FUTURE SCALING OF DATACENTER POWER-EFFICIENCY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jacob Barton Leverich
March 2014

This dissertation is online at: http://purl.stanford.edu/gd102bj0840

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mendel Rosenblum

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Datacenters are critical assets for today's Internet giants (Google, Facebook, Microsoft, Amazon, etc.). They host extraordinary amounts of data, serve requests for millions of users, generate untold profits for their operators, and have enabled new modes of communication, socialization, and organization for society at-large. Their steady growth in scale and capability has allowed datacenter operators to continually expand the reach and benefit of their services. The motivation for the work presented in this dissertation stems from a simple premise: *future scaling of datacenter capability depends upon improvements to server power-efficiency*. That is, without improvements to power-efficiency, datacenter operators will soon face a limit to the utility and capability of existing facilities that might result in (1) a sudden boom in datacenter construction, (2) a rapid increase in the cost to operate existing datacenters, or (3) stagnation in the growth of datacenter capability. This limit is akin to the "Power Wall" that the CPU industry has been grappling with for nearly a decade now.

Like the CPU "Power Wall," the problem is not that we can't build datacenters with more capability in the future *per se*. Rather, it will become increasingly difficult to do this economically; even now, the cost to provision power and cooling infrastructure is a substantial fraction of datacenter "Total Cost of Ownership." The root cause of this problem is the recent failure of Dennard scaling for semiconductor technologies smaller than 90 nanometers. Even though Moore's Law continues to march on at a steady pace, granting us exponential growth in transistor count in new processors, we can no longer make full use of that transistor count without also increasing the power density of those new processors. Thus, in order for datacenter operators to sustain the rate of growth in capability that they have come to expect, they must either provision new power and cooling infrastructure to support future servers (at exceptional cost), or find other ways to improve the power-efficiency of datacenters that do not depend on semiconductor technology scaling. Indeed, the initial onset of this problem led to rapid improvements to the efficiency of power delivery and cooling within datacenters, reducing non-server power consumption by an order of magnitude. Unfortunately, those improvements were essentially one-time benefits and have now been exhausted.

In this dissertation, we show that most of the future opportunity to improve datacenter power-efficiency lies in improving the power-efficiency of the servers themselves, as most of the inefficiency in the rest of a datacenter has largely been eliminated. Then, we explore four compelling opportunities to improve server power-efficiency: two hardware proposals that explicitly reduce the power consumption of servers, and two

software proposals that improve the power-efficiency of servers operating as a cluster.

First, we present Multicore DIMM (MCDIMM), a modification to the architecture of traditional DDRx main memory modules optimized for energy-efficiency. MCDIMM modules divide the wide, 64-bit *rank* interface presented by ordinary DIMMs into smaller *rank subsets*. By accessing rank subsets individually, fewer DRAM chips are activated per column access (i.e. cache-line refill), which greatly reduces dynamic energy consumption. Additionally, we describe an energy-efficient implementation of error-correction codes for MCDIMMs, as well as "chipkill" reliability, which tolerates the failure of entire DRAM devices. For ordinary server configurations and across a wide range of benchmarks, we estimate more than 20% average savings in memory dynamic power consumption, though the impact on total system power consumption is more modest. We also describe additional, unexpected performance and static-power consumption benefits from rank subsetting.

Second, we propose an architecture for per-core power gating (PCPG) of multicore processors, where the power supply for individual CPU cores can be cut entirely. We propose that servers running at low to moderate utilization, as is common in datacenters, could operate with some of their cores gated off. Gating the power to a core eliminates its static power consumption, but requires flushing its caches and precludes using the core to execute workloads until power is restored. In our proposal, we improve the utility of PCPG by coordinating power gating actions with the operating system, migrating workloads off of gated cores onto active cores. This is in contrast to contemporary industry implementations of PCPG that gate cores *reactively*. We control the gating of cores with a dynamic power manager which continually monitors CPU utilization. Our OS-integrated approach to PCPG maximizes the opportunities available to utilize PCPG relative to OS-agnostic approaches, and protects applications from incurring the latency of waking a sleeping core. We show that PCPG is beneficial for datacenter workloads, and that it can reduce CPU power consumption by up to 40% for underutilized systems with minimal impact on performance.

The preceding hardware proposals seek to improve the power-efficiency of individual servers directly. The improvements are modest, however, as there are many factors that contribute to the inefficiency of servers (i.e. cooling fans, spinning disks, power regulator inefficiency, etc.). More to the point, techniques like PCPG only address the power-inefficiency of underutilized CPUs, and do little to address the inefficiency of the rest of the components within a server when it is at low utilization. To address this short-coming, this dissertation then explores a different tack and holistically assesses how utilization across clusters of servers can be manipulated to improve power-efficiency.

First, we describe how contemporary distributed storage systems, such as Hadoop's Distributed File System (HDFS), expect the perpetual availability of the vast majority of servers in a cluster. This artificial expectation prevents the use of low-power modes in servers; we cannot trivially turn servers off or put them into a standby mode without the storage system assuming the server has failed. Consequently, even if such a cluster is grossly underutilized, we cannot disable servers in order to reduce its aggregate power consumption. Thus, these clusters tend to be tragically power-inefficient at low utilization. We propose a simple set of modifications to HDFS to rectify this problem, and show that these storage systems can be built to be

power-proportional. We find that running Hadoop clusters in fractional configurations can save between 9% and 50% of energy consumption, and that there is a trade-off between performance energy consumption.

Finally, we set out to determine why datacenter operators chronically underutilize servers which host *latency-sensitive* workloads. Using memcached as a canonical latency-sensitive workload, we demonstrate that latency-sensitive workloads suffer substantial degradation in quality-of-service (QoS) when co-located with other datacenter workloads. This encourages operators to be cautious when provisioning or co-locating services across large clusters, and this ultimately manifests as the low server utilization we see ubiquitously in datacenters. However, we find that these QoS problems typically manifest in a limited number of ways: as increases in queuing delay, scheduling delay, or load imbalance of the latency-sensitive workload. We evaluate several techniques, including *interference-aware provisioning* and replacing Linux's CPU scheduler with a scheduler previous proposed in the literature, to ameliorate QoS problems when co-locating memcached with other workloads. We ultimately show that good QoS for latency-sensitive applications can indeed be maintained while still running these servers at high utilization. Judicious application of these techniques can greatly improve server power-efficiency, and raise a datacenter's effective throughput per TCO dollar by up to 53%.

All told, we have found that there exists considerable opportunity to improve the power-efficiency of datacenters despite the failure of Dennard scaling. The techniques presented in this dissertation are largely orthogonal, and may be combined. Through judicious focus on server power-efficiency, we can stave off stagnation in the growth of online services or an explosion of datacenter construction, at least for a time.

Acknowledgements

My most cherished T-shirt was produced by the American Association for the Advancement of Science, and given to me by my father. It reads “6 Steps to Completing Your Dissertation” and is followed by six cartoons depicting: Drudgery, Procrastination, Panic, Despair, Drudgery, and Printing. I doubt there exists a more compact and accurate portrayal of my experience in pursuing a PhD. However, I feel it should not be interpretted as a strictly linear path. Instead, the first five steps are waypoints each student travels through along their own unique route. With determination and a little luck, printing is the final destination.

Figure 1: My favorite T-shirt design.

I took a little longer than average to complete my PhD, so I may have transited the Panic and Despair stations a few more times than normal. I’d like to briefly acknowledge the two people who made sure I kept moving along. First, Christos, my advisor, was quick to assure me that I had accumulated plenty enough work to earn a PhD. When he shared this with me it was a huge revelation and a profound relief. I felt empowered to complete my dissertation on my own terms, and to tie up the loose ends I cared most passionately about. This could not have happened at a better moment. Second, Theresa, my wife, was both sympathetic and dogged in her determination that I should stick to it, and always encouraged me when I buckled down for a conference deadline. Written words will never convey my feelings here so I won’t torture this page much more except to say “loves.”

It goes without saying, countless others contributed to my growth and well-being during my graduate studies: Mom, Dad, Jubi, Jorge, Felix, Steve, Larriann, Mike, Della, Joel, Alexis, Jared, Adam, Scot, Ryon, Jason, Whit, Genesis, The Kozyrakis Group, The Horowitz Group, The Olukotun Group, and Team Dawg. Cheers!

Contents

Abstract **iv**

Acknowledgements **vii**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Total Cost of Ownership . . . 2
1.2.1 Energy's Contribution to TCO . . . 4
1.2.2 Power's Contribution to TCO . . . 6
1.3 Datacenter Capability is Limited by Critical Power . . . 7
1.3.1 Other Strategies to Increase Datacenter Capability . . . 8
1.3.2 The End of Dennard Scaling and Advent of Dark Silicon . . . 9
1.4 Future Improvements to Datacenter Power-Efficiency . . . 10
1.4.1 Typical Server Power Consumption . . . 11
1.4.2 Raising Server Utilization to Improve Power-Efficiency . . . 13
1.5 Contributions of this Dissertation . . . 14
1.5.1 Energy-Efficient DRAM module . . . 15
1.5.2 Per-Core Power Gating . . . 15
1.5.3 Power-Proportional Hadoop . . . 16
1.5.4 High Server Utilization and Sub-millisecond Quality-of-Service . . . 16
1.5.5 Summary . . . 16

2 Improving Power-Efficiency with Memory Rank Subsetting **17**

2.1 Energy Efficient and Reliable Memory Modules . . . 19
2.1.1 Background . . . 19
2.1.2 Implications of Rank Subsetting . . . 23
2.1.3 Adding Reliability to Multicore DIMMs . . . 27
2.2 Experimental Setup . . . 30
2.3 Results . . . 33

2.3.1 Impact of the Number of Ranks on the Frequency of Row-Buffer Conflicts 33
2.3.2 Single-Rank Performance and Power Efficiency 34
2.3.3 Four-Rank Performance and Power Efficiency 37
2.3.4 Power and Performance of Chipkill-level Reliability 38
2.4 Related Work 40
2.5 Conclusion 41

3 Per-Core Power Gating for Datacenters **43**
3.1 Introduction 44
3.2 Background on Power Management Techniques 46
3.2.1 Voltage Scaling 46
3.2.2 C-states 46
3.2.3 Multiple Voltage Domains 47
3.2.4 Motivation for our work 48
3.3 Per-Core Power Gating Architecture 48
3.3.1 System Architecture 48
3.3.2 Circuits for Per-core Power Gating 49
3.3.3 Implementation 51
3.3.4 Power Gating Process 52
3.4 Management Policies for Per-Core Power Gates 52
3.4.1 Static Policies 52
3.4.2 Dynamic Policies 52
3.4.3 Interaction with Existing Voltage Scaling Techniques 53
3.5 Methodology 54
3.5.1 Workloads 54
3.5.2 Performance Model 55
3.5.3 Hybrid Power Model 55
3.5.4 Dynamic Power Management Daemon 56
3.6 Evaluation 57
3.6.1 Characterization of Per-core Power Gating 57
3.6.2 Evaluation of PCPG with Realistic Utilization Traces 59
3.7 Related Work 63
3.7.1 Intel Nehalem and AMD Bobcat 64
3.8 Conclusions 65

4 On the Energy (In)efficiency of Hadoop Clusters **66**
4.1 Introduction 67
4.2 Improving Hadoop's Energy-Efficiency 68

4.2.1 Data Layout Overview . . . 68
4.2.2 A Replication Invariant for Energy-Efficiency . . . 69
4.2.3 Implementation . . . 70
4.3 Evaluation . . . 71
4.3.1 Methodology . . . 71
4.3.2 Results . . . 72
4.4 Towards an Energy-Efficient Hadoop . . . 74
4.4.1 Data Layout . . . 74
4.4.2 Data Availability . . . 74
4.4.3 Reliability and Durability . . . 74
4.4.4 Dynamic Scheduling Policies . . . 75
4.4.5 Node Architecture . . . 75
4.4.6 Workloads and Applications . . . 75
4.5 Related Work . . . 75
4.6 Conclusions . . . 76

5 High Server Utilization and Sub-millisecond Quality-of-Service 77
5.1 Introduction . . . 77
5.2 Improving Utilization through Co-location . . . 80
5.3 Analysis of QoS Vulnerabilities . . . 81
5.3.1 Analysis Methodology . . . 82
5.3.2 Analysis of Memcached's QoS Sensitivity . . . 83
5.3.3 Scheduling Delay Analysis . . . 89
5.3.4 Discussion . . . 91
5.4 Addressing QoS Vulnerabilities . . . 92
5.4.1 Tolerating Queuing Delay . . . 92
5.4.2 Tolerating Load Imbalance . . . 93
5.4.3 Tolerating Scheduling Delay . . . 94
5.5 Co-location Benefits . . . 97
5.5.1 Facebook Scenario . . . 98
5.5.2 Google Scenario . . . 100
5.6 Related Work . . . 101
5.7 Conclusions . . . 101

6 Concluding Remarks 103
6.1 Synergy Amongst Contributions . . . 103
6.2 Future Work . . . 104
6.3 Summary . . . 105

Bibliography **106**

List of Tables

1.1 Datacenter “Total Cost of Ownership” model proposed by James Hamilton. 3

2.1 Comparison of DRAM schemes for high-reliability. 29
2.2 Power and performance parameters of the memory hierarchy used in this chapter. 31
2.3 SPLASH-2 datasets and SPEC 2006 application mixes used in this chapter. 32

3.1 Comparison of the main alternatives to per-core power gating. 47
3.2 Results applying PCPG to a variety of traces of commercial server utilization. 61

5.1 Latency breakdown of an average memcached request. 85
5.2 Results of co-locating memcached with SPEC CPU2006 workloads in the Facebook scenario. 98
5.3 Results of co-locating memcached with SPEC CPU2006 workloads in the Google scenario. . 100

List of Figures

1 My favorite T-shirt design. vii

1.1 Monthly datacenter expenses by category. 4
1.2 Site of Google's datacenter in The Dalles, OR. 5
1.3 Combined impact of Moore's Law and Dennard scaling on capability and power consumption. 7
1.4 Impact of the demise of Dennard Scaling. 9
1.5 Power vs. CPU utilization for a mid-range server. 11
1.6 The relationship between power consumption, utilization, and efficiency. 12
1.7 Power consumption by server subsystem. 13
1.8 An alternative strategy to improve a given server's power-efficiency is to raise its utilization. 14

2.1 A canonical representation of a DRAM chip with 8 banks. 19
2.2 A conventional memory channel where a memory controller and two memory modules are connected through a shared bus. 20
2.3 Random access latencies of chips from successive DRAM generations. 22
2.4 DRAM power breakdown of a Micron 2Gb DDR3-1333 SDRAM chip. 23
2.5 Comparison between conventional DIMMs and DIMMs with rank-subsetting. 24
2.6 A memory channel with two Multicore DIMMs. 25
2.7 System architecture assumed for evaluation of rank-subsetting. 30
2.8 Frequency of row-buffer conflicts and relative IPC when number of active threads and ranks are varied. 34
2.9 Memory and system level power and performance on a system with 1 rank per memory channel. 35
2.10 Memory and system level power and performance on a system with 4 ranks per memory channel. 37
2.11 Power and performance of applications on systems with chipkill-level reliability. 39

3.1 Relationship between Per-Core Power Gating, DVFS, and global deep-sleep states. 45
3.2 The system architecture for PCPG. 49
3.3 Circuit and details of the power-gate design. 50

3.4 Power measurements of our Phenom X4 9850 system as it goes through P-state transitions. . 56
3.5 Performance and power consumption for SPECpower_ssj2008 with and without PCPG. . . . 57
3.6 Trace of CPU load while running SPECpower_ssj2008. 58
3.7 Performance and power consumption for SPECpower_ssj2008 with dynamic PCPG. 59
3.8 Trace of power measurements while running SPECpower_ssj2008. 60
3.9 CPU load and power consumption as the SAP05 trace is played. 60
3.10 Results of applying PCPG to the HCOM19 trace. 62
3.11 Even at exceptionally low load, Nehalem rarely uses its per-core power gating capability when serving memcached requests. 65

4.1 Opportunities for power-efficiency improvements in Hadoop workloads. 67
4.2 Impact of block layout policy on data availability. 69
4.3 Runtime and energy results of a scaled-down cluster. 72
4.4 System inactivity distribution of a scaled-down cluster. 73

5.1 Typical server utilization leads to poor power-efficiency. Reproduced from Chapter 1. . . . 78
5.2 Trace of CPU and memory utilization and allocation over a 30-day period in a 12,000-node cluster at Google. CPU and memory resources are greatly underutilized but over-reserved. Adapted from [133]. 79
5.3 Life-cycle of a memcached request. 84
5.4 Impact of heavy L3 interference on latency. Interference causes substantial queuing delay at high load, but has little impact at low to moderate load (e.g., at 30%). 87
5.5 Impact of load imbalance on memcached QoS. 88
5.6 Impact of context-switching with other workloads on memcached latency. 89
5.7 Demonstration of scheduler wait-time induced on memcached when run concurrently with an antagonist. 90
5.8 Achieved QoS when co-scheduling two latency-sensitive services on a processor core with Linux's CFS scheduler. 91
5.9 A depiction of CFS run-queue when a task wakes up. 92
5.10 Pinning memcached threads to distinct cores greatly improves load balance, consequently improving tail latency. 93
5.11 Comparison of scheduling algorithms. 96
5.12 A depiction of BVT run-queue when a task wakes up. 97

Chapter 1

Introduction

1.1 Motivation

Datacenters are critical assets for today's Internet giants (Google, Facebook, Microsoft, Amazon, etc.). They host extraordinary amounts of data, serve requests for millions of users, generate untold profits for their operators, and have enabled new modes of communication, socialization, and organization for society at-large. Their steady growth in scale and capability has allowed datacenter operators to continually expand the reach and benefit of their services. The motivation for the work presented in this dissertation stems from a simple premise: *future scaling of datacenter capability depends upon improvements to server power-efficiency*. That is, without improvements to power-efficiency, datacenter operators will soon face a limit to the utility and capability of existing facilities that might result in (1) a sudden boom in datacenter construction, (2) a rapid increase in the cost to operate existing datacenters, or (3) stagnation in the growth of datacenter capability. This limit is akin to the "Power Wall" that the semiconductor industry has been grappling with for nearly a decade now [38]. In this chapter, we will argue the merit of this premise, discuss at a high-level the contemporary opportunities available to improve power-efficiency, and introduce the contributions of this dissertation towards addressing this new "Datacenter Power Wall".

Before narrowing this argument down to the concerns of datacenter operators, it is important to acknowledge the stake that humanity at-large has in addressing datacenter power consumption. Our long-term comfort and survival as a species depends upon addressing anthropogenic climate change [125]. Datacenters consume a significant and growing fraction of all electric power generated in the world, and are responsible for substantial carbon dioxide emissions [82]. It is incumbent upon us to reduce datacenter power consumption, as well as its rate of growth. Concurrently, the cost to own and operate a datacenter can be significantly reduced by improving its power-efficiency. It is therefore in a datacenter operator's own interest to reduce energy consumption, and such action can additionally afford a competitive advantage relative to other datacenters. Thus, we find ourselves in the fortunate position that business and environmental issues align.

Given this general alignment in goals, this chapter focuses on the short-term business motivations for

improving power-efficiency: reducing datacenter "Total Cost of Ownership", and enabling continued growth (i.e. scaling) in the aggregate capability of datacenter facilities.

For most of this dissertation, we pay attention to the problems faced by operators of very large-scale datacenters—so-called "Warehouse-Scale Computers" (WSC) [11]. Unlike server OEMs (like Dell, HP, Supermicro, etc.) and chip vendors (like Intel and AMD), operators of WSCs have a substantial, direct financial interest in improving power-efficiency. For instance, Google reports that they have saved over a billion dollars from their work on energy-efficiency[1]. They have already invested significant resources, research, and effort into improving datacenter power-efficiency, and typically make use of all state-of-the-art best-practices available. As a result, they now confront more fundamental issues in Computer Architecture and Distributed Systems that we attempt to tackle within this dissertation.

This rest of this chapter is organized as follows: Section 1.2 introduces the "Total Cost of Ownership" metric, and shows how power and energy are crucially important to the economics of operating a datacenter. Section 1.3 argues that datacenter capability is intimately tied to trends in server power efficiency, and that server power efficiency has abruptly stopped scaling. Section 1.4 surveys the likely sources of improvements to datacenter power-efficiency in the future. Finally, Section 1.5 introduces the specific contributions of this dissertation towards addressing the new "Datacenter Power Wall".

1.2 Total Cost of Ownership

Contemporary warehouse-scale datacenters are massive affairs, housing tens of thousands of servers and costing several hundreds of millions of dollars to construct [110, 163]. Reducing the cost to own and operate datacenters is desirable both because they are expensive in absolute terms, and also because a cost-efficient infrastructure is a competitive advantage.

One metric commonly used to quantify datacenter-related expenses is Total Cost of Ownership (TCO). TCO is a holistic metric used to describe the overall expenses related to a datacenter. Formally, datacenter TCO is the sum of capital expenses (Capex) and operating expenses (Opex) incurred in purchasing and operating a datacenter facility over its lifetime. Typically, the cost of real estate, building structure, mechanical equipment (such as air handlers and chillers), power delivery equipment (such as power distribution units, generators, and backup batteries), and IT equipment (such as routers, switches, storage appliances, and servers) are included in Capex. The cost of energy consumption (such as by IT equipment, mechanical equipment, and lighting), interest payments on capital expenses, and human resources are included in Opex.

Although there exist several models to estimate TCO for a datacenter [9, 127, 159], the one proposed by James Hamilton is perhaps the most illustrative and simplest to reason about [52]. Given the assumptions originally suggested by Hamilton and listed in Table 1.1, we can easily figure the portion of monthly expenses to purchase and operate a large-scale datacenter, as illustrated in Figure 1.1. One of the key assumptions of Hamilton's model is that servers have a 3-year lifetime.

[1] Google Green: The Big Picture: `http://www.google.com/green/bigpicture/`, December 2013

(a) Basic Datacenter Assumptions

Critical Load of Facility (CL)	8,000,000 W
Facility Cost per Critical Watt	$9/W
Cost of Facility (FC)	$72,000,000
"Other" Portion of Facility Cost (O)	18%
Power Usage Effectiveness (PUE)	1.45
Average Critical Load Usage (U)	80%
Cost of Energy (EC)	$0.07/kWh
Cost per Server	$1,450
Watts per Server (SP)	165 W
Number of Servers	45,978 = (CL - NP)/SP
Cost of Servers (SC)	$66,668,100

(b) Financial Assumptions

Annual Cost of Money (CM)	5%
Facility Amortization (FA)	10 years
Server Amortization (SA)	3 years
Network Amortization (NA)	4 years

(c) Networking Assumptions

Servers per rack	40
Discount off MSRP	60%

(d) Networking Gear

Equipment	Qty.	Cost	Power
Cisco 7609 (Border)	2	$362,000	5,000 W
Cisco 6509E (Core)	2	$500,000	5,000 W
Juniper Ex8216 (Agg.)	22	$750,000	10,000 W
Cisco 3560-48TD (Rack)	1,150	$11,995	151 W
Total		(NC) $12,807,300	(NP) 413,650 W

(e) Monthly Expenses

Servers	$1,998,097	= PMT(CM/12, SA, SC, 0)
Networking	$294,943	= PMT(CM/12, NA, NC, 0)
Power/Cooling	$626,211	= PMT(CM/12, FA, FC, 0) × (1-O)
Energy	$474,208	= $CL \times U \times PUE \times EC/1000 \times 24 \times 365/12$
Other	$137,461	= PMT(CM/12, FA, FC, 0) × O

Table 1.1: James Hamilton's datacenter TCO model [52]. Basic assumptions are provided by Hamilton based on his experience working for Amazon Web Services and Microsoft Foundation Services. Switch count is derived from a linear system of two unknowns taking into account switch count, switch power, server count, server power, and facility critical load. Monthly expenses are simple monthly payments given an amortization period and annual cost of money. The Microsoft Excel formulas used by Hamilton to calculate these are given.

Altogether, Hamilton suggests that costs related to servers themselves are usually the single most expensive component of TCO, often accounting for over 50% of it. Expenses related to power delivery and energy consumption comprise the bulk of the remainder, but neither is overwhelming. No other component (i.e. real estate, labor, etc.) represents a substantial fraction of datacenter TCO, so we do not consider them in this

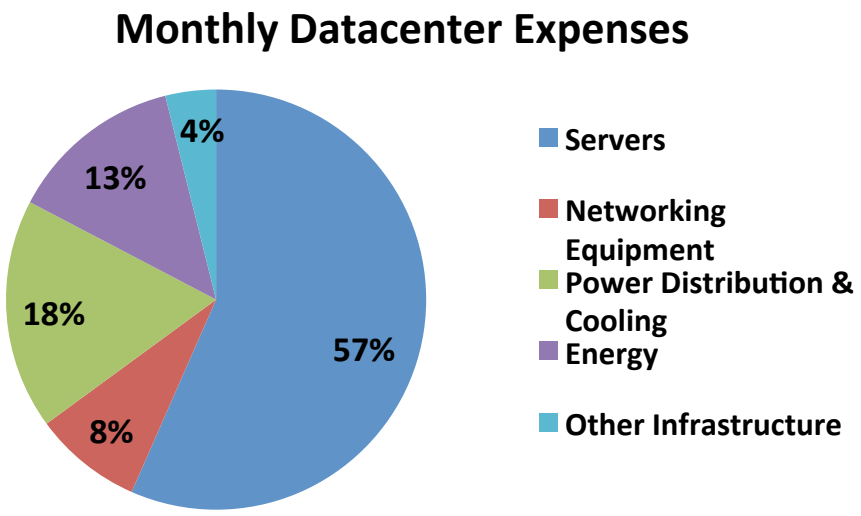


Figure 1.1: Monthly datacenter expenses by category based on the assumptions in Table 1.1.

work. We will revisit the cost of servers when we discuss server utilization.

1.2.1 Energy's Contribution to TCO

Energy is generally the principal component of datacenter Opex. It is perhaps unintuitive that the cost of energy is such a modest fraction of TCO, especially given the attention paid to energy-efficiency, as evidenced by government initiatives like Energy Star [16]. Indeed, outside of the context of a large, warehouse-scale datacenter, it is easy to estimate the cost of energy for a server would be quite high. For example, let us assume we have a $2,000 server which consumes on average 200 Watts over an expected 3-year lifespan. Such a server would itself consume 5,256 kWh during this period. As the server is likely housed in a datacenter facility, we must also consider the overheads involved to deliver power to the server and to provide adequate cooling of it. This overhead is usually accounted for with a metric called Power Usage Effectiveness (PUE), which is the ratio of total power consumed by a facility (including power delivery losses and power spent on cooling) to power consumed by IT equipment alone (referred to as the "Critical Load"). Thus, a PUE of 1 indicates zero overhead, and a PUE of 2 indicates that there is 1 W of overhead for every 1 W of critical load.

Recent surveys have found industry average PUE to be around 1.8-1.9 [11, 151]. Thus, the server in our example is perhaps responsible for $5,256 \times 1.9 = 9,986$ kWh of energy consumption. Given that the average price of electricity in the U.S. in 2012 is $0.12/kWh [162], it could then cost $9,986 \times 0.12 = \$1,198.37$ in energy consumption to operate this server. Assuming the server originally cost $2,000, the energy consumption alone would add 60% to the total cost to own and operate the server!

In light of the preceding example, how is it possible that Hamilton can figure that the cost of power is only 13% of TCO? The answer is that the operators of today's largest datacenters have very carefully optimized the design and placement of their facilities to minimize PUE and the cost of energy, respectively.

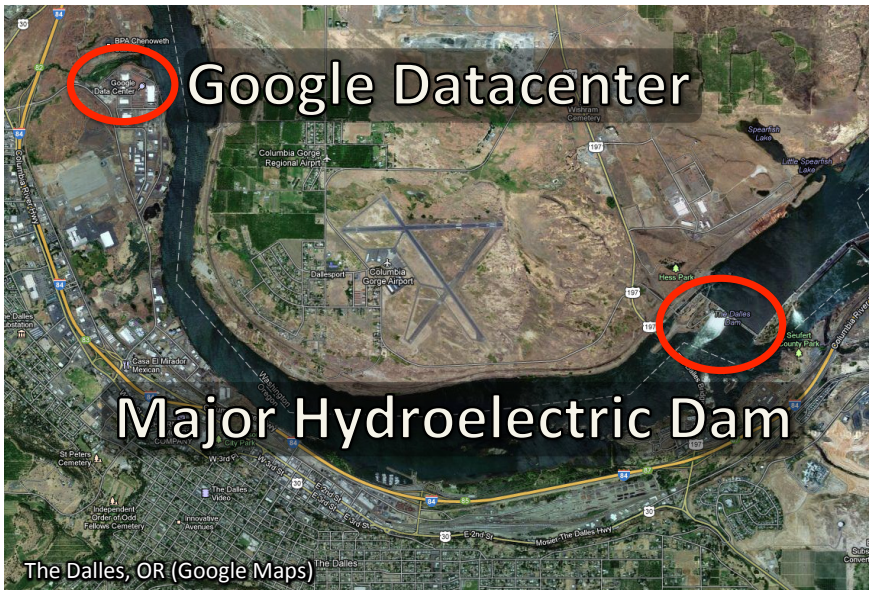


Figure 1.2: Google's datacenter in The Dalles, OR is a short distance from a large hydroelectric dam. Datacenters have sprung up all along the Columbia River gorge as aluminum smelting plants have shuttered, replacing lost jobs and making use of surplus energy production [97].

Although industry average PUE is 1.8-1.9, modern WSC facilities achieve PUE under 1.2 (Hamilton pessimistically assumes 1.45) [11, 45]. Thus, instead of wasting 0.8 W for every 1 W of IT equipment, state-of-the-art facilities waste less than 0.2 W, reducing total energy consumption for our server to less than 6,307 kWh. These newer facilities achieve such low PUE through careful engineering of their mechanical cooling and power delivery systems. For instance, PUE-optimized facilities eschew centralized, in-line UPS (uninterruptible power supply) designs which require an inefficient AC-DC-AC conversion, and instead use distributed UPS solutions which minimize such conversions. They also optimize cooling efficiency by ensuring effective hot-/cold-aisle separation, efficient air handling, and making use of "free" cooling resources like cold outside air and large bodies of water. See [11] for a more complete survey of state-of-the-art techniques to reduce PUE.

Datacenter operators are also very conscientious about the physical placement of facilities for multiple reasons. First, datacenters ultimately need good Internet connectivity. They can save on the cost of laying fiber optic cable by being near major points-of-presence (POPs) or existing "dark fiber" deployments. Second, siting a datacenter in a cold climate has benefits for PUE due to air-side economization, as mentioned previously. Finally, operators can optimize placement in order to negotiate discounted rates from power generating utilities. DCs have a notoriously steady rate of power consumption, which makes them excellent industrial "base load" customers for power plants with low dynamic range (i.e. hydroelectric dams, nuclear power plants, etc.). It is common for datacenters to be sited near these types of generation facilities (as illustrated in Figure 1.2), and they likely are able to pay on the low-end of industrial electricity rates for energy

(i.e. $0.07 per kWh [162]).

Altogether, with a PUE of 1.2 and an electricity rate of $0.07 per kWh, the server from our earlier example would only cost $441 in energy to operate over its lifetime, or 22% of its initial capital cost. While the assumptions to arrive at this lower cost are aggressive and assume a state-of-the-art facility, it elucidates the argument that server cost, and not energy cost, dominate the TCO for large-scale datacenters. This is not to say that energy consumption is not an important matter. Rather, it alone may not be the strongest motivating factor to improve power-efficiency for operators of large datacenters.

Note that this argument relies on the assumption that a server will be replaced or retired after a 3-year period, as is common practice presently [11]. If a server is left in operation for a longer period of time, its energy cost will ultimately dominate.

1.2.2 Power's Contribution to TCO

Power provisioning represents the bulk of datacenter Capex. Even though the cost of energy does not dominate TCO, the cost of power is ironically another story. I say ironically, since any student of rudimentary physics can tell you that power is merely energy over time. So why should they be treated distinctly? The answer is that datacenters are engineered and provisioned with respect to a specific *peak power* budget. The mechanical equipment responsible for cooling and the electrical equipment responsible for power distribution have finite peak capacity, and this capacity is independent of how loaded or unloaded the equipment is at any instant. Thus, peak power provisioning requires up-front capital expenditure, while energy consumption requires ongoing operational expenditure.

Although Hamilton suggests that power distribution and cooling accounts for only 18% of datacenter TCO, this is due to the assumption that the Capex is amortized over a 10-year period. In reality, it costs operators on the order of $9-13 per critical watt (i.e. power drawn by IT equipment) to provision reliable power and cooling in a datacenter [11]. This figure accounts for the cost of transformers, power-distribution units (PDUs), circuit breakers, UPS units, batteries, conduits, wiring, backup generators, fuel storage, chillers, pumps, pipes, cooling towers, plus building space to house all of this equipment. The Uptime Institute finds that the cost per critical watt can even be as high as $25/W, should the datacenter design to their "Tier IV" standard which stipulates redundant power delivery and cooling for each rack in a facility [159, 160]. In practice, the cost to provision all of this equipment scales with peak critical load (the maximum provisioned power draw of the IT equipment), so it is common to relate the cost of power provisioning in terms of dollars per critical watt. Note that modern datacenters are now being built to accommodate a critical load of over 60 MW [163], which, for comparison, dwarves the 45 MW power plant of Ohio-class nuclear submarines [35]. This drives the cost of new datacenters, and even datacenter expansion projects, into the 100s of millions of dollars.

Recall our server from before that cost $2,000 to buy, draws 200 W on average, and cost $441 in energy. To provision power for this server at $10 per watt could cost an additional $2,000! Obviously, this is an expensive and unrealistic cost to incur for each server. In reality, the Capex related to power distribution and

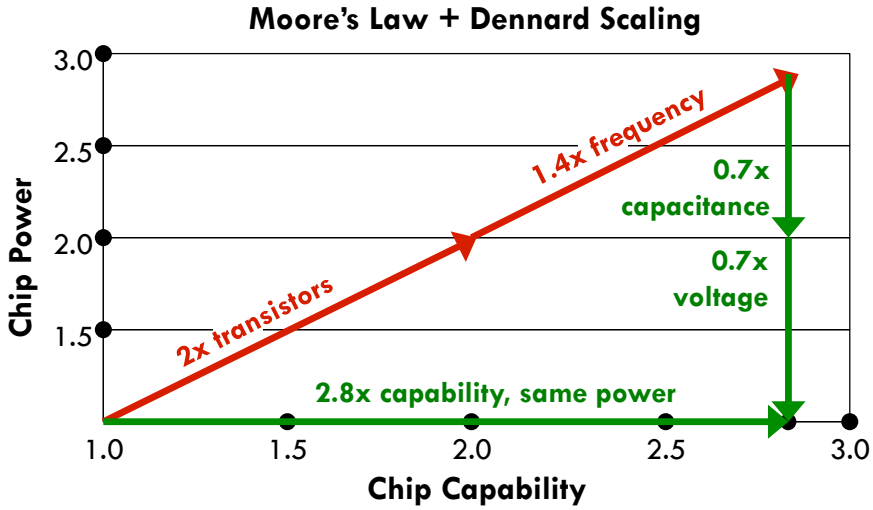


Figure 1.3: Moore's Law and Dennard scaling's combined impact has been to provide a potential 2.8× improvement in capability each CMOS generation with no change in power consumption.

cooling is amortized over a much longer period of time (i.e. 10-15 years) than the servers, which assumes that the same facility will be used by several generations of servers. Under this assumption of a long amortization period, the portion of TCO related to power provisioning is thus closer to 18%.

1.3 Datacenter Capability is Limited by Critical Power

The preceding section has outlined how the costs of energy and power in current datacenters are significant, but not overwhelming. We now consider the interplay between the capability scaling of a datacenter and how these costs might grow over time.

First and foremost, there is a hidden assumption behind the expectation that the facility Capex can be amortized over a longer period of time than the server Capex. This assumption is that each new generation of server is more capable than the last. That is, for a given power budget, a new server has more CPU cores, a faster clock, new architectural features, or whatever it takes in order to support more users, handle more data, or otherwise provide a richer service to end users. Otherwise, there is scant motivation for a datacenter operator to replace their existing server fleet. If operators were to slow the rate of turn-over in their server fleet because new servers offer no material benefit over older servers, the assumption that facility Capex is amortized over a longer period of time than server Capex would no longer hold. Consequently, the cost to provision power and cooling for servers would become a far more substantial fraction of TCO.

So far, this hidden assumption of long relative amortization for facilities has been a safe and efficacious one to make. Moore's Law [112, 143] and the actualization of Dennard scaling [29] have conspired to provide us with new servers each CMOS generation that are materially more capable than the last, while staying within a similar power budget (around 100 W per CPU). The combined effect of Moore's Law and Dennard

scaling is illustrated succinctly in Figure 1.3. Moore's Law observes that subsequent process generations double in transistor count every 2 years. Optimistically, let's assume this doubling of transistor count leads to a doubling of capability through chip multi-processing (i.e. doubling the number of cores). Dennard scaling (the scaling principles regarding gate size, voltage, and circuit speed that have served as a blueprint for actualizing Moore's Law in the sub-micron CMOS era) provides a further increase in capability through a higher clock rate (reduction in gate and interconnect delay). Moreover, the reduction in gate size and operating voltage from scaling leads to reduced switching capacitance and energy consumption, respectively. Thus, ideal application of Dennard scaling leads to a new process generation with $2.8\times$ the capability as the previous generation, all with no change in power consumption.

Altogether, this has afforded a straight-forward strategy for datacenter operators to deploy and grow their infrastructure: build and populate a datacenter facility up-front, and replace servers every few years. Studious realization of Moore's Law and Dennard scaling by CPU vendors has so far guaranteed that these new servers would fit within the critical power budget of the facility and be substantially faster.

1.3.1 Other Strategies to Increase Datacenter Capability

Replacing servers with newer, faster servers is by no means the only strategy available to the datacenter operator to increase overall capability. For instance, to accommodate rapid growth of a service, one could simply deploy new servers within an existing datacenter facility. While straight-forward sounding, this strategy is actually terribly expensive to realize. First, even if the facility has the square footage to put the servers, it will still need spare racks, PDUs, and the mechanical infrastructure in order to accommodate the new servers. Second, the peak critical power of a facility is a fixed budget, and the new servers will necessarily consume additional power. Unless spare capacity was already provisioned for the facility, new power and cooling capacity must be deployed in order to accommodate the new servers (at a cost of $9-13/W). Finally, there may in fact be a practical limit to the critical power that a facility can deploy. Aside from the difficulties in expanding a site's mechanical infrastructure (from both the physical and regulatory points of view), it may be impossible or unaffordable to obtain increased service from electric utilities [24] due to over-subscription of distribution lines, substations or generating plants.

An alternative strategy to accommodate growth is to build new datacenters. Again, while straight-forward sounding, it is at least as difficult and expensive as increasing the number of servers within an existing datacenter. Aside from the regulatory difficulty and financial risk in siting a new datacenter [24], new facilities require access to substantial power and cooling resources (water or cold outside air). While these resources have been readily obtained in locales such as Oregon's Colombia River Gorge (in part due to the retirement of local aluminum smelting plants [97]), they may become more difficult to find in the future. Crucially, future datacenter construction might also require new power generating capacity, whose cost would be incurred by either the public-at-large or as a higher Capex cost per critical watt for datacenter operators.

