'T Tweak the Hardware: A New Compiler Approach to Voltage-Frequency Scaling. In *Proc. of the 12th IEEE/ACM Intl. Symp. on Code Generation and Optimization*, 2014. Referenced at page 12.

[62] Changyeon Jo, Erik Gustafsson, Jeongseok Son, and Bernhard Egger. Efficient Live Migration of Virtual Machines Using Shared Storage. In *Proceedings of the 9th International Conference on Virtual Execution Environments*, VEE '13, pages 41–50, New York, NY, USA, 2013. ACM. doi: 10.1145/2451512.2451524. Referenced at page 32.

[63] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. of the 12th intl. conf. on Extending Database Technology*, 2009. Referenced at page 87.

[64] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications. In *Proc. of the IEEE Intl. Symp. on Workload Characterization*, 2014. Referenced at pages 61, 64, and 65.

[65] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proc. of SoCC*, 2012. Referenced at page 15.

[66] Harshad Kasture and Daniel Sanchez. Ubik: Efficient Cache Sharing with Strict Qos for Latency-critical Workloads. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 729–742, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541944. Referenced at pages XXVII, XXVIII, 5, 8, 9, 15, 16, 17, 19, 20, 24, 60, 61, 62, 63, 64, 65, 67, 68, 86, and 88.

[67] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Power management for highly utilized, latency-critical systems. *Under double-blind review for ISCA 2015*, 2014. Referenced at pages 21 and 59.

[68] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time DVFS orchestration in superscalar processors. In *Proc. of the 7th ACM international conference on Computing frontiers*, 2010. Referenced at page 71.

[69] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Raj Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *Proc. RTAS*, 2014. Referenced at pages 6 and 63.

[70] Wonyoung Kim, Meeta Sharma Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proc. of the 14th IEEE intl. symp. on High Performance Computer Architecture*, 2008. Referenced at pages 61 and 65.

[71] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010. Referenced at pages 6 and 63.

[72] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007. Referenced at pages 27 and 39.

[73] Brian Kocoloski, Jiannan Ouyang, and John Lange. A Case for Dual Stack Virtualization: Consolidating HPC and Commodity Applications in the Cloud. In *Proceedings of the 3rd Symposium on Cloud Computing*, SoCC '12, pages 23:1–23:7, New York, NY, USA, 2012. ACM. doi: 10.1145/2391229.2391252. Referenced at page 53.

[74] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Proc. of the 45th annual meeting of the ACL*, 2007. Referenced at page 87.

[75] Konstantinos Koukos, David Black-Schaffer, Vasileios Spiliopoulos, and Stefanos Kaxiras. Towards More Efficient Execution: A Decoupled Access-execute Approach. In *Proc. of the Intl. Conf. on Supercomputing*, 2013. Referenced at page 81.

[76] Nasser A Kurd, Subramani Bhamidipati, Christopher Mozak, Jeffrey L Miller, Timothy M Wilson, Mahadev Nemani, and Muntaquim Chowdhury. Westmere: A family of 32nm IA processors. In *Proc. of the IEEE Intl. Solid-State Circuits Conf.*, 2010. Referenced at page 86.

[77] Jacob Leverich. *Future Scaling of Datacenter Power-Efficiency*. PhD thesis, Stanford University, 2014. Referenced at pages 3, 4, and 5.

[78] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Sub-millisecond Quality-of-Service. In *Proceedings of the 9th European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. ACM. Referenced at pages 26 and 56.

[79] B. Li, L. Zhao, R. Iyer, L.S. Peh, M. Leddige, M. Espig, S.E. Lee, and D. Newell. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *Journal of Parallel and Distributed Computing*, 71(5), 2011. Referenced at pages 6, 61, 63, and 68.

[80] J. Li and J.F. Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *Proc. of the 12th IEEE intl. symp. on High Performance Computer Architecture*, 2006. Referenced at page 64.

[81] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proc. of the 42nd annual IEEE/ACM intl. symp. on Microarchitecture*, 2009. Referenced at page 80.

[82] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining Insights into Multicore Cache Partitioning: Bridging the Gap between Simulation and Real Systems. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*, HPCA '08, pages 367–378, Washington, DC, USA, 2008. IEEE Computer Society. doi: 10.1109/HPCA.2008.4658653. Referenced at page 55.