In any event, while it is presently possible to increase datacenter capability by either expanding existing facilities or building new facilities, these strategies will become more difficult or expensive with time. This

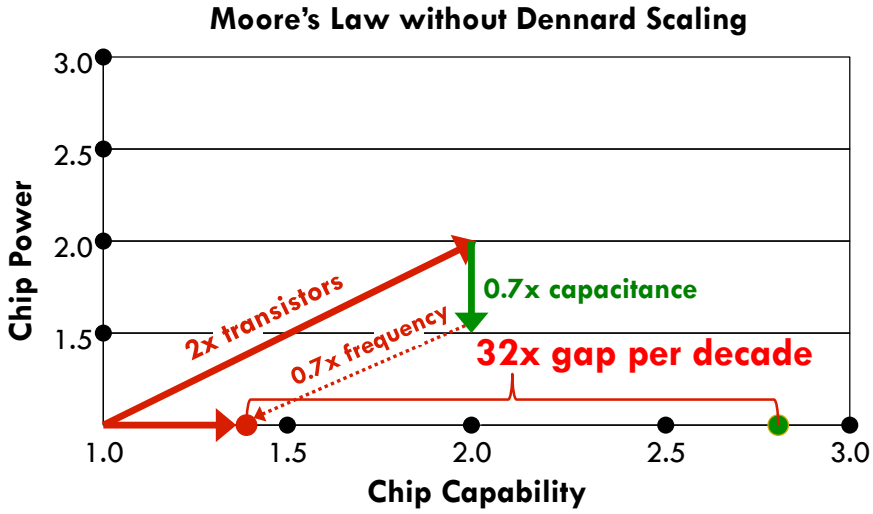


Figure 1.4: Due to the demise of Dennard scaling, Moore's law affords far less improvement in capability per Watt each generation than it did previously.

essentially constitutes a "Datacenter Power Wall": a limit to the amount of power that a datacenter can economically supply to servers. The conundrum is similar to the CPU "Power Wall", the apparent limit to the TDP (Thermal Design Power) of CPU chips: even though we know how to build chips with higher clock rates and power consumption, they require exotic and costly cooling solutions (i.e. water-cooling, oil immersion, etc.) [121]. Most importantly, the affordability of expanding capability in datacenters still depends on amortizing the cost per critical watt of new capacity over several generations of servers. If these strategies are being pursued in lieu of replacing existing servers, both the Capex related to provisioning critical power and the Opex related to energy consumption increase as a fraction of TCO.

1.3.2 The End of Dennard Scaling and Advent of Dark Silicon

It should be clear at this point how important increases to server capability and regular turn-over of server stock are to the economics of datacenters. Without these trends, energy consumption and power provisioning easily dominate datacenter TCO. Unfortunately, the recent failure of Dennard scaling beyond 90nm [28] and the specter of Dark Silicon [33] indicate that these trends are about to come to an abrupt end.

The scaling relationships described by Dennard et al. assumed that device leakage (i.e. transistor "off" current) were insignificant. Unfortunately, this is no longer the case; starting with the 90nm technology node, further decreases to threshold voltage seem unlikely. In nodes smaller than 90nm, scaling down both operating voltage and threshold voltage results in a net increase in power consumption. To maintain constant power density with a decreased operating voltage, threshold voltage must, in fact, be increased. This leads to a reduction in performance relative to ideal scaling (depicted in Figure 1.4). Whereas ideal scaling suggested a $2.8\times$ improvement in capability per generation in a fixed power budget, current scaling trends will achieve

at best $1.4\times$ improvement. As users are unlikely to tolerate a reduction in clock frequency (and consequently, single-thread performance), we are more likely to see the advent of Dark Silicon [33] as the means to maintain constant power-density, whereby growing fractions of a chip must necessarily be under-utilized to limit power consumption. In any event, the break-down of Dennard scaling translates to a $32\times$ opportunity-loss in capability per Watt over the next decade.

The challenge for datacenter operators going forward is thus to find ways to reconcile this $32\times$ loss in capability per Watt while contending with the "Datacenter Power Wall". We must find other ways to improve the *power-efficiency* of datacenters without the aid of Dennard scaling. Without such reconciliation, datacenter operators will either (1) spend considerably more than they have in the past in order to grow their services, or (2) their growth in capability will stagnate, as the return from replacing servers diminishes.

1.4 Future Improvements to Datacenter Power-Efficiency

The preceding sections have established motivation for improving datacenter power-efficiency. In this section, we will enumerate where opportunities to improve efficiency lie, and how the contributions of this dissertation take advantage of these opportunities. In summary, we find that most of the opportunity to improve datacenter power-efficiency rests with improving the power-efficiency of servers themselves. Additionally, we find that there are three main opportunities to improve server power-efficiency in the near future: (1) reduce per-server static power consumption, (2) reduce per-server dynamic power consumption, and (3) raise server utilization by consolidating workloads or growing workloads on existing hardware.

Unlike the period from 2000-2010 during the infancy of commercial warehouse-scale datacenters, most of the present opportunity to improve datacenter power-efficiency rests with improving power-efficiency of the servers themselves. As datacenter PUE has dropped from 1.8 to as low as 1.06 (in the case of Facebook's Prineville Datacenter [106]), there is little inefficiency left to address by developing novel power delivery or cooling architectures. The one exception, however, would be some new design which considerably reduced the Capex cost to provision power. That notwithstanding, we focus our attention on the IT equipment.

There are two categories of IT equipment to consider: networking equipment, and servers. Hamilton's TCO model predicts that networking gear costs around 14% as much as the servers in terms of monthly expenses, mostly from the considerable expense of aggregation switches. However, the same gear only consumes around 6% as much power as the servers. Since there is a large number of servers per top-of-rack switch, and a large number of racks relative to aggregation and core switches, the power consumption of networking gear is well-amortized over the fleet of servers. Thus, there appears to be little opportunity in optimizing the power-efficiency of the networking equipment. Therefore, the remainder of this dissertation focuses only on individual servers.

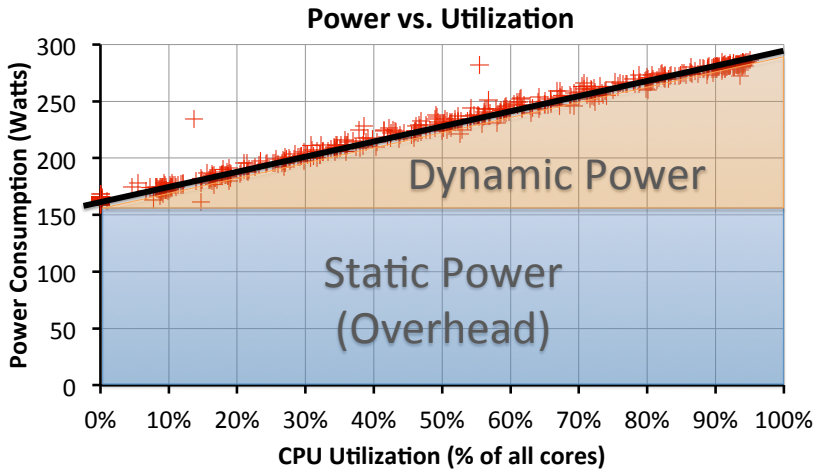


Figure 1.5: Scatter-plot of power relative to CPU utilization on a contemporary mid-range server (2-socket "Westmere" Xeon L5640, each with 6 cores at 2.26 GHz, and 48GB of DDR3 ECC UDRAM). The data were collected during a run of SPEC power_ssj2008 [47]. On this server, total power consumption is essentially a linear function of utilization. Roughly half of the power consumption of this server at peak load is due to static power consumption.

1.4.1 Typical Server Power Consumption

To survey the opportunities available to improve server power-efficiency, we now take a look at power consumption for a typical server. Analogous to CMOS circuits, power consumption for servers can be broadly divided into two categories: static power and dynamic power. Dynamic power is power spent doing useful work, whereas static power is power spent just by having the server powered on. Dynamic power includes power spent by the CPU while executing instructions, by memory while servicing requests, by hard drives while seeking, reading, or writing, by the network interface while handling packets, and so forth. Static power is composed of all of the activities that enable a server to maintain availability, reliability, and respond quickly to new client requests. For instance, it includes power spent by fans to move cold air through the chassis, by disks to keep their platters spinning, by network interfaces to keep links established, and the static power (e.g. leakage, clock distribution, etc.) of CPUs and caches, memory, and auxiliary chips in the server.

Figure 1.5 shows a scatter-plot of server power consumption versus CPU utilization for a mid-range server from 2010 (2-socket Westmere, 12 DIMMs). As a crude first-approximation, we can assume that all of the power consumption at idle (i.e. 0% load) is static power, and any increase in power consumption as load increases is dynamic power consumption. Note that the slope of this plot is almost linear with CPU utilization, and can be well-approximate with a straight line. Prior surveys show that CPU usage is a good predictor of server power consumption [135]. Memory utilization (and by extension, power) largely coincides with CPU utilization, so it doesn't need to be considered separately. Other components, such as disk drives and network interfaces, either consume little power or have very little dynamic range. Also, unlike desktop systems, server systems eschew high-power GPUs or lack video graphics altogether.

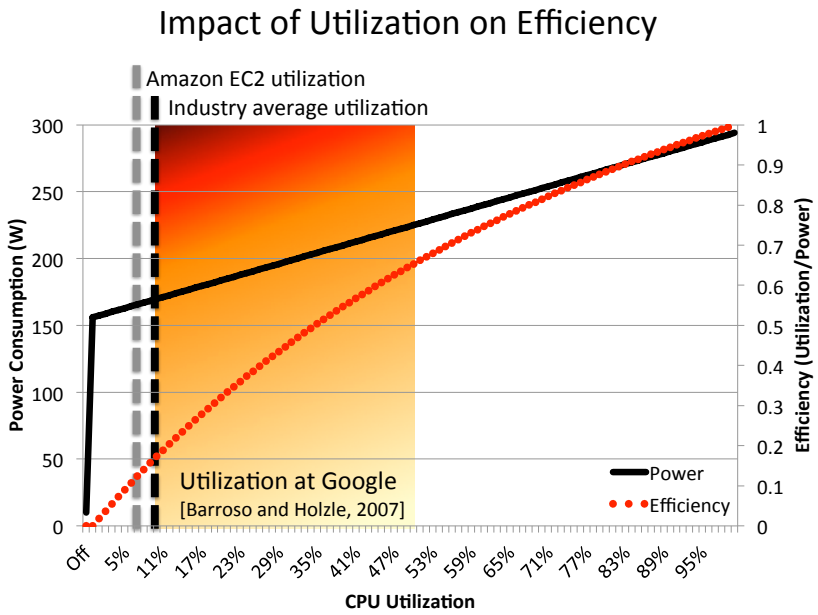


Figure 1.6: The relationship between power consumption, utilization, and efficiency.

It may be striking that static power is slightly more than half of the peak power consumption of this server; this server consumes considerable power, even when it is doing nothing. This is analogous to a automobile driver braking at a stop light while simultaneously holding the gas pedal half-way down. In the context of the home computer user, this waste is not terribly significant; it is perhaps the equivalent of leaving one 45 Watt light bulb on while the computer is idling. The overhead is colossal, however, when it is multiplied over 100,000 servers in a large facility.

Figure 1.6 shows this plot in terms of *efficiency*, normalized to the peak efficiency of this server. A dictionary definition of efficiency is "effective operation as measured by a comparison of production with cost (as in energy, time, and money)"[2]. With respect to power-efficiency, this essentially equates to utilization over power consumption. Note that typical server hardware, with its high static power consumption, tends to be quite inefficient at low utilization. This is especially tragic, as typical server utilization is quite low in commercial datacenters. Barroso reports that utilization is nominally between 10 and 50% in a large Google cluster [10], but Google is far ahead of the industry in this case. Various analyses estimate industry-wide utilization between 6% [74] and 12% [40, 164], and there are even studies that suggest utilization is closer to 7% in Amazon's EC2, the largest public cloud service [93].

Overall, high static power contributes significantly to the overall power-inefficiency of datacenters. Barroso and Hölzle introduced this problem to a wide audience with their influential paper, "The Case for Energy-Proportional Computing" [10]. This led to an industry-wide effort to reduce static power and build "power-proportional" servers: servers which consume power proportional to how much work they are doing.

Assuming servers become more power-proportional over time, dynamic power consumption will grow in

[2] http://www.merriam-webster.com/dictionary/efficiency/

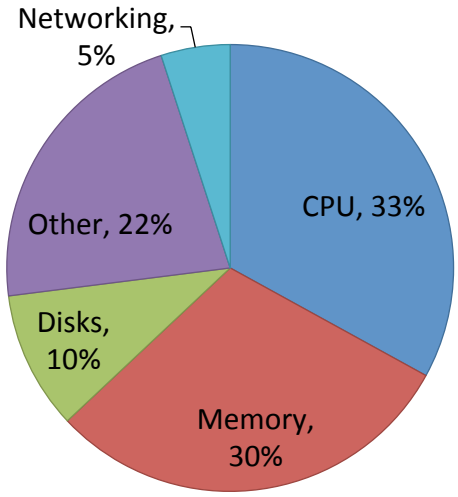


Figure 1.7: Power consumption by subsystem for a representative server at Google in 2009 [11]. "Other" includes fans, power supply losses, southbridge, and other miscellaneous components.

importance. Future reductions to dynamic power consumption will come from microarchitectural improvements and architectural specialization, as we can no longer rely on the steady march of Dennard scaling. The present challenge is that no single hardware component dominates server power consumption. Figure 1.7 depicts power consumption by server subsystem, as reported by Google in 2009 [11]. There is no "silver bullet" component of the server that needs to be fixed, so instead we must undertake the arduous work of improving them all, starting with the CPU and main memory. Nevertheless, reducing server static and dynamic power consumption are the most direct strategies to improve datacenter power-efficiency.

1.4.2 Raising Server Utilization to Improve Power-Efficiency

Hardware power consumption improvements notwithstanding, there is an alternative strategy to improve efficiency: raise server utilization. In Figure 1.6, we showed that power-efficiency is poor at low utilization and that servers tend to be gravely underutilized. This suggests that raising server utilization is a straightforward, hardware agnostic strategy to improve datacenter power-efficiency (Figure 1.8).

In practice, raising server utilization implies many things. First, we don't mean for operators to dispatch work to a server solely to raise its utilization; all other things being equal, increasing utilization will only increase power consumption. Instead, a marginal increase in one server's utilization should represent a commiserate reduction in utilization of other servers. This reduction can either be in space (i.e. reduce the utilization of the server next to this one) or in time (i.e. reduce the utilization of some server in the future, as in absorbing future growth). The shifting of load in this manner leads to an aggregate reduction in datacenter power consumption if, by shifting load away from some servers, we can power them off entirely and prevent them from consuming static power. Second, it implies that load can be redistributed across servers and that

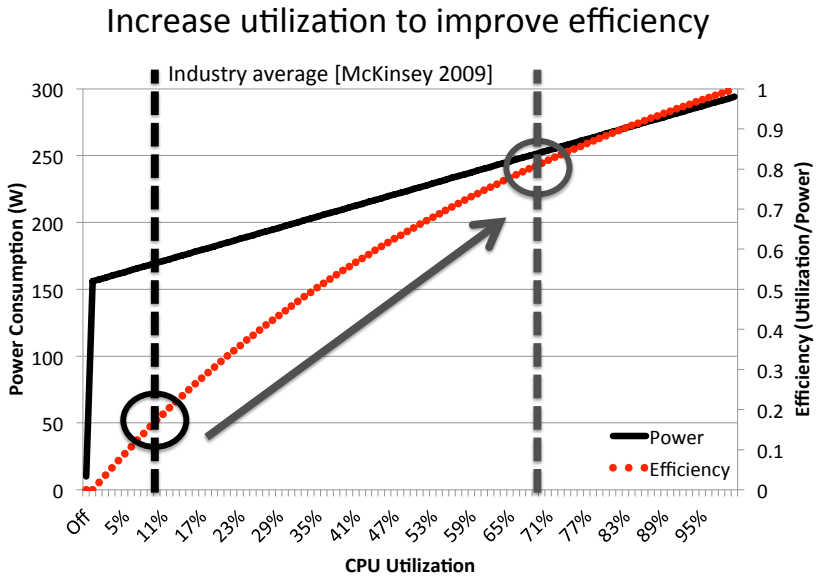


Figure 1.8: An alternative strategy to improve a given server's power-efficiency is to raise its utilization. Modest increases in utilization result in large improvements to efficiency. Moreover, these improvements are orthogonal to other hardware optimizations. Even though an increase in utilization leads to an increase in power consumption of a given server, it would coincide with a reduction in the number of servers needed to deploy a distributed service, leading to cluster-wide benefits.

we can provide acceptable quality of service on servers at high utilization. This requires designing software amenable to load redistribution, and carefully provisioning workloads to not overload any servers. Finally, if by raising server utilization in the present we can avoid having to turn on some server in the future, we may be able to avoid purchasing that server in the first place. Thus, the TCO consequences of raising server utilization contributes both to the Capex to provision power and to the Capex to purchase servers.

1.5 Contributions of this Dissertation

The preceding sections of this chapter have laid out the case that there exists a new "Datacenter Power Wall" and holistically described the form in which new improvements to datacenter power-efficiency will come. As datacenter mechanical and power distribution facilities are now quite efficient, the most viable path toward improving overall datacenter power-efficiency lies in improving the efficiency of the servers themselves. We identified three broad opportunities available in order to achieve this: (1) reduce per-server dynamic power consumption, (2) reduce per-server static power consumption, and (3) raise server utilization by consolidating workloads or growing workloads on existing hardware.

With this framework in mind, the specific contributions of this dissertation are easy to put into context. The subsequent chapters explore novel, promising embodiments of the three opportunities identified above:

- Chapter 2 proposes a novel DRAM module architecture to reduce the dynamic power consumption of main memory.
- Chapter 3 introduces a novel control mechanism for per-core power gating in order to reduce the static power consumption of multi-core CPUs.
- Chapter 4 describes how contemporary distributed storage systems preclude the use of low-power modes and hamper efforts to raise server utilization.
- Chapter 5 addresses head-on the quality-of-service challenges in raising the utilization of servers hosting low-latency applications.

The following sections summarize the salient aspects of these contributions.

1.5.1 Energy-Efficient DRAM module

In Chapter 2, we present Multicore DIMM (MCDIMM), a modification to the architecture of traditional DDRx main memory modules optimized for energy-efficiency. MCDIMM modules divide the wide, 64-bit *rank* interface presented by ordinary DIMMs into smaller *rank subsets*. By accessing rank subsets individually, fewer DRAM chips are activated per column access (i.e. cache-line refill), which greatly reduces dynamic energy consumption. Additionally, we describe an energy-efficient implementation of error-correction codes for MCDIMMs, as well as "chipkill" reliability, which tolerates the failure of entire DRAM devices. For ordinary server configurations and across a wide range of benchmarks, we estimate more than 20% average savings in memory dynamic power consumption, though the impact on total system power consumption is more modest. We also describe additional, unexpected performance and static-power consumption benefits from rank subsetting.

1.5.2 Per-Core Power Gating

In Chapter 3, we propose an architecture for per-core power gating (PCPG) of multicore processors, where the power supply for individual CPU cores can be cut entirely. We propose that servers running at low to moderate utilization, as is common in datacenters, could operate with some of their cores gated off. Gating the power to a core eliminates its static power consumption, but requires flushing its caches and precludes using the core to execute workloads until power is restored. In our proposal, we improve the utility of PCPG by coordinating power gating actions with the operating system, migrating workloads off of gated cores onto active cores. This is in contrast to contemporary industry implementations of PCPG that gate cores *reactively*. We control the gating of cores with a dynamic power manager which continually monitors CPU utilization. Our OS-integrated approach to PCPG maximizes the opportunities available to utilize PCPG relative to OS-agnostic approaches, and protects applications from incurring the latency of waking a sleeping core. We show that PCPG is beneficial for datacenter workloads, and that it can reduce CPU power consumption by up to 40% for underutilized systems with minimal impact on performance.

1.5.3 Power-Proportional Hadoop

Chapters 2 and 3 propose hardware modifications to improve the power-efficiency of individual servers directly. Chapters 4 and 5 take a different tack and holistically assess how utilization across clusters of servers can be manipulated to improve power-efficiency.

In Chapter 4, we describe how contemporary distributed storage systems, such as Hadoop's Distributed File System (HDFS), expect the perpetual availability of the vast majority of servers in a cluster. This artificial expectation prevents the use of low-power modes in servers; we cannot trivially turn servers off or put them into a standby mode without the storage system assuming the server has failed. Consequently, even if such a cluster is grossly underutilized, we cannot disable servers in order to reduce its power consumption. Thus, these clusters tend to be tragically power-inefficient at low utilization. We propose a simple set of modifications to HDFS to rectify this problem, and show that these storage systems can be built to be power-proportional. We find that running Hadoop clusters in fractional configurations can save between 9% and 50% of energy consumption, and that there is a trade-off between performance energy consumption.

1.5.4 High Server Utilization and Sub-millisecond Quality-of-Service

Finally, Chapter 5 sets out to determine why datacenter operators chronically underutilize servers which host *latency-sensitive* workloads. Using memcached as a canonical latency-sensitive workload, we demonstrate that latency-sensitive workloads suffer substantial degradation in quality-of-service (QoS) when co-located with other datacenter workloads. This encourages operators to be cautious when provisioning or co-locating services across large clusters, and this ultimately manifests as the low server utilization we see ubiquitously in datacenters. However, we find that these QoS problems typically manifest in a limited number of ways: as increases in queuing delay, scheduling delay, or load imbalance of the latency-sensitive workload. We evaluate several techniques, including *interference-aware provisioning* and replacing Linux's CPU scheduler with a scheduler previous proposed in the literature, to ameliorate QoS problems when co-locating memcached with other workloads. We ultimately show that good QoS for latency-sensitive applications can indeed be maintained while still running these servers at high utilization. Judicious application of these techniques can greatly improve server power-efficiency, and raise a datacenter's effective throughput per TCO dollar by up to 53%.

1.5.5 Summary

All told, we have found that there exists considerable opportunity to improve the power-efficiency of datacenters despite the failure of Dennard scaling. The techniques presented in this dissertation are largely orthogonal, and may be combined. Through judicious focus on server power-efficiency, we can stave off stagnation in the growth of online services or an explosion of datacenter construction, at least for a time.

Chapter 2

Improving Power-Efficiency with Memory Rank Subsetting

We begin the technical proceedings of this dissertation by describing a technique to reduce per-server dynamic power consumption called *rank subsetting*. Semiconductor process technology scaling has enabled dramatic improvements in the capacity and peak bandwidth of DRAM devices. However, current standard DDRx DIMM memory interfaces are not well tailored to achieve high power-efficiency and performance in modern chip-multiprocessor-based computer systems. As discussed in Section 1.4.1, main memory accounts for a large and growing fraction of system power [91, 111]. There are three drivers of this trend: growing power dissipation in DRAM devices (i.e. DRAM chips), growing number of DRAM devices per system, and growing interconnect power. DRAM devices are not under a tight TDP constraint like CPUs, so there is little industry pressure to reduce the power consumption per cell or prevent the power per device from increasing. Moreover, DRAM processes are not expected to scale as fast as logic processes, so there is upward pressure on the number of DRAM chips per processor. Finally, memory access involves inter-chip communication, which improves more slowly than storage and local transistor operations in terms of speed and energy consumption [61].

The power-efficiency of accessing main-memory data is suboptimal on chip multiprocessors (CMPs) employing contemporary DDRx processor-memory interfaces. Several DRAM chips, which compose a rank in a dual-inline memory module (DIMM), are involved per main-memory access, and the number of bits activated and restored per access could be more than 100 times that of a typical cache line size. As shown in Section 2.1.1, since memory access streams from multicore and manycore processors have lower correlation in access address (in other words, it looks more random) than those from a single core, conventional techniques of utilizing more data from activated bits by reordering memory access requests [136] are less useful. Therefore, the energy to activate data and restore them is largely wasted, causing a problem called memory *overfetch* [2].

There are several recent proposals [2, 166, 182] that share a main goal of saving dynamic main-memory access energy by dividing a memory rank into subsets, and making a subset, rather than a whole rank, serve a memory request. This saves dynamic power by reducing memory overfetch. We refer to this category of proposals as *rank subsetting*. While promising, these studies on rank subsetting are limited. Our early work published in *Computer Architecture Letters*, Multicore DIMM (MCDIMM) [2], did not address memory capacity issues, or evaluate DRAM low-power modes. Module threading [166] focused on micro-architectural details and bus utilization, but did not evaluate system-level impacts. Mini-rank [182] treated memory power and processor power in isolation, which overstates its efficacy. Since a subset of a rank effectively has a narrower datapath than a full rank, data transfers take longer, which can negate the benefit of dynamic power saving. It is therefore hard to judge if rank subsetting provides enough benefits to computer systems when both processor and main memory are considered and other architectural techniques are taken into account. Moreover, none of these previous studies analyzed or devised solutions for high-reliability, which is critical for datacenter computing.

In this chapter, we holistically assess the effectiveness of rank subsetting on the performance, energy-efficiency, and reliability of a whole system, not just of individual components. We first model memory system power analytically including the degree of rank subsetting, memory capacity, system performance, and reliability. We validate these analyses by simulating a chip-multiprocessor system using multithreaded and consolidated workloads. We also develop a novel solution which extends our Multicore DIMM design to support high-reliability systems, and show that compared with conventional chipkill [27] approaches it can lead to much higher system-level energy-efficiency and performance at the expense of additional DRAM capacity. Throughout this evaluation, we consistently use the *system energy-delay product* metric to judge the efficacy of rank subsetting in the context of a whole system.

Our key findings and contributions regarding rank subsetting in modern and future processor-memory interfaces are as follows:

- From model analyses and simulations, we show that 2 to 4 rank subsets is a sweet spot in terms of main-memory power and system energy-delay product. The best configuration depends on application characteristics and memory system capacity.

- Our evaluation shows that subsetting memory ranks and exploiting DRAM low-power modes are largely complementary, since the former technique is applied to saving dynamic energy on memory accesses while the latter one is effective when DRAM chips mostly stay idle. They can be synergistic as well, which is especially apparent on high-reliability systems since both access and static power of main memory take a large portion of total system power.

- Finally, we demonstrate that rank subsetting affords a new category of trade-off in the design of high-reliability memory systems. Traditional chipkill solutions achieve energy-*inefficient* chip-level error recovery at no capacity or component cost relative to conventional ECC, whereas rank subsetting enables energy-*efficient* reliability in exchange for reduced capacity-efficiency.

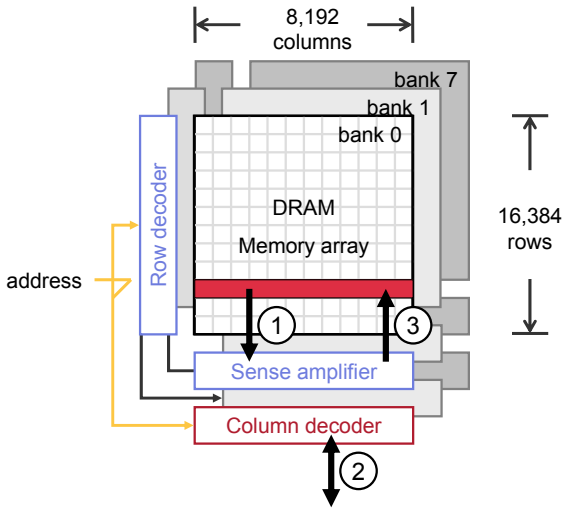


Figure 2.1: A canonical representation of a DRAM chip with 8 banks. The movement of data during a typical memory transaction consists of an ACTIVATE command (1), READ or WRITE commands (2) and PRECHARGE command (3).

In a typical high-throughput server configuration with 4 ranks per memory controller and 4 memory controllers in a system, dividing a memory rank into four subsets and applying DRAM low-power modes provides 22.3% saving in memory dynamic power, 62.0% in memory static power, and 17.8% improvement in system energy-delay product with largely unchanged IPC (instructions per cycle) on tested applications. In high-availability "chipkill" systems, which work correctly even with a failed chip in every memory rank, a Multicore DIMM-based high-reliability memory system uses 42.8% less dynamic memory power, and provides a 12.0% better system energy-delay product compared with a conventional chipkill system of the same data capacity, but needs 22.2% more DRAM devices.

2.1 Energy Efficient and Reliable Memory Modules

In this section we first review modern processor-memory interfaces and the concept of rank subsetting, which has been recently proposed to improve the energy efficiency of main-memory accesses. System-level impacts of rank subsetting are analyzed on performance, energy-efficiency, and reliability, which are all tightly coupled. Then we extend the Multicore DIMM design for high-reliability systems.

2.1.1 Background

Organizations of Modern DRAM Devices and Memory Channels

Main-memory systems are built from DRAM modules, which in turn are comprised of DRAM chips [69]. DRAM is used to store main-memory data since its storage density (smaller than $10F^2$ per cell, where F is

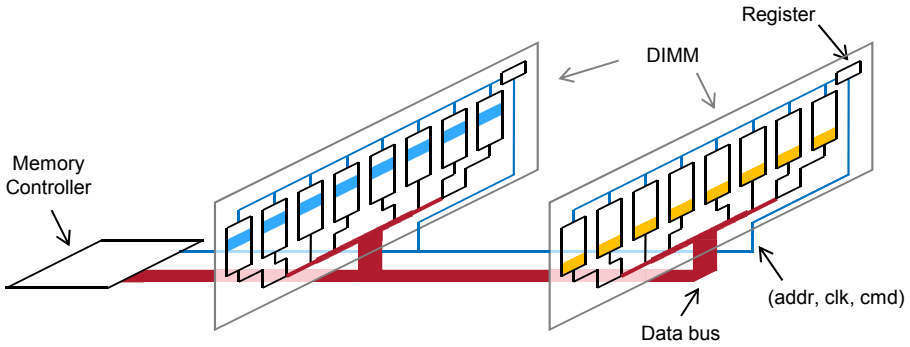


Figure 2.2: A conventional memory channel where a memory controller and two memory modules are connected through a shared bus. Each memory module is composed of one or more ranks, each rank with 8 or 16 DRAM chips.

the minimum feature size of a fabrication process) is much higher than that of SRAM (bigger than $100F^2$ per cell) [155], and its random access time (lower than $100ns$) is much lower than that of NAND flash (in the order of us) and rotating magnetic storage (in the order of ms). Modern DRAM chips contain billions of bit cells, organized as a number of two-dimensional banks. The data output, command, and address signals of a DRAM chip are all shared by its banks. Figure 2.1 shows a canonical representation of a DRAM bank, numbered to indicate the salient phases of a typical interaction with a bank. First, a row-level command called ACTIVATE (1) is issued to the chip, instructing a specified bank to latch all of the bits in a given row into sense amplifiers. Although the readout from a row is destructive, the sense amplifiers will clamp the bitlines of the bank to whatever value they detected, which restores the contents of the original bit cells. Subsequently, one or more column-level commands (READ or WRITE) may follow (2), initiating data transfers. The number of bits transferred per READ or WRITE command is the product of the data path width of the chip (typically 4, 8, or 16 bits per chip, termed $\times 4$, $\times 8$, and $\times 16$) and the burst length (8 in DDR3 [71]). Once a sequence of read or write operations is over, a row-level command called PRECHARGE (3) is sent to the bank in order to precharge the bitlines in preparation for the next ACTIVATE command. When two consecutive memory accesses to the same bank of the same DRAM rank select different rows, a DRAM *row-buffer conflict* [180] occurs requiring PRECHARGE and ACTIVATE commands before the second row can be accessed. Were the accesses to go to the same row (as is common in single-threaded workloads), these row-level commands could be omitted.

The bandwidth and capacity demands from a modern microprocessor keep increasing, since the processor has more cores, the cache size per core does not increase much, and emerging applications such as in-memory databases need even higher bandwidth, more capacity, or both from main memory. A single DRAM chip therefore cannot satisfy the latency, bandwidth, and capacity demands of a microprocessor as a main memory. As a result, several (typically 8 or 16) DRAM chips compose an access unit called a *rank*. All DRAM chips in a rank operate in unison, i.e. receiving the same control (address and command) signals and transferring data in parallel. One or more ranks are packaged on a printed circuit board and called a memory module.

Dual in-line memory modules (DIMMs) are widely used in contemporary computer systems that have 64-bit data buses. Memory modules are connected to a memory controller, which feeds control signals to ranks and exchanges data through a shared bus to form a memory channel.

Figure 2.2 shows a conventional memory channel which contains two DIMMs attached to a memory controller through a bus. In a memory channel, commands generated from the memory controller may be issued to one rank while transferring data to or from another rank in a pipelined fashion. A module with several ranks is logically similar to having several modules, each with only one rank. Command, clock, and address signals from a memory controller are broadcast to all DRAM chips in all ranks on a memory channel. As a result, all of the DRAM chips within a rank act like a single DRAM chip with a wider data path and longer rows. Chip select signals are used to mask unintended recipients from commands. Memory controllers have historically been placed outside of microprocessors (in a chip called the northbridge), but are more recently integrated into the CPU package [14, 77].

Trends in Modern DRAM Devices and the Overfetch Problem

As the throughput of microprocessors has increased, significant effort has been invested in improving the data path bandwidth of DRAM chips. This is primarily achieved by boosting the signaling rate of the data bus and by internally coarsening the granularity of access in a DRAM chip (essentially, enlarging the number of bits that are fetched from rows in parallel). Despite the increase in the capacity and bandwidth of DRAM chips, the latency of a random access (`tRC`) composed of a sequence of PRECHARGE, ACTIVATE, and READ/WRITE in a chip has remained relatively constant in terms of absolute time, and has increased in terms of bus cycles, as seen in Figure 2.3. As data transfer rates improve, the maximum number of ranks attached to a bus decreases and control signals are registered per rank, due in both cases to signal integrity issues. To enhance energy-efficiency, memory controllers may utilize the low-power states of DRAMs when the requests from the processors are infrequent [64].

An error correcting code (ECC) is often employed to cope with hard and soft errors on data storage and communication paths. Single bit error correction and double bit error detection (SECDED) is the most common scheme, but some computer systems need higher levels of error correction. The most well known technique to correct multi-bit errors is chipkill [27], which protects against failure of an entire memory chip.

While the organizational philosophy of DRAM chips has changed little over the past several technology generations, their capacity and communication throughput have followed Moore's law [69]. As the capacity of DRAM chips has grown over successive generations, the number of rows and the number of banks in a chip have each increased to keep pace. The length of a row (termed the *page size*) of modern DRAM chips is between 4K and 16K bits. Assuming that the page size is 8K bits and a rank consists of 8 DRAM chips, 64K bits are activated by an ACTIVATE command to the rank. Considering that the unit of most accesses to DRAM ranks is a cache line and the size of a cache line is typically 1K bits or below, only a few bits are used by column accesses relative to the page size following a row activation. We call this phenomenon *overfetch*.

This memory overfetch can significantly lower the energy efficiency of accessing main-memory data

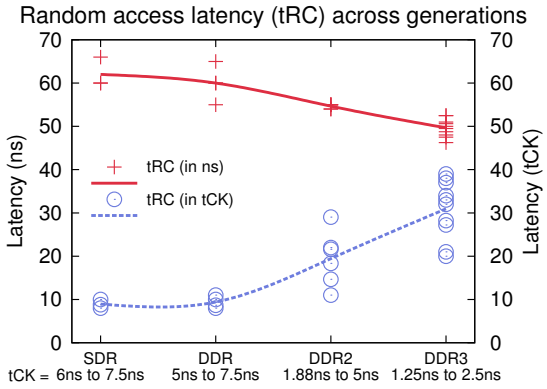


Figure 2.3: Random access latencies of chips from several speed bins of successive DRAM generations are shown. While random access latency has fallen slightly in absolute terms from SDRAM to DDR3, it has increased significantly in terms of clock cycles (`tCK`).

stored in modern DDRx DRAM chips. Figure 2.4 shows a power breakdown of a Micron 2Gb DDR3 DRAM chip [107]. The labels $\times 8$ and $\times 16$ indicate the width of a data path and row/col means the ratio between row-level command pairs (ACTIVATE/PRECHARGE) and column-level commands (READ or WRITE). Row-level commands not only take a long time, but also consume a lot of power. When row/col = 1, ACTIVATE and PRECHARGE take more than half the DRAM power. Thus, it is desirable to decrease the row/col ratio. In modern DRAM chips, refresh power is much smaller than other components; refresh power consumption is mainly noticeable on large memory capacity configurations such as those in Section 2.3.3.

Alleviating Overfetch by Memory Access Scheduling and Rank Subsetting

Out-of-order execution with non-blocking caches, simultaneous multithreading, chip multiprocessing, and direct-memory accesses [55] are now common in microprocessors. Such processors support multiple outstanding memory requests. A naïve memory controller serves these requests first-come-first-serve (FCFS). More advanced memory controllers adopt memory access scheduling schemes to reorder and group these requests to lower the DRAM row-buffer conflicts and to minimize the performance degradation due to various timing constraints on DRAM accesses. Reducing row-buffer conflicts helps with overfetch, too. There have been multiple memory access scheduling proposals [115, 119, 136] to exploit these characteristics to achieve higher performance.

New processor-memory interfaces such as module threading [166], mini-rank [182], and Multicore DIMM [2] address the overfetch issue. These proposals alleviate the overfetch problem by dividing the DRAM chips within a rank into multiple subsets and making a subset (not a whole rank) serve a memory access, as illustrated in Figure 2.5. We call this technique *rank subsetting*. Rank subsetting requires minimal changes (a few additional address lines if existing address lines are too few) to the existing processor-memory interface, since conventional DRAM chips can be used without modification. Memory controllers treat each subset as a separate rank with longer data transfer time, so that the modifications to the memory controllers are minor.

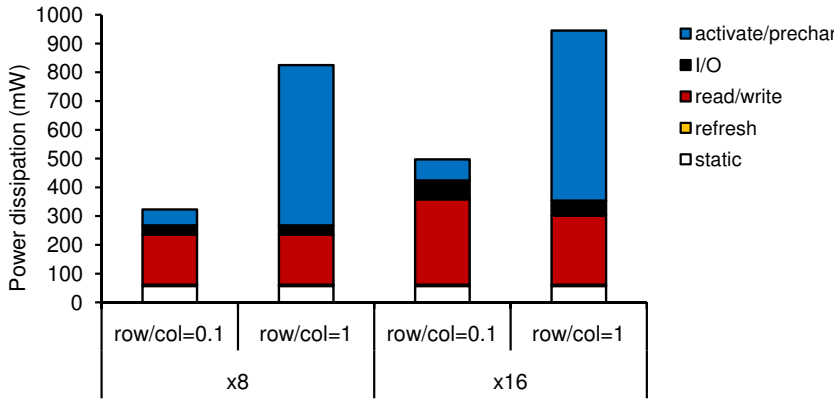


Figure 2.4: DRAM power breakdown of a Micron 2Gb DDR3-1333 SDRAM chip [107]. row/col means a ratio between row-level commands (ACTIVATE/PRECHARGE) and column-level commands (READ/WRITE). The DRAM is assumed to output read data for 50% of clock cycles and input data for 50% of clock cycles except the case of $\times 16$ and row/col = 1, where it inputs and outputs data for 40% of clock cycles each to satisfy the minimum delay between ACTIVATE commands to different banks.

Narrowing each data channel and providing more channels has effects similar to rank subsetting, but requires substantial changes to memory module standards, more memory controllers, and more control signals from microprocessors to provide the same main-memory throughput. These new interface proposals primarily save DRAM access energy, but have additional costs and benefits. Rank subsetting increases DRAM access latency and changes the effective bandwidth of memory channels. Since it changes the number of DRAM chips involved in a memory access, traditional reliability solutions such as chipkill must be revisited as well. So the effectiveness of rank subsetting must be assessed in the context of the performance, energy efficiency, and reliability of a whole system, not just of individual components.

2.1.2 Implications of Rank Subsetting

Rank subsetting alleviates the overfetch problem by dividing each rank into smaller subsets of chips and sending memory commands to only one subset at a time. Figure 2.6 shows an exemplary Multicore DIMM memory channel with two memory ranks, one per DIMM. Each rank is divided into two rank subsets called virtual memory devices (VMDs). Physically the same data bus as in a conventional memory channel is used ($\times 64$ in a DIMM memory channel), but it is divided into two logical data buses, each occupying half of the physical bus ($\times 32$ per logical data bus). A demux register is placed on each rank, which routes (demultiplexes) control signals to the proper rank subset to enable independent operation.

The primary goal of rank subsetting is to improve the energy efficiency of memory systems by saving DRAM access energy, which is important since main-memory DRAM power can reach or surpass the processor power in systems with high memory capacity or high reliability, as shown in Section 2.3. In order to understand how much energy can be saved by rank subsetting, we first identify the sources of DRAM power consumption. DRAM power can be categorized into two parts—static power and dynamic power. Static

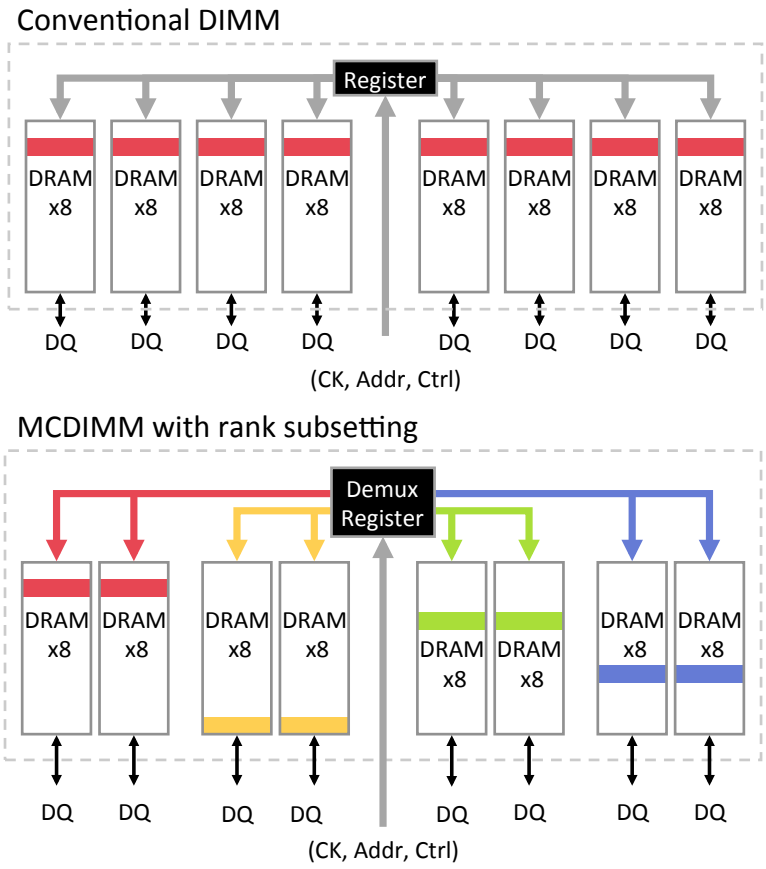


Figure 2.5: Illustrative comparison between conventional DIMMs and DIMMs with rank-subsetting. Conventional DIMMs service requests by accessing all DRAM devices in a rank simultaneously. DIMM architectures with rank-subsetting service requests by routing commands to a subset of devices in a rank. Signaling on the memory channel is otherwise unchanged.

power is independent of activity, and mainly comprised of power consumed from peripheral circuits (like DLL and I/O buffers), leaky transistors, and refresh operations. Dynamic power can further be categorized to two parts since DRAM access is a two step process. First, bitlines in a bank of DRAM chip are precharged, and data in a row of the bank is delivered to the bitlines and latched (activated) to sense amplifiers by row-level commands. This consumes activate-precharge power. Then, a part of the row is read or updated by column-level commands. This consumes read-write power. Dynamic power consumption is proportional to the rate of each operation. However since a row can be read or written multiple times once it is activated, the rates of activate-precharge and read-write operations can be different.

We can model the total power consumed in a memory channel as follows. When D is the number of

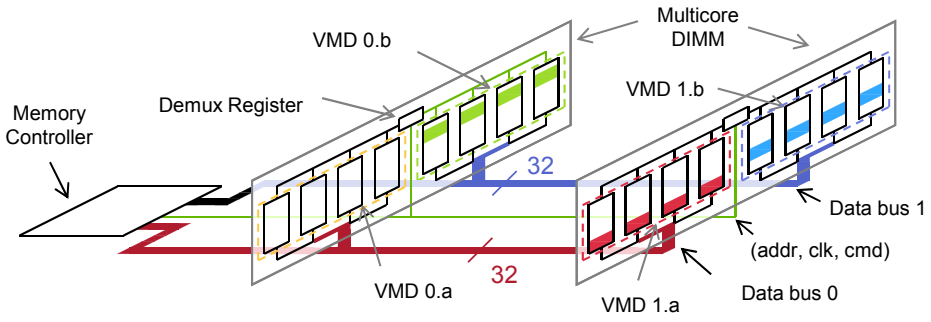


Figure 2.6: A memory channel with two Multicore DIMMs (MCDIMMs) with each divided into two subsets called virtual memory devices (VMDs). Each MCDIMM has a demux register instead of a normal register routing control signals to each VMD.

DRAM chips per subset, S is the number of subsets per rank, and R is the number of ranks per channel,

$$\text{Total main} - \text{memory power} = D \cdot S \cdot R \cdot SP + E_{RW} \cdot BW_{RW} + D \cdot E_{AP} \cdot f_{AP}, \tag{2.1}$$

where SP is the static power of a DRAM chip, E_{RW} is the energy needed to read or write a bit [1], BW_{RW} is the read-write bandwidth per memory channel (measured, not peak), E_{AP} is the energy to activate and precharge a row in a DRAM chip, and f_{AP} is the frequency of the activate-precharge operation pairs in the memory channel. The first term of Equation (2.1) is the static power portion of the total memory power, the second is the read-write power, and the third is the activate-precharge power which, due to overfetch, grows linearly with D. Assuming that misses from last-level caches are the dominant portion of memory access requests, $BW_{RW} = f_{CM} \cdot CL$ where f_{CM} is the frequency of cache misses and CL is the line size of the last-level caches. If we analyze f_{AP} further,

$$f_{AP} = \frac{f_{AP}}{f_{CM}} \cdot f_{CM} = \frac{f_{AP}}{f_{CM}} \cdot \frac{BW_{RW}}{CL} = \beta \cdot \frac{BW_{RW}}{CL}, \tag{2.2}$$

showing that the dynamic power of main memory is proportional to the read-write bandwidth per memory channel and β, the ratio of the number of rows being activated to the number of memory requests to the memory channel. β indicates the frequency of row-buffer conflicts.

Rank subsetting lowers D while keeping $D \cdot S$ constant. Equation (2.1) and (2.2) indicate that this mainly decreases activate-precharge power, which can be more than half of DRAM power [108] as shown in Figure 2.4. (We show below that SP and BW_{RW} are affected by rank subsetting as well.) Saving activate-precharge power is more significant if β is higher. The frequency of row-buffer conflicts depends on various factors including memory access patterns of applications and the number of applications running on computation cores, memory address interleaving across memory controllers, memory access scheduling policies in

[1] The major portion of read or write power consumption is from wires transferring control and data signals through chip-to-chip I/O and DRAM chip global interconnects, which is similar for both read and write operations. We therefore assume that both operations consume the same power as a first order approximation.

memory controllers, and the number of DRAM banks in memory systems. Section 2.3.1 further evaluates the dependency of row-buffer conflicts on the access patterns of running applications and the number of DRAM banks in an example chip-multithreaded (CMT) processor. Activate-precharge power can also be lowered by increasing the cache line size (CL). However CL is around 64bytes while the size of a row in a rank is typically 8 or 16Kbytes, which is orders of magnitude smaller. Also larger cache line sizes can increase miss rate and harm application performance [41, 169].

Both dynamic power terms in (2.1) are proportional to the bandwidth of memory channels BW_{RW}, which is in turn proportional to instructions per cycle (IPC). Rank subsetting increases the access latency of memory requests. However, modern throughput-oriented CMPs such as the Sun UltraSparc T2 [73] can amortize or tolerate this latency with features like simultaneous multithreading or speculative out-of-order execution. They allow memory controllers to reorder and pipeline memory requests, to lower the correlation between memory latency and bandwidth.

Effective bandwidth depends on multiple factors such as load-balancing across memory channels and DRAM banks, memory access patterns, and inherent timing constraints of DRAM chips. In DDR3 [107], for example, 7.5ns is the minimum time from the end of a previous write transaction to the issuance of a following read command (`tWTR`). Recent DRAM chips are limited in the rate at which they can activate different banks in a device (`tRR` and `tFAW`). This limitation can lower the effective throughput since the row-buffer conflict ratio is high on modern CMPs. This throughput degradation can be alleviated when R or S increases. As R increases, there are more banks a memory controller can utilize and there is no timing constraint on activating rows in different ranks. Rank subsetting also effectively increases the number of independent DRAM banks on a module, since each subset can be accessing different banks. As S increases, cache line transfer time per rank subset increases reaching or surpassing the minimal inter-activate time, which effectively renders it irrelevant. Increasing S reduces the effect of other timing constraints as well, such as switches between read and write operations and bus ownership changes. So the impact of rank subsetting (varying S) on system performance is determined by the interplay of these different factors, and also depends on the characteristics of the applications running on the system.

Furthermore, the number of ranks in a memory channel can significantly affect the memory system power. When R is large or the memory channel is not utilized frequently (BW_{RW} is small), static power dominates. The memory controller can then utilize low-power DRAM modes to decrease the static power (SP). This typically lowers BW_{RW} as well since it takes time for DRAM chips to enter or exit low-power modes, and commands to utilize low-power modes compete with other commands to DRAM.

It can be seen that rank subsetting changes both the energy-efficiency and performance of computer systems. So we need a metric to measure the effectiveness of rank subsetting at the system level, combining both performance and energy-efficiency instead of presenting memory system power and processor performance separately. We pick system energy-delay product (EDP), which is a product of the execution time of a workload running on the system and the energy consumed by computation cores and memory systems during execution [44].

2.1.3 Adding Reliability to Multicore DIMMs

Rank Subsets with Single-Bit Error Correction and Double-Bit Error Detection

Soft and hard errors occur frequently on modern DRAM devices [144]. Memory systems adopt error correcting codes (ECCs [129]) that add redundant or parity symbols (typically bits) to data symbols in order to recover from these errors. Since data stored in DRAM devices are accessed at a block granularity such as an OS page or a cache line, linear block codes that are processed on a block-by-block basis are popular. An n-symbol codeword of a linear block code is composed of k data symbols and $(n - k)$ parity symbols that are encoded and decoded together. The redundancy of a code is described in terms of its code rate, k/n, which is a ratio of data symbols to overall symbols. The minimal Hamming distance of two codewords is the number of locations where the corresponding symbols are different and determines the strength (the number of detectable or correctable errors) of the code. Codes with lower code rates or higher redundancy have higher Hamming distance, so they can correct or detect more errors. The most popular codes used in memory systems enable single-bit error correction and double-bit error detection (SECDED), in which 8 parity and 64 data bits occur in a codeword of 72 bits, so the code rate is $8/9$. It is a Hamming code [129] with the minimal Hamming distance of 4, which is needed to correct one symbol error and detect two symbol errors. Since it has 64 data bits, it is used on a rank and one $\times 8$ DRAM chip or two $\times 4$ chips are added to the rank to provide the parity bits.

There are two ways to support SECDED-level reliability on Multicore DIMMs. One way is to add a DRAM chip per subset to supply parity bits. When a rank is divided into S rank subsets, each subset has a data path width of $64/S$. 64 data bits of the 72-bit SECDED codeword are hence from S bursts of the rank subset, so 8 parity bits can be provided by a DRAM chip with a data path width of $8/S$. When a rank has more than 2 subsets, the data path width needed for parity is lower than 4. Since DRAM chips with this low data path with are not popular, the code rate (a.k.a. coding efficiency) of supporting SECDED-level reliability is lower on Multicore DIMMs compared to conventional DIMMs. One exception is the case of 2 subsets having 9 $\times 4$ DRAM chips each. The other way is to use a DRAM device with a wider data path, such as $\times 9$, instead of $\times 8$. Though not popular, some DRAM vendors provide such configurations (RLDRAM [109] and RDRAM).

Rank Subsets with Single-Chip Error Correction and Double-Chip Error Detection

SECDED schemes protect against single-bit errors, such as DRAM bit cell faults and wire stuck-at faults in DRAM ranks or rank subsets. High-availability systems often demand the stronger reliability of single-chip error correct, double-chip error detect (SCCDCD) schemes, sometimes called *chipkill*. These schemes can correct the failure of an entire DRAM chip, and detect the failure of two DRAM chips. There are two common practices for implementing SCCDCD reliability in memory systems: interleaving SECDED-quality codewords across multiple ranks [27], as implemented by IBM's xSeries, or employing stronger error correcting codes [3], as found in the AMD Opteron.

The first scheme observes that codewords can be interleaved across ranks such that no more than 1 bit of a codeword comes from a single DRAM chip. For example, if a rank supporting SECDED-level reliability consists of 18 $\times 4$ DRAM chips, we combine 4 of these into a single conceptual rank consisting of 72 chips. In this case, four SECDED codewords of 72-bits interleaved across 288-bits are sufficient to recover from whole chip failures, since each bit from a DRAM chip belongs to a different codeword. While conceptually simple, this solution suffers from the fact that all DRAM chips in the conceptual rank (72 chips in the example above) must be activated on each access, exacerbating the memory overfetch problem and leading to poor dynamic energy efficiency. This type of SCCDCD solution is also impractical with $\times 8$ and $\times 16$ chips, since they would require 8 and 16 ranks per transaction, respectively. It should be noted that DRAMs with long minimum burst lengths (like DDR3, which has a minimum burst length of 8) render SCCDCD schemes that interleave transactions across several ranks unreasonable, since they grow the memory transaction length beyond that of a typical cache line. For example, were the IBM chipkill solution to be implemented with $\times 4$ DRAM chips with burst-length 8, each memory transaction would contain four 64-byte cache lines.

The other approach observes that DRAM chip failures manifest as a burst of bit-errors in codewords with a burst-length the same as or multiples of the width of the chip's data path (e.g. failure of a $\times 4$ chip is a burst-error of length 4). In practice, non-binary cyclic error-correcting codes like Reed-Solomon (RS) codes can be used to support the SCCDCD feature [172]. While a bit is a symbol in the SECDED Hamming code, multiple bits constitute a symbol in the RS codes, and a natural symbol size for the chipkill protection is the data path width of a DRAM chip so that errors from a chip can be regarded as a single symbol error, which is correctable. Using m-bit symbols, the maximum codeword size is $2^m - 1$ symbols. An RS code is a maximum distance separable code that has the minimum Hamming distance $k + 1$ with k parity symbols, so that at least 3 parity symbols (Hamming distance 4) in addition to the data symbols are needed to detect two symbol errors and correct one symbol error. An RS code with 4-bit symbols can't be used for standard DIMMs since its maximum code size is $((2^4 - 1) \times 4 =)60$ bits, which is lower than the 72-bit data path of a standard DIMM. Hence a RS code with 8-bit symbols should be used instead, with all the bits from a given DRAM chip's output contributing to the same symbol. When $\times 4$ chips are used, data from 2 bursts of each chip compose an 8-bit symbol. Note that the AMD Opteron uses 2 ranks of $\times 4$ chips to construct a 144-bit cyclic code [3]. The Opteron solution, with 4 parity chips, comes close to this theoretic lower bound. However, this result also means that the Opteron solution is only practical with $\times 4$ chips, since 2 ranks of $\times 8$ chips would only provide 2 parity chips. Both the IBM and Opteron solutions achieve a high code rate of $8/9$. Compared to the IBM chipkill solution that needs 72 DRAM chips per transactions, the Opteron solution needs only half. However, it still activates 36 DRAM chips per transaction, leading to poor energy efficiency relative to SECDED schemes.

The approach of using multi-bit symbols can be applied to Multicore DIMMs to support SCCDCD-level reliability by adding 3 chips per rank subset to achieve the minimum Hamming distance of 4. In the case of 2 subsets ($S = 2$) with $\times 4$ chips, this equates to 11 chips per subset (8 for data and 3 for parity). Compared to the conventional chipkill solutions described earlier, this worsens the code rate from $8/9$ to $8/11$. But only 11

	No parity	IBM	Opteron	MCDIMM $S=2$			MCDIMM $S=4$			MCDIMM $S=8$	
Chip width	$\times4$	$\times4$	$\times4$	$\times4$	$\times8$	$\times16$	$\times4$	$\times8$	$\times16$	$\times4$	$\times8$
Min TX (bytes)	64	256	128	32			16			8	
Chips/TX	16	72	36	11	7	5	7	5	4	5	4
Code Rate	1.0	0.89	0.89	0.73	0.58	0.4	0.58	0.4	0.25	0.4	0.25

Table 2.1: This table compares a baseline system without parity, conventional chipkill solutions [3, 27], and several configurations of SCCDCD MCDIMMs (reliability-enhanced MCDIMM). Min TX is the minimum memory transaction size assuming DDR3's burst length of 8. Chips/TX is the number of DRAM chips activated per memory transaction. The IBM chipkill solution can correct certain types of multi-chip errors while the Opteron and MCDIMM solutions provide protection against equivalent categories of single-chip errors.

DRAM chips are activated per transaction, instead of 36 (Opteron) or 72 (IBM), leading to substantial access energy savings. The number of chips per transaction for several configurations of MCDIMMs is compared with that of conventional chipkill solutions in Table 2.1. While the Opteron and IBM chipkill solutions are only practical for $\times4$ chips, it is feasible to implement SCCDCD MCDIMMs across several potential configurations ranging from 2 to 8 rank subsets and using either $\times4$, $\times8$, or $\times16$ chips. Each configuration represents a compromise between code rate and energy efficiency. For example, when comparing SCCDCD with 2 subsets of $\times4$ and $\times8$ chips, the $\times4$ configuration activates 11 chips per transaction while the $\times8$ configuration only activates 7, leading to improved energy efficiency. On the other hand, the $\times4$ configuration has the code rate of 73%, while the $\times8$ configuration has the code rate of 58%. Another advantage of using $\times4$ DRAM chips for Multicore DIMMs is that a RS code with 4-bit symbols can be used, lowering the complexity of encoding and decoding parts of the codewords [142].

Implications and Limitations of Adding Reliability to Multicore DIMMs

We can apply Equation (2.1) and (2.2) to model the power consumption of a Multicore DIMM memory channel augmented with SCCDCD reliability. Here D is the number of DRAM chips activated to provide both data and parity bits, and R is the number of ranks that operate independently within a channel. For example, $R = 1$ on a memory channel with 2 ranks employing the chipkill protection scheme in the AMD Opteron since all 36 chips in both ranks operate in unison. When the total memory capacity excluding parity bits of a system stays constant, implementing SCCDCD-level reliability increases D, decreases R, and increases E_{RW} due to parity overhead. Dynamic power plays a bigger role in the memory system power due to these changes.

Practical implementations of the high-reliability techniques discussed in this section need to take into account issues of form-factor, module compatibility, and memory channel wire-count into consideration. For instance, while it is easy to imagine a single physical module specification that could be shared between modules with 2, 4, or 8 rank subsets, SCCDCD protection presumes a distinct datapath width and part count between different numbers of rank subsets. First, we observe that module slots already have pins dedicated to

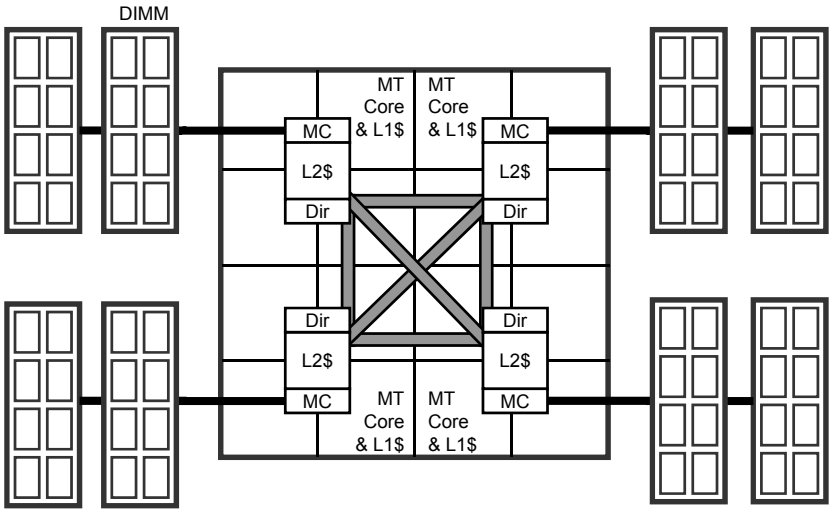


Figure 2.7: System architecture assumed for evaluation of rank-subsetting. A processor consists of 16 in-order cores, 4 threads per core, 16 L1I and L1D caches, 4 L2 caches, 4 memory channels (MCs), and 4 directories (Dirs). From one rank to four ranks (i.e., two dual-rank DIMMs) are connected per memory channel.

ECC even though many systems use DIMMs that do not have ECC chips. Similarly, we expect that module standards for DIMMs which incorporate rank-subsetting would include some number of pins for ECC chips. Second, like the Opteron SCCDCD solution, the implementation of SCCDCD MCDIMMs need not match the theoretic upper bound for code rate. For example, on an SCCDCD MCDIMM with 2 rank subsets and using $\times 4$ or $\times 8$ chips, SCCDCD reliability could be achieved by spreading codewords across both rank subsets. While counterproductive towards the goal of minimizing overfetch, this solution is still far more energy efficient than the conventional SCCDCD alternatives, and could reduce transfer times.

2.2 Experimental Setup

To evaluate the impact of rank subsetting in the context of performance, energy-efficiency, and reliability, we model a Niagara-like system [81] (Figure 2.7) with multiple cores and memory channels. A processor has 16 in-order cores and 4 threads per core, for a total of 64 concurrent threads. The cores run at 2GHz and process up to 1 instruction and 1 memory access per cycle. Each core has its own separate L1 instruction and data cache, while there is an L2 cache shared by each cluster of 4 cores. L2 caches are not shared between clusters. All caches have 64B cache lines. There is a crossbar between the 4 L2 caches and 4 memory controllers. A hierarchical MESI protocol is used for cache coherency and a reverse directory, similar to what was implemented in the Niagara processor, is associated with each memory controller. A memory controller is connected to from 1 to 4 ranks and there are either 1, 2, 4, or 8 rank subsets per memory rank (S). When $S = 1$, cache line transfer time in a data bus is 4ns or 8 cycles. As S doubles, cache line transfer time

	Cache			Directory	DRAM chip	
	L1 I	L1 D	L2		×4	×8
Capacity	16KB×16	32KB×16	1MB×4	×4	4Gb	8Gb
Associativity	4	4	8	32	N/A	N/A
Access time	1 cycle	2 cycles	4 cycles	4 cycles	93 cycles	95 cycles
Cycle time	1 cycle	1 cycle	2 cycles	2 cycles	55 cycles	59 cycles
Dynamic read energy	0091nJ	0.095nJ	0.183nJ	0.021nJ	1.32nJ	1.52nJ
Static power	4.8mW	8.9mW	185mW	86.5mW	75.6mW	104.8mW

Table 2.2: Power and performance parameters of the memory hierarchy used in this chapter. On DRAM chips, dynamic read energy includes precharge and activation energy assuming random access sequences. All caches use 64-byte blocks.

doubles as well. The controller has 32-entry scheduling buffers (or windows) and employs the Parallelism-Aware Batch Scheduling algorithm [115], where the requests in the current batch or group have the highest priority, requests that can be served by single column-level commands are prioritized within a batch, and an older request is served earlier than a younger one when both have the same priority. An XOR-based [39] interleaving scheme is used to map memory addresses across memory controllers, ranks, and rank subsets pseudo-randomly in rank-subset page granularity. (A rank-subset page is the product of the number of DRAM devices per subset and the number of columns in a DRAM bank.)

A 32nm process technology based on ITRS projections is assumed for both processor and memory chips. We use CACTI 5.3 [154] to compute access time, cycle time, dynamic energy, and static power of uncore parts such as caches, directories, and DRAM chips, as summarized in Table 2.2. To compute the power of processor cores, we use McPAT [89], whose technology modeling is built on CACTI 5.3's technology model. DDR3 DRAM chips are assumed for the main memory, with a prefetch size of 8, a row size of 16,384 bits, 8 banks per chip, and 2Gbps data pin bandwidth. The product of prefetch size and data-path size is the smallest possible number of bits accessed when a row is read or written. The energy dissipated by a data bus of a memory channel is calculated as the DC power of the output driver and termination resistance specified in the DDR3 standard [108] after scaling the operating voltage from 1.5V to 1.0V in order to account for a 32nm process. As more ranks are attached per memory channel, more energy is dissipated per data transaction. The energy consumption of the address and command bus is calculated by computing the capacitance of driver, wire, and load [43]. A memory controller puts a DRAM in a low-power state (similar to the precharge power-down mode in DDR3) when all banks in it are at the precharge state. We assume that a DRAM chip in a low-power state consumes 20% of normal static power and needs two cycles to enter and exit the state [107].

We developed a multi-core simulation infrastructure in which a timing simulator and a functional simulator are decoupled in a way similar to GEMS [100, 147]. A user-level thread library [126], which was developed as a Pin [95] (version 2.4) binary instrumentation tool, is augmented to support additional pthread library APIs, such as `pthread_barriers`, and used as a functional simulator to run multi-threaded applications. An event-driven timing simulator, which models in-order cores, caches, directories, and memory

SPLASH-2

Application	Dataset	L2 MPKI
Barnes	16K particles	0.3
Cholesky	tk17.O	3.0
FFT	1024K points	5.1
FMM	16K particles	0.6
LU	512×512 matrix	0.4
Ocean	258×258 grids	7.3
Radiosity	room	0.3
Radix	8M integers	18.6
Raytrace	car	1.7
Volrend	head	0.7
Water-Sp	512 molecules	0.1

SPEC CPU2006

Set	Applications	L2 MPKI
CFP		
high	433.milc, 450.soplex, 459.GemsFDTD, 470.lbm	21.9
med	410.bwaves, 434.zeusmp, 437.leslie3d, 481.wrf	9.9
low	436.cactusADM, 447.dealII, 454.calculix, 482.sphinx3	7.3
CINT		
high	429.mcf, 462.libquantum, 471.omnetpp, 473.astar	18.9
med	403.gcc, 445.gobmk, 464.h264ref, 483.xalancbmk	4.6
low	400.perlbench, 401.bzip2, 456.hmmer, 458.sjeng	3.7

Table 2.3: SPLASH-2 datasets and SPEC 2006 application mixes are listed with misses per kilo-instruction (MPKI) in the L2 cache.

channels, controls the flow of program execution in the functional simulator and effectively operates as a thread scheduler.

We perform experiments with the SPLASH-2 [169], PARSEC [12], and SPEC CPU2006 [102] benchmark suites. For multi-threaded workloads, 64 threads are spawned per workload and each thread is mapped to a hardware thread statically. All 11 SPLASH-2 applications are used while only 6 PARSEC applications (canneal, streamcluster, blackscholes, facesim, fluidanimate, and swaptions) are used due to a Pin limitation. Currently, the functional simulator (Pin) doesn't run the other PARSEC applications due to its limited support for vector operations and library-embedded pthread API calls. PARSEC applications are executed with the simlarge dataset while our SPLASH-2 inputs are summarized in Table 2.3. To model multiprogrammed workloads, we consolidate applications from SPEC CPU2006. The SPEC CPU2006 benchmark suite has single threaded applications consisting of integer (CINT) and floating-point (CFP) benchmarks. We made 3 groups each of integer and floating-point benchmarks, 4 applications per group, based on their L2 cache miss ratio [56], which are listed in Table 2.3. It also shows the number of L2 misses per instruction, which is measured by using the baseline configuration in Section 2.3.2. Simpoint 3.0 [148] is used to find several

simulation phases (100 million instructions per phase) and weights. For each CINT and CFP set, 16 simulation phases per application are consolidated so that each hardware thread executes a simulation phase. The number of instances per phase of each SPEC 2006 benchmark is proportional to its weight. We simulate each workload until the end or up to 2 billion instructions. We skip initialization phases of the SPLASH-2 and PARSEC benchmark suites. gcc 4.2 is used to compile all the benchmark programs.

2.3 Results

We evaluate the impact of rank subsetting on memory power, processor performance, and system energy-delay product using the configurations described in the previous section. In particular, we focus on understanding the interplay between subsetting DRAM ranks and utilizing DRAM power-down modes as the capacity and the reliability level of the memory systems are varied.

2.3.1 Impact of the Number of Ranks on the Frequency of Row-Buffer Conflicts

The parallelism-aware batching scheduling algorithm [115] we chose for memory access scheduling does not specify when to precharge a DRAM bank. Rixner et al. [136] evaluated two policies called *closed* and *open* with regard to the precharge timing that showed noticeable differences on media applications. A closed-policy controller closes a row of a DRAM bank that has no pending requests. In the open-policy, an open row remains open until a request arrives for a different row of the same rank. There won't be much of performance difference between the two policies when memory controllers have enough pending requests so that there exist multiple requests to most of DRAM banks and it is not needed to speculatively open or close rows, which is the case for [1]. However, the number of pending requests from computation cores is limited due to various factors such as the lack of memory-level parallelism of applications, limited buffer sizes, or core microarchitecture. For example, the microprocessor we evaluate can have up to 64 pending memory requests since it has in-order cores with 64 concurrent threads. So unless the number of pending requests is much more than the number of DRAM banks in the system, speculative decisions are needed on deciding the bank precharging time and could have a noticeable influence on system performance and energy efficiency.

Figure 2.8 shows the frequencies of DRAM row-buffer conflicts (β) and the relative IPCs of 3 benchmark suites on systems that the number of active threads or the number or ranks per memory controller is varied. There are five configurations on each application. The two leftmost configurations have one rank per memory controller with 4 and 16 active threads. Three remaining configurations have 64 active threads with 1, 4, and 16 ranks per memory controller. For each suite, applications which do not access main memory frequently are not shown due to space limitations, but they are included when average values are computed. The open policy is used to measure the frequency of row-buffer conflicts. The relative IPC of an application is the ratio of the application IPC with the open policy over the one with the closed policy.

Average β decreases in all three cases with increasing S or fewer active threads. This is because memory requests from the same thread on many applications have spatial locality. Requests from other threads that

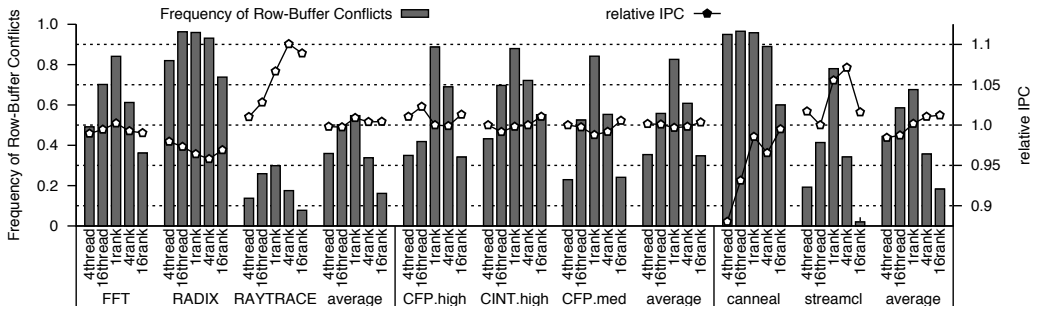
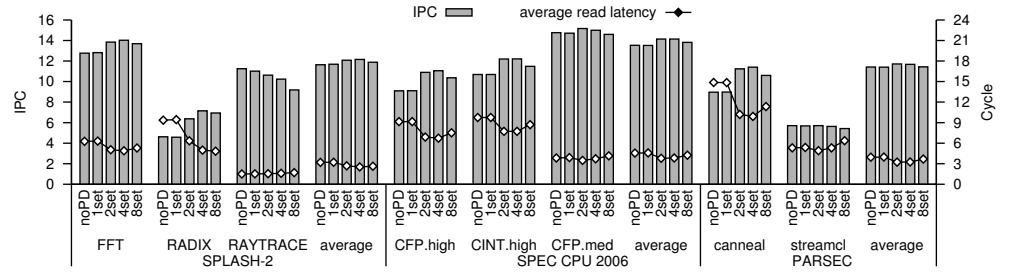


Figure 2.8: Frequency of row-buffer conflicts (β) and relative IPC when the number of active threads and ranks are varied. `nthread` configurations have `n` active threads with 1 rank per memory controller. `mrank` configurations have `m` ranks per memory controller with 64 active threads. Relative IPC is the ratio of the IPC with an *open* policy over the IPC with a *closed* policy on an application.

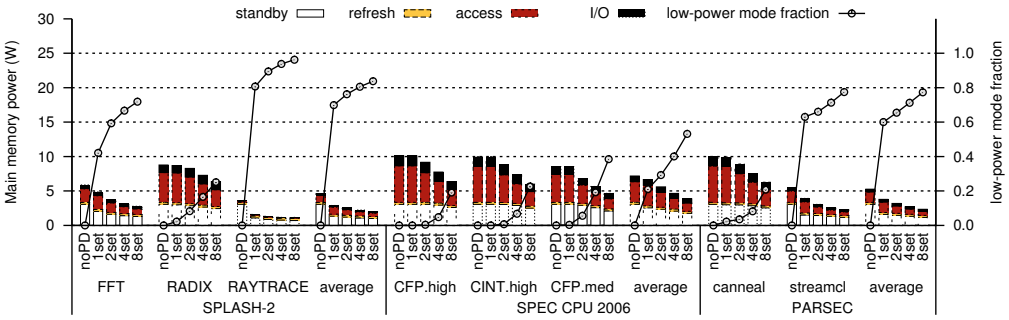
disrupt spatial locality due to a row-buffer conflict are less likely when there are more ranks. When, as in RADIX and canneal, there is low spatial locality, this effect is slightly less. When β is close to unity, the closed policy lowers the average access latency because the precharge delay is off the critical path. On the contrary, the open policy is beneficial if β is low since a memory request can often be served by a single column-level command. β is not always tightly coupled to the relative IPC though, because the difference in memory system access latency affects the core performance only when the latency hiding techniques in the cores (chip multithreading in our test system) cannot tolerate the cache miss latency. Since RADIX and canneal are memory bandwidth intensive applications, the closed policy provides higher IPC. On many applications, however, the open policy provides similar or even higher performance than the closed policy, especially when the system has more ranks. Considering that lower β will dissipate less main-memory power and rank subsetting will provide more DRAM banks to the system, we choose the open policy for the remainder of the evaluation.

2.3.2 Single-Rank Performance and Power Efficiency

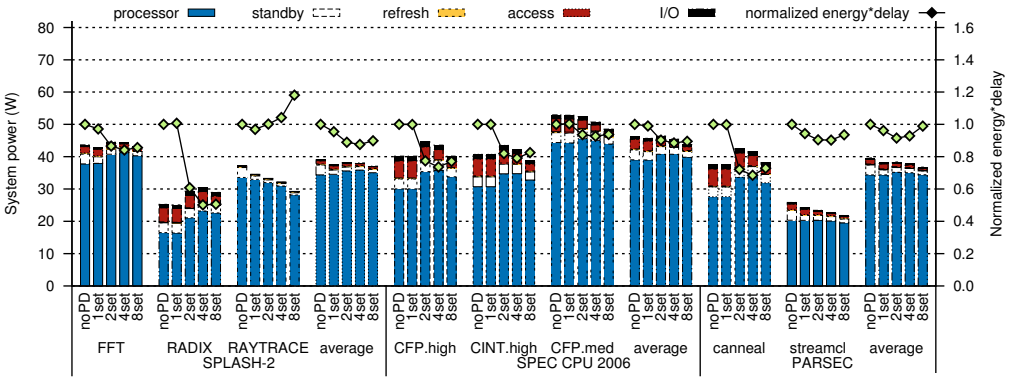
We first explore the optimal numbers of rank subsets that provide higher IPCs and lower system energy-delay products when we place 1 memory rank per memory controller ($R = 1$). We also identify major factors that improve the system energy-delay products. Figure 2.9 shows the performance and power result of 3 benchmark suites on a system with Multicore DIMMs where each memory controller has 1 memory rank. There are five configurations on each application. The left-most configuration has one subset per memory rank and does not exploit the low-power mode of DRAM chips, which is the baseline configuration. Four remaining configurations have their memory controllers exploit the low-power mode and the number of rank subsets are varied from 1 to 8. For each suite, applications whose performance is not sensitive to the number of rank subsets are not shown due to space limitations, but they are included when average values are computed. Off-chip memory demands depend on the instructions per cycle (IPC), the number of memory requests per instructions, and the last-level cache miss rates. Figure 2.9(a) shows the IPC and the average read latency



(a) IPC and average read latency. `noPD` configuration does not utilize DRAM power-down modes, and has 1 subset per rank. `nset` configurations utilize DRAM power-down modes, and have `n` subsets per rank.



(b) Memory power breakdown and mean fraction DRAM chips stay in a low-power mode



(c) System power breakdown and energy×delay (lower is better)

Figure 2.9: Memory and system level power and performance on a system with 1 rank per memory channel on 3 benchmark suites. For each suite, applications whose performance is not sensitive to the number of rank subsets are not shown due to space limitations, but they are included when average values are computed.

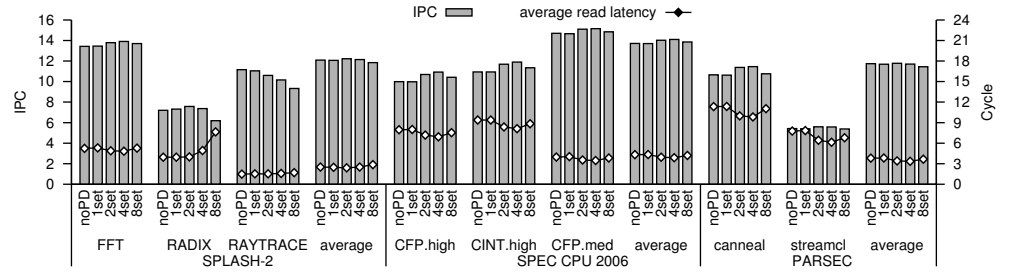
while Figure 2.9(b) shows the memory power breakdown of applications. Static power is divided into refresh and standby power, while dynamic power is divided into chip I/O power and access power within DRAM chips performing read, write, activate, and precharge operations.

RADIX, CFP.high, CINT.high, and canneal are applications having high main-memory bandwidth demand, which consume more DRAM dynamic power than other applications. The performance of these applications, which is closely correlated with their average read latency due to their high cache miss rate, strongly depends on the number of rank subsets per rank. Except for RADIX, the IPC increases until there are 2 or 4 subsets per rank and then decreases. As explained in Section 2.1, it is primarily due to interaction of two factors: access latency and effective bandwidth of memory channels. RADIX, an integer sorting application, takes advantage of higher memory bandwidth so that its performance keeps improving as S increases until there are 8 subsets per rank. In contrast, RAYTRACE doesn't stress the memory channel bandwidth, but it is very sensitive to access latency, so its IPC drops as a memory rank is split into more subsets.

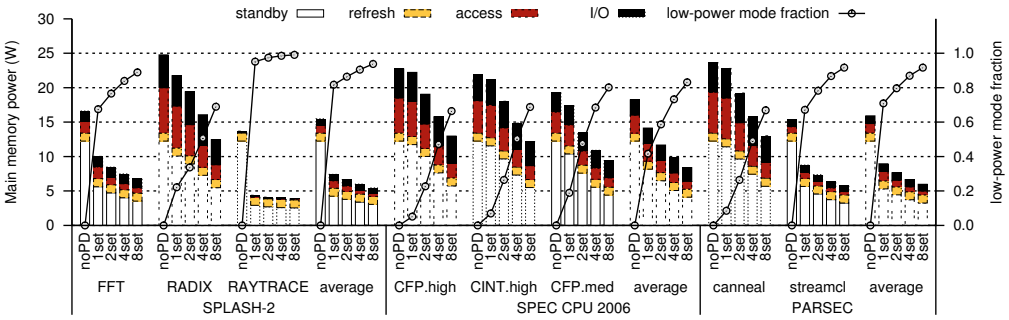
FFT and CFP.med are applications with medium main memory bandwidth demand. The relationship between the IPC and the number of rank subsets is similar to that of the applications with high bandwidth demand but with smaller variation. However, compared to other applications, the fraction of time that DRAM chips stay in a low-power mode (which is shown in Figure 2.9(b)) substantially increases as the number of rank subsets increases. Nevertheless, on applications with low bandwidth demand, 1 subset per rank is already enough for the memory controller to put DRAMs in a low-power state most of the time. Conversely, applications with high bandwidth demand rarely leave rank subsets idle, regardless of the number of subsets, so the power-down mode is not often used.