[83] Yanpei Liu, Stark C Draper, and Nam Sung Kim. SleepScale: Runtime Joint Speed Scaling and Sleep States Management for Power Efficient Data Centers. In *Proc. of the 41st Intl. Symp. on Computer Architecture*, 2014. Referenced at page 64.

[84] David Lo and Christos Kozyrakis. Dynamic Management of TurboMode in Modern Multi-core Chips. In *Proc. of the 20th IEEE intl. symp. on High Performance Computer Architecture*, 2014. Referenced at pages 20, 61, 64, 65, and 95.

[85] David Lo, Liqun Cheng, Rama Govindaraju, Luiz Barroso, and Christos Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proc. of the 41st Intl. Symp. on Computer Architecture*, 2014. Referenced at pages 11, 15, 61, 62, 63, 64, 66, 68, 88, 89, and 99.

[86] Jacob R. Lorch and Alan Jay Smith. Improving Dynamic Voltage Scaling Algorithms with PACE. *SIGMETRICS Perform. Eval. Rev.*, 29(1), 2001. Referenced at pages 64 and 66.

[87] Krishna T Malladi, Ian Shaeffer, Liji Gopalakrishnan, David Lo, Benjamin C Lee, and Mark Horowitz. Rethinking DRAM power modes for energy proportionality. In *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture*, 2012. Referenced at page 65.

[88] R Manikantan, Kaushik Rajan, and R Govindarajan. Probabilistic shared cache management (PriSM). In *Proc. of the 39th Intl. Symp. on Computer Architecture*, 2012. Referenced at pages 6 and 63.

[89] N. Mankiw. *Principles of Microeconomics*. Cengage Learning, 2011. Referenced at page 78.

[90] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proc. of the EuroSys Conf.*, 2012. Referenced at page 87.

[91] Dan C. Marinescu. *Cloud Computing: Theory and Practice*. 1st edition, 2013. Referenced at pages 1 and 6.

Bibliography

[92] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th International Symposium on Microarchitecture*, MICRO '11, pages 248–259, New York, NY, USA, 2011. ACM. doi: 10.1145/2155620.2155650. Referenced at pages 5, 6, 24, 29, 57, 62, 63, and 67.

[93] David Meisner and Thomas F Wenisch. Stochastic queuing simulation for data center workloads. *EXERT*, 2010. Referenced at page 87.

[94] David Meisner and Thomas F. Wenisch. DreamWeaver: Architectural Support for Deep Sleep. In *Proc. of the 17th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2012. Referenced at pages 10, 61, and 65.

[95] David Meisner, Brian T. Gold, and Thomas F. Wenisch. The PowerNap Server Architecture. In *Proc. of the 14th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, volume 29, pages 3:1–3:24, New York, NY, USA, February 2011. ACM. doi: 10.1145/1925109.1925112. Referenced at pages 10, 28, 60, 61, and 65.

[96] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power Management of Online Data-intensive Services. In *Proc. of the 38th Intl. Symp. on Computer Architecture*, 2011. Referenced at pages 64, 65, and 87.

[97] Paul Menage. CGROUPS. `http://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`, 2013. Referenced at pages 27 and 39.

[98] C. Metz. Facebook Lets You Spy on Its Data Centers. *Wired Enterprise*, 2013. Referenced at page 4.

[99] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N Patt. Predicting performance impact of DVFS for realistic memory systems. In *Proc. of the 45th annual IEEE/ACM intl. symp. on Microarchitecture*, 2012. Referenced at page 71.

[100] Miquel Moreto, Francisco J Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. FlexDCP: A QoS framework for CMP architectures. *SIGOPS Operating Systems Review*, 43(2), 2009. Referenced at pages 6, 63, and 68.

[101] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proc. of the 44th annual IEEE/ACM intl. symp. on Microarchitecture*, 2011. Referenced at pages 6, 63, 67, and 87.

[102] Amnon Naamad. New challenges in performance engineering. In *Proc. ICPI*, 2012. Referenced at page 2.

[103] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 237–250, New York,

NY, USA, 2010. ACM. doi: 10.1145/1755913.1755938. Referenced at pages 25, 39, and 53.