Regardless of memory demands of the application, there are substantial savings in dynamic energy since fewer bits are activated/precharged per memory access as the number of rank subsets increases. However, since dynamic power is proportional to the performance (IPC) of an application, this reduction in dynamic power is less apparent when its performance improves. Figure 2.9(c) shows the system power breakdown and the system energy-delay product of workloads. The energy-delay product improves substantially on applications with high main-memory bandwidth demand, on average 25.7% among four applications when $S = 4$. Across the workloads, the energy-delay product is improved by 4.5%, 0.9%, and 3.8% on SPLASH-2, SPEC CPU 2006, and the PARSEC benchmarks by utilizing a DRAM power-down mode. Rank subsetting brings additional 8.2%, 10.6%, and 3.4% improvement when $S = 4$. It shows that the effectiveness of these two techniques is complementary. The main-memory power always improves as the number of rank subsets increases. However, the system energy-delay product degrades on most applications when S increases from 4 to 8. When memory channels are highly utilized, dynamic power is much larger than static power, and rank-subsetting provides more improvement on system energy-delay product than DRAM power-down modes.

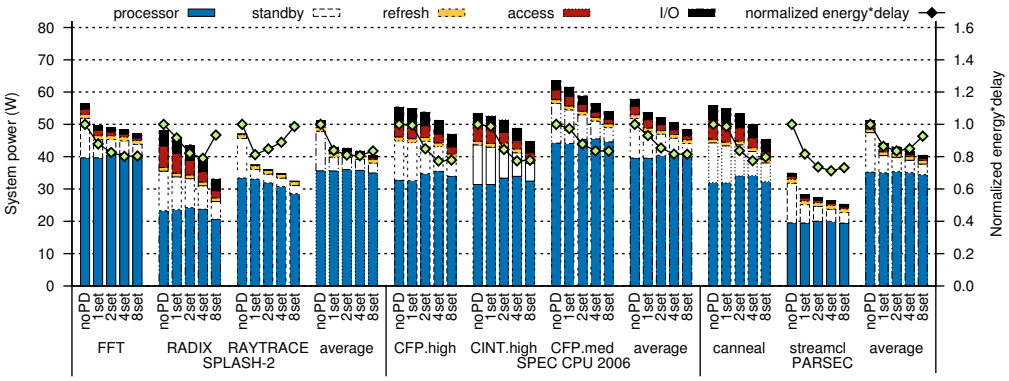
In summary, when each memory controller has 1 memory rank, 2 or 4 subsets per rank have higher IPC and lower memory access latency on average compared to other configurations. As the number of subsets per rank increases, dynamic power consumption of the memory systems decreases and DRAM chips stay in a low-power mode more frequently. Since the memory system power is substantially lower than the processor power, higher IPC due to rank subsetting affects EDP more than the energy-saving of utilizing a DRAM power-down mode.



(a) IPC and average read latency. `noPD` configuration does not utilize a DRAM power-down mode, and has 1 subset per rank. `nset` configurations utilize a DRAM power-down mode, and have `n` subsets per rank.



(b) Memory power breakdown and mean fraction DRAM chips in a low-power mode



(c) System power breakdown and energy×delay (lower is better)

Figure 2.10: Memory and system level power and performance on a system with 4 ranks per memory channel on 3 benchmark suites. For each suite, applications whose performance is not sensitive to the number of rank subsets are not shown due to space limitations, but they are included when average values are computed.

2.3.3 Four-Rank Performance and Power Efficiency

When more ranks are attached per memory channel so that the main-memory capacity increases, the relationship between application performance, memory power, and system energy-delay product changes in a way that it becomes more important to exploit DRAM power-down modes to achieve better EDP. Figure 2.10 shows the power and performance of the 3 benchmarks on a system with 2 dual-rank DIMMs per channel

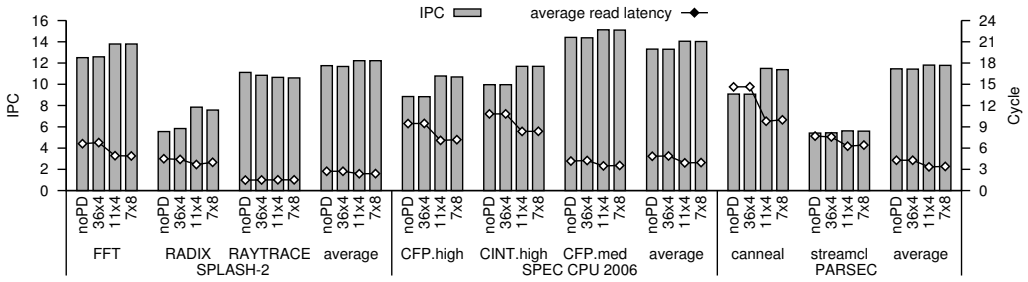
($R = 4$). The increase in the IPC from 1 to 2 rank subsets on applications with high memory bandwidth demand is not as much as in the previous system configuration. As analyzed in Section 2.1, with 4 times more independent DRAM banks per channel, the activate-to-activate time constraint becomes a smaller problem as a memory controller can issue commands to other ranks, leading to high performance even without rank subsetting. Still, 2 rank subsets perform better than 1, since the timing constraints on each switch of bus ownership limit performance, and this is alleviated with multiple subsets as each DRAM transaction takes longer.

With 4 ranks per channel, static memory power (such as standby and refresh power) and I/O power increase substantially, becoming a significant part of the total memory power as shown in Figure 2.10(b). Since the peak bandwidth per channel is the same as with 1 rank per channel, banks are idle more often, hence it is more likely that the memory controller can exploit low-power modes. I/O power increases since there are more termination resistors per data bus, and sometimes I/O power even surpasses the access power within DRAM chips, highlighting the need for more energy-efficient technologies, such as differential signaling or point-to-point connections. The total memory power becomes comparable to the processor power on applications with high memory demand (Figure 2.10(c)). However, since the performance of these applications varies less than in the single rank case as the number of rank subsets changes less than before, the energy-delay product improves less as multiple subsets are used: 3.8% on SPLASH-2 with 4, 12.3% on SPEC CPU 2006 with 4, and 1.8% on PARSEC with 2 rank subsets all compared to the configuration utilizing a low-power mode but no rank subsetting. Rather, there are bigger savings by utilizing DRAM low-power modes even without rank subsetting: 16.1% and 13.6% on SPLASH-2 and PARSEC. The SPEC CPU 2006 benchmarks access main memory more often than others, so a 6.9% improvement in energy-delay product from putting DRAMs in a low-power mode is less than the additional improvement due to the rank subsets.

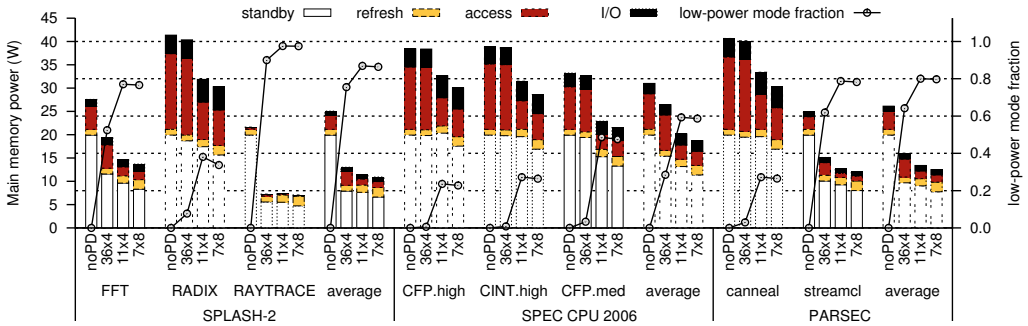
As the number of ranks per memory controller increases from 1 to 4, the number of DRAM banks in the memory system quadruples as well, becoming 128 and surpassing the number of concurrent threads (64) of the CPU. Additional increase in the number of DRAM banks by rank subsetting helps less on improving the IPC. Rather, serialization in cache line transfers due to rank subsetting lowers the IPC when $S = 4$ or 8. As a result, utilizing a DRAM power-down mode provides more impact on EDP compared to rank subsetting in general. Still, the applications with high memory bandwidth demands exploit rank subsetting effectively. The difference in EDP by the choice of specific rank subsetting implementation decreases as more ranks are deployed.

2.3.4 Power and Performance of Chipkill-level Reliability

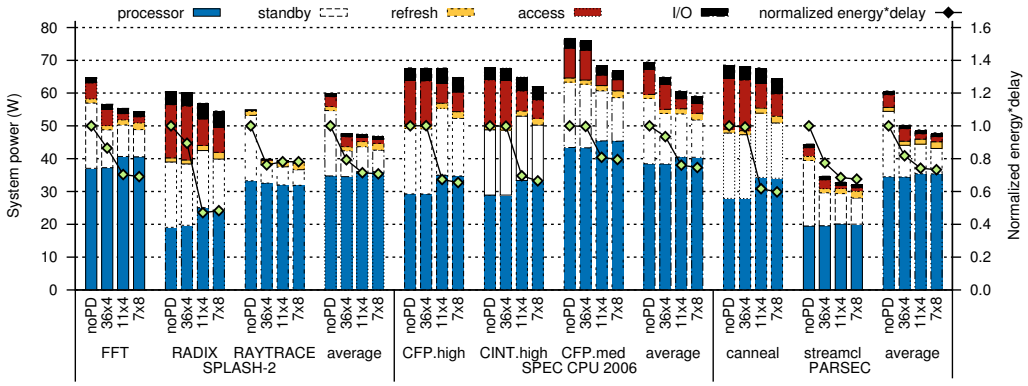
Both rank subsetting and the exploitation of DRAM low-power modes improve the energy-delay products of systems with chipkill-level reliability, because supporting chipkill increases the number of DRAM chips involved per memory access while providing the same memory capacity. Figure 2.11 shows the performance and energy efficiency of 4 different memory systems supporting chipkill level reliability. On each application, the first two columns (`noPD` and `36x4`) have the values for a conventional chipkill solution, in which each



(a) IPC and average read latency.



(b) Memory power breakdown and mean fraction DRAM chips in a low-power mode



(c) System power breakdown and energy×delay (lower is better)

Figure 2.11: Power and performance of applications on systems with chipkill-level reliability. There are four configurations per application: both `noPD` and `36x4` for the conventional system with 36 ×4 4Gb DRAM chips per rank while power-down modes are applied only to `36x4`, `11x4` with 11 ×4 4Gb DRAM chips per MCDIMM rank, and `7x8` with 7 ×8 8Gb DRAM chips per MCDIMM rank.

rank consists of 36 ×4 4Gb DRAM chips (32 chips for data and 4 for parity and other information). The first column does not utilize the DRAM low-power mode while the second column does. Since each DRAM chip has a prefetch length of 8, a minimum transfer size $36 \times 8 \times 4 = 1152$bits of data should be read or written. The cache line size of the system is 72B = 576bits including ECC in internal caches, so half of the data are not used. Although burst chopping [107] can be used to save I/O power by not transferring unused

data, substantial DRAM dynamic energy is still wasted. In the second memory system, denoted `11x4`, each Multicore DIMM rank consists of 2 subsets, each with 11 $\times 4$ 4Gb DRAM chips. As a result, 2/9 = 22.2% more DRAM chips are used for error correction over a system that only provides parity or ECC. On the other hand, only 11 chips are used per memory access, less than 1/3 of DRAM chips compared to the conventional chipkill DIMMs. The third system, denoted `7x8`, has 2 Multicore DIMM VMDs (subsets) per rank, each with 7 $\times 8$ 8Gb DRAM chips. Three more DRAM chips are used for error correction, but it needs fewer than 1/5th as many DRAM chips per access compared to `36x4`. All configurations have the same data capacity. The `36x4` configurations have 2 ranks per channel, and the last two have 4 ranks (two dual-rank DIMMs) per channel and 2 subsets per rank.

In traditional chipkill systems, memory power can surpass the processor power, amplifying the importance of improving energy efficiency of processor-memory interfaces. The `36x4` configurations clearly performs worse than others since its effective per-rank bandwidth is lower while `11x4` and `7x8` obtain benefits from having multiple rank subsets — more banks and less frequent timing constraint conflicts, as shown in Section 2.3.3. There are major savings in DRAM dynamic power on configurations with high-reliability MCDIMMs (Figure 2.11(b)) since far fewer DRAM chips are used per access. This also helps memory controllers to idle subsets so that the static power of MCDIMMs can be even lower than conventional chipkill DIMMs unless the whole memory system is largely idle like on RAYTRACE and streamcl. This is true even though systems with high-reliability MCDIMMs have more DRAM chips than `36x4`. Therefore both subsetting DRAM ranks and exploiting DRAM low-power modes are equally important to enhancing energy-delay product. By utilizing DRAM low-power modes, system energy delay product on `36x4` is improved by 16.3%. Rank-subsetting leads to additional improvement on system energy delay product: 12.0% and 13.1% on `11x4` and `7x8` compared to the `36x4` with a low-power mode utilized.

2.4 Related Work

A large body of computer architecture research aims to improve the performance, energy efficiency, and reliability of main-memory systems. One key idea is to group together memory accesses with the same access type and similar addresses by reordering, in order to minimize the performance degradation due to various timing constraints on DRAM accesses. There have been proposals to exploit these characteristics to achieve higher performance on vector [101], stream [136], and single-core and multicore processors [114, 119]. Higher performance typically leads to higher energy efficiency by reducing execution time and saving static power. We use one of the latest proposals, Parallelism-Aware Batch Scheduling [115], in our evaluation.

Multiple power states were introduced in Rambus DRAM (RDRAM).Some studies try to exploit these low power states in RDRAMs by allocating and migrating OS pages in order to put DRAM chips into a low power state for longer periods [62, 86]. Modern DDRx DRAM chips also have multiple low power states. Hur et al. suggest ways to exploit them in a memory scheduler [64]. These are complementary to our idea of saving dynamic energy per memory access, which can also be synergistic as shown in Section 2.3. Ghosh and

Lee suggest a memory controller design with smart refresh [43] to save refresh power. This is complementary to our idea as well.

Among the proposals advocating rank subsetting, module threading [166] relies on high speed signaling. The memory controller outputs separate chip select signals for selecting a subset of devices. Zheng et al. called a subset a mini-rank [182] and proposed a design in which all mini-ranks in a memory rank send and receive data to/from a memory controller through a mini-rank buffer. They did not consider processor power and reliability in their evaluation of their architecture. The key difference between Multicore DIMM and mini-rank is the placement of the data mux and address/command demux. Mini-rank has a demux per memory rank while Multicore DIMM has one per memory channel. As a result, mini-rank is more costly in energy and component count. Multicore DIMM has one address/command demux per memory rank, while mini-rank does not have any. Since address and command signals must be registered per rank due to signal integrity issues, the incremental cost of the demux register is minimal. Both proposals need chip select signals per rank subset.

There are recent proposals to improve the efficiency of main-memory systems either by reducing OS page sizes [153] or by modifying the internal microarchitecture of DRAM chips [161]. Sudan et al. [153] suggested to collocate cache blocks that are frequently utilized into the same row of the DRAM bank by reducing OS page sizes and providing hardware/software mechanisms for data migration within main memory in order to reduce DRAM row-buffer conflicts. Udipi et al. [161] proposed changes to the DRAM microarchitecture to alleviate the overfetch problem while sacrificing area efficiency. Their techniques either delay DRAM row activation until both row-level and column-level commands reach the DRAM chip in order to activate DRAM cells that only correspond to the cache line to be accessed, broaden the internal DRAM data path so that an entire cache line can be fetched from a small portion of a bank in a DRAM chip, or provide a checksum logic per cache line to provide RAID-style fault tolerance. Rank subsetting does not require changes to OS or DRAM chip microarchitecture and can be supplementary to these techniques.

2.5 Conclusion

Memory power is becoming an increasingly significant portion of system power. In this chapter, we holistically assessed the effectiveness of rank subsetting on the performance, energy efficiency, and reliability at the system level rather than only looking at the impact on individual components. We also quantified the interactions between DRAM power-down modes and rank subsetting. For single-rank and four-rank memory systems, across the SPLASH-2, SPEC CPU 2006, and PARSEC benchmarks, we found that power-down modes without rank subsetting could save an average of 3.5% and 13.1% in system energy-delay product. When rank subsetting is added, additional average savings of 7.7% and 6.6% are obtained. In a typical high-throughput server configuration with 4 ranks per memory controller and 4 memory controllers in a system, dividing a memory rank into four subsets and applying DRAM low-power modes provides 22.3% saving in memory dynamic power, 62.0% in memory static power, and 17.8% improvement in system energy-delay

product with largely unchanged IPC (instructions per cycle) on tested applications. The cost of rank subsetting is very low: for example, in the Multicore DIMM approach, a latch on each DIMM is converted to a demux latch. Thus, given the insignificant investment required, these system energy-delay product savings are remarkably high.

Rank subsetting increases the amount of data read out of a single chip, so it also can increase the probability of a large number of bit errors when an entire chip fails. Therefore, we extended the MCDIMM design for high-reliability systems. Enhancing reliability involves a tradeoff between energy and code rate. Traditional chipkill solutions optimize code rate at the cost of reduced energy efficiency. A Multicore DIMM-based high-reliability memory system uses 42.8% less dynamic memory power, and provides a 12.0% better system energy-delay product compared with a conventional chipkill system of the same data capacity, but needs 22.2% more DRAM devices. Given the substantial portion of datacenter TCO related to power provisioning and energy consumption, solutions which strike a balance between code rate and energy efficiency make more sense.

Overall, we expect rank subsetting, especially reliability-enhanced Multicore DIMM, to be a compelling alternative to existing processor-memory interfaces for future DDR systems due to their superior energy efficiency, tolerance for DRAM timing constraints, similar or better system performance, and ability to provide high reliability. Moreover, as DRAM semiconductor processes scale more slowly than logic processes, memory power will grow as a share of overall power consumption. Thus, the need for techniques like rank subsetting will only increase over time.

Chapter 3

Per-Core Power Gating for Datacenters

Orthogonal to dynamic power consumption, static power consumption is a large component of overall server power consumption, and the processor is a large contributor to this. Dynamic Voltage and Frequency Scaling (DVFS) and deep sleep states represent the state of the art in power management for processor chips. Nevertheless, they are likely to suffer from limitations in the future. With continued semiconductor scaling, the dynamic range of DVFS is going to be limited and ineffective at addressing the increasing fraction of leakage power dissipation. Moreover, modern workloads exhibit highly variable resource usage, frequently operating at CPU utilizations ranging from 10% to 80%. Current deep sleep states do not allow a reduction in the leakage power of underutilized resources without forcing the whole chip into an idle mode.

To address these challenges, this chapter proposes and evaluates per-core power gating (PCPG) as an additional power management solution for multi-core processors. PCPG enables the elimination of leakage power in underutilized cores, while the rest of the system (other cores and shared caches) offer the resources necessary to handle moderate or low performance requirements. We present two practical implementations for power gates using on-chip and package-level sleep transistors, as well as the hardware and software infrastructure necessary to manage per-core power gating. Using a test bed based on a commercial 4-core chip and a set of real-world application traces from enterprise environments with varying processor utilization, we have evaluated the potential of PCPG for power and energy savings. We show that PCPG can significantly reduce CPU energy consumption (up to 40%) without significant performance overheads. We also demonstrate that PCPG works well with dynamic voltage-frequency scaling techniques and that the combination provides a holistic approach to address both dynamic and leakage power in multi-core chips.

The work presented in this chapter was originally submitted to the *15th International Symposium on High-Performance Computer Architecture* on August 17, 2008. Three days later on August 20, 2008, Intel announced at the *Intel Developers Forum* its new "Nehalem" architecture [150], which included the first implementation of per-core power gating for general-purpose processors. Understandably, the response to our paper was mixed during peer-review months later, ranging from "very timely... detailed analysis" (Accept) to "has already been adopted by industry" (Strong Reject). The HPCA submission was ultimately rejected, but

an abbreviated version of the manuscript was eventually accepted to *Computer Architecture Letters* [87] and published in 2009.

In this chapter, we present the original, full description and evaluation of our work on per-core power gating for datacenter workloads. In Section 3.7.1, we compare commercial examples of per-core power gating to our own design, and offer new critique of the Nehalem implementation.

3.1 Introduction

Dynamic Voltage and Frequency Scaling (DVFS) is one of the most successful power management mechanisms provided by modern processors. DVFS has been extensively used to fight the dynamic component of CPU power consumption [37, 67, 123, 131, 132]. However, the increasing contribution of leakage to total power consumption as semiconductor technology scales and the pervasive use of multi-core chips for all types of workloads provide an opportunity for additional innovation in processor power management.

The efficacy of DVFS is limited by its dynamic range, ultimately fixed by the minimum voltage necessary to operate the transistors and the maximum voltage that can be thermally tolerated. For instance, the voltage range for DVFS for a 65nm Intel Core 2 Duo processor is 0.75V-1.35V. The range is constrained by several characteristics of the manufacturing process, like the threshold voltage (V_{th}) and noise margins, that are not likely to scale significantly in the future [28]. Consequently, the relatively high voltage of the minimum DVFS state limits the effectiveness of this technique in reducing leakage power [85].

Voltage scaling is also less applicable to multi-core environments running heterogeneous workloads. Since there is typically a common voltage plane shared across all cores, a lower supply voltage cannot be employed unless all cores are simultaneously ready to use it. Alternatively, multiple voltage planes can be implemented at additional cost and design complexity. Thus, any heterogeneity in the characteristics of disparate workloads consolidated onto a multi-core chip forces a compromise in either performance, power consumption, or cost. More importantly, prior study of per-core DVFS mechanisms have focused primarily on minimizing energy consumption for workloads that utilize every core in the system [68, 79, 88]. Hence, they place particular emphasis on dynamic power consumption at peak utilization. However, many deployments of multi-core chips exhibit only moderate utilization. For example, Barroso observes that processors in data centers operate mostly within a utilization range of 10% to 50% [10]. Likewise, end-user (client) systems often remain mostly idle, perhaps waiting for the user to click on the next web link. This trend suggests that the contribution of leakage power to the overall energy consumption will increase.

These issues motivate research on power management techniques that enable real differentiation in the power consumption of individual cores and very low-power modes. Power gating is a technique that allows one to shut off—i.e. *gate*—the power supply of a logic block by inserting a gate (or sleep transistor) in series with the power supply. Gating the power supply results in virtually no power consumption in the gated block. This technique can thus radically reduce a core's power consumption, providing a very effective per-core deep sleep state. In this chapter, we specifically make the case for *per-core power gating (PCPG)*.

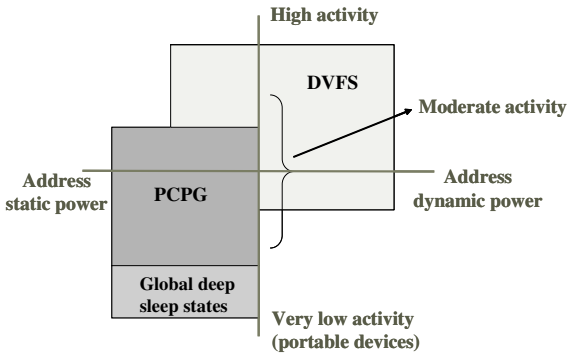


Figure 3.1: Per-Core Power Gating (PCPG) in relation to DVFS and global deep-sleep states. PCPG targets moderate to low activity workloads and addresses static power savings. It also has full per-core granularity as opposed to DVFS (limited core granularity) and global deep sleep states (chip-level granularity)

Per-core power gating compliments existing voltage scaling techniques by providing an effective mechanism to reduce leakage power in a multi-core system when it operates at moderate utilization or when the aggregate workload exhibits high variability in resource usage. Figure 3.1 illustrates the relationship of PCPG to voltage scaling techniques. DVFS reduces dynamic power at high levels of utilization, while global sleep states can deal with extremely low-activity operation modes. PCPG, on the other hand, efficiently deals with moderate-activity workloads, aggressively reducing the leakage power at the expense of limited performance loss. Being an orthogonal technique, PCPG can be used in conjunction with these other power management solutions (such as DVFS) in the same processor.

This work makes the following contributions. We present a per-core power gating architecture for multi-core processors. The power gates are practical to implement in a cost-effective manner using on-chip sleep transistors or die-stacking technology. The gated cores consume negligible leakage power while the remaining system, cores and shared caches, provide full functionality. A power control architecture allows software to turn on and off individual cores using PCPG, as utilization and quality-of-service requirements vary. We have developed example management policies that use PCPG to achieve desired power savings. These policies can be applied statically or dynamically, and can run in conjunction with DVFS policy managers. Using a test bed based on a commercial quad-core chip and a set of real-world application traces with varying processor utilization, we have quantified the potential of per-core power gating. We demonstrate that PCPG leads to significant energy reduction (up to 40%) and power savings (up to 20%) without significant impact on performance. Moreover, we show that combining PCPG and DVFS leads to additional advantages (up to 59% energy savings and 26% power reduction) by providing a holistic approach to address both dynamic and leakage power in multi-core chips.

The remainder of this chapter is organized as follows. Section 3.2 presents background on DVFS and deep sleep states on modern processor. Section 3.3 discusses the architectural implications of PCPG and discuss the circuit model that we used to model PCPG. Section 3.4 discusses static and dynamic policies for PCPG.

Section 3.5 discusses our methodology and Section 3.6 presents our experimental evaluation. Section 3.7 discusses related work and Section 3.8 concludes the chapter.

3.2 Background on Power Management Techniques

The power consumption of a chip can be optimized at design time using circuit techniques, such as gate sizing, transistors with alternative thresholds, or architectural techniques such as simplified pipelines or drowsy caches [48, 117]. At runtime, power consumption can be managed using techniques such as clock gating and voltage scaling.

3.2.1 Voltage Scaling

Contemporary systems use an off-chip voltage regulator module (VRM) to provide the voltage levels needed for the performance (P) and sleep (C) states of the ACPI standard [59]. The *P states* correspond to DVFS levels and are directly controlled by the processor using the voltage identification (VID) pins [66].

As a first approximation, processor dynamic power consumption is given by the equation:

$$P_{dynamic} = CfV^2 \tag{3.1}$$

where C is switching capacitance (assumed to be constant), f is frequency, and V is voltage. The performance relationship between frequency and voltage is largely linear [65]. DVFS can optimize power and energy efficiency by two means: by sacrificing performance to exploit the quadratic relationship between voltage and dynamic power consumption, or by observing that memory-bound workloads are not significantly slowed by lowering the processor's clock rate. For instance, a notebook system may use the lowest voltage/frequency setting when running on battery in order to minimize power consumption. Intentions aside, care must be taken when attempting to minimize *energy* consumption (and thus maximize battery life) with DVFS. When in a low P state, the integration of power consumption (especially considering full system power consumption) over the longer duration of a workload can result in a net increase in energy consumption. Therefore, statically assigning a system to its lowest P state is usually best only if it is known that the system will remain mostly idle (as notebooks often are). In other scenarios, it is best to dynamically monitor processor utilization and adaptively switch between P states [46, 58, 63, 94, 174].The OS or a hypervisor requests the P states that meets the current performance requirements and power constraints given some control algorithm.

3.2.2 C-states

C-states target power reduction during long idle periods with no useful code being executed. Some chips implement a few C states without involving the VRM, instead using aggressive clock gating or by disabling clock generation. Deep sleep C states that aggressively reduce voltage to mitigate leakage involve an external

	Advantages	Disadvantages
Global deep sleep states	Easy to implement, no impact on the chip's power network	Whole chip granularity, driven by external bulky VRM
Per-core power planes	Limited impact on power network	Driven by external bulky VRM, no decoupling capacitor sharing
Per-core switching regulators	Core granularity, fast DVFS	Large on-chip area, bulky passive elements
Per-core voltage regulators	No external inductors	Limited efficiency
PCPG	Core-granularity, deep sleep states	No per-core DVFS

Table 3.1: Comparison of the main alternatives to per-core power gating.

entity, such as the Southbridge or a platform-specific microcontroller, to control the VRM. This entity coordinates the powering down of the processor's VRM with the powering down of other system components, and monitors for wake-up events. In general, deep sleep states lead to maximum power savings when the voltage level supplied by the VRM is well below the threshold voltage of the chip or is entirely off. As a consequence, the processor may lose most or all of its volatile state (register contents, modified cache lines, etc.), so the transition to and from a deep sleep state must be carefully coordinated with the operating system, whose job it is to transfer and restore volatile state to and from main memory or disk.

Almost all modern processors provide whole chip deep sleep states. All cores in a multi-core with a shared voltage domain must reach the lowest C state before the voltage can be reduced, to ensure that the voltage is not reduced prematurely [67, 123].

3.2.3 Multiple Voltage Domains

The evolutionary approach to power management for multi-core chips would be to provide per-core voltage domains and implement DVFS and deep sleep states. After all, many chips already have multiple power planes that enable distinct power management of resources like cores, shared caches, I/O interfaces, and analog sections [116, 139]. Following this approach, the AMD's Griffin mobile platform has separate voltage domains in order to enable per-core DVFS [123]. Nevertheless, Griffin still implements deep sleep states for the whole chip, not per-core. In general, separate voltage domains introduce several design issues. They require more supply voltages from the external VRM(s), which will increase its complexity. Also, within the chip, having separate voltage domains prevents the sharing of decoupling capacitance between separate domains, which makes the power supply noisier. For this reason, the power network for the Power6 chip uses a single voltage plane for cores and caches [70].

Kim et al. have proposed the use of on-chip per-core switching voltage regulators [79]. This solution enables very fast voltage transitions among P-states (DVFS). On the other hand, due to area efficiency and thermal constraints, on-chip switching regulators can only provide limited voltage variations, making them less suitable for per-core deep sleep states. They also require external inductors and thus pose challenging layout, packaging, and overhead issues [90]. Linear on-chip regulators that do not require passive elements have also been demonstrated [54]. Nevertheless, their efficiency dramatically drops for high step-down ratios, limiting their useful voltage range [79]. Their efficiency is limited by the V_{out}/V_{in} ratio, which is 58% when

converting from 1.3V to 0.75V. Typical switching regulators operate at 80% efficiency.

3.2.4 Motivation for our work

Existing power management methodologies have been based upon the external VRM to provide efficient power consumption at both high utilization (DVFS) and at idle utilization (deep sleep states). However, they do not sufficiently manage power consumption for moderately utilized multi-core chips, where a small number of cores may be sufficient to meet performance needs. In this case, DVFS would consume static power across all cores, while the system cannot enter a deep sleep state unless it expects all cores to be idle for significant duration. Also, as discussed above, it is unlikely that providing separate power domains with off-chip or on-chip regulators will be a practical solution for power management in a multi-core chip at moderate utilization. Finally, the efficacy of DVFS altogether is in question, as voltage scaling is reaching its limits and the dynamic range of DVFS is much lower than it has been in the past [85].

The per-core power gating technique in the following sections enables fine-grain deep sleep states within a multi-core chip, which are necessary for power efficiency at moderate utilization. The technique is practical and complements existing per-chip DVFS, as well as per-chip deep sleep states.

3.3 Per-Core Power Gating Architecture

This section describes the hardware components and the control mechanisms necessary to implement per-core power gating (PCPG). For simplicity, we initially describe PCPG independent of traditional voltage scaling (P and C states). In Section 3.4.3, we explain how PCPG can be used in conjunction with existing techniques for voltage scaling.

3.3.1 System Architecture

Figure 3.2 presents the system components for PCPG. We focus on managing the power of cores, but the technique is applicable to all major blocks in a multi-core chip (e.g. shared cache tiles, encrypt or protocol offloading engines, and memory or I/O channels). Each core is associated with a power gate and the circuitry necessary to control and drive the gate. Each power controller makes the corresponding power gate control architecturally visible so that privileged software (firmware, hypervisor, or OS) can control the gate through register or memory operations. The power manager implements a control algorithm that determines when each core is power gated given measured utilization, workload characteristics, or QoS requirements.

The power controllers, the power manager, and control channels that connect them are abstractions that can be implemented in many different ways, depending on the technology and specific system. For example, the power interfaces may be additional states or fields encoded in the existing registers of the ACPI hardware. The power manager may be implemented as a periodic service running on a core or as a separate microcontroller. The control bus may be a stand-alone bus (e.g., I^2C-inspired) or an existing system interconnect that

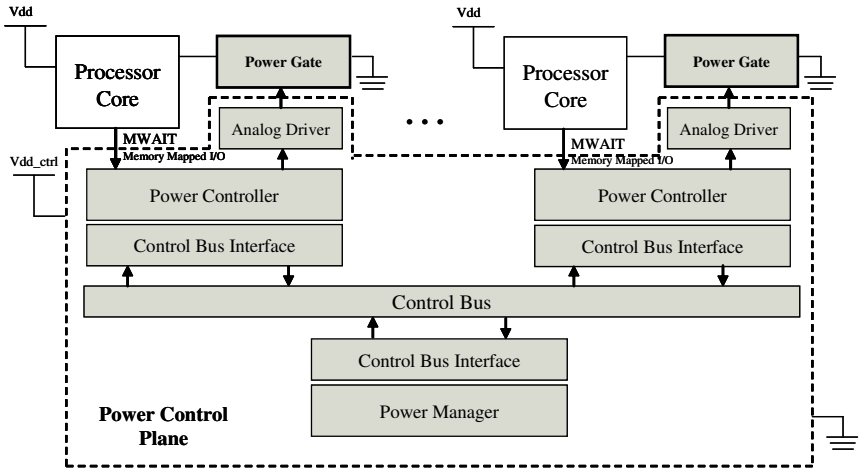


Figure 3.2: The system architecture for PCPG.

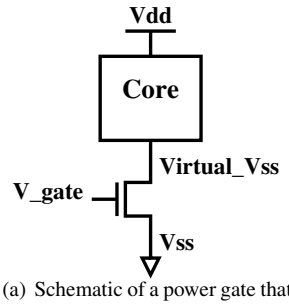
provides access to memory-mapped registers.

A core may message its associated power controller directly through the use of a special instruction, like MWAIT, or by writing to a memory-mapped register. The power manager may communicate with any power controller via the control bus. The former mechanism allows each core to have the final say in its own loss of power and ensure that the OS had ample time to prepare. The latter mechanism allows the power manager to actuate the awakening of gated cores.

All of these PCPG control circuits should be connected to an uninterrupted power supply that is not gated as they need to operate even when cores are power gated. For example, the PCPG circuits could use the I/O power plane. The digital circuits associated with the power gate should be relatively small and low power, so we expect that their presence should not significantly impact the total power consumption. We discuss the power consumption of the power gate itself in Section 3.3.2.

3.3.2 Circuits for Per-core Power Gating

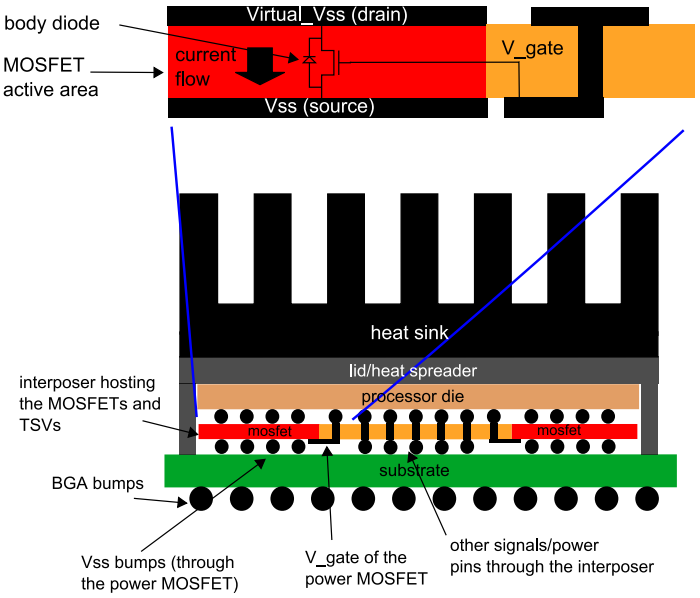
Power gates can be implemented using sleep transistors [117]. Sleep transistors are significantly simpler than voltage regulators (used for voltage scaling) as they do not require passive elements nor active control and have smaller area and power overhead. A voltage regulator, such as the one discussed by [79], is at least $4\times$ bigger and exhibits twice as much power loss as a sleep transistor, even without considering the external inductor. In this chapter, we use Spice models to characterize the process of gating a core for performance and power. Figure 3.3(a) illustrates the basic circuit we assume, consisting of a NMOS transistor to gate the V_{ss}. We conservatively estimated the model parameters from published data [50, 79, 152] and measurements of an AMD 4-core Phenom X4 9850, which is the processor used in our test bed. We use NMOS devices because they are smaller than PMOS for the same channel resistance, but our results apply to PMOS devices as well.



(a) Schematic of a power gate that employs an N-MOSFET as a sleep transistor.

Core and Package	
Technology node	65nm
Area of the core	25 mm^2
I_{max}	33 A
V_{dd}	1.06 – 1.35V
I_{leak}	4.2A (1.06V), 5.3A (1.35V)
T	333K
C_{core}	50nF
C_{dec_int}	100nF
C_{dec_ext}	500nF
L_{ext}	150pH
Power gate	
R_{on}	1.8m Ω (60mV)
T_{on}	100ns
T_{off}	190ns
E_{swon}/E_{swoff}	160nJ
P_{max}	59mW
P_{leak}	1mW

(b) Circuit-level assumptions and parameters for each core and for the target power gate.



(c) On-package sleep transistor design. From the bottom, the stack includes: an organic substrate to connect to the Ball Grid Array (BGA), the interposer die with integrated power MOSFETs and through silicon vias (TSVs), the multi-core processor die, a heat spreader, and heat sink. The top of the figure shows a detailed cross-section of the power MOSFET, illustrating the drain, source, and gate (control) pads. The gate terminal of the MOSFET is on the bottom side (the active area of the device), hence the gate signal must go through TSV in order to drive it from the processor die. The orientation of the power MOSFET is as such in order to guarantee that the body diode (shown above) is correctly back-biased.

Figure 3.3: Circuit and details of the power-gate design.

Figure 3.3(b) reports our estimates based on SPICE simulations for the switching time and the energy consumption. Since we sized the sleep transistor quite aggressively to have very little resistance (1.8mΩ, targeting 60mV of voltage drop at 33A), switching energy and time primarily depend on the charging/discharging process of the capacitance seen from the *Virtual_Vss* node, i.e. the core's capacitance and the internal decoupling capacitors (C_{dec_int}). In the off→on transition, the core's capacitance is charged by the current flowing through the transistor channel. This transition is *per se* quite fast, since the transistor resistance is quite small. Nevertheless, the actual speed must be limited to control the dI/dt voltage noise on the power supply. We assume that the gate voltage can be driven "slowly" (10ns). In the presence of a 500nF external decoupling capacitor (C_{dec_ext}) per core, the voltage variation after closing the gate is within 10% of the V_{dd}, and the rush current is limited to 1A. The total settling time to have a supply voltage within 1% of its nominal value is less than 100ns. The on→off transition is slower (approximately 200ns) since the core's capacitance is discharged through the core's leakage current (4.2A), which is smaller than the current that can be driven in the sleep transistor.

The energy overhead of a transition is quite low, on the order of a hundred nanojoules. When a gate

is off (power is gated), it introduces leakage power of roughly a milliwatt, while its active power when the gate is on is roughly 6% of the core peak power consumption. Such low-resistance power gates do not cause significant thermal issues. We ensure that the power density developed within a power gate is always much lower than the power density of a core.