[104] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical Modeling of Pipeline Parallelism. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 281–290, Washington, DC, USA, 2009. IEEE Computer Society. doi: 10.1109/PACT.2009.28. Referenced at page 44.

[105] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006. Referenced at pages 6 and 63.

[106] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic Distributed Resource Scaling for Infrastructure-as-a-Service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7. URL `https://www.usenix.org/conference/icac13/technical-sessions/presentation/nguyen`. Referenced at page 53.

[107] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the 24th Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM. doi: 10.1145/2517349.2522716. Referenced at page 53.

[108] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th European Conference on Computer Systems*, EuroSys '09, pages 13–26, New York, NY, USA, 2009. ACM. doi: 10.1145/1519065.1519068. Referenced at pages 25, 32, 34, 37, 51, 52, and 54.

[109] M.K. Qureshi and Y.N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. of the 39th annual IEEE/ACM intl. symp. on Microarchitecture*, 2006. Referenced at pages 6, 61, 63, 64, 68, and 88.

[110] K. Rajamani, F. Rawson, M. Ware, H. Hanson, J. Carter, Todd Rosedahl, Andrew Geissler, Guillermo Silva, and Hong Hua. Power-performance management on an IBM POWER7 server. In *Proc. of the ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010. Referenced at page 61.

[111] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A Comparison of High-level Full-system Power Models. In *Proc. of the 2008 Conference on Power Aware Computing and Systems*, 2008. Referenced at page 80.

[112] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Doron Rajwan, and Eliezer Weissmann. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2), 2012. Referenced at pages 61 and 95.

[113] Barry Rountree, David K Lowenthal, Martin Schulz, and Bronis R de Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *Proc. of the Intl. Green Computing Conf.*, 2011. Referenced at page 71.

[114] Daniel Sanchez and Christos Kozyrakis. The ZCache: Decoupling ways and associativity. In *Proc. of the 43rd annual IEEE/ACM intl. symp. on Microarchitecture*, 2010. Referenced at page 87.

[115] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. of the 38th Intl. Symp. on Computer Architecture*, 2011. Referenced at pages 6, 63, 68, and 87.

[116] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *Proc. of the 40th Intl. Symp. on Computer Architecture*, 2013. Referenced at pages 16, 86, and 87.

[117] Conrad Sanderson. Armadillo: An Open Source C++ Linear Algebra Library for Fast Prototyping and Computationally Intensive Experiments. Technical report, NICTA, 2010. Referenced at page 37.

[118] Hiroshi Sasaki, Teruo Tanimoto, Koji Inoue, and Hiroshi Nakamura. Scalability-Based Manycore Partitioning. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 107–116, New York, NY, USA, 2012. ACM. doi: 10.1145/2370816.2370833. Referenced at pages 34, 40, 47, and 56.

[119] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. Coordinated Power-performance Optimization in Manycores. In *Proc. of the 22nd Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2013. Referenced at page 64.

[120] Eric Schurman and Jake Brutlag. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search. In *Velocity*, 2009. Referenced at pages 15 and 63.

[121] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM. doi: 10.1145/2465351.2465386. Referenced at page 31.

[122] Alberto Scolari, Filippo Sironi, Davide B. Bartolini, Donatella Sciuto, and Marco D. Santambrogio. Coloring the Cloud for Predictable Performance. In *Proceedings of the 4th Symposium on Cloud Computing*, SoCC '13, pages 47:1–47:2, New York, NY, USA, 2013. ACM. doi: 10.1145/2523616.2525955. Referenced at page 55.

[123] Akbar Sharifi, Shekhar Srikantaiah, Asit K. Mishra, Mahmut Kandemir, and Chita R. Das. METE: Meeting End-to-End QoS in Multicores Through System-Wide Resource Management. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '11, pages 13–24, New York, NY, USA, 2011. ACM. doi: 10.1145/1993744.1993747. Referenced at pages 6, 42, 54, 61, 63, 64, and 68.

[124] Kai Shen, Arrvindh Shriraman, Sandhya Dwarkadas, Xiao Zhang, and Zhuan Chen. Power Containers: An OS Facility for Fine-grained Power and Energy Management on Multicore Servers. In *Proc. of the 18th intl. conf. on Architectural Support for Programming Languages and Operating Systems*, 2013. Referenced at page 81.