3.3.3 Implementation

Sleep transistors can be implemented on the processor die or on a distinct die that is stacked below the processor chip. The two approaches have similar characteristics in terms of latency and power overheads. They differ primarily in area, as the on-package approach does not introduce any overhead on the processor die.

On-chip power gates can be implemented using the high-V_{th} transistors available in all logic processes. The transistors are connected in parallel to form the gate. They can be laid out surrounding each core. Hattori et al. [53] give a detailed description of how this scheme can be practically implemented in a low-power processor. In our scenario, the transistor is much bigger, since it must sustain higher current, but we assume a very similar circuit and layout organization. According to our estimates, derived from Predictive Technology Models (PTM) [181] for 65-nm, the power gate occupies 7.65mm^2, or 22% of the area of a modern core including private L1 and L2 caches.

The on-package approach avoids the area overhead on the processor die by using die stacking technology and advanced packaging solutions [80]. In this case, we can also use power MOSFET technology for the sleep transistors [7]. Power MOSFET devices have been extensively demonstrated and are widely used for a large variety of power electronics applications including existing off-chip power regulators [21, 96]. The typical overall characteristics of power MOSFET devices in terms of channel resistance or current capability are similar to those of power gates built using logic-process CMOS transistors. The advantage of power MOSFET devices for on-package gates is their vertical nature, which simplifies the connection between the processor die and the power gate.

Figure 3.3(c) illustrates how power MOSFETs can be stacked on the processor die within a package enclosure, exploiting the vertical nature of these devices. The MOSFETs necessary for all power gates are integrated on a dedicated die, a silicon interposer that also hosts through silicon vias (TSVs) needed for other power and signal pins. A MOSFET is thus placed on the top of each core. This kind of flip-chip solutions with a silicon interposer has been widely explored in the literature for Systems-on-Package [80]. Figure 3.3(c) shows the details of the bonding between the two dies. The drain of the power MOSFET die is connected directly to the external V_{ss}, while its source goes directly to the power pads on the logic die (area pads) without the need for any vertical vias or additional wiring.

3.3.4 Power Gating Process

Analogous to when a user pushes the ”soft” power off button on a computer, power gating is not an immediate process. Under the control of privileged software, the core must go through a series of intermediate states to safely enter the power gated state. The intermediate steps involve i) saving the core's registers, ii) flushing dirty data from its private caches to the next level of the memory hierarchy, and iii) turning off its clock tree. To power on a core, the reverse process must be followed. To avoid significant spikes in supply voltage or memory contention as multiple cores flush or re-warm the caches, the manager could stagger transitions if multiple cores should be gated or turned on based on some workload change.

The latency of power gating is determined by many factors other than the physical latency of driving the power gate. We found that the software and hardware procedures involved take much longer than the circuit time. We measured the time that it takes to enable or disable a core in Linux as roughly 100ms. This time accounts for all the steps needed to stop the core and prepare it to be shut off. Transitions across P states for DVFS, for comparison, are much faster (microseconds), since the state of the processor does not need to be saved/restored.

3.4 Management Policies for Per-Core Power Gates

Similar to chip-wide DVFS (P states), we can manage PCPG using either static or dynamic policies. Since PCPG, like DVFS, enables a number of states that trade-off performance (number of active cores) for lower leakage power, the control algorithms for both can also be similar. Nevertheless, the implementation of PCPG involves saving and restoring state and must involve privileged software. DVFS, on the other hand, can be transparent to the OS as the number of active cores and the state of the system remain unaffected during voltage/frequency switches.

3.4.1 Static Policies

PCPG can be be used with static policies by allowing the hypervisor, operating system, or the user to statically select the number of cores in the system that are gated off. For instance, this approach may be appropriate for a notebook computer while on battery in order to minimize leakage power consumption. If the gated cores were underutilized, there is no significant effect on performance. This is similar to DVFS low voltage states, but with one important difference. While PCPG decreases the system throughput by reducing the number of active cores, single-thread performance is unaffected as the remaining core(s) can run at full frequency.

3.4.2 Dynamic Policies

While static policies can provide good power savings, PCPG, like DVFS, performs best when controlled dynamically. This is particularly important for workloads with high variability or systems that spend a significant percentage of their time at low utilization [10]. There are several options in constructing a dynamic

control scheme: the policy can be reactive to system utilization or may use predictive models [132]; the policy may utilize profiling information or compiler feedback [170, 173]; the policy may use an objective function that weighs performance, power, and energy in different ways; and so forth. The policy designer must carefully take into account the latency and energy cost of power gating cores in order to reach a stable control scheme. The overall power and energy efficiency depend primarily on whether the savings in leakage from the gated cores outweighs any increase in execution time or in dynamic power in the remaining cores that handle the consolidated workload of the system.

Dynamic schemes for DVFS exploit memory bound workloads in order to reduce power without compromising performance. Using a lower frequency while the chip is waiting for memory to fulfill requests does not slow down applications. Similarly, PCPG can exploit I/O bound workloads, such as database servers, in order to reduce leakage power without compromising performance. Consolidating on to a single core multiple processes that mostly wait for I/O provides power saving without affecting system throughput. On the other hand, consolidating cache intensive workloads on a single core can lead to increased cache interference that increases the execution time of each application, even if the processor core is mostly idle.

In Section 3.6, we evaluate PCPG with a simple dynamic algorithm. Our scheme is reactive and uses the CPU utilization as measured by the OS to determine the number of cores to power gate. A daemon polls CPU utilization (`/proc/stat`) at the rate of 20Hz by summing the time spent not idling or waiting for I/O requests. A simple high/low watermark scheme is used to determine whether to power gate an additional core or re-enable one. When the average utilization increases above the mark of $(active\_cores - 0.2)$ an extra core is enabled. When the average utilization decreases below $(active\_cores - 1.4)$, a core is power gated. We always have at least one core on in order to implement the dynamic management algorithm in software. We leave for future work the evaluation of more advanced schemes similar to those proposed for DVFS.

3.4.3 Interaction with Existing Voltage Scaling Techniques

PCPG essentially provides an additional set of fine-grain C-states that have part of the chip in deep sleep and the rest fully functional. A power manager can progressively move between C-states with higher savings and higher latencies in the following order: *Idle*, when executing the HLT (halt) instruction; *Stop Clock*, where the clock tree is gated; **PCPG**, where the core is shut down. Once no code is available to run on the chip, the manager can transition into package-level C-states: *Deep Sleep*, where the shared cache is turned off; *Package-level Soft Off*, where the package is finally shut down (the external voltage regulator is powered off).

On the other hand, the efficient combination of PCPG and DVFS (P-states) is subtle. Suppose that an idle system (lowest P-state, minimum number of cores active) is presented with a mild system load that it cannot handle in its reduced power state. If it is to raise the P-state, static power will increase slightly and dynamic power consumption will increase moderately. If it is to instead enable a gated core, static power will increase moderately and dynamic power may remain unchanged. The most energy efficient choice depends on the ranges of change to static and dynamic power. In spite of this subtlety, we choose to increase the P-state before a core is enabled by our dynamic PCPG manager, due to the long latency of enabling a core.

3.5 Methodology

We evaluated the performance and power impact of PCPG using a 2.5 GHz AMD Phenom X4 9850 system. This 65nm chip has 4 cores, each with a private 512KB L2 cache, a shared 2MB L3 cache, and an on-chip Northbridge. Our performance results are taken directly from this system, while our power results are a hybrid of real power measurements and an estimated reduction in leakage power for when we gate cores.

3.5.1 Workloads

We use the recently released SPECpower_ssj2008 benchmark [47] as well as a set of eight system utilization traces from various systems in real-world deployments. SPECpower_ssj2008 is an interesting benchmark for this study as it is designed to impart partial load on a system, rather than peak load. It is similar to SPECjbb2005, except it scales its average throughput (in ops/sec) from peak utilization down to idle in 10% steps. We use this workload to reason about the ability to consolidate non-peak workloads onto fewer numbers of cores.

Our traces are comprised of samples of CPU utilization taken once per second in a variety of commercial settings, including highly utilized SAP servers, web servers with varied utilization, and enterprise desktop machines. We selected 8 traces, each 120 seconds in length, to use as loads to place on our test system through a load generator that we developed. As will become evident, publicly available load generators, such as gamut [113], are inadequate for our study. The load generator takes the utilization samples from a trace and assigns work to several worker threads, which alternate at the rate of 100 Hz between executing dummy loops and sleeping. To promote fine-grained variance in the workload, the sampled utilization is used as a probability for whether or not to sleep at each time step, so that average utilization approaches the sampled utilization over one second time quanta. Work is distributed non-uniformly across threads, which we have found significantly improves the load generator's ability to replicate the power profile measured from real workloads.

The trace generator is *work conserving*. Every run of a trace will execute the same number of instructions, regardless of its variance in performance. A trace sample of $x\%$ utilization is interpreted as a finite quantity of work (i.e. # of instructions) rather than simply a utilization to match at a given point in time. If the load generator is unable to issue all of its work during a one-second time window, it records the left-over work as a deficit, which it will attempt to pay-down immediately in the next time window. Each time the generator retires a deficit, it records in a histogram the amount of time it took to retire that work. If a deficit remains following the end of a trace run, the load generator continues to run until the entire deficit is cleared. We validated our trace methodology by recording utilization traces of real workloads, replaying them, and then comparing the power consumption of the real workload to the power consumption of the load generator. We find mean error of -7 Watts and mean percent error of -3.4% for our experiments.

3.5.2 Performance Model

To study the performance impact of gated cores, we use Linux 2.6.24's built-in CPU "hotplug" support and measure actual performance of the system. Originally intended for high-end server systems with hardware support to remove and install CPU modules into servers without interruption, the hotplug support mimics precisely the behavior necessary to model core gating. Specifically, when a core is hot-unplugged, pending interrupts are serviced, running processes are migrated away from it, incoming interrupts are routed to other cores, its caches are flushed, and the core enters a low-power idle state using a HALT instruction. Likewise, when a core is hotplugged, it receives an inter-processor-interrupt directing it to an initialization vector, it re-enables local interrupts, the kernel's multiprocessor processor scheduler adjusts to the availability of the new core, and resumes executing processes. With the exception of activating the circuit to switch a core's associated power gate, the code path executed by the kernel when hotplugging cores should closely match a real PCPG implementation.

3.5.3 Hybrid Power Model

Our power model is a hybrid of real power measurements, a model of the power consumption of our power gates, and estimates of leakage power for the Phenom X4 9850. Starting with the physical power measurement, we incorporate the inefficiency of our power gates as well as the discrete energy cost each time they are switched, and subtract the leakage power of any cores that are sleeping at that moment. Energy consumption is calculated through discrete integration of our power estimates.

Measurements of CPU power consumption are made by measuring the voltage drop (V_{sr}) across a 0.00258Ω sense resistor placed in series with the CPU's private +12V power rail from the computer's power supply. The measurements are made by a Measurement Computing PC-CARD-DAS16/16-AO 16-bit 16-channel analog to digital converter, and are sampled at 4 Hz. Simultaneously with the voltage drop across the sense resistors, a voltage measurement is taken of the +12V rail (V_{12}). From these two measurements, the power consumption on the +12V rail is calculated as $P_{12} = V_{sr}/0.00258\Omega \times V_{12}$. This power consumption figure includes losses introduced by the CPU's VRM (voltage regulator module), whose efficiency we estimate at 80%. Our measurements were validated by comparing our estimated CPU power consumption with the power consumption measured at the wall socket by a Brand Electronics 1850-20 power meter.

We have estimated leakage power based on measurements of our system while at idle. Our assumption that leakage power mostly comprises the total power consumption of the Phenom X4 9850 at idle is derived from the results plotted in Figure 3.4, which shows power consumption of the chip at idle as cores are individually transitioned from the 2.5 GHz P-state to the 1.25 GHz P-state (note that the Phenom X4 9850 has a private PLL for each core). The lack of substantive change in power consumption as cores switch at half their previous frequency indicates that the chip is consuming very little dynamic power. This is likely due to the extensive use of clock gating. On the other hand, the large drop in power consumption when the CPU voltage driven by the shared VRM drops from 1.33V to 1.06V (since all cores are in the P1 P-state) is

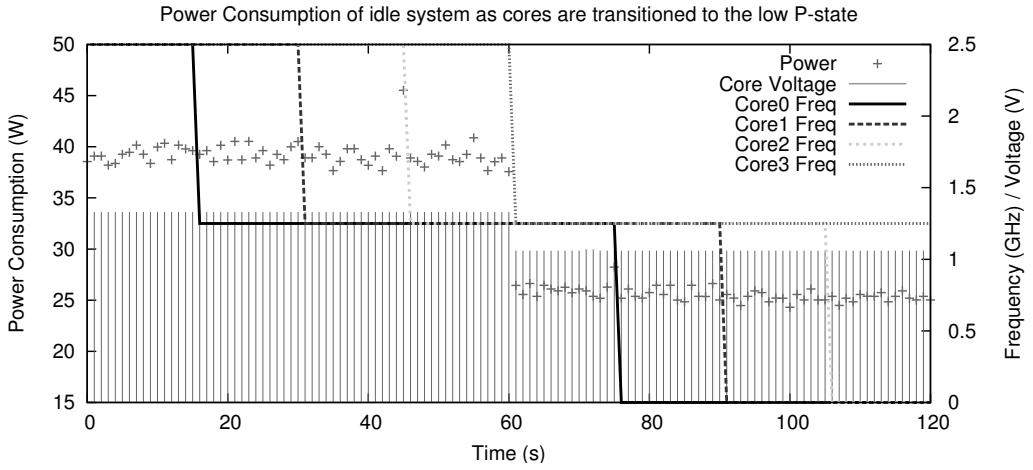


Figure 3.4: Power measurements of the system at idle as cores are transitioned individually from the P0 P-state to P1. The points show power consumption and vertical lines show operating voltage. The other lines indicate what frequency each core is operating at. The absence of significant change in power consumption as cores are individually set to a lower frequency suggests that they draw very little dynamic power at idle.

consistent with the nature of leakage current.

We assume that the 4 cores, the integrated Northbridge, and the shared L3 cache are the most significant contributors to leakage on the Phenom X4 9850. We additionally assume, based on their area similarity, that the leakage of the Northbridge is equal to that of one of the cores. To account for the L3 cache's leakage, we derived leakage figures for a 2MB cache matching the characteristics of the Phenom's cache using CACTI 5.3 [155], which came out to 2.7 Watts at the maximum P-state and 2.2 Watts at the minimum P-state.

All together, we calculate the per-core leakage in the P0 P-state as 7.13 Watts, and the leakage in the P1 P-state as 4.45 Watts. Power consumption results in Section 3.6 for PCPG configurations are post-processed to account for the amount of leakage eliminated when cores are gated.

3.5.4 Dynamic Power Management Daemon

We implement a dynamic PCPG manager as a userspace daemon. The daemon polls processor utilization (`/proc/stat`) at the rate of 20 Hz by summing the time spent not idling or waiting for I/O. This utilization factor is averaged over a one second sliding window. The decision to enable or disable a core is based on the simple high watermark/low watermark control algorithm discussed in Section 3.4.2. In general, the maximum utilization possible on a system is limited by the number of active cores. This affects the rate at which our daemon responds to sudden spikes in utilization; the daemon must first turn on a core and sample utilization for a short period of time before it can determine that it should enable yet another core. As stated, we enable and disable cores by using Linux 2.6.24's CPU hotplug support. Core x can be disabled by writing "0" to `/sys/devices/system/cpu/cpu`x`/online`, and re-enabled by writing "1".

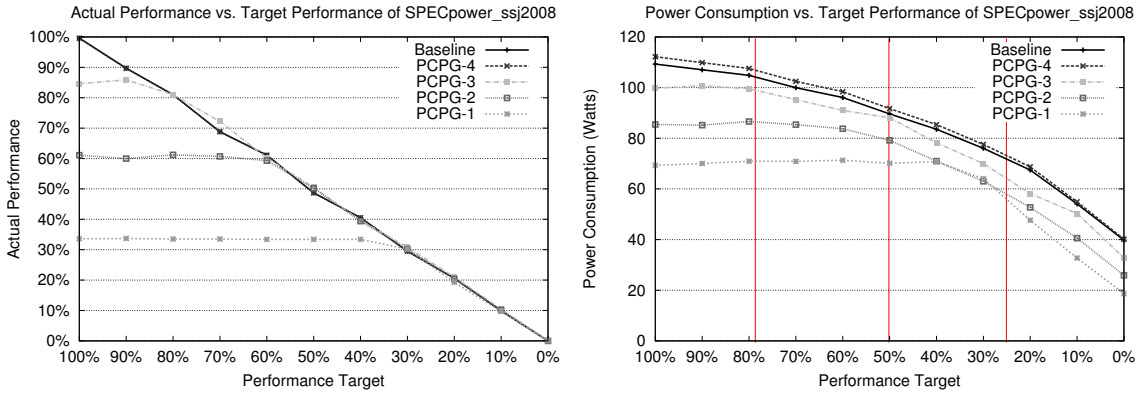


Figure 3.5: Actual performance vs. target performance and power consumption vs. target performance for SPECpower_ssj2008. Static PCPG settings are labeled "PCPG-x", where x is the number of active cores. Vertical lines correspond to intercepts between target performance and actual performance.

For comparisons with DVFS in Section 3.6, we use Linux 2.6.24's "ondemand" DVFS governor.

3.6 Evaluation

We evaluate PCPG in two steps, first with with SPECpower_ssj2008 to characterize its efficacy, and then by replaying utilization traces to measure typical energy savings and performance impact.

3.6.1 Characterization of Per-core Power Gating

Static Per-core Power Gating

SPECpower_ssj2008 allows us to explore the potential of PCPG at different levels of CPU utilization. Figure 3.5 shows performance (left) and power consumption (right) of static PCPG configurations compared to the baseline system (no PCPG, maximum P-state). Performance is presented as a comparison between a "performance target" that the SPECpower_ssj2008 is attempting to achieve and the actual performance that the benchmark exhibits. 100% performance was calibrated as 57,573 ops/s on the baseline system. Power is presented as the average power consumption during the execution of each performance target.

The performance results serve as a rough validation of our performance measurement methodology. The measurements for configurations with fewer cores plateau at lower performance levels. For example, while PCPG-4 performs identically to the baseline, PCPG-1 (3 cores gated off) never crests over 34% performance. At performance targets below each plateau, each PCPG configuration performs as well as those above it (i.e. all configurations achieve 20% performance during the 20% target run). The non-linearity in performance degradation is due to the memory intensive nature of the benchmark. For processor-intensive microbenchmarks, we observe perfect linear scaling in performance as cores are gated off.

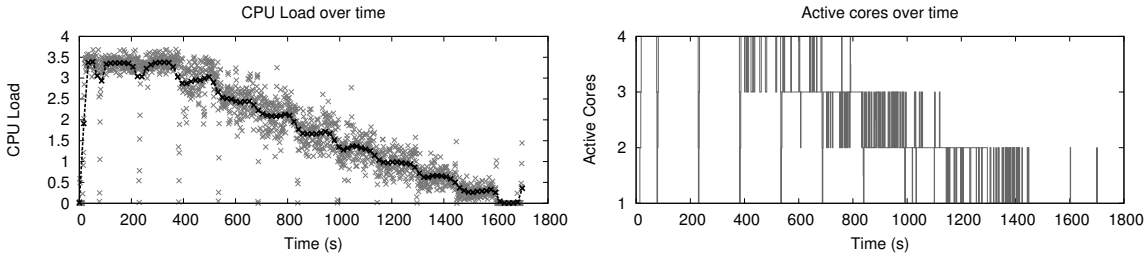


Figure 3.6: A trace of CPU Load and active cores when running SPECpower_ssj2008 with dynamic power gating enabled. The gating of cores by the dynamic PCPG manager due to limited utilization is clearly evident. CPU load is presented as discrete samples and as a bezier smoothing.

The power results in Figure 3.5 characterize power consumption at different levels of load. Vertical lines correspond to intercepts between target performance and actual performance. For example, the power measurements to the left of the vertical line at 34% target performance are not significant for PCPG-1, since this configuration cannot achieve the target performance in this region. Note that PCPG-4 consumes slightly more power than the baseline at all performance targets due to the resistive load of the power gates. Disregarding measurements where actual performance is less than target performance, all other PCPG configurations consume less power than the baseline—between 21.1 Watts (53% reduction) and 5.0 Watts (5% reduction) fewer at 0% and 80% performance, respectively. Overall, this experiment shows the potential power consumption savings of PCPG, and establishes a trade-off between power savings and performance.

Dynamic Per-core Power Gating

We now evaluate the potential of PCPG with dynamic control. Figure 3.6 shows a trace of measured CPU Load during a run of SPECpower_ssj2008 (left) correlated with how many cores are active at any given time (right). A CPU Load of 4 corresponds to 4 cores working all of the time. The variability seen in CPU Load is a design goal of SPECpower_ssj2008 [47], the observation being that realistic server workloads exhibit burstiness and volatility, or even self-similarity [20]. As a consequence, the dynamic PCPG manager is often toggling cores on and off during measurement runs.

Figure 3.7 presents performance and power consumption measurements for the baseline system, two static PCPG configurations (PCPG-1 and PCPG-4) and the dynamic configuration (PCPG). With respect to performance, the PCPG configuration performs identically to the baseline with the exception of some slowdown at the 100% performance target. This is due to the dynamic PCPG manager slowly responding to the sudden increase in CPU utilization at the beginning of this performance measurement. After this ramp-up period, the PCPG manager constantly decides to keep 4 cores enabled.

The power measurements in Figure 3.7 succinctly demonstrate the design goal of per-core power gating: a smooth transition from maximum power consumption at peak performance to minimum achievable power consumption at idle. The PCPG-1 configuration is plotted as a reference for the minimum idle power

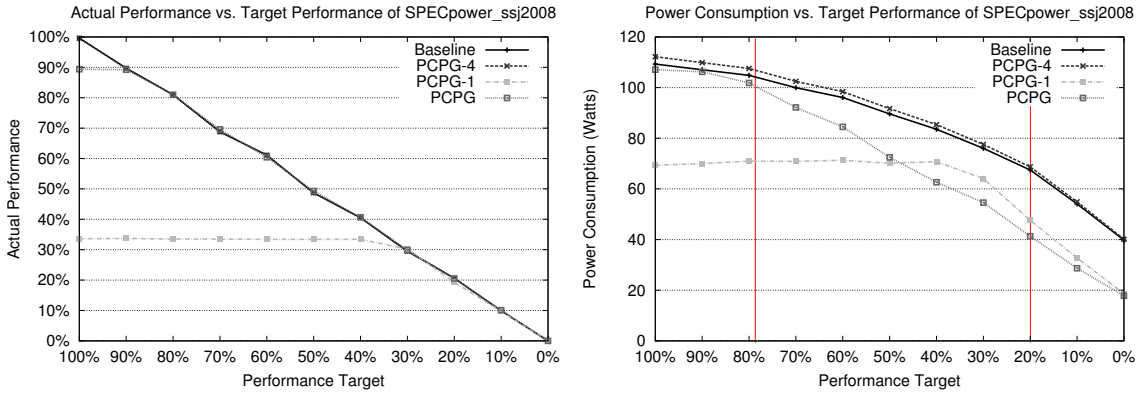


Figure 3.7: Actual performance vs. target performance and power consumption vs. target performance for SPECpower_ssj2008. The Dynamic PCPG configuration is labeled "PCPG".

consumption achievable by our system. An unexpected result is that the dynamic PCPG system sometimes achieves lower power consumption than the static PCPG-1 setting. During these measurements, the *dynamic* power consumption of the system with only 1 active core was higher than any other configuration.

For further exhibition of this interesting result, Figure 3.8 presents a power consumption trace of dynamic PCPG and PCPG-1 while running SPECpower_ssj2008. At the 40% performance target (indicated by vertical lines), it can be seen that the dynamic PCPG configuration is often dropping to idle power consumption while the PCPG-1 configuration is pegged at full power consumption. In short, the PCPG-1 configuration is so pressured with work that it never idles, where as the other configurations spread the same work across multiple cores, and occasionally the idle periods of each core coincides with the others. These idle periods allow the CPU to employ aggressive clock gating.

Combining DVFS and PCPG

We omit the presentation of a detailed characterization of SPECpower_ssj2008 when DVFS and PCPG are combined, but present quantitative results for the trace-based evaluation in the next section. In summary, we find that the combination leads to a further drop in performance at the 100% performance target, but otherwise matches the baseline system. In terms of power consumption, PCPG+DVFS achieves lower power consumption at idle than PCPG alone (11.8 Watts vs. 17.8 Watts) and moderately reduced power consumption for performance targets up to 40%.

3.6.2 Evaluation of PCPG with Realistic Utilization Traces

We now evaluate PCPG with the traces and the methodology discussed in Section 3.5. Unlike SPECpower_ssj2008, which defines synthetic performance targets, the traces allow us to look at actual processor load in real-world scenarios.

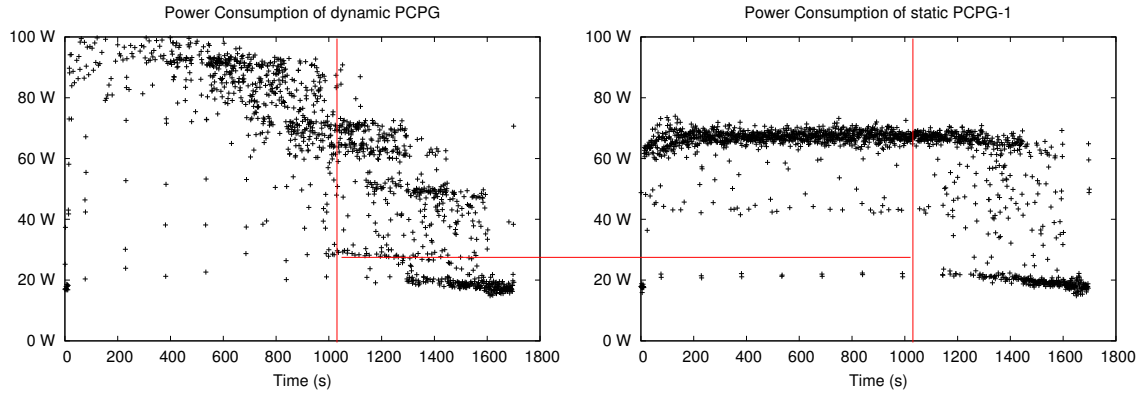


Figure 3.8: Power consumption trace of dynamic PCPG and the static PCPG-1 setting running SPECpower_ssj2008. Vertical lines show where the 40% performance run is occuring. The dynamic PCPG system occasionally idles, bringing its average power consumption down. The PCPG-1 system is pegged at maximum utilization and consumes more power.

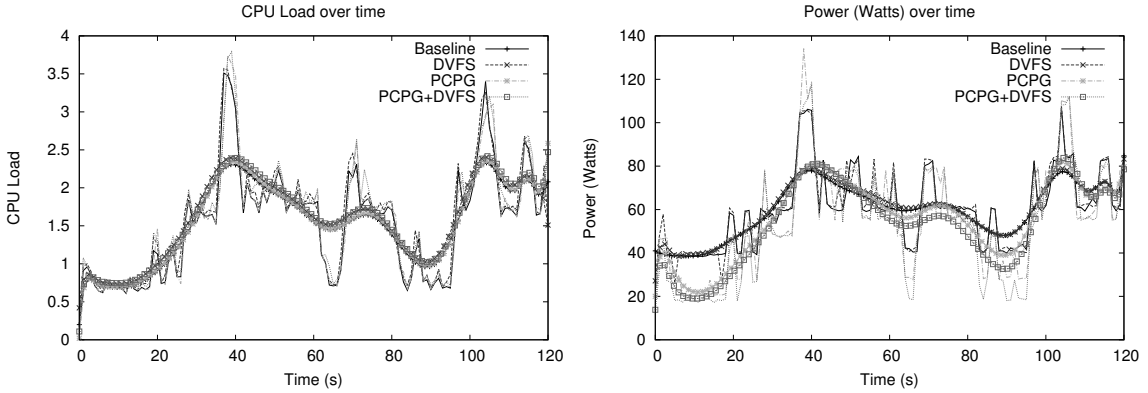


Figure 3.9: CPU load and power consumption as the SAP05 trace is played. Thin lines indicate instantaneous measurements, while thicker lines show a bezier smoothing.

Figure 3.9 is a representative sample of what the duration of one of our trace runs looks like, in this case, the SAP05 trace. The left graph shows measured CPU load of the baseline system (no PCPG), the baseline system with DVFS, the dynamic PCPG system, and the combination of DVFS and PCPG (PCPG+DVFS). A CPU load of 4 corresponds to 4 cores working all of the time. The right graph shows power consumption over the trace for each configuration. Naturally, there is significant correlation between CPU load and power consumption. Overall, each configuration exhibits uniform CPU Load, but PCPG configurations consume less power at low utilization.

While the differences in CPU load at first glance seem insignificant, they do show interesting behavior. Notably, the PCPG configurations tend to peak in load slightly after the non-PCPG configurations (see Figure 3.9 at 40s). These slight delays correlate to lag in the PCPG manager increasing the number of active

Trace	Max Load	Min Load	Avg Load	Technique	Avg Power (W)	High Δ Power (W)	Low Δ Power (W)	Avg Δ Power (W)	Energy (J)	% Δ Energy	% delayed 1s	% delayed 2s	% delayed >2s	% 4 cores	% 3 cores	% 2 cores	% 1 core
SAP05	3.51	0.20	1.53	Base	59.66	-	-	-	7099	-	0.18	0.00	0.00	-	-	-	-
				DVFS	60.45	3.05	-1.50	0.79	7194	1.33	2.60	0.00	0.00	-	-	-	-
				PCPG	56.02	14.70	-16.68	-3.63	6667	-6.09	3.58	0.00	0.00	5.0	47.1	35.5	12.4
				Both	53.41	11.71	-21.60	-7.25	6356	-10.47	8.54	0.00	0.00	5.0	48.7	45.4	0.8
SAP11	2.71	0.55	1.26	Base	61.29	-	-	-	7294	-	0.07	0.00	0.00	-	-	-	-
				DVFS	61.83	4.37	-2.92	0.54	7358	0.87	2.08	0.00	0.00	-	-	-	-
				PCPG	49.13	-3.28	-20.28	-12.17	5846	-19.85	2.94	1.54	2.98	1.7	15.7	71.9	10.7
				Both	46.62	-4.90	-25.76	-14.89	5548	-23.94	4.60	2.28	1.71	1.7	15.0	80.0	3.3
SAP12	3.45	0.89	2.22	Base	82.84	-	-	-	9858	-	0.05	0.00	0.00	-	-	-	-
				DVFS	91.66	23.38	-7.10	8.82	10907	10.64	4.63	0.00	0.00	-	-	-	-
				PCPG	82.85	12.63	-11.27	0.01	9859	0.01	4.72	1.08	1.72	21.5	66.1	11.6	0.8
				Both	79.99	6.10	-12.30	-3.38	9519	-3.44	10.20	1.08	0.37	24.2	65.8	9.2	0.8
PHARMA04	2.84	0.00	0.42	Base	47.36	-	-	-	5636	-	0.22	0.00	0.00	-	-	-	-
				DVFS	37.42	-0.53	-17.97	-9.95	4452	-21.00	2.04	0.00	0.00	-	-	-	-
				PCPG	29.13	-9.20	-26.92	-18.23	3466	-38.50	9.93	3.39	0.00	0.0	9.1	7.4	83.5
				Both	25.49	-2.09	-31.20	-21.87	3034	-46.18	7.26	0.00	0.00	0.8	9.9	10.7	78.5
HCOM10	1.77	0.10	0.51	Base	45.30	-	-	-	5391	-	0.02	0.00	0.00	-	-	-	-
				DVFS	31.97	-0.07	-21.37	-13.33	3804	-29.43	0.92	0.00	0.00	-	-	-	-
				PCPG	25.90	-9.16	-27.77	-19.41	3082	-42.84	5.43	0.00	0.00	0.0	0.0	19.8	80.2
				Both	18.71	-13.80	-34.72	-26.81	2227	-58.70	5.69	0.00	0.00	0.0	0.8	25.0	74.2
HCOM19	2.79	0.00	1.45	Base	68.98	-	-	-	8208	-	0.06	0.00	0.00	-	-	-	-
				DVFS	58.40	-0.06	-23.44	-10.58	6949	-15.34	3.46	0.00	0.00	-	-	-	-
				PCPG	50.87	-4.02	-30.85	-18.10	6054	-26.25	5.66	2.00	1.79	5.0	43.0	2.5	49.6
				Both	49.61	0.82	-36.06	-20.04	5904	-28.07	8.26	2.13	1.94	15.0	32.5	5.0	47.5
ECOM3	1.78	0.01	0.84	Base	47.84	-	-	-	5693	-	0.03	0.00	0.00	-	-	-	-
				DVFS	48.30	18.17	-12.83	0.46	5748	0.97	1.27	0.00	0.00	-	-	-	-
				PCPG	32.09	-8.51	-25.99	-15.75	3818	-32.93	2.88	0.00	0.00	0.0	0.8	68.6	30.6
				Both	29.84	-10.13	-25.95	-18.00	3551	-37.62	3.05	0.00	0.00	0.0	0.8	78.5	20.7
DESKTOP	0.87	0.00	0.14	Base	40.62	-	-	-	4833	-	0.00	0.00	0.00	-	-	-	-
				DVFS	27.56	-4.49	-16.21	-13.05	3280	-32.14	0.19	0.00	0.00	-	-	-	-
				PCPG	19.12	-19.81	-23.13	-21.49	2276	-52.92	0.69	0.00	0.00	0.0	0.0	0.8	99.2
				Both	14.33	-21.72	-28.79	-26.29	1705	-64.73	1.20	0.00	0.00	0.0	0.0	1.7	98.3

Table 3.2: Results for a variety of utilization traces, including commercial application servers (SAP05 SAP11, SAP12, PHARMA04), web servers (HCOM10, HCOM19, ECOM3), and a desktop machine (DESKTOP). A few positive (lighter green) and negative (darker red) results are highlighted. The presented metrics can be divided into 4 categories: power, energy, delay, and PCPG state. For power, we present average power consumed during the trace, the highest power consumption above and the lowest power consumption below the baseline configuration (90^{th} percentile), and average power difference. For energy, we present the absolute consumption in Joules (which, since each trace runs for the same amount of time, is proportional to average power consumption), and the percent change in energy consumption. For delay, we show the percent of total work delayed for one second, for two seconds, and for more than two seconds. For the PCPG state, we present the percent of time spent with 4, 3, 2, or 1 core active.

cores, as it makes decisions based on 1s samples of average CPU load. The brief delay also results in a backlog of work, during which the dynamic power consumption goes up.

Other than the aforementioned spikes in power consumption following the accumulation of a work deficit, power consumption for the PCPG configurations stay generally at or below the power consumption of the non-PCPG configurations. During periods of relative inactivity (load < 1), significant improvements of up to 20W are observed. These improvements result in net energy savings.

Table 3.2 shows the evaluation results for a variety of traces, including commercial application servers, web servers, and a desktop machine. See its caption for a description of each column. As none of these traces exhibited 100% load for their entire duration, opportunities were generally found to gate cores and reduce power consumption. These energy consumption savings range from 3.4% for SAP12, a trace with

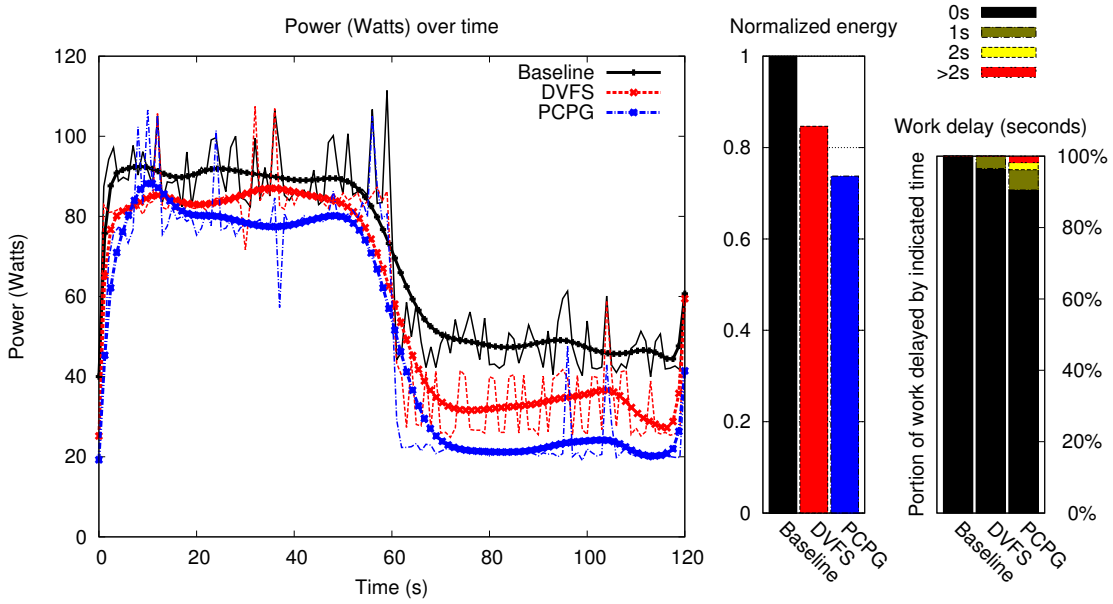


Figure 3.10: Summary of the HCOM19 trace. Power, normalized energy consumption, and work delay history are presented. Power savings are most evident following a drop in CPU load that occurs at 60s.

substantial utilization, to 64.73% for DESKTOP, a trace taken from a mostly idle desktop system. While nominal power consumption of the PCPG configurations is well below the baseline system, there are some anomalous measurements. For example, the High Δ Power (the highest amount by which a configuration overshoots the baseline system at any correlated point during the trace) for SAP05 and SAP12 show that the PCPG configuration at some point consumes 14.7 Watts and 12.63 Watts, respectively, over the baseline. These readings correspond to periods immediately following low throughput, where the load generator is attempting to pay-down an accumulated work deficit.

In nearly all cases, the PCPG configurations come with some delay in work execution, ranging from 0.69% work delayed one second or more for the DESKTOP trace to 12.33% for the HCOM19 webserver trace. The amount of this delay correlates more strongly with the maximum load during a trace rather than the average load. For example, PHARMA04 exhibits significant work delay, yet the average load during that trace is only 0.42. Note that these work delays do not correspond to discrete "slow-downs" or increases in execution time. Indeed, work deficits notwithstanding, all of the traces complete in the same amount of time. Rather, these work delays can best be described as quality-of-service (QoS) metrics, and can mainly be used for comparison between different configurations. Nevertheless, they serve as a suitable lower-bound on performance.

As a visual reference for the data presented in Table 3.2, Figure 3.10 shows power consumption over time, as well as a summary of energy consumption and performance degradation of HCOM19, a trace from a commercial webserver. For this trace, before the drop in load at 60s, there is only minimal difference in

power consumption between all four configurations. After the drop, significant differences can be observed as the PCPG configurations gate cores. The PCPG configurations take advantage of this period to accrue significant energy savings (28%). The PCPG configurations exhibit several brief losses of performance during the beginning of the trace, but eventually reach equilibrium. This trace incurred the most severe loss of performance that we observed (12.33% of work delayed by one second or more).

3.7 Related Work

Most of the existing work on power gating for high-end processors deals with low-leakage memory design. The Intel Dual Core Xeon presented in [139] uses sleep transistors to implement low-power modes in the L3 cache. Similarly, Zhang *et al.* present a SRAM design that employs power gating to reduce the leakage power [178].

Tschanz *et al.* propose several techniques to design a low-leakage ALU, including power gating [157]. Unlike our work, they adopt a different design strategy by redesigning a basic processor block and providing fine-grain control of the leakage. Our solution, on the other hand, permits the reuse of processor cores, leaving unmodified their logic and layout. Power gating has been demonstrated for low-power mobile processors [53, 120]. These devices feature current densities that are order of magnitude less than high-end processors, making sleep transistors extremely small.

Significant effort has focused on algorithms for managing P and C states. In particular, many papers have been published on DVFS starting from initial efforts by Transmeta [37]. Some of the studies most related to our work are as follows. Donald and Martonosi explore DVFS for dynamic thermal management in multi-core processors [30]. Canturk *et al.* explore the design space of per-core DVFS, assuming that this comes at a negligible hardware cost [68]. Li *et al.* [88] present a simulation study of per-core DVFS and sleep states by leveraging an embedded microcontroller. Although the latter includes sleep states in its evaluation, it does not analyze the many hardware/software implications of implementing per-core power gating, but assumes it as a given technology. All these studies target high-activity workloads (SPEC and HPC benchmarks), optimizing towards a different design point with respect to us. We evaluate PCPG in the context of variable and low-utilization workloads.

In the context of datacenters, most research has concentrated on leveraging existing power management hooks, like DVFS and migration, or even turning off individual servers. Fan *et al.* [34] explore how DVFS can be used in the Google infrastructure. Ramos and Bianchini present a software infrastructure that uses load redistribution and DVFS to respond thermal emergencies [132]. Raghavendra *et al.* introduce a global management solution that coordinates individual power management approaches [131]. Tolia *et al.* [156] suggests how ensemble-level power proportionality could be achieved by turning off servers in an ensemble. Meisner *et al.* [103] focuses on idle power of servers, suggesting new architectural features to enable efficient on/off switching of a server.

This work also differentiates from the previous studies in its experimental methodology. Similarly to us,

but in a different context, Wu *et al.* evaluated their dynamic compilation-directed DVFS by performing real measurements [170]. Lloyd and John [13] propose an experimental analysis of dynamic power management (DVFS) for an AMD 4-core processor on desktop-oriented benchmarks.

As far as we know, our work is the first one at evaluating per-core deep sleep states and showing its advantages for commercial applications.

3.7.1 Intel Nehalem and AMD Bobcat

Since the original manuscript describing this work was written, per-core power gating has been implemented for high-performance, general-purpose processors and shipped by both Intel (starting with its Nehalem architecture) and AMD (most recently in its Bobcat architecture). We now draw a few comparisons with our work and two existing implementations: Intel's Nehalem [138, 150] and AMD's Bobcat [83, 137]. Both expose per-core power gating as a "C6" C-state.

First, the circuit overheads associated with PCPG are far lower in real implementations than we estimated. AMD reports that power gates occupy only 1-3% of chip area on the Bobcat processor [137], as opposed to the 20% we predicted assuming a generic high-V_{th} process. With area overhead so low, a MOSFET interposer like we propose is not likely necessary. Kosonocky also reports that AMD sizes their power gating network to achieve approximately 1% voltage drop [83]. Thus, the power and frequency overheads we assumed in our study are overly pessimistic.

Policy-wise, both AMD and Intel have adopted an approach to PCPG which requires little operating system intervention. Rather, the operating system simply requests the C6 C-state when it calls `MWAIT` as part of its idle-loop, the same as was done for other low-power C-states: C1, C1E, C3, etc. A "package control unit" (PCU) then makes the final decision regarding which C-state to enter, and takes care of saving and restoring all architecturally-visible state [138]. The PCU can choose not to enter the C6 state if it predicts that the core will need to wake up within a short period of time (such that the latency to exit C6 will be significantly noticeable), or when entering and exiting C6 will consume more energy than simply staying put. This strategy largely hides the latency of entering and exiting a power-gated state from the end-user, and has greatly simplified the adoption of PCPG by operating system developers. Unfortunately, the PCU's second-guessing can mean that C6 is used far less often than it otherwise could be. For instance, Meisner et al. find that "C6 ... does not provide significant power reduction." [104] in their study of power-management for Google workloads.

As a demonstration, Figure 3.11 shows how often a Nehalem-architecture core is in the C6 state when exposed to a modest memcached workload [36]. The shocking result is that at 3% load, the core almost never actually enters the C6 state, even though the CPU is idle (not in the C0 state) more than 80% of the time. Memcached is a particularly antagonistic workload for Nehalem, as its requests can be handled in 10s of microseconds; even at very low load, requests are arriving at several kHz. However, it is not far removed from several workloads at Google, so it is not surprising that Meisner et al. found little benefit from C6.

Note that the PCPG policy presented in this chapter does not suffer from this problem. The key distinction

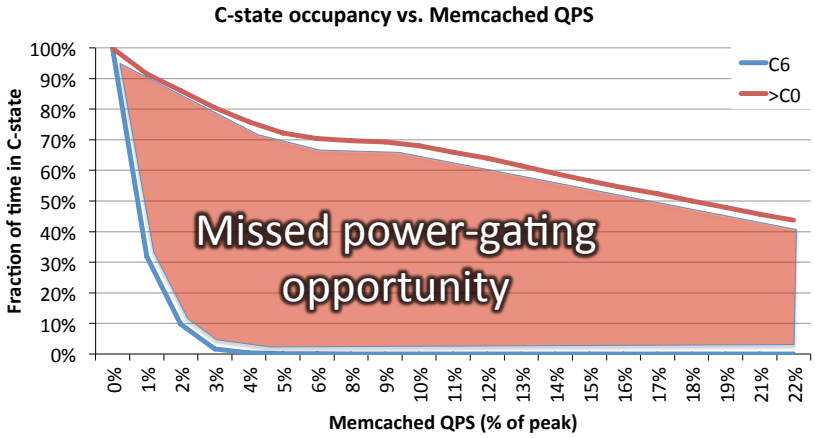


Figure 3.11: Even at exceptionally low load, Nehalem rarely uses its per-core power gating capability when serving memcached requests.

is that we actively consolidate a CMP's workload onto a smaller number of cores, rather than reactively wait for suitably long periods of idleness. By forcing the operating system to migrate workloads, we guarantee that sleeping cores will be idle for a long period of time.

The one caveat is that by consolidating workloads to fewer cores, we are essentially forcing them to timeshare more aggressively. This may introduce quality-of-service problems for some workloads that did not exist before. We investigate in-depth the maintenance of good quality-of-service on highly utilized cores in Chapter 5.

3.8 Conclusions

This chapter has made the case for per-core power gating (PCPG) as an effective technique to reduce leakage power in multi-core systems. PCPG exploits the fact that most server and client systems spend significant periods of time under moderate utilization, during which load can be aggregated on fewer cores. PCPG is orthogonal to existing power management techniques, such as DVFS, which targets dynamic power reduction at high utilization, and deep sleep states, which target leakage power consumption when a chip is idle.

We presented two practical alternatives to implement the sleep transistors necessary for power gating and the hardware and software components needed to implement PCPG. Our evaluation demonstrates that, under dynamic control, PCPG can lead to significant savings in static power and energy consumption, without significant performance reduction, for a wide range of workloads with periods of moderate utilization. We have also shown that PCPG is complementary to chip-wide DVFS techniques. Our results establish that PCPG is a practical and effective technique for static power management and that there is exciting future work on PCPG control algorithms, on techniques for combining PCPG with existing voltage scaling, and on evaluation of detailed implementations.

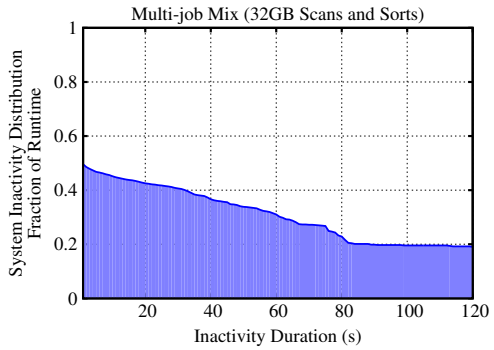
Chapter 4

On the Energy (In)efficiency of Hadoop Clusters

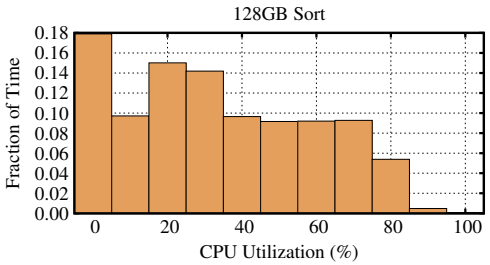
The previous two chapters have sought to improve server power-efficiency by proposing new hardware mechanisms to reduce main memory and CPU power consumption, respectively. While the previous chapter made progress towards addressing CPU static power consumption, it did little else for the rest of the server. To reduce the power consumption of the myriad other components of a modern server, we need a drastically different approach. Thus, we now shift the focus of this dissertation towards *server utilization*, and the role that utilization plays in determining datacenter power-efficiency.

As discussed in Section 1.4, server utilization is quite low in typical datacenters (between 6% and 12%), and servers are power-inefficient at low utilization. A seemingly obvious question might be, "Why don't they just turn off the servers they aren't using?" In this chapter, we begin to answer this question by looking at the role that contemporary distributed processing frameworks play in it.

Distributed processing frameworks, such as Yahoo!'s Hadoop [168] and Google's MapReduce [23], have been successful at harnessing expansive datacenter resources for large-scale data analysis. However, their effect on datacenter power-efficiency has not been scrutinized. Moreover, the filesystem component of these frameworks effectively precludes *scale-down* of clusters deploying these frameworks (i.e. operating at reduced capacity). This chapter presents our work on modifying Hadoop to allow scale-down of operational clusters. We find that running Hadoop clusters in fractional configurations can save between 9% and 50% of energy consumption, and that there is a trade-off between performance energy consumption. We also outline further research into the energy-efficiency of these frameworks.



(a) Distribution of the lengths of system inactivity periods across a cluster during a multi-job batch workload, comprised of several scans and sorts of 32GB of data. A value of .38 at $x = 40$ seconds means that 38% of the time, a node was idle for 40 seconds or longer.



(b) Histogram of CPU utilization across the cluster when sorting 128GB of data.

Figure 4.1: Opportunities for power-efficiency improvements in Hadoop workloads. Figure (a) shows that many opportunities to utilize low-power modes for entire servers are available. Figure (b) shows that opportunities exist to use fractional low-power modes within servers, and perhaps also that these servers are *imbalanced* for MapReduce workloads.

4.1 Introduction

A growing segment of datacenter workloads is managed with MapReduce-style frameworks, whether by privately managed instances of Yahoo!'s Hadoop [168], by Amazon's Elastic MapReduce[1], or ubiquitously at Google by their archetypal implementation [23]. Therefore, it is important to understand the power-efficiency of this emerging workload.

The power-efficiency of a cluster can be improved in two ways: by matching the number of active nodes to the current needs of the workload, placing the remaining nodes in low-power standby modes; by engineering the compute and storage features of each node to match its workload and avoid energy waste on oversized components. Unfortunately, MapReduce frameworks have many characteristics that complicate both options.

First, MapReduce frameworks implement a distributed data-store comprised of the disks in each node, which enables affordable storage for multi-petabyte datasets with good performance and reliability. Associating each node with such a large amount of state renders state-of-the-art techniques that manage the number of active nodes, such as VMWare's VMotion [118], impractical. Even idle nodes remain powered on to ensure data availability [10]. To illustrate the waste, Figure 4.1(a) depicts the distribution of the lengths of system inactivity periods across a cluster during a multi-job Hadoop workload, comprised of several scans and sorts of 32GB of data. We define inactivity as the absence of activity in all of the CPU, disk, and network. While significant periods of inactivity are observed, the need for data availability prohibits the shutting down of idle nodes.

[1] `http://aws.amazon.com/elasticmapreduce/`

Second, MapReduce frameworks are typically deployed on thousands of commodity nodes, such as low-cost 1U servers. The node configuration is a compromise between the compute/data-storage requirements of MapReduce, as well as the requirements of other workloads hosted on the same cluster (e.g., front-end web serving). This implies a mismatch between the hardware and any workload, leading to energy waste on idling components. For instance, Figure 4.1(b) shows that for an I/O limited workload, the CPU utilization is quite low and most of the energy consumed by the CPU is wasted.

Finally, given the (un)reliability of commodity hardware, MapReduce frameworks incorporate mechanisms to mitigate hardware and software failures and load imbalance [23]. Such mechanisms may negatively impact power-efficiency.

This chapter presents a first effort towards improving the power-efficiency of MapReduce frameworks like Hadoop. First, we argue that Hadoop has the global knowledge necessary to manage the transition of nodes to and from low-power modes. Hence, Hadoop should be, or cooperate with, an power controller for a cluster. Second, we show it is possible to recast the data layout and task distribution of Hadoop to enable significant portions of a cluster to be powered down while still fully operational. We report our initial findings pertaining to the performance and power-efficiency trade-offs of such techniques. Our results show that energy can be conserved at the expense of performance, such that there is a trade-off between the two. Finally, we establish a research agenda for a broad power-efficient Hadoop, detailing node architecture, data layout, availability, reliability, scheduling, and applications.

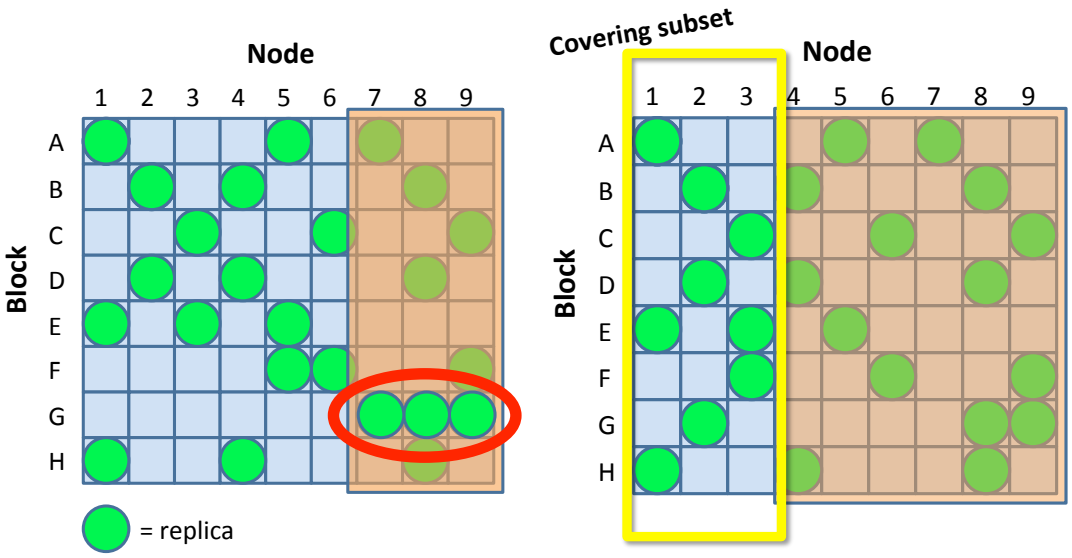
4.2 Improving Hadoop's Energy-Efficiency

Our first efforts focus on the impact of data layout. While this discussion is specifically targeted at Hadoop, many of our observations apply to the Google Filesystem [42] and similar cluster filesystems [145].

4.2.1 Data Layout Overview

Hadoop's filesystem (HDFS) spreads data across the disks of a cluster to take advantage of the aggregate I/O, and improve the data-locality of computation [15]. While beneficial in terms of performance, this design principle complicates power-management. With data distributed across all nodes, any node may be participating in the reading, writing, or computation of a data-block at any time. This makes it difficult to determine when it is safe to turn a node or component (e.g. disk) off. The utility of any particular node in making data available is opaque to both users and nodes themself, such that neither can make informed decisions regarding whether or not it is permissible to disable a node. For instance, while at some point in time a node may appear to be idle, it may house a data block that an application will request soon.

Tangentially, Hadoop must also handle the case of node failures, which can be frequent in large clusters. To address this problem, it implements a data-block replication and placement strategy to mitigate the effect of certain classes of common failures; namely, single-node failures and whole-rack failures. When data is stored in a HDFS, the user specifies a *block replication factor*. A replication factor of n instructs HDFS



(a) With Hadoop's default block replication policy, there is an exceptionally high probability that the failure of any three nodes leads to the unavailability of at least some data. In this case, taking nodes 7, 8, and 9 offline makes block G disappear.

(b) With our proposed "Covering Subset" replication policy, all nodes not in the covering subset can be taken offline without affecting data availability. This maximizes the opportunity to turn-off underutilized nodes.

Figure 4.2: Impact of block layout policy on data availability.

to ensure that n identical copies of any data-block are stored across a cluster (by default $n = 3$). Whilst replicating blocks, Hadoop maintains two invariants: (1) no two replicas of a data-block are stored on any one node, and (2) replicas of a data-block must be found on at least two racks. These two invariants directly address the two common failures modes cited above. Should a single node fail, it is guaranteed that any data that node stores will be found on another node. Should an entire rack fail (due to a switch, network cabling, or power delivery failure), it is guaranteed that any data stored within that rack will be found in some other rack. In the event of any failure, Hadoop proactively re-replicates any data-blocks found within the affected node(s) to ensure n replicas exist and to restore the aforementioned layout invariants.

4.2.2 A Replication Invariant for Energy-Efficiency

The fact that Hadoop maintains replicas of all data affords an opportunity to save energy on inactive nodes. That is, there is an expectation that if an inactive node is turned off, the data it stores will be found somewhere else on the cluster. However, this is only true up to a point. Should the n nodes that hold the n replicas of a single block be selected for deactivation, that piece of data is no longer available to the cluster. In fact, we have found through examination of our own Hadoop cluster that when configured as a single rack, removing *any* n nodes from the cluster (where n is the replication factor) will render some data unavailable, illustrated in Figure 4.2(a). Thus the largest number of nodes we could disable without impacting data availability is

$n - 1$, or merely two nodes when $n = 3$. While Hadoop's autonomous re-replication feature can, over time, allow additional nodes to be disabled, this comes with severe storage capacity and resource penalties (i.e. significant amounts of data must be transferred over the network and condensed onto the disks of the remaining nodes). Hadoop's rack-aware replication strategy mitigates this effect only moderately; at best a single rack can be disabled before data begins to become unavailable.

To address this short-fall in Hadoop's data-layout strategy, we propose a new invariant for use during block replication: at least one replica of a data-block must be stored in a subset of nodes we refer to as a *covering subset*. The premise behind a covering subset is that it contains a sufficient set of nodes to ensure the immediate availability of data, even were all nodes not in the covering subset to be disabled. An example is shown in Figure 4.2(b).

This invariant leaves the specific designation of the covering subset as a matter of policy. The purpose in establishing a covering subset and utilizing this storage invariant is so that large numbers of nodes can be gracefully removed from a cluster (i.e. turned off) without affecting the availability of data or interrupting the normal operation of the cluster; thus, it should be a minority portion of the cluster. On the other hand, it cannot be too small, or else it would limit storage capacity or even become an I/O bottleneck. As such, a covering subset would best be sized as a moderate fraction (10% to 30%) of a whole cluster, to balance these concerns.

Just as replication factors can be specified by users on a file by file basis, covering subsets should be established and specified for files by users (or cluster administrators). In large clusters (thousands of nodes), this allows covering subsets to be intelligently managed as current activity dictates, rather than as a compromise between several potentially active users or applications. Thus, if a particular user or application vacates a cluster for some period of time, the nodes of its associated covering subset can be turned off without affecting the availability of resident users and applications.

4.2.3 Implementation

We made the following changes to Hadoop 0.20.0 to incorporate the covering subset invariant. HDFS's `ReplicationTargetChooser` was modified to first allocate a replica on the local node generating the data, then allocate a replica in the covering subset, and finally allocate a replica in any node not in the first node's rack. We changed Hadoop's strategy in choosing "excess" replicas for invalidation (for instance, when temporarily faulty nodes rejoin a cluster) to refrain from deleting replicas from a covering subset unless there is at least one other replica in the subset. To prevent Hadoop from autonomously re-replicating blocks for nodes which have been intentionally disabled, we have added a hook to Hadoop's HDFS `NameNode` (i.e. master node) which behaves similarly to Hadoop's *decommissioning* of a live node. For each block stored by a node being disabled, our routine removes the node from Hadoop's internal list of storage locations for that block and stores it in a separate list of offline storage locations. In contrast to node decommissioning, replications are not scheduled for each block that goes offline. Furthermore, we adjust all queries regarding the number of live replicas of a block to account for offline replicas, so that extraneous activity does not

trigger superfluous block replications (e.g. should a separate node legitimately fail, Hadoop will globally evaluate how many replicas it should generate for each block on that failed node).

Disabled nodes can gracefully rejoin a cluster after either a short period of stand-by, or even after a complete operating system reboot, by sending a heartbeat or node registration request to the `NameNode`.

In addition to these core changes in HDFS's NameNode, we have modified Hadoop's HDFS client to detect when it is attempting to access a block for which no replicas are available, and to ask the NameNode to revive a node so that the block can be retrieved. This can mitigate instances where a node in the covering subset has failed. Upon receiving such a request, the NameNode refers to its list of offline block locations and selects a node to be woken. The NameNode has hooks to execute arbitrary commands to revive this node (for instance, issuing Wake-on-LAN or IPMI[2] power-management requests). We have also implemented a dynamic power manager to activate nodes whenever there is a backlog of Hadoop Map or Reduce tasks on the JobTracker, but do not evaluate it here.

Our changes to HDFS are largely mirrored in Hadoop MapReduce. We added RPCs to the JobTracker in order to enable and disable the scheduling of tasks to nodes. We stop sending compute tasks to a node in unison with HDFS being disabled on it. In the following experiments, nodes are disabled and enabled manually.

4.3 Evaluation

We evaluate our changes to Hadoop through experiments on a 36-node cluster coupled with an energy model based linearly on CPU utilization [135]. Each node is an HP ProLiant DL140G3 with 8 cores (2 sockets), 32GB of RAM, a gigabit NIC, and two disks. The nodes are connected by a 48-port HP ProCurve 2810-48G switch (48 Gbps bisection bandwidth). This cluster was also used for the results in Section 4.1. We made every effort to optimize the raw performance of all workloads we executed and our Hadoop environment before experimenting with power-efficiency. To more accurately model Hadoop's rack-based data layout and task assignment, we arbitrarily divided the cluster into 4 "racks" of 9 nodes each. We selected one of these racks to be the covering subset for our input datasets.

4.3.1 Methodology

In our experiments, we statically disable a number of nodes before running a Hadoop job and observe the impact on performance, energy, power consumption, and system inactivity. We assume the power consumption of a disabled node to be nil, and it contributes neither to the energy consumption nor performance of an experiment. Since the covering set for our input dataset is comprised of 9 nodes, we can gracefully disable up to 27 nodes.

Power models based on a linear interpolation of CPU utilization have been shown to be accurate with I/O workloads ($< 5\%$ mean error) for this class of server [135], since network and disk activity contribute

[2]Intelligent Platform Management Interface: `http://www.intel.com/design/servers/ipmi/`

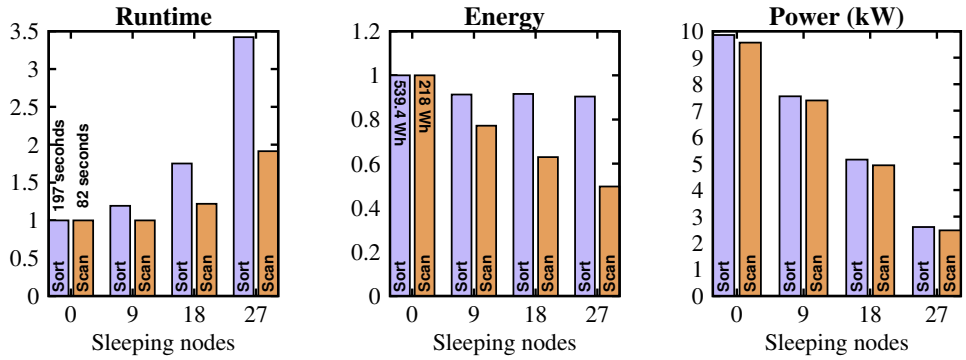


Figure 4.3: Runtime, Energy Consumption, and Average Power Consumption for the 32GB Sort and 32GB Scan workloads as nodes are disabled. Runtime and Energy are normalized to when all nodes are active.

negligibly to dynamic power consumption. Our nodes consume 223 W at idle and 368 W at peak utilization. Moreover, the use of a power model enables us to evaluate hypothetical hardware characteristics or capabilities in the future.

Our primary workloads are webdata_sort and webdata_scan from Hadoop's own "gridmix" batch-throughput benchmark. We generate several datasets (from 16GB to 128GB) using the same methodology as gridmix. We additionally generate batch-compute traces of several sorts and scans issued randomly over a 30 minute period (also seen in Section 4.1). We schedule enough jobs to occupy 75% of the 30 minute period, had the jobs been issued sequentially. This workload is loosely inspired by SPECpower_jbb2008 [47], which first determines the peak "operations per second" of a machine, and then measures power consumption at several lower operating points. Note that this workload more closely mimics a multi-user/tenant *throughput-sensitive* environment, rather than a single-user/tenant *latency-sensitive* environment.

4.3.2 Results

Figure 4.3 depicts performance, energy consumption, and power consumption over the duration of a Sort and Scan of a 32GB dataset, given different static configurations of the cluster. "0 sleeping nodes" is a baseline result, where all nodes are active. The other configurations show the effect of disabling an increasing number of nodes. We first observe that our mechanism works, and that a significant fraction of the cluster can be disabled with no impact on data availability, contrary to the position of [10].

Quantitatively, we find that disabling nodes in all cases leads to energy savings, from 9% with Sort to 51% with Scan. On the other hand, with the exception of disabling 9 nodes during the Scan, there is deleterious impact to performance as nodes are disabled (up to 71% when 27 nodes are disabled with Sort). Although in some cases severe, the penalty does not show a perfect slowdown. Overall, nodes tend to contribute less to performance than they do to energy consumption. Dramatic reduction is seen in the aggregate power consumption of the cluster as nodes are disabled. This clearly demonstrates that this mechanism can be used to implement cluster-level power-capping [34] (with a commensurate reduction in service-level).

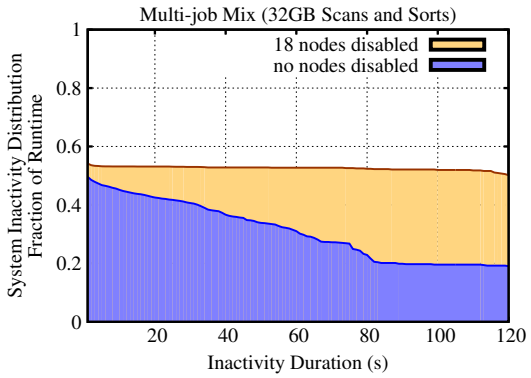


Figure 4.4: System Inactivity Distribution of the Multi-job workload when all nodes are active and when 18 nodes are disabled.

A similar large-scale experiment would process perhaps 1 TB of data, occupying a thousand node cluster. However, it would exhibit the same result as our small-scale experiment: at some point, nodes contribute less to performance than they do to energy consumption. Moreover, we would expect to observe bottlenecks in such a large-scale experiment that we don't observe in our cluster, such as contention at the network's root-switch/router or individual rack uplinks. Such bottlenecks would further soften the performance penalty of disabling nodes and further improve energy-efficiency.

To study the impact of disabling nodes on *fractional-utilization* workloads, we performed experiments with batches of Hadoop jobs, described earlier. Figure 4.4 depicts the distribution of system inactivity periods of the job trace when 18 nodes are disabled, and the distribution when all nodes are active is overlaid for comparison. Disabling 18 nodes significantly increases the length of time spent idle for more than 82 seconds (by 154%), and moderately increases the length of time spent idle for more than 10 seconds (18%). The increase at 82 seconds represents more opportunity to utilize brute-force energy management knobs, such as fully shutting a machine down, to reduce energy consumption. This is in contrast to the improvement at 10 seconds, which represents time potentially available to use less aggressive modes, such as spinning down disks and putting the platform and power supply into a standby mode. Note that while the increase at 10 seconds is modest, a significant fraction of the total time spent there is now contributed by *intentionally inactive* nodes, and it is trivially permissible to disable them. This is not the case for the run where all nodes are enabled, as most of this inactive time is due to *spontaneously inactive* nodes, for whom it is harder to determine whether or not it is permissible to disable. Note that since this is a job trace over 30 minutes, both experiments complete in the same amount of time, even though the constituent jobs may run for different lengths of time. The run with 18 nodes disabled consumes 44% less energy than the run with no nodes disabled.

4.4 Towards an Energy-Efficient Hadoop

This work to date has just scratched the surface of research into optimizing the energy-efficiency of Hadoop and similar distributed processing frameworks. The following describes numerous future issues to address.

4.4.1 Data Layout

Section 4.2 described our implementation of an energy-aware replication invariant, but it is easy to imagine a multitude of alternatives. Certain types of data may exhibit temporal locality, which could be taken into account in choosing whether or not to keep those data in a covering subset. Next, the covering subset of a cluster should rotate among datanodes to manage durability, about which we discuss below. Finally, should a covering subset become a write bottleneck, a lazy replication strategy could be adopted (first create a replica wherever convenient or performant, and later move it to the covering subset).

4.4.2 Data Availability

We have assumed that it is not permissible for requested data to only reside on disabled nodes. The concept of a covering subset and the design of our proposed replication invariant ensure some minimum availability of all requested data. However, it is reasonable to imagine that nodes could be enabled *on-demand* to satisfy availability. The efficacy of this approach would depend on how quickly a node can be enabled, or how well the task scheduler copes with an unruly delay. Such a solution may permit an energy-efficient Hadoop with no changes to data layout.

In addition, we currently handle the failure of a covering subset node by enabling all nodes and reconstituting replicas of the data. A less dramatic solution should be sought, or else a very minor equipment failure can turn into a very major power spike.

4.4.3 Reliability and Durability

A supposition of this work is that the data stored on intentionally inactive nodes are indelible. The degree to which this is true should be bounded, or the degree on which this property is depended should be limited. At a minimum, it is prudent for sleeping nodes to be woken periodically so that the integrity of the data they contain can be verified.

Second, the fact that HDFS (and the Google Filesystem) materializes n replicas of all data is blindly accepted as the cost of reliability. However, spraying replicas throughout a cluster during a computation reduces the performance of the computation (by consuming disk and network bandwidth), which ultimately translates to wasted energy (by not spending energy on useful work). Design principles regarding reliability should be revisited, and the incorporation of some other data durability mechanism could be considered (e.g. an enterprise SAN). Ideally, we would quantify the trade-off between reliability and energy consumption, such that users can make informed decisions. Note that some newer distributed processing frameworks, like

Spark, explicitly distinguish between transient intermediate results and permanent final outputs, and provide appropriate levels of reliability respectively [177].

4.4.4 Dynamic Scheduling Policies

A full implementation of an energy-efficient Hadoop should contain a dynamic power controller which is able to intelligently respond to changes in utilization of a Hadoop cluster. Different jobs may have disparate service levels requirement. Moreover, one must identify the best real-time signals to use for actuating power management activities, and how these impact job execution. As a most basic (but fundamental) example, inactive nodes that previously participated in a currently running job should not necessarily be disabled until after the job is complete; these nodes may contain transient output from Map tasks that will later be consumed by some separate Reduce task. There should be some intelligent cooperation between Hadoop's job scheduler and a power controller. Furthermore, the job scheduler has information that can improve the decisions made by the power controller. For example, all of the data blocks that a job will access are determined a priori, and this knowledge can be used to preemptively wake nodes or to hoist tasks with data on active nodes to the front of the task queue.

4.4.5 Node Architecture

Future technology, such as PowerNap [103] and PCRAM, may make short inactivity periods as useful as long inactivity periods in reducing energy consumption. In either case, this work is grounded in *increasing* the total inactive time available. Moreover, PCRAM and solid-state disks may not soon be economical for capacity-maximizing, cost-sensitive MapReduce clusters.

4.4.6 Workloads and Applications

While this work considered Hadoop's MapReduce, note that HBase [168] and BigTable [17] build upon the same filesystem infrastructure as their sibling MapReduce frameworks. While they have similar requirements of the storage layer as MapReduce (massively scalable, distributed, robust), the nature of their workload and activity is far removed. MapReduce computations are characterized by long, predictable, streaming I/O, massive parallelization, and non-interactive performance. On the other hand, the aforementioned structured data services are often used in real-time data serving scenarios [17]. As such, quality-of-service and throughput become more important than job runtime. The energy management strategies discussed in this paper should be evaluated for these structured data services.

4.5 Related Work

Power-proportional storage has been proposed at many layers of the storage stack, including individual disks, RAID arrays, and cluster-wide distributed storage services.

Gurumurthi et al. described server-class hard drives with variable spindle rates, able to save power at low load [51]. Weddle et al. evaluated a power-aware RAID layout that clusters disks into "gear groups" (analogous to the gears of an automobile transmission) [167]. In "low gears", data is spread across a small number of disks to consume a small amount of power with proportionally poor performance. In higher gears, data is spread wider for higher performance, but worse power consumption.

Kaushik et al. proposed an alternative to our covering subset policy in GreenHDFS, which uses statistical data-classification to identify data amenable for cold-storage [76]. Rather than guaranteeing availability of all data, they guarantee availability of hot data only.

Amur et al. extended our original HDFS work with Rabbit, which could scale-up and down its power consumption with robustly modeled impacts on performance [4]. In addition to designating a covering subset, they skew the layout of subsequent replicas using a power law to favor some nodes relative to others. Thus, some nodes contain much more data than others, and these can be powered-on early to gain access to large amounts of data with little relative increase in power consumption.

Krioukov et al. proposed NapSAC to manage power-proportional clusters of web servers [84]. They coordinate power-management actions with the load-balancer at the root of a web serving cluster. However, they assume all shared storage is distinct from the web servers, which greatly simplifies the problem.

Finally, Cidon et al. independently discovered that Hadoop's random replication policy leads to data unavailability when a small number of nodes are taken offline [18]. They proposed the MinCopySets algorithm to group storage nodes into replication sets, statistically minimizing the number of distinct failure domains in the whole storage system. They suggest that the parameters of this algorithm can be chosen so-as to provide power-proportionality as well.

4.6 Conclusions

This chapter presents a first effort towards improving the power-efficiency of MapReduce frameworks like Hadoop. First, we argued that Hadoop has the global knowledge necessary to manage the transition of cluster nodes to and from low-power modes. Hence, Hadoop should be, or cooperate with, the energy controller for the cluster. Second, we show that it is possible to recast the data-layout and task-distribution of Hadoop to enable significant portions of a cluster to be powered down while still fully operational. We find that running Hadoop clusters in fractional configurations can save between 9% and 50% of energy consumption, and that there is a trade-off between performance energy consumption. Despite this early success, we find that there is significant work to be done in understanding and improving the energy-efficiency of MapReduce frameworks.

Chapter 5

High Server Utilization and Sub-millisecond Quality-of-Service

We conclude the technical portion of this dissertation by analyzing the role that "quality of service" (QoS) plays in causing server underutilization. To put it succinctly, the simplest strategy to guarantee good quality of service (QoS) for a latency-sensitive workload with sub-millisecond nominal latency in a shared cluster environment is to never run other workloads concurrently with it on the same servers. Unfortunately, this inevitably leads to low server utilization, reducing the power-efficiency, capability, and cost effectiveness of the cluster as a whole.

In this chapter, we analyze the challenges of maintaining high QoS for low-latency workloads when sharing servers with other workloads. We show that workload co-location leads to latency QoS violations due to increases in queuing delay, scheduling delay, and thread imbalance. We present techniques that address these vulnerabilities ranging from provisioning the latency-critical service in an interference aware manner to replacing the Linux CFS scheduler with a scheduler that provides good latency guarantees and fairness for co-located workloads. Ultimately, we demonstrate that latency-critical workloads like memcached can be aggressively co-located with other workloads, achieve good QoS, and that such co-location can improve a datacenter's effective throughput per TCO-$ by up to 52%.

5.1 Introduction

Warehouse-scale datacenters host tens of thousands of servers and consume tens of megawatts of power [11]. These facilities support popular online services such as search, social networking, webmail, video streaming, online maps, automatic translation, software as a service, and cloud computing platforms. We have come to expect that these services provide us with instantaneous, personalized, and contextual access to terabytes of data. Our high expectations of these services are largely due to the rapid rate of improvement in the capability

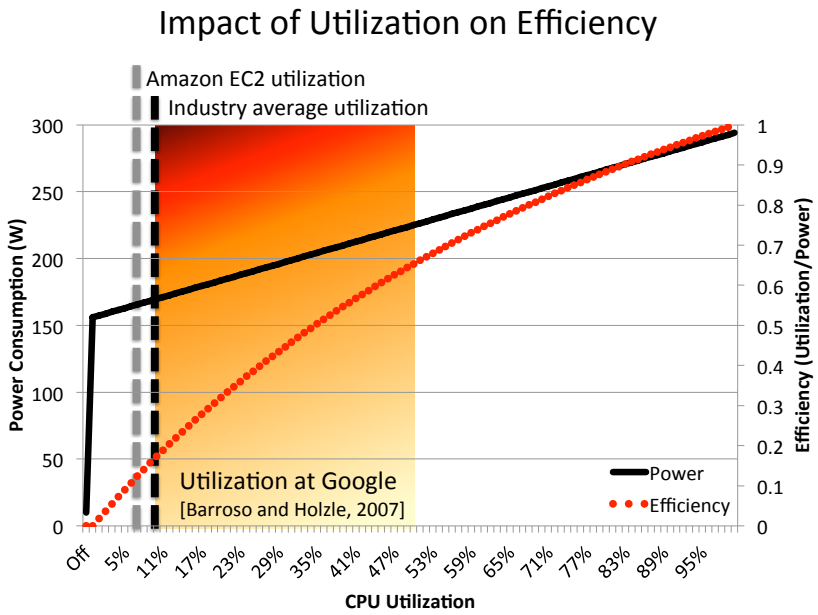


Figure 5.1: Typical server utilization leads to poor power-efficiency. Reproduced from Chapter 1.

(performance) and total cost of ownership (TCO) of the datacenters that host them.

Many factors that led to TCO and capability improvements are reaching the point of diminishing returns. Cost was initially reduced by switching from high-end servers to commodity x86 hardware and by eliminating the overheads of power distribution and cooling (PUE has dropped from 3.0 to 1.1) [11]. Unfortunately, these are both one-time improvements. In the past, we could also rely on deploying new servers with processors offering higher performance at the same power consumption; in effect, by replacing servers year after year, we could achieve greater compute capability without having to invest in new power or cooling infrastructure. However, the end of voltage scaling has resulted in a significant slowdown in processor performance scaling [33]. A datacenter operator can still increase capability by building more datacenters, but this comes at the cost of hundreds of millions of dollars per facility and is fraught with environmental, regulatory, and economic concerns.

These challenges have led researchers to pay attention to the utilization of existing datacenter resources. Various analyses estimate industry-wide utilization between 6% [74] and 12% [40, 164]. A recent study estimated server utilization on Amazon EC2 in the 3% to 17% range [93]. Even for operators that utilize advanced cluster management frameworks that multiplex workloads on the available resources [60, 146], utilization is quite low. For instance, Reiss et al. showed that a 12,000-server Google cluster managed with Borg consistently achieves aggregate CPU utilization of 10-20% and aggregate memory utilization of 40% [133]. This pervasive underutilization means that the vast majority of servers are operated in a power-*inefficient* manner (see Figure 5.1), and that resources in these datacenters are grossly overprovisioned. Hence, an obvious path towards improving both the capability and cost of datacenters is to make use of underutilized resources; in effect, raise utilization [26, 98, 99].