[125] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *Proceedings of the 2nd Symposium on Cloud Computing*, SoCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM. doi: 10.1145/2038916.2038921. Referenced at pages 25, 32, 34, 37, 48, and 52.

[126] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM Journal of Research and Development*, 55(3), 2011. Referenced at page 86.

[127] Filippo Sironi, Davide B. Bartolini, Simone Campanoni, Fabio Cancare, Henry Hoffmann, Donatella Sciuto, and Marco D. Santambrogio. Metronome: Operating System Level Performance Management via Self-Adaptive Computing. In *Proceedings of the 49th Design Automation Conference*, DAC '12, pages 856–865, New York, NY, USA, 2012. ACM. doi: 10.1145/2228360.2228514. Referenced at pages 31, 32, 55, and 56.

[128] Filippo Sironi, Donatella Sciuto, and Marco D. Santambrogio. A Performance-Aware Quality of Service-Driven Scheduler for Multicore Processors. *SIGBED Rev.*, 11(1):50–55, February 2014. ISSN 1551-3688. doi: 10.1145/2597457.2597464. Referenced at pages 32, 34, and 47.

[129] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for Continuously Adaptive DVFS. In *Proc. of the 2011 International Green Computing Conference (IGCC)*, 2011. Referenced at page 81.

[130] Standard Performance Evaluation Corporation. SPECvirt_sc2013. `http://www.spec.org/virt_sc2013/`, 2013. Referenced at page 29.

[131] Justin Teo, Jonathan P How, and Eugene Lavretsky. Proportional-integral controllers for minimum-phase nonaffine-in-control systems. *IEEE Trans. on Automatic Control*, 55(6), 2010. Referenced at page 73.

[132] Paul Turner, Bharata B. Rao, and Nikhil Rao. CPU Bandwidth Control for CFS. In *Proceedings of the Linux Symposium*, pages 245–254, 2010. Referenced at pages 39 and 40.

[133] Aniruddha N Udipi, Naveen Muralimanohar, Niladrish Chatterjee, Rajeev Balasubramonian, Al Davis, and Norman P Jouppi. Rethinking DRAM design and organization for energy-constrained multi-cores. In *Proc. of the 37th Intl. Symp. on Computer Architecture*, 2010. Referenced at page 65.

[134] Daniel Wong and Murali Annavaram. KnightShift: Scaling the Energy Proportionality Wall Through Server-Level Heterogeneity. In *Proc. of the 45th annual*

IEEE/ACM intl. symp. on Microarchitecture, 2012. Referenced at pages 11, 61, and 65.

[135] Xen Project. Credit Scheduler – Xen. `http://wiki.xen.org/wiki/CreditScheduler`, 2013. Referenced at page 39.

[136] Ruibin Xu, Chenhai Xi, Rami Melhem, and Daniel Moss. Practical PACE for Embedded Systems. In *Proc. of the 4th ACM International Conference on Embedded Software*, 2004. Referenced at pages 64 and 66.

[137] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proc. of the 40th Intl. Symp. on Computer Architecture*, 2013. Referenced at pages 6, 62, and 63.

[138] Wanghong Yuan and Klara Nahrstedt. Energy-efficient Soft Real-time CPU Scheduling for Mobile Multimedia Systems. In *Proc. of the 19th Symp. on Operating System Principles*, 2003. Referenced at pages 64 and 66.

[139] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. MemGuard: Memory Bandwidth Reservation System for Efficient Performance Isolation in Multi-Core Platforms. In *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium*, RTAS '13, pages 55–64, Washington, DC, USA, 2013. IEEE Computer Society. doi: 10.1109/RTAS.2013.6531079. Referenced at page 55.

[140] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. ControlWare: A Middleware Architecture for Feedback Control of Software Performance. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS '02, pages 301–310, Washington, DC, USA, 2002. IEEE Computer Society. doi: 10.1109/ICDCS.2002.1022267. Referenced at pages 32 and 54.

[141] X. Zhang, S. Dwarkadas, and K. Shen. Hardware Execution Throttling for Multi-core Resource Management. In *Proc. of the USENIX Annual Technical Conf.*, 2009. Referenced at pages 6 and 63.

February 16, 2015

Made with LaTeX