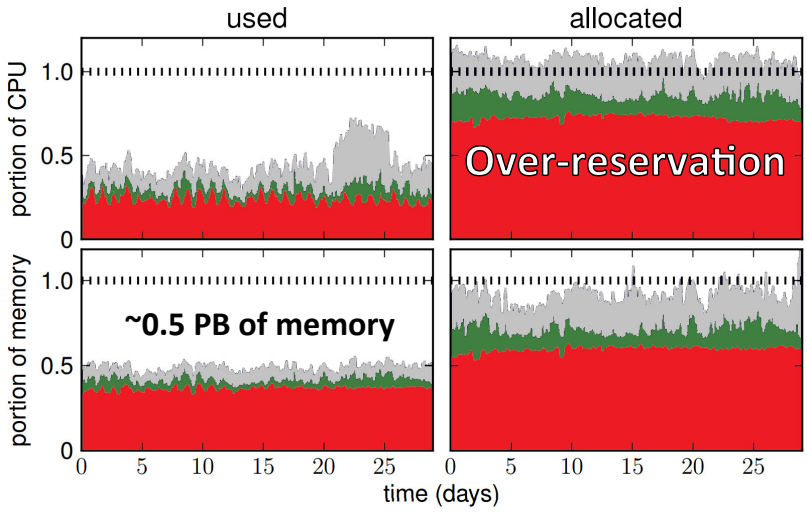


Figure 5.2: Trace of CPU and memory utilization and allocation over a 30-day period in a 12,000-node cluster at Google. CPU and memory resources are greatly underutilized but over-reserved. Adapted from [133].

High utilization is straight-forward to achieve if system throughput is the only performance constraint: we can co-locate multiple workloads on each server in order to saturate resources, switching between them in a coarse-grain manner to amortize overheads. In contrast, high utilization is difficult to achieve in the presence of complex, latency-critical workloads such as user-facing services like search, social networking, or automatic translation [8]. For instance, updating a social networking news feed involves queries for the user's connections and his/her recent status updates; ranking, filtering, and formatting these updates; retrieving related media files; selecting and formatting relevant advertisements and recommendations; etc. Since multiple tiers and tens of servers are involved in each user query, low average latency from each server is not sufficient. These services require low tail latency (e.g., low 95th or 99th percentile) so that latency outliers do not impact end-to-end latency for the user [22].

The conventional wisdom is that latency-critical services do not perform well under co-location. The additional workloads can interfere with resources such as processing cores, cache space, memory or I/O bandwidth, in a manner that introduces high variability in latency and violates each service's QoS constraints. This concern drives operators to deploy latency-sensitive services on dedicated servers, or to grossly exaggerate their resource reservations on shared clusters. For instance, the Google cluster studied by Reiss showed average CPU and memory reservation of 75% and 60% of available resources, respectively, while actual utilization was only 20% and 40%, shown in Figure 5.2.

The goal of this work is to investigate if workload co-location and good quality-of-service for latency-critical services are fundamentally incompatible in modern systems, or if instead we can reconcile the two. Using memcached, a widely deployed distributed caching service as a representative workload with aggressive QoS requirements (hundreds of microseconds in many commercial deployments), we study the challenges and opportunities in co-locating latency-critical services with other workloads on the same servers.

First, we analyze three common sources of QoS degradation due to co-location: (1) increases in queuing delay due to interference on shared resources (e.g., caches or memory), (2) long scheduling delays when timesharing processor cores, and (3) poor tail latency due to thread load imbalance. Second, we propose techniques and best practices to address these problems in existing servers. To manage queuing delay due to shared resource interference, we suggest interference-aware provisioning of the latency-critical service, so that interference leads to predictable decrease in throughput instead of intolerable spikes in latency. To address long scheduling delays, we find that Linux's CFS scheduler is unable to provide good latency guarantees while maintaining fairness between tasks. We demonstrate that a modern version of the BVT scheduler [31] affords predictable latency and fair sharing of CPU amongst tasks and allows for aggressive co-location of latency-critical services. Finally, we find that thread-pinning and interrupt routing based on network flow-affinity resolve much of memcached's vulnerability to load imbalance.

We put the observations above together to evaluate the potential savings from co-locating latency-critical services with other workloads under two important scenarios. For example, with a memcached cluster utilized at 30%, co-location of batch workloads gives you the equivalent throughput of a 47% larger cluster that would otherwise cost 29% more. On a general-purpose cluster with 50% CPU utilization, we can co-locate memcached to harness stranded memory, and achieve memcached performance that would otherwise require a 17% increase in TCO. Overall, we show that high server utilization and strong QoS guarantees for latency-critical services are not incompatible and that co-location of low-latency services with other workloads can be highly beneficial in modern datacenters.

5.2 Improving Utilization through Co-location

There are several reasons for the low utilization observed in datacenters today. Many companies have thousands of servers dedicated to latency-critical tasks, such as webserving and key-value stores for user-data. These servers are under-utilized during periods of low traffic, such as evenings or weekends. Even worse, the number of deployed servers is typically determined by the requirements of traffic spikes during uncommon events (e.g., Black Friday, celebrity mishaps), so these servers are often under-utilized, even during periods of high nominal traffic. This creates an opportunity to use the spare capacity for other workloads. For instance, a company like Facebook, which has thousands of dedicated servers for memcached, could use spare CPU capacity to run analytics jobs that would otherwise run on a separate set of machines. This co-location scenario can be quite beneficial in terms of computing capability and cost (discussed in Section 5.5) as long as the analytics jobs do not unreasonably impact the quality-of-service of memcached. Such a system might be managed by a cluster scheduler that can adjust the number of analytics jobs as the load on memcached varies throughout the day [26, 165].

Underutilization also occurs because of the difficulty in allocating the right number of resources for jobs of any kind, whether latency-critical or throughput-oriented. Similarly, it is difficult to build servers that have the perfect balance of resources (processors, memory, etc) for every single workload. The analysis of

a 12,000-server Google cluster by Reiss et al. [133] shows that while the average reservation of CPU and memory resources is 75% and 60% of available resources respectively, the actual utilization is 20% and 40% respectively. One could deploy memcached as an additional service that uses the underutilized processing cores to export the underutilized memory on such a cluster for other uses, such as a distributed disk cache. For a 12,000-server cluster, this could provide a DRAM cache with capacity on the order of 0.5 petabytes at essentially no extra cost, provided the original workload is not disturbed and that memcached can serve this memory with good QoS guarantees.

One can imagine several additional scenarios where co-locating latency-critical services with other workloads will lead to significant improvements in datacenter capability and TCO. Nevertheless, most datacenters remain underutilized due to concerns about QoS for latency-critical services. Ultimately, this is why most companies deploy latency-critical services, like memcached, on dedicated servers, and why latency-critical services running on shared machines have exaggerated resource reservations (as seen from Reiss' analysis of Google's cluster). Several recent works aim to discover when latency-critical services and other workloads do not interfere with each other, in order to determine when co-location is permissible [26, 99, 179]. However, this approach is pessimistic; it dances around the symptoms of interference but does not address the root causes. Our work provides a deeper analysis of the causes of QoS problems, which allows us to propose more effective mechanisms to raise utilization through co-location without impacting QoS.

5.3 Analysis of QoS Vulnerabilities

In this section, we perform an analysis of the QoS vulnerabilities of latency-critical services when co-located with other workloads. We focus on interference local to a server (processor cores, caches, memory, I/O devices and network adapters); QoS features for datacenter networks are studied elsewhere and not considered in this work [72].

The experimental portion of this analysis focuses on memcached. We chose memcached for several reasons. First, it has exceptionally low nominal latency, as low as any other widely deployed distributed service found in datacenters today (excluding certain financial "high-frequency trading" workloads). As such, it is quite sensitive to interference from co-located work, making it easy to identify and analyze their causes. Second, it is a concise, generic example of an event-based service, and many other widely deployed services (including REDIS, node.js, lighttpd, nginx, etc.) share many aspects of its basic architecture and use the same basic kernel mechanisms (epoll via libevent). Indeed, only 17% of CPU time when running memcached is spent in user-code, versus 83% in the kernel, so the particulars of memcached are less important than how it uses kernel mechanisms. Thus, we believe that a thorough analysis of memcached yields knowledge applicable to other services as well. Third, memcached consists of less than 10,000 lines of code, so it is easy to deconstruct. Finally, memcached is an important service in its own right, a cornerstone of several large web applications, and deployed on many thousands of servers. Knowledge gleaned from its careful analysis is useful, even outside the context of understanding interference from co-located work.

5.3.1 Analysis Methodology

There are a couple of prerequisite steps to our analysis. First, it is important to set realistic expectations about guarantees for latency under realistic load. For memcached, there is sufficient information about traffic patterns in large-scale commercial deployments (key and data sizes, put/get distributions, etc) in order to recreate realistic loads [6]. These deployments typically monitor tail latency and set QoS requirements thereof, such as requiring 95th-percentile latency to be lower than 500 to 1000 microseconds under heavy load. Second, it is equally important to carefully measure the single-request latency on an unloaded server. This measurement helps in multiple ways. First, if there is significant variability, it will be difficult to meet an aggressive QoS target even without interference from other workloads unless requests with high and low service times are prioritized and treated separately within the application. Second, the behavior of a well-tuned server can be approximated with a basic M/M/n queuing model (independent arrival and service time distributions), where n is the number of worker threads concurrently running on the server [49, 78, 105]. The unloaded latency (service time) allows us to estimate the performance potential of a well-tuned system and pinpoint when observed behavior deviates due to interference or any other reason.

The main part of our analysis consists of separating the impact of interference from co-located workloads into its three key causes: (1) *queuing delay*, (2) *scheduling delay*, and (3) *load imbalance*.

Queuing delay occurs due to coincident or rapid request arrivals. Even without co-location, the M/M/1 model suggests that, given a mean service rate of μ and mean arrival rate of λ, the average waiting time is $1/(\mu - \lambda)$ and 95th-percentile latency roughly $3\times$ higher ($ln(\frac{100}{100-95})/(\mu - \lambda)$). Interference from co-located workloads impacts queuing delay by increasing service time, thus decreasing service rate. Even if the co-located workload runs on separate processor cores, its footprint on shared caches, memory channels, and I/O channels slows down the service rate for the latency-critical workload. Even if a single request is only marginally slower in absolute terms, the slowdown is compounded over all queued requests and can cause a significant QoS problem. As λ approaches μ (i.e. at high load), wait time asymptotically increases, experienced as significant request latency by clients. It is common practice to take this asymptotic response into account when provisioning servers for a latency-sensitive service in order to ensure that no server is ever subjected to load that leads to excessive latency (i.e. provision servers for a load of at most 70-80% of μ). The impact of co-location on queuing delay can be thoroughly characterized by running the latency-critical service concurrently with micro-benchmarks that put increasing amounts of pressure on individual resources [25].

Distinct from queuing delay, scheduling delay becomes a QoS vulnerability when a co-located workload contends for processor cores with a latency-critical workload. That is, if the OS or system administrator assigns two tasks to the same core, they become subject to the scheduling decisions of the OS, which may induce long-enough delays to cause QoS violations. There are two parts to scheduling delay that must be considered independently: scheduler wait time, which is the duration of time that the OS forces a task to wait until it gets access to the core while another task runs, and context switch latency, which is the time it takes the OS and hardware to switch from running one application to another after the scheduling algorithm

has determined that it is the others' turn to run. The impact of co-location on scheduling delay can be characterized by constructing co-location scenarios that exacerbate the priorities and idiosyncrasies of the scheduling algorithm. We present such scenarios for commonly used Linux schedulers in Section 5.3.2.

Finally, load imbalance across threads of a multi-threaded service can lead to poor overall tail latency, even when the cumulative load on a server is low and average or median latency is asymptomatic. Co-located work can exacerbate load imbalance in a multi-threaded service in numerous ways: by causing additional queuing delay on only the threads that share hardware resources with the interfering work, by incurring scheduling delay on a subset of threads due to the interfering work, or when the OS migrates threads of the latency-sensitive application on top of each other to make way for the co-located work, causing the application to effectively interfere with itself. A latency-sensitive service's vulnerability to load imbalance can be easily ascertained by purposefully putting it in a situation where threads are unbalanced. For example, we can achieve this by running it with $N + 1$ threads on an N-core system, by forcing only one thread to share a hyper-threaded core with another workload, or by timesharing one core with another workload. By the pigeon-hole principle, such arrangements necessarily reduce the performance of at least one thread of the latency-sensitive service. If the service does not employ any form of load-balancing, this will cause at least one thread of the service to exhibit asymptotic queuing delay at far lower load than the other threads, and is readily observable in measurements of tail latency.

Note that queuing delay, scheduling delay, and imbalance should not be characterized at a fixed load, e.g., at the maximum throughput the server can support or the maximum throughput at which good QoS is still maintained. Since the goal of co-location is to increase utilization when the latency-critical service is at low or medium load, the characterization must be performed across all throughput loads, from 0% to 100%. This provides valuable data across the whole range of likely loads for the latency-critical service. As we show in Section 5.5, it also allows us to work-around interference issues by using cluster-level provisioning instead of server-level optimizations.

5.3.2 Analysis of Memcached's QoS Sensitivity

Experimental Setup

We now make the preceding analysis concrete by applying it to memcached. For all of the following measurements, we run memcached version 1.4.15 on a dual-socket server populated with Intel Xeon L5640 processors (2× 6 cores, 2.27 Ghz), 48GB of DDR3-1333 memory, an Intel X520-DA2 10GbE NIC and running Linux 3.5.0 (with ixgbe driver version 3.17.3). We disable the C6 C-state [138] during our experiments, as it induces high latency (100s of microseconds) at low load due to its wakeup penalty. We also disable DVFS, as it causes variability in our measurements at moderate load, and as its efficacy has diminished as voltage scaling has slowed [85]. Finally, we disable the `irqbalance` service and manually set the IRQ affinity for each queue of the X520-DA2 to distinct cores (using Intel's `set_irq_affinity.sh` script). Otherwise, memcached performance suffers due to grave softIRQ imbalance across cores. In this configuration, memcached achieves

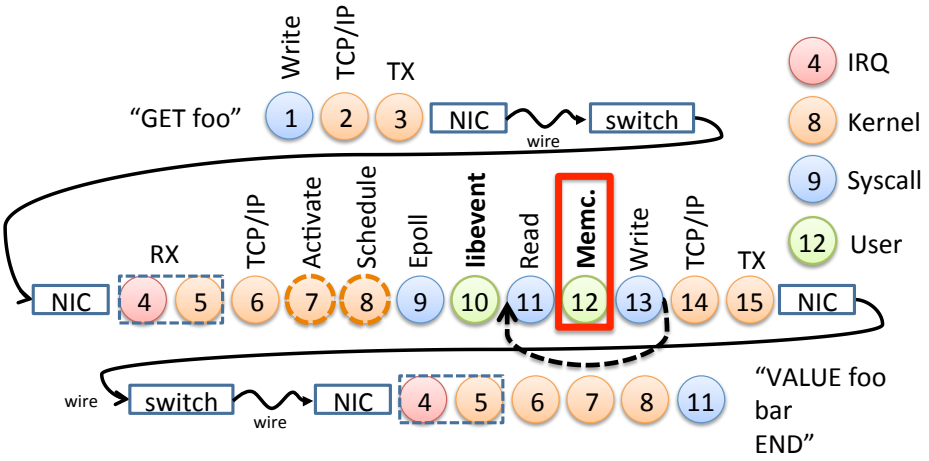


Figure 5.3: Life of a memcached request. Note that actual request processing (highlighted) is but one step on a long journey.

1.1M queries per second (QPS) for non-pipelined requests, and over 3.0M QPS for pipelined requests.

We generate client load for the server with *mutilate*[1], a high-performance, distributed memcached load-generator we developed that can recreate the query distributions at Facebook reported by Atikoglu et al. [6]. Requests are paced using an exponential distribution, and access 200-byte values uniformly at random from a set of 1,000,000 30-byte keys (sized to match the APP pool studied by Atikoglu). Surprisingly, we found no difference in latency with larger datasets or non-uniform access patterns. We do not study larger value sizes as we are focused on interference due to co-located workloads, not adversarial request patterns. As memcached is commonly accessed synchronously, we do not pipeline requests. Even in environments that make extensive use of parallel *multi-GETs*, keys are sharded across multiple servers, such that the number of requests handled by any one server is low. In any case, we configure mutilate to make hundreds to thousands of connections to the memcached server, spread across 20 client machines. This ensures that we can generate enough load on the server to saturate it and observe *server-side* queuing delay, while never taxing the clients enough to see *client-side* queuing delay.

Memcached Request Pipeline

We begin the analysis by presenting a detailed description of memcached's request pipeline. There is substantially more to executing a memcached query than just looking up a value in a hash-table. In order to gain a more detailed understanding, we traced the life-cycle of a memcached request in Linux. Figure 5.3 depicts the basic steps involved.

A client initiates a request by constructing a query and calling the `write()` system call (1). The request undergoes TCP/IP processing (2), is transmitted by the client's NIC (3), and is then sent to the server's NIC via cables and switches, where upon the processor core running memcached receives an interrupt (since the Intel

[1] https://github.com/leverich/mutilate/

Who	What	Unl.	Ctx Sw.	Loaded	L3 int.
Server	RX	0.9us	0.8us	1us	1us
	TCP/IP	4.7us	4.4us	4us	4us
	EPoll	3.9us	3.1us	2,778us	3,780us
	libevent	2.4us	2.3us	3,074us	4,545us
	Read	2.5us	2.1us	5us	7us
	memcached	2.5us	2.0us	2us	4us
	Write⋆	4.6us	3.9us	4us	5us
	Total	**21.5us**	**18.7us**	**5,872us**	**8,349us**
Client	End-to-end	49.8us	47.0us	6,011us	8,460us
	TX-to-RX	36us	30us	-	-
Switch	RX-to-TX	3us	-	-	-

Table 5.1: Latency breakdown of an average request when the server process is unloaded (Unl), when it is context-switching with another process (Ctx Sw), when it is fully loaded (Loaded), and when it is subjected to heavy L3 cache interference while fully loaded (L3 int). Client "TX-to-RX" is the time from when the client has posted a packet to its NIC until it receives a reply, and "End-to-end" is the time reported by mutilate on the clients. Switch latency is estimated by connecting the client directly to the server and measuring the difference in latency. ⋆For brevity, we include TCP/IP and TX time in Write.

X520-DA2 NIC maintains flow-to-core affinity with its "Flow Director" hardware [128, 171]). Linux quickly acknowledges the interrupt, constructs a `struct skbuff`, and calls `netif_receive_skb` in *softIRQ* context (4). After determining it is an IP packet, *ip_rcv* is called (5), and after TCP/IP processing is complete, *tcp_rcv_established* is called (6). At this point, the memcached process responsible for handling this packet has been identified and activated (marked as runnable) (7). Since there is no more packet processing work to be done, the kernel calls `schedule` to resume normal execution (8). Assuming memcached is asleep waiting on an `epoll_wait` system call, it will immediately return and is now aware that there has been activity on a socket (9). If memcached is not asleep at this point, it is still processing requests from the last time that `epoll_wait` returned. Thus, when the server is busy, it can take a while for memcached to even be aware that new requests have arrived. If `epoll_wait` returns a large number of ready file descriptors, it executes them one by one and it may take a long time for memcached to actually call `read` on any particular socket (10). We call this *libevent* time. After returning from `epoll_wait`, it will eventually call `read` on this socket (11), after which memcached finally has a buffer containing the memcached request. After executing the request by looking up the key in its object hash-table, memcached constructs a reply and `write`'s to the socket (13). Now TCP/IP processing is performed (14) and the packet is sent to the NIC (15). The remainder of the request's life-cycle at the client-side plays out similar to how the RX occurred at the server-side. It is interesting to note that a large portion of this request life-cycle is played out in the Linux kernel. We find that only 17% of memcached's runtime is spent in user code vs. 83% in kernel code. 31% of runtime is spent in Soft IRQ context.

Using SystemTap [32], we have instrumented key points in the Linux kernel to estimate how long each step of this process takes. By inspecting the arguments passed to kernel functions and system calls, we are able to create accurate mappings between `skbuff`s, file descriptors, and sockets. Using this information, we

can track the latency of *individual requests* as they work their way through the kernel, even though hundreds of requests may be outstanding at any given time. We take measurements for an unloaded case (where only one request is outstanding), a context switching case (where a cpu-bound task is running and the OS must context-switch to memcached after receiving a packet), a loaded case (where memcached is handling requests at peak throughput), and an interference case (where we subject the loaded memcached to heavy L3 cache interference).

Manifestation of Queuing Delay

Table 5.1 presents the latency breakdown by condensing measurements into key periods: driver RX time, TCP/IP processing, waiting for epoll to return (which includes process scheduling and context switching if memcached isn't already running), libevent queuing delay, read system-call time, memcached execution time, and write system-call time. In the unloaded case there are no surprises: TCP/IP processing, scheduling, and epoll take a plurality of time. We discuss the context-switching case in Sec. 5.3.2. Distinct from the unloaded case, our measurements of the loaded case gives us a key insight: *the vast majority of the latency when memcached is overloaded is queuing delay*. This queuing delay manifests itself in the measurement of "libevent" time, but also "epoll" time. When overloaded, `epoll_wait` is returning hundreds of ready file descriptors. Thus, it will take a while to get to any one request (long "libevent" time). Second, since so many requests are being received by memcached at once, it will take a long time to process them all and call `epoll_wait` again. This shows up in the long "epoll" time measured for subsequent packets. When subjected to interference (for instance, L3 interference), the moderate growth in the time it takes to process each individual request (read, memcached) results in a substantial increase in this queuing delay (epoll, libevent).

This increase in queuing delay due to L3 interference, however, is observed most distinctly at high load. At low load, the interference is hardly measurable. Figure 5.4 plots latency vs. QPS for memcached when subjected to a range of loads. When subjected to L3 cache interference, its service rate is effectively reduced, resulting in asymptotic queuing delay at lower QPS. However, note that whether or not L3 interference is present, latency at 30% QPS is quite similar. This indicates that the impact on per-request latency due to interference is in and of itself insufficient to cause QoS problems at low load. Instead, queuing delay at high load is the real culprit. We build on this observation in Section 5.4.1.

Manifestation of Load Imbalance

As discussed in Section 5.3.1, it is a simple exercise to determine if a service is vulnerable to multi-threaded load imbalance: simply run it with $N + 1$ threads on an N-core system. Figure 5.5 shows the impact on memcached QPS vs. latency in this scenario (13 threads on a 12-core system). A pronounced spike in tail latency is observed at around 50% of peak QPS not seen in lower-order statistics (i.e. median or average). Interestingly, we found that a similar spike in tail latency is observed even when memcached is run with just 12 threads (instead of 13) on this 12-core system (i.e. N instead of $N + 1$ on an N-core system). Upon

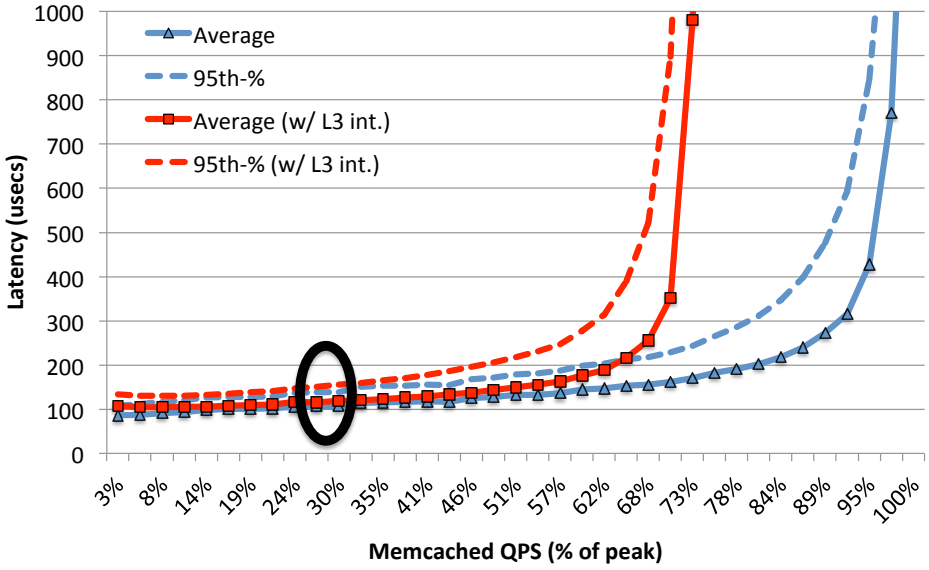


Figure 5.4: Impact of heavy L3 interference on latency. Interference causes substantial queuing delay at high load, but has little impact at low to moderate load (e.g., at 30%).

inspection, we found that Linux frequently migrated threads from one core to another, often ending up with some cores with two threads and other cores with no threads, leading to essentially the same tail latency regressions we observed with 13 threads. Linux's "affine wakeup" mechanism appears to be most responsible for these migrations, causing memcached threads to migrate closer to each other, even when there is only minor contention for shared mutexes. Moreover, Linux's CPU load-balancing mechanism operates at the time scale of 100s of milliseconds, so any mistake in thread placement lasts long enough to be observed in measurements of tail latency. Linux's thread migration issues notwithstanding, these experiments demonstrate that memcached performs no dynamic load balancing across threads. Indeed, inspection of its source code shows that it assigns incoming client connections to threads statically in a round-robin fashion. This lack of load balancing is the root cause of memcached's vulnerability to load imbalance, and a contributing factor to its sensitivity to queuing delay and scheduling delay.

Interestingly, UDP connections to memcached do not suffer from this load imbalance problem in the same way as TCP connections. Memcached monitors the same UDP socket across all threads, and UDP requests are handled by whichever thread reads the socket first. However, kernel lock contention on this UDP socket limits peak UDP throughput at less than 30% of that using TCP. Indeed, Facebook rearchitected memcached's UDP support to use multiple sockets in order to work around this throughput problem [140].

Manifestation of Scheduling Delay

As discussed in Section 5.3.1, scheduling delay consists of context-switch latency and wait time. Seen in Table 5.1, our SystemTap traces show that not only does memcached not seem to incur any overhead due

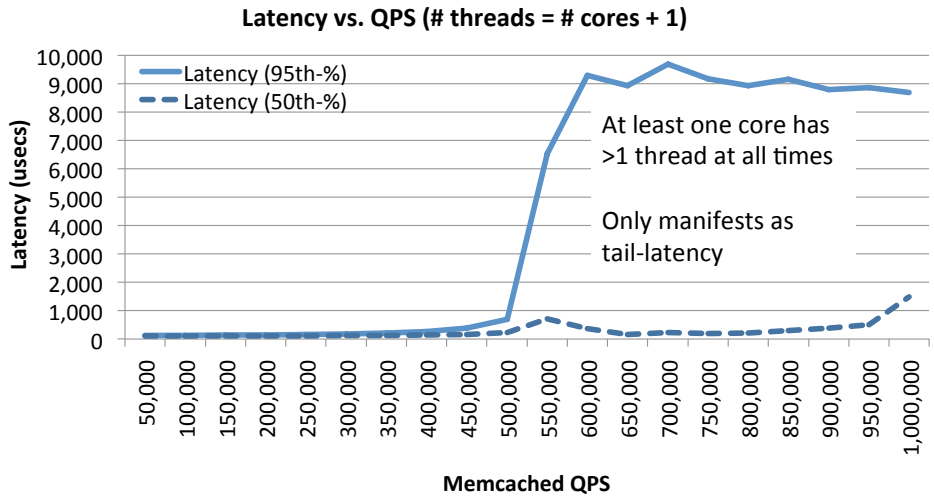


Figure 5.5: Impact of load imbalance on memcached QoS.

to the context switch after receiving a request, *it actually goes slightly faster*. We limit the CPU to the C1 C-state for this experiment, so this is not C3 or C6 transition overhead. We suspect that this overhead has to do with the C1 state or with Linux's management of timers for "tickless" idle support. In any event, the upshot is that Linux can quickly switch to memcached and service a request with little impact on end-to-end latency, even when another application is running on the CPU core when the request arrives.

Interestingly, contention for the L1 caches or TLBs when timesharing a CPU appears to not be an issue for memcached. Figure 5.6 shows latency vs. QPS for memcached when timesharing a CPU with a wide variety of SPEC CPU2006 workloads. In this experiment, we run memcached with "nice -n -20", so that it essentially never incurs wait time; we are only observing the impact of context switching (direct and indirect) with another workload. The fact that memcached is affected so little by contention for the L1 caches and TLBs may be unexpected, but it can be explained intuitively. First, per-request latency is inconsequential compared to the latency incurred from queuing delay, so any increase in per-request latency at low-load is effectively negligible. Second, at high-load, memcached is running often enough to keep its cache footprint warm, so it sees essentially no interference. Finally, since the majority of memcached's runtime is spent in kernel code (83% of runtime), the TLB flushes due to each context switch have little impact; the kernel's code is mapped using large pages. Our conclusion from this experiment is that *context-switching is not itself a large contributor to memcached latency*. If memcached is vulnerable to scheduling delay, it is almost entirely due to wait time induced by the OS scheduling algorithm.

In contrast to context-switch time, memcached is quite vulnerable to scheduler wait time. We demonstrate the scale of the danger by forcing memcached to timeshare a core with a "square-wave" workload. Figure 5.7 shows latency vs. QPS for memcached when timesharing a CPU with such a square-wave antagonist, where the antagonist runs in a tight loop (i.e. `while(1);`) for 6ms, sleeps for 42ms, and then repeats; its average CPU usage is only 12.5%. As can be seen, when using CFS (Linux's default scheduler), memcached starts

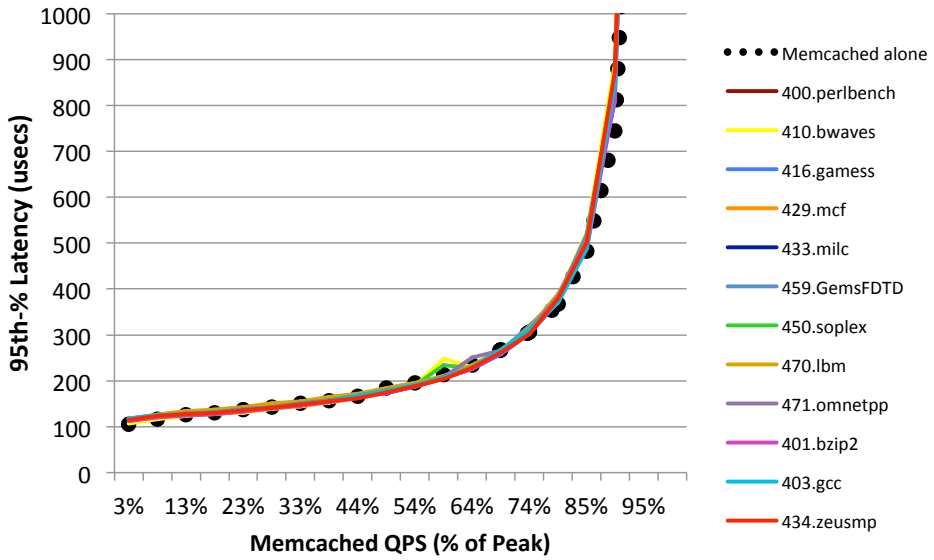


Figure 5.6: Impact of context-switching with other workloads on memcached latency.

exhibiting exceptionally long tail-latency starting at around 12% of peak QPS, where the aggregate load on the CPU (including both memcached and the antagonist) is only around 25%. The fact that latency spikes when memcached is at approximately the same CPU usage as the antagonist is an interesting artifact of CFS's algorithm; we explore this in detail in Section 5.3.3. At 87% QPS (∼100% CPU load), memcached must yield involuntarily to ensure the antagonist gets fair access to the CPU, so it necessarily experiences long latency.

To illustrate the consequences of this vulnerability in a more realistic setting, we measured QoS for two latency-sensitive services running simultaneously on one server: memcached (with average latency measured on the order of 100us) and a synthetic event-based service similar to memcached with average latency on the order of 1ms. We ran both services concurrently in every combination of QPS from 10% to 100% and monitored latency and throughput. The results are charted in Figure 5.8, which reports which of the two services were able to maintain acceptable QoS at each load point. Note that memcached fails to achieve good QoS, even when the co-scheduled workload is offered exceptionally low load (as circled in Fig. 5.8). Additionally, the 1ms service fails to meet its QoS requirement at higher loads, as it is sensitive to excessive wait time as well. The consequence of this result is that *neither* latency-sensitive service can safely be co-located with the other. Overall, we can conclude that memcached is particularly sensitive to scheduler wait time, and that addressing it (either by adjusting the scheduler or prohibiting CPU time-sharing) is paramount to running memcached with good QoS.

5.3.3 Scheduling Delay Analysis

We noted in Section 5.3.2 that memcached experienced a spike in scheduler wait time right around the point when its CPU usage exceeded that of a co-scheduled application, despite the fact that the CPU is underutilized

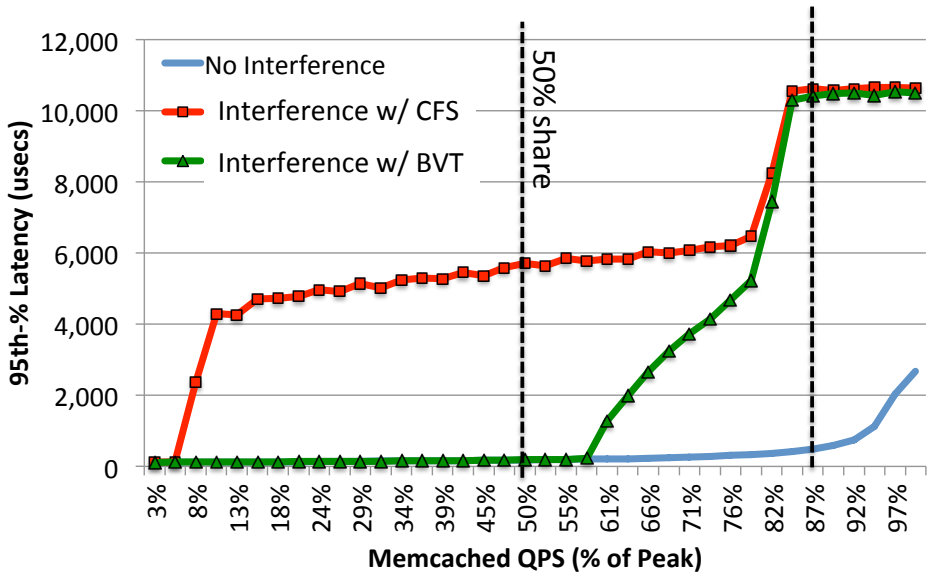


Figure 5.7: Demonstration of scheduler wait-time induced on memcached when run concurrently with a "square-wave" antagonist. The antagonist runs for 6ms every 48ms (12.5% load), and is assigned an equal fair share weight as memcached. Using the CFS scheduler, memcached exhibits unacceptably high tail latency even at low load. With BVT (discussed in Sec. 5.4.3), memcached tail latency is as good as if there were no interference until it exceeds its fair share of CPU time. Both schedulers cap memcached's throughput at ~87%, where the CPU is 100% utilized.

overall. We describe this behavior in detail in this section.

The essence of the "Completely Fair Scheduler" (CFS) [124], Linux's general-purpose scheduling algorithm is quite simple: all non-running tasks are sorted by "virtual runtime" and maintained in a sorted run-queue (implemented as a red-black tree). Tasks accumulate virtual runtime (a weighted measure of actual runtime) when they run. Whenever there is a scheduling event (i.e. a periodic timer tick, an I/O interrupt, a wakeup notification due to inter-process communication, etc.) the scheduler compares the virtual runtime of the current running task with the virtual runtime of the earliest non-running task (if the event is a periodic timer tick) or the task being woken (if the event is a wakeup). If the non-running task has a smaller virtual runtime, the scheduler performs a context switch between the two tasks. Overall, the algorithm attempts to guarantee that the task with the lowest virtual runtime is always the running task, within some error bounded by the maximum period between scheduling events times the number of running tasks. CFS also uses several heuristics to prevent overscheduling (i.e. `sched_min_granularity`). Users may assign "shares" to tasks, and a task's accumulation of virtual runtime when running is weighted inversely proportional to its number of shares. In the long term, tasks receive a fraction of CPU time proportional to the fraction of total shares they are assigned.

An important detail in the design of CFS is how virtual runtime is assigned for a task that wakes up and becomes runnable. The assignment of virtual runtime for waking tasks balances two properties: (1) allowing the waking task to run promptly, in case the event that caused its wakeup needs to be handled urgently, and

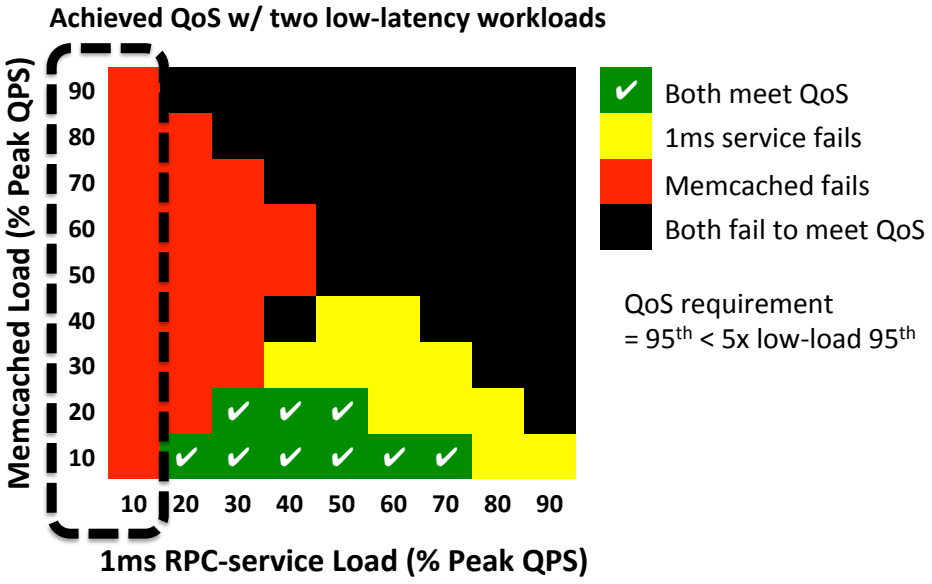


Figure 5.8: Achieved QoS when co-scheduling two latency-sensitive services on a processor core with Linux's CFS scheduler. For both services, we require that its 95th-% latency never exceed 5× its 95th-% latency as measured at low-load. Both services achieve tolerable QoS only in a narrow range of loads, so it is unwise to co-locate them together without taking corrective action.

(2) not giving an unfair share of processor time to waking tasks. CFS balances these two goals by clamping the virtual runtime of a waking task to the minimum virtual runtime of all non-sleeping tasks minus an offset L (called "thresh" in the kernel), which defaults to one half of the target scheduling latency (L = 24ms/2 = 12ms): $\text{vruntime}(T) = \max(\text{vruntime}(T), \min(\text{vruntime}(*)) - L)$.

Unfortunately, CFS's wakeup placement algorithm allows sporadic tasks to induce long wait time on latency-sensitive tasks like memcached. The fundamental problem is that the offset CFS uses when placing the waking task is larger than memcached's nominal request deadline. Illustrated in Figure 5.9, if memcached (A) is serving a request when another task B wakes up, it must wait at least for L time before it can resume processing requests. The only way memcached can be guaranteed to never see this delay is if its virtual runtime never exceeds that of the other task. Coming back to Figure 5.7, this fully explains why memcached achieves good quality of service when its load is lower than 12%; it is accumulating virtual runtime more slowly than the square-wave and always staying behind, so it never gets preempted when the square-wave workload wakes.

5.3.4 Discussion

The preceding sections have concretely demonstrated how memcached experiences significant reductions in QoS due to queuing delay, scheduling delay, or load imbalance in the presence of co-located workloads. In the course of this study, we additionally evaluated the impact of interference due to network-intensive co-located

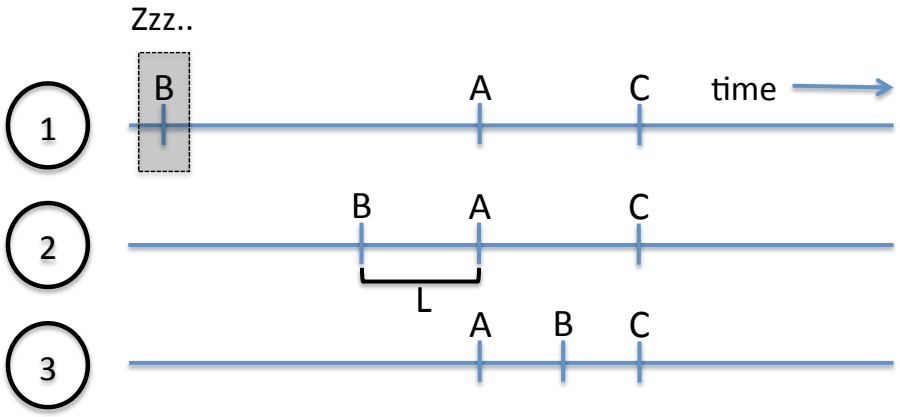


Figure 5.9: A depiction of CFS run-queue when a task wakes up. Task B is initially asleep (1). When it wakes up, it is placed L behind task A and preempts task A (2), inducing wait time on A while B runs (3).

workloads. Interestingly, these workloads caused far less interference than expected for the following reasons: (1) memcached, particularly with realistic request size distributions, saturates the server's CPUs before it saturates the 10GbE network, and (2) Intel's Flow Director [171] largely isolates co-located workloads from interference due to interrupt handling for large network loads; network interference was indeed a problem when we disabled Flow Director and only used RSS.

It's worth noting that there are a large variety of scenarios where memcached's QoS could suffer even in the absence of co-located workloads. For instance, a misbehaving client could flood the server with requests, denying service to other clients; a client could try to mix large and small requests together, causing long serialization latency for the small requests; the request stream could contain an unusually high number of "SET" requests, which induces frequent lock contention between threads; or the server could genuinely be overloaded from provisioning an insufficient number of servers for a given load. We do not provide a detailed characterization of these vulnerabilities here, as this work is focused on interference caused by co-located workloads.

5.4 Addressing QoS Vulnerabilities

This section describes robust strategies that can be employed to address the vulnerabilities identified in Section 5.3 when co-locating latency-sensitive services on servers.

5.4.1 Tolerating Queuing Delay

In Figure 5.4, we demonstrated that interference within the memory hierarchy from co-located workloads only causes tail latency problems when it exacerbates queuing delay. The upshot is that workloads may be safely co-located with each other, despite interference, so long as they don't induce unexpected asymptotic queuing delay. Since queuing delay is a function both of throughput (service rate) and *load*, we can tolerate

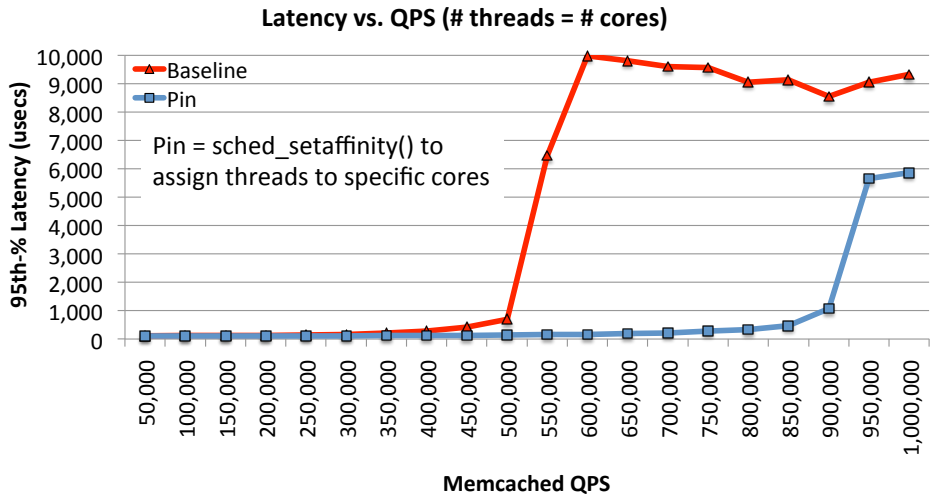


Figure 5.10: Pinning memcached threads to distinct cores greatly improves load balance, consequently improving tail latency.

a reduction in throughput (due to interference) if we also reduce the load on the service for any given server. Additional servers can be added to pick up the slack for the low-latency service. Thus, we propose that load be provisioned to services in an *interference-aware* manner, that takes into account the reduction in throughput that a service might experience when deployed on servers with co-located workloads.

Load provisioning is already an important aspect of the life-cycle of distributed datacenter services. Provisioning entails selecting how many and what type of servers are needed to handle a given aggregate request load. While interference-aware provisioning may be pessimistic with respect to how much load an individual server could support for a given service, it accommodates the potential for interference on shared hardware resources and is very permissive in terms allowing other workloads to be co-located on the same hardware. Hence, while interference-aware provisioning may use more servers for the latency-critical service, the TCO for the cluster as a whole may actually be lower because any capacity underutilized by the latency-critical service can be used by other workloads. We give a concrete example of how interference-aware provisioning can improve overall cluster utilization and TCO in Section 5.5.

5.4.2 Tolerating Load Imbalance

In Section 5.3.2, we reported that memcached exhibits a profound spike in latency at around 50% load when running with N threads on an N-core system due to spurious thread-migrations placing multiple threads on a single core. Note that this latency spike is in the absence of co-located work; we cannot operate this server at high load, even if it is dedicated to this workload. One solution to this problem is particularly straight-forward and effective: threads can be pinned explicitly to distinct cores, so that Linux can never migrate them on top of each other. We made a simple 20-line modification to memcached 1.4.15 to query its available CPU set at startup and to statically pin threads (using `sched_setaffinity()`) as they are

spawned to each core in round-robin order. After this simple modification, memcached handles up to near 90% of peak load (over 900,000 QPS) with 95th-% latency under 1ms (see Figure 5.10). We should also note that the flow-to-core affinity provided by Intel's ixgbe driver for their X520-DA2 NIC also contributes substantially to maintaining balance amongst memcached threads: it ensures that the interrupt load on each core is proportional to its request rate, and that network interrupts destined for co-located workloads do not disturb the cores running memcached [128, 171].

Although thread-pinning is of exceptional utility in this case, it does not address the deeper problem: that memcached makes no attempt to perform load-balancing amongst threads on its own. Thus, it is important to heed the following guidelines when deploying memcached with co-located work. First, it is imperative to pin memcached threads to distinct cores, as demonstrated above. Second, spawn at most N memcached threads on an N-core system; memcached achieves no higher performance with more threads, and tail latency is adversely affected if some cores have more threads than other cores. Third, if you wish to share a CPU core with other workloads (either via OS scheduling or hyper-threading), you might as well share all of the cores that memcached is running on; memcached's tail latency will ultimately be determined by its slowest thread. Finally, if thread load imbalance is unavoidable but certain requests require minimal latency, issue those requests over a UDP connection (Sec. 5.3.2). In the long-term, memcached should be rearchitected to proactively load balance either requests or connections across threads, like Pisces [149].

5.4.3 Tolerating Scheduling Delay

As described in Section 5.3.3, the root cause of scheduler-related tail latency lies with CFS's wakeup placement algorithm, which allows workloads which frequently sleep to impose long wait times on other co-located workloads at arbitrary times. Fortunately, there are several strategies one can employ to mitigate this wait time for latency-sensitive services, including (1) adjusting task share values in CFS, (2) utilizing Linux's POSIX real-time scheduling disciplines instead of CFS, or (3) using a general purpose scheduler with support for latency-sensitive tasks, like BVT [31].

Adjusting task shares in CFS

Assigning an extremely large share value to a latency-sensitive task (i.e. by running it with "nice -n -20" or by directly setting its CPU container group's shares value) has the effect of protecting it from wakeup-related wait time. Recall from Section 5.3.3 that a task's virtual runtime advances inversely proportional to the fraction of shares that a particular task possesses relative to all other tasks. Thus, if a task A has a very high shares value, its virtual runtime advances at a crawl and every other task advances at a relative sprint. Thus, each time a task other than A runs, it accrues significant virtual runtime and leaps far ahead of A. Consequently, these tasks are essentially never eligible to preempt A or induce wait time on it, and effectively only run when A yields.

While effective at mitigating wait-time for latency-sensitive tasks, such a strategy presents a conundrum for servers with co-located workloads: an unexpected spike in load on the latency-sensitive service, or even a

bug, could cause its CPU usage to spike and starve the other task. In effect, a task's immunity to wait time and its share of CPU time are tightly coupled in CFS, since the only tunable parameter available to the end-user (shares) affects both. This limitation is fine for some situations (i.e., co-locating best-effort analytics jobs with a memcached cluster at Facebook), but is inappropriate when co-locating multiple user-facing services (i.e. at Google).

POSIX real-time scheduling

An alternative to CFS are the POSIX real-time scheduling disciplines implemented in Linux, `SCHED_FIFO` or `SCHED_RR`.[2] These schedulers are priority-based, where higher-priority tasks may never be preempted by lower-priority tasks, as opposed to the general-purpose fair-share nature of CFS. Tasks scheduled with the POSIX real-time schedulers are implicitly higher-priority than CFS tasks in Linux, so they are never preempted by CFS tasks. Thus, latency-sensitive tasks scheduled using the real-time schedulers never incur CFS-induced wait-time due to other co-located tasks. Additionally, multiple latency sensitive tasks can be safely run concurrently up to an aggregate load of $\sim$70% so long as they are assigned priorities *rate monotonically* (i.e. lower latency begets a higher priority) [92].

Similar to using CFS with a high shares value, the real-time schedulers enable high-priority tasks to starve other tasks; a load-spike or bug in a latency-sensitive service could lead to unfair use of the CPU. Again, this is tolerable when co-located with best-effort workloads, but not with other workloads with throughput or latency requirements.

CPU Bandwidth Limits to Enforce Fairness

For both CFS and the real-time schedulers, Linux allows users to specify a CPU bandwidth limit [158], where a user specifies a runtime "quota" and "period" for a task. This mechanism can prevent high-share tasks from starving others, but it comes with its own drawback: it eliminates the property of "work conservation" from the schedulers. So even if CPU time is unused and available, a task would not be permitted to run if it exceeded its bandwidth quota. While not terribly important at low loads, non-work-conserving means that a task operating at close to its share of CPU time can be hit with long latency as it waits for its quota to be refilled, even if the server is otherwise underutilized.

We demonstrate the negative consequences of non-work-conservation by repeating the experiment of Figure 5.8, where two latency-sensitive tasks are co-located on the same server, but using the real-time SCHED_FIFO scheduler for both tasks instead of CFS. We assigned memcached a higher priority than the 1ms server (i.e. rate-monotonically). We assigned quotas to each of the two services proportional to how high of QPS they were offered at each point (i.e. at 30%/30%, each service is given a quota of 50% of total

[2]The principle difference between SCHED_FIFO and SCHED_RR is that SCHED_FIFO tasks run indefinitely until they either voluntarily yield or a higher-priority task becomes runnable, where as SCHED_RR tasks time-share in round-robin order with a slice duration of 100ms.

Feature	Completely Fair Scheduler	POSIX Realtime + Bandwidth Limit	BVT [Duda'99] + Grace Period
Reservations	✔	✔	✔
Configurable preemption	✖	✔	✔
Work conserving	✔	✖	✔

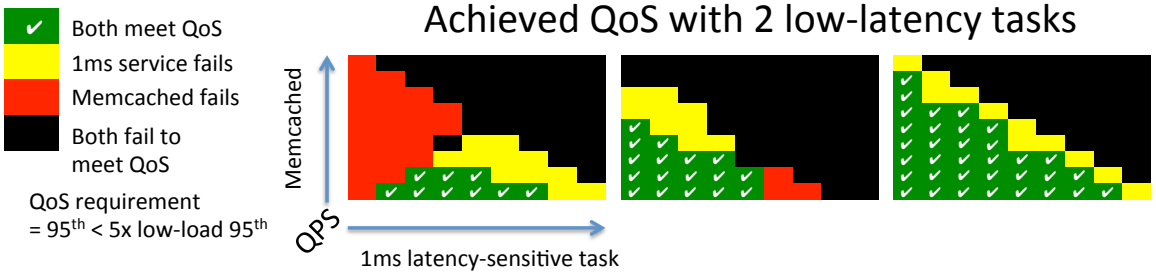


Figure 5.11: Comparison of scheduling algorithms. "Reservations" indicates whether CPU time can be guaranteed for distinct tasks. "Configuration Preemption" indicates whether the scheduler allows the user to indicate which task may preempt another. See Figure 5.11 for axes on the QoS tables.

CPU time), and a period of 24ms (equal to CFS's scheduling period). The result is depicted in the middle column of Figure 5.11. In this configuration, the co-location is substantially improved compared to the baseline CFS: there is a large range of moderate loads where both services achieve acceptable QoS. Still, it leaves the server quite underutilized: at least one service degrades when aggregate load exceeds 70%. The degradation is entirely due to the bandwidth limit and non-work-conservation: even though CPU time is available, the services are prohibited from running by fiat.

Borrowed Virtual Time (BVT)

In contrast to CFS and the POSIX real-time schedulers, Borrowed Virtual Time (BVT) [31] affords a more sensible option for operators looking to co-locate latency-sensitive tasks with others. BVT extends virtual-time scheduling by adding a user-provided "warp" parameter that specifies a static offset to be applied to a task's virtual runtime when making scheduling decisions; this adjusted virtual runtime is termed *effective virtual runtime*. This adjustment biases short-term preemption actions, but has minimal impact on long-term throughput fairness, which is still dictated by the fair-share weight assigned to different tasks. This behavior can be seen in Figure 5.12. Essentially, BVT allows preemption priority and fair-share allocation to be controlled independently, rather than being tightly coupled as in CFS.

We have implemented BVT as a concise patch to CFS (approx. 150 lines changed) which allows users to

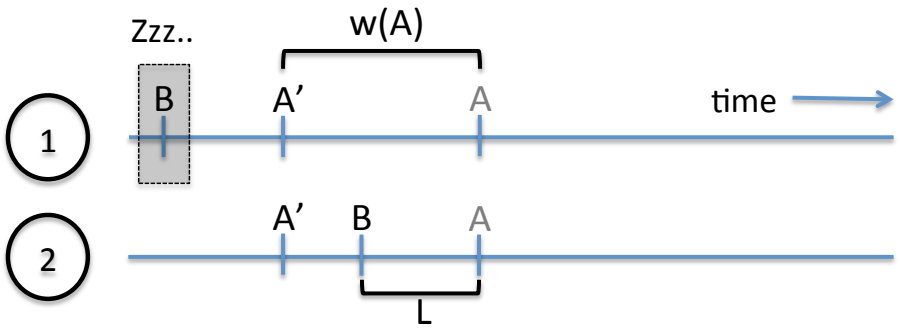


Figure 5.12: A depiction of BVT run-queue when a task wakes up. Task B is initially asleep. When it wakes up, it gets placed L behind A, as it would with CFS. However, since task A's warp (w(A)) is larger than L, A does not get preempted. Long-term throughput is still determined by the tasks' relative share weights.

assign a "warp" parameter for tasks (or CPU container groups). We additionally added a simple extension to BVT that uses Linux's high-resolution tick infrastructure (HRTICK) to re-evaluate scheduling decisions after a short grace period if a waking task barely misses preempting the current running task; this improves its worst-case scheduling error. Tasks with a warp of 0 behave no differently than regular CFS. The efficacy of BVT is readily apparent when the previous square-wave and latency-sensitive service co-location experiments are rerun using BVT as the scheduler. As seen in Figure 5.7, when memcached is run concurrently with the square-wave antagonist using BVT, memcached no longer suffers unacceptable tail latency at low loads. It is not until memcached exceeds 50% load (and starts using more than its fair-share of CPU time) that it begins to incur long latencies due to wait-time. In addition, warp values can be assigned rate-monotonically much as priorities can be assigned rate-monotonically in POSIX real-time scheduling. In the case of co-locating memcached and a 1ms latency-sensitive service, BVT achieves good QoS for both services up to high aggregate load (80%-90%), as seen in Figure 5.11.

Overall, there are several viable strategies to mitigate scheduling delay for latency-sensitive services when co-located with other workloads. When the co-located service is merely "best-effort", it is sufficient to run the latency-sensitive service with high CFS shares. If the co-located service is latency-sensitive or requires a guaranteed portion or fair-share of CPU throughput, a general-purpose scheduler with support for latency-sensitive tasks, like BVT, is required.

5.5 Co-location Benefits

The preceding sections have shown that a latency-sensitive workload like memcached can be quite resilient to interference caused by co-located workloads. In this section, we analyze the additional utility (performance, power-efficiency, and TCO) that can be extracted from clusters through co-location, either by deploying other workloads onto an existing memcached cluster, or deploying memcached onto an existing batch-compute cluster. We use a combined experimental/analytical approach. We measure latency vs. QPS for memcached

	Memcached QPS (% of peak)				
	10%	**30%**	**50%**	**70%**	**90%**
Workload	Memcached 95th-% Latency (usecs)				
none	122	132	156	209	387
400.perlbench	141	161	195	270	474
470.lbm	325	302	309	380	642
mean	217	215	242	307	512
Workload	CPU2006 Instr. per second (norm.)				
400.perlbench	74%	42%	22%	10%	14%
401.bzip2	79%	48%	41%	17%	22%
403.gcc	75%	41%	23%	14%	18%
410.bwaves	65%	29%	17%	9%	10%
416.gamess	78%	49%	28%	17%	17%
429.mcf	77%	54%	38%	28%	26%
433.milc	80%	56%	38%	29%	26%
434.zeusmp	73%	43%	24%	13%	14%
450.soplex	86%	53%	34%	18%	22%
459.GemsFDTD	79%	56%	38%	24%	22%
470.lbm	76%	53%	35%	23%	23%
471.omnetpp	61%	48%	40%	28%	39%
geo. mean	75%	47%	30%	18%	20%
Workload	Perf/TCO-$ improvement (%)				
geo. mean	52%	29%	18%	11%	18%

TCO [127] / server = server + power (3yrs). server = $2,000,
power = $(1 + K_1 + L_1 + K_2 * L_1) * U_{\$,grid} * E$.
$K_1/L_1/K_2 = 1.33/0.8/0.25$, $U_{\$,grid} = \$0.08$/kWh.
E = P * 24 hrs * 365 days * 3 yrs
Average power = P = 0.3 kW * ($0.5 + 0.5*$CPU util.)

Table 5.2: Memcached tail latency and SPEC CPU2006 instruction throughput when co-located under a range of memcached loads. Both memcached and SPEC run on all 24 threads of the server and are timesharing. For SPEC CPU throughput, all measurements are of instruction throughput normalized to the benchmark running alone. Perf/TCO-$ improvement compares co-location to operating distinct clusters for the two workloads.

with load varying 0% to 100%), while concurrently running SPEC CPU2006 benchmarks on the same server and measuring instruction throughput with Linux's `perf` tool (normalized to the benchmark running alone). We utilize the techniques discussed in Section 5.4 to minimize the interference between the co-located workloads. We use the results to calculate the utility of co-location under two different scenarios explained below.

5.5.1 Facebook Scenario

The first scenario represents the opportunity in companies like Facebook (see discussion in Section 5.2). Memcached is deployed on a large number of dedicated servers that have long periods of underutilization due to diurnal patterns and provisioning for spikes. We'd like to reclaim the unused server resources by also

running other workloads (e.g., analytics) as background tasks. Memcached is the high priority workload and there should be no impact on its QoS.

Table 5.2 presents the analysis, where the columns represent the memcached service load (as % of peak). The top portion of the table shows the 95th-percentile latency of memcached with and without co-located workloads. While this metric is affected by co-location, it never devolves into asymptotic queuing delay. If the QoS constraint is 1msec, we can co-locate memcached with other workloads at any load. If the QoS limit is 0.5msec, the most memory intensive workloads (470.lbm) can cause memcached to violate its QoS at high load loads (90%). Hence, a conservative approach may be to disallow colocation in the rare case where memcached reaches high levels of utilization.

The table also shows the performance achieved by the SPEC workloads. Since CPU utilization closely matches memcached load, we'd ideally expect a co-located workload to achieve (100-M)% of its peak throughput, where M% is the memcached load. However, SPEC workloads do not achieve ideal performance. 400.perlbench, for instance, only achieves 74.1% throughput when memcached is at 10% load. Curiously, many workloads (i.e. 401.bzip2) achieve slightly higher throughput at 90% memcached load compared to 70% load. We measured substantially fewer context-switches at 90% load even though the request rate was higher, indicating that memcached is servicing more requests per wakeup, and may be the result of interrupt rate throttling by the Intel 10GbE NIC. This results in less time spent scheduling and context-switching and fewer TLB flushes for the co-located workload.

Given the observations above, we calculate the benefit of co-location by comparing the TCO of a single cluster co-locating both of these workloads vs. operating two separate clusters for the two workloads. For the split scenario, we size the cluster for SPEC to match the SPEC throughput provided by the co-located cluster. For instance, if we assume 12,000 servers for memcached and since 400.perlbench achieves 74.1% throughput when co-located with memcached at 10% load, this throughput can be matched by 8,892 servers dedicated to running 400.perlbench. For TCO, we use the methodology of Patel et al. [91, 127] to compute the fully-burdened cost of power using the parameters in Table 5.2 and the assumptions that (1) server capital cost is $2,000, (2) server power consumption scales linearly with CPU utilization, and static power is 50% of peak power, (3) peak power consumption is 300W, and (4) datacenter PUE is 1.25. This methodology takes into account the fact that a server operating at higher utilization consumes more power, hence it has higher overall cost.

Despite the fact that SPEC workloads do not achieve ideal performance when co-located with memcached, the TCO benefit of co-location in all cases is positive and substantial. When memcached operates at 10% load, the co-located workloads achieve an average 75.4% of the throughput of an equivalent-sized cluster (12,000 servers). To match this performance with a separate cluster, it would cost an additional 52% increase in TCO, taking into account all capital and operational expenses. To put it differently, it is as-if we achieve the capability of 9,048 servers running SPEC at a 34% discount (1-$\frac{1}{.52+1}$). Even at 90% load for memcached, there is still a 18% TCO benefit to co-locating workloads.

	Memcached QPS (% of peak)				
	50%	**60%**	**70%**	**80%**	**90%**
Workload	Memcached 95th-% Latency (usecs)				
none	156	176	210	250	360
400.perlbench	156	177	206	266	380
401.bzip2	156	181	213	268	402
403.gcc	162	183	217	282	457
410.bwaves	163	188	233	386	4703
416.gamess	154	173	204	252	364
429.mcf	186	228	318	4,246	6,775
433.milc	178	210	291	904	5,686
434.zeusmp	170	197	236	367	3,143
450.soplex	175	206	274	497	5,410
459.GemsFDTD	201	253	408	6,335	6,896
470.lbm	208	274	449	7,799	6,429
471.omnetpp	167	191	234	320	2,568
mix	157	199	207	334	4,711
L3 μbench	156	197	246	5,400	4,944
Workload	CPU2006 Instr. per second (norm.)				
400.perlbench	99%	98%	98%	98%	98%
470.lbm	91%	90%	89%	88%	89%
mix	98%	98%	98%	96%	94%
geo. mean	98%	97%	96%	97%	97%

Table 5.3: Memcached tail latency and SPEC CPU2006 instruction throughput when co-located on distinct cores. Each workload gets half of the cores of the server. Memcached peak QPS is measured when running alone on its half of the cores. Shaded cells indicate an unacceptable drop in QoS.

5.5.2 Google Scenario

In the second scenario, we assume an underutilized cluster similar to the 12,000-server cluster analyzed by Reiss et al. [133] (20% CPU utilization, 40% memory utilization). We conservatively assume that half (50%) of the CPU cores in the cluster are utilized running some primary workload and we would like to evaluate the benefit of deploying memcached on the other available cores in order to take advantage of the $\sim$0.5 PB of unallocated memory in the cluster.

Again, we evaluate the efficacy of co-location by comparing a co-located cluster to a pair of separate clusters where one cluster runs the existing primary workload (SPEC workloads at 50% core utilization) and the other cluster runs memcached. The impact on latency and throughput of co-location at various memcached loads is presented in Table 5.3. For many workloads (perlbench, bzip2), the latency impact of co-location is negligible. For others (lbm, mcf, etc.), substantial queuing delay is observed at loads above 70%, as previously seen in Sec. 5.3.2. Moreover, the SPEC workloads in some cases also see a moderate slowdown (up to 12% for lbm). If we assume that 500 μsec 95th-% latency is the maximum latency we are willing to tolerate from memcached, and 10% is the maximum slowdown we are willing to tolerate from

the SPEC workloads, then the maximum memcached load we can provision for co-located servers is 60% of peak (*interference-aware provisioning*).

Were we to build a separate memcached cluster to accommodate this load, we would allow memcached to use all of the cores of those servers. Thus, we would need to increase the size of the cluster by an additional 30%, optimistically assuming that performance scales linearly with cores. Put another way, if our original cluster is 12,000 servers large and 50% is occupied with SPEC workloads, we can safely co-locate up 3,600 servers worth of additional memcached load to serve unallocated memory, and guarantee good quality of service for both workloads. Taking into account TCO and the pessimistic power assumption that the memcached service is always at 100% load, the co-located cluster achieves 17% improvement in TCO compared to two separate clusters with the same performance for the two workloads. Also note that the Google cluster studied by Reiss et al. had 60% of memory unallocated; a separate memcached cluster of this *capacity* would require 7,200 servers at substantially higher TCO.

5.6 Related Work

There have been several works addressing QoS for co-located workloads in warehouse-scale datacenters [26, 98, 99, 175, 176, 179]. Several of these, including Paragon, Bubble-Up, and Bubble-Flux, focus on identifying workloads which interfere with each other and avoiding co-locating them together. Our work is distinct in that we (1) concretely describe how this interference manifests in a canonical low-latency workload, and (2) accommodate co-locations that may cause substantial interference by considering the overall impact on load provisioning in our analytical study. Other works focus on minimizing latency or improving tail latency for specific low-latency services [75, 122]. They accomplish this by bypassing the OS and using user-level network stacks, which complicates workload consolidation; these services would invariably have low utilization during diurnal troughs, which is counter-productive for the goals of this work. Tessellation [19] and Akaros [134] improve QoS in operating systems through radical changes to the kernel and process abstractions. Our work instead delves deep into the challenge of co-locating workloads using existing OS abstractions, and presents specific examples of where the epoll interface and CFS fall short in Linux. Pisces [149] presents a holistic solution to QoS for membase; however, they do not consider workload co-location, just multi-tenancy for membase. Finally, hardware partitioning techniques have been studied for caches [130, 141], hyper-threads [57], and memory channels [114, 119]. Those works are orthogonal to our own.

5.7 Conclusions

In this paper, we address the conflict between co-locating workloads to increase the utilization of servers and the challenge of maintaining good quality-of-service for latency-sensitive services. We showed that workload co-location leads to QoS violations due to increases in queuing delay, scheduling delay, and thread

load imbalance. We also demonstrated concretely how and why these vulnerabilities manifest in a canonical low-latency service, memcached, and described several strategies to mitigate the interference due to them. Ultimately, we demonstrated that latency-critical workloads like memcached can be aggressively co-located with other workloads, achieve good QoS, and that such co-location can improve a datacenter's effective throughput per TCO-$ by up to 52%.

Chapter 6

Concluding Remarks

We conclude this dissertation with a discussion of the synergies amongst its contributions, an outline of future work, and a brief summary.

6.1 Synergy Amongst Contributions

All of the techniques presented herein are *orthogonal*; none depends upon or precludes the application of another. Indeed, the road forward to improve server power-efficiency will likely be a messy affair, requiring innovation on several fronts, as the sources of inefficiency are myriad. There are, however, a handful of synergies between the techniques we have proposed that merit attention.

First, aggressive application of per-core power gating (Chapter 3) leads to high utilization of some cores in a multi-core server. If that server hosts latency-sensitive applications, this may lead to a degradation in quality of service, even though the server still affords the same throughput. Such a degradation may be avoided, however, if the techniques presented in Chapter 5 are applied. Thus, designing latency-sensitive services to tolerate interference helps both to improve server utilization and to gracefully mitigate the inefficiency of low utilization.

Similarly, aggressive power-down of servers in distributed file systems (Chapter 4) leads to higher utilization of the remaining active servers. The performance consequences of this are less severe if those services are designed with high utilization in mind (Chapter 5).

Finally, main memory rank subsetting unexpectedly improved performance for several multi-programmed workloads (Chapter 2), despite the increase in memory access time it causes. This is due to the large number of independent banks it exposes to the memory controller, which allows it to perform better memory access scheduling. Should server utilization indeed increase due to workload co-location in future datacenters (Chapter 5), memory systems with rank subsetting are both more power-efficient and more performant.

6.2 Future Work

There are several opportunties for future work which builds upon the contributions of this dissertation. First, even though per-core power gating is now commonplace in general-purpose CPUs, it is actuated using reactive mechanisms (i.e. only gating core power after observing idleness). Consequently, it errs towards preserving performance rather than improving power-efficiency. However, since the hardware is now readily accessible, the opportunity exists to evaluate proactive policies, like our own, and provide empirical evidence regarding their impact on performance and power-efficiency. Moreover, since the transition speed for PCPG is faster than we anticipated, more aggressive policies than our own watermark policy can be implemented. Such policies could even be integrated with the OS scheduler. The methodology is straight-forward: build a power-measurement testbed to study a modern Nehalem-derived system and reimplement a controller for our policy (or other, more aggressive policies). Then, measure the performance, quality-of-service, and power consumption for a range of latency-sensitive and batch-oriented workloads which fractionally-utilize a server.

Second, our work on power-proportional Hadoop left many avenues for future exploration open. Indeed, Rabbit [4] and GreenHDFS [76] largely address the data layout and data availability issues we identified in Section 4.4, respectively. Other issues, however, remain unresolved. Most importantly, all related work in this area has focused on block-based storage systems. None has, to our knowledge, evaluated dynamic power management of higher-level storage systems, like HBase [5], BigTable [17], or similar column-oriented distributed databases. Similar to Hadoop and HDFS, these systems are all designed to be tolerant of complete server failure by migrating ownership of data among cluster nodes. However, the vast majority of nodes are still expected to be available at all times, so this fault-tolerance does not lend itself to effective power management. Thus, even if the servers are underutilized, they may not be gracefully quiesced. Future work in this area should identify a set of *design patterns* or *rules* in order to enable effective power-management of distributed services. Such patterns or rules should be advocated for existing and new data stores, or else cluster-scale power management will remain impractical for datacenters which host significant amounts of data.

Third, in our study of memcached, we identified that its static assignment of incoming connections to server threads ultimately precipitated its intolerance to interfering workloads. There appears to be an opportunity here to study alternative software architectures for multi-threaded servers, and evaluate how well they cope with different types of interference (i.e. load imbalance, co-scheduled tasks, etc.). Such architectures might use forms of work stealing, connection stealing, or even a broker to assign requests to workers. Ultimately, we would like to know if acceptable performance and quality of service can be achieved with current system call interfaces (i.e. epoll, kqueue, /dev/poll, etc.), or if a new interface should be proposed. As the pattern used by memcached is found in several other services (i.e. MySQL, REDIS, node.js, lighttpd, nginx, etc.), the result of such a study would be broadly applicable.

Finally, in keeping with the tradition of contemporary Computer Architecture research, this dissertation has treated all workloads "as given," and evaluated how we can improve server power-efficiency without

materially modifying them. This is a reasonable restriction, as it suggests that the solutions herein are general-purpose and widely applicable. However, when semiconductor scaling has finally run its course and every effort has been expended on optimizing the servers that workloads run on, the logical next step is to engineer more power-efficient workloads. Thus, there will eventually be considerable future work in improving the performance and power-efficiency of the storage systems, distributed processing frameworks, and user-facing services of future datacenters.

6.3 Summary

We began this dissertation by arguing that future scaling of datacenter capability depends upon improvements to server power-efficiency. The richness of contemporary online services is largely enabled by the amount of brute computing power we are able to bring to bear in warehouse-scale datacenters. Without growth in datacenter capability, such online offerings will stagnate. Even though there are viable ways to continue growing datacenter capability today, they largely entail additional cost; it is almost as-if computing power is a scarce natural resource which is becoming harder to come by. Thus, this dissertation proposed several novel strategies to improve datacenter power-efficiency, so as to perpetuate the process of technological innovation we are experiencing online today.

We have argued that future improvements to datacenter power-efficiency largely fit into three categories: reductions in server dynamic power consumption, reductions in server static power consumption, and increases to average server utilization. We then described in detail four proposals that span these categories: a novel architecture for main memory modules that improves dynamic power consumption, a new control mechanism for per-core power gating of multicore processors to reduce static power consumption, a modification to distributed file systems to ameliorate the inefficiency of underutilized clusters, and a blueprint to maintain good quality of service on highly utilized servers.

All told, we have found that there exists considerable opportunity to improve the power-efficiency of datacenters despite the failure of Dennard scaling. Through judicious focus on server power-efficiency, we can stave off stagnation in the growth of online services or an explosion of datacenter construction, at least for a time.

Bibliography

[1] AHN, J., EREZ, M., AND DALLY, W. J. The Design Space of Data-Parallel Memory Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2006).

[2] AHN, J., LEVERICH, J., SCHREIBER, R. S., AND JOUPPI, N. P. Multicore DIMM: An Energy Efficient Memory Module with Independently Controlled DRAMs. *IEEE Computer Architecture Letters 8*, 1 (2009).

[3] AMD. *BIOS and Kernel Developer's Guide for AMD NPT Family 0Fh Processors*, July 2007.

[4] AMUR, H., CIPAR, J., GUPTA, V., GANGER, G. R., KOZUCH, M. A., AND SCHWAN, K. Robust and Flexible Power-Proportional Storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010), ACM, pp. 217–228.

[5] APACHE. Hbase. http://hadoop.apache.org/hbase/.

[6] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th International Conference on Measurement and Modeling of Computer Systems* (2012).

[7] BARKHORDARIAN, V. Power MOSFET Basics. White paper, International Rectifier, 2008.

[8] BARROSO, L. A. Warehouse-Scale Computing: Entering the Teenage Decade, 2011. Plenary talk at the ACM Federated Computing Research Conference.

[9] BARROSO, L. A., CLIDARAS, J., AND HÖLZLE, U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 2nd Edition. *Synthesis Lectures on Computer Architecture 8*, 3 (2013).

[10] BARROSO, L. A., AND HOLZLE, U. The Case for Energy-Proportional Computing. *IEEE Computer 40*, 12 (2007), 33–37.

[11] BARROSO, L. A., AND HÖLZLE, U. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, 1st Edition. *Synthesis Lectures on Computer Architecture 4*, 1 (2009).

[12] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compiler Techniques* (2008).

[13] BIRCHER, W. L., AND JOHN, L. K. Analysis of Dynamic Power Management on Multi-Core Processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing* (2008).

[14] BOHR, M. The New Era of Scaling in an SoC World. In *Proceedings of the International Solid-State Circuits Conference* (2009).

[15] BORTHAKUR, D. The Hadoop Distributed File System: Architecture and Design, 2007.

[16] BROWN, R., WEBBER, C., AND KOOMEY, J. G. Status and Future Directions of the ENERGY STAR Program. *Energy 27*, 5 (2002), 505–520.

[17] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design & Implementation* (2006).

[18] CIDON, A., STUTSMAN, R., RUMBLE, S., KATTI, S., OUSTERHOUT, J., AND ROSENBLUM, M. MinCopysets: Derandomizing Replication In Cloud Storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013).

[19] COLMENARES, J. A., ET AL. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. In *Proc. of the Design Automation Conference* (2013).

[20] CROVELLA, M. E., AND BESTAVROS, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *IEEE/ACM Transactions on Networking 5* (1997), 835–846.

[21] DAI, N. Effects of Powering Low-Voltage, High-Current Load on Components from Power Design Perspective. In *Proceedings of the 15th Annual IEEE Applied Power Electronics Conference* (2000).

[22] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Commununications of the ACM 56*, 2 (Feb. 2013), 74–80.

[23] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design & Implementation* (2004).

[24] DEGRANDIS, W. Power Considerations in Siting Data Centers. `http://www.datacenterknowledge.com/archives/2011/02/08/power-considerations-in-siting-data-centers/`, December 2013.

[25] DELIMITROU, C., AND KOZYRAKIS, C. iBench: Quantifying Interference for Datacenter Workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization* (2013).

[26] DELIMITROU, C., AND KOZYRAKIS, C. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[27] DELL, T. J. The Benefits of Chipkill-Correct ECC for PC Server Main Memory. White paper, IBM Microelectronics Division, Nov 1997.

[28] DENNARD, R. H., CAI, J., AND KUMAR, A. A Perspective on Todays Scaling Challenges and Possible Future Directions. *IEEE Journal of Solid-State Electronics 51*, 4 (2007), 518–525.

[29] DENNARD, R. H., GAENSSLEN, F. H., RIDEOUT, V. L., BASSOUS, E., AND LEBLANC, A. R. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (1974), 256–268.

[30] DONALD, J., AND MARTONOSI, M. Techniques for Multicore Thermal Management: Classification and New Exploration. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture* (2006), pp. 78–88.

[31] DUDA, K. J., AND CHERITON, D. R. Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles* (1999), pp. 261–276.

[32] EIGLER, F. C., PRASAD, V., COHEN, W., NGUYEN, H., HUNT, M., KENISTON, J., AND CHEN, B. Architecture of Systemtap: A Linux Trace/Probe Tool, 2005.

[33] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual ACM International Symposium on Computer Architecture* (2011), pp. 365–376.

[34] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power Provisioning for a Warehouse-Sized Computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (2007), pp. 13–23.

[35] FEDERATION OF AMERICAN SCIENTISTS. Ohio-Class SSGN-726 Overview. `http://www.fas.org/programs/ssp/man/uswpns/navy/submarines/ssgn726_ohio.html`, Dec 12, 2013.

[36] FITZPATRICK, B. Distributed caching with memcached. *Linux Journal 2004*, 124 (2004).

[37] FLEISCHMANN, M. LongRun Power Management. White paper, Transmeta, 2001.

[38] FLYNN, M. J., AND HUNG, P. Microprocessor Design Issues: Thoughts on the Road Ahead. *IEEE Micro 25*, 3 (2005), 16–31.

[39] FRAILONG, J. M., JALBY, W., AND LENFANT, J. XOR-Schemes: A Flexible Data Organization in Parallel Memories. In *Proceedings of the International Conference on Parallel Processing* (1985).

[40] Gartner Says Efficient Data Center Design Can Lead to 300 Percent Capacity Growth in 60 Percent Less Space. `http://www.gartner.com/newsroom/id/1472714`.

[41] GEE, J., HILL, M. D., PNEVMATIKATOS, D. N., AND SMITH, A. J. Cache Performance of the SPEC92 Benchmark Suite. *IEEE Micro 13* (1993).

[42] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (2003), pp. 29–43.

[43] GHOSH, M., AND LEE, H.-H. S. Smart Refresh: An Enhanced Memory Controller Design for Reducing Energy in Conventional and 3D Die-Stacked DRAMs. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2007).

[44] GONZALEZ, R., AND HOROWITZ, M. Energy Dissipation in General Purpose Microprocessors. *IEEE Journal of Solid-State Circuits 31*, 9 (1996), 1277–1284.

[45] GOOGLE. Efficiency: How we do it. `https://www.google.com/about/datacenters/efficiency/internal/`, 2013.

[46] GOVIL, K., CHAN, E., AND WASSERMAN, H. Comparing Algorithm for Dynamic Speed-Setting of a Low-Power CPU. In *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking* (1995), pp. 13–25.

[47] GRAY, L., KUMAR, A., AND LI, H. Characterization of SPECpower_ssj2008 Benchmark. In *SPEC Benchmark Workshop* (2008).

[48] GRAYBILL, R., AND MELHEM, R. *Power Aware Computing*. Springer, 2002.

[49] GROSS, D., SHORTLE, J. F., THOMPSON, J. M., AND HARRIS, C. M. *Fundamentals of Queueing Theory*. Wiley, 2013.

[50] GUPTA, M. S., OATLEY, J. L., JOSEPH, R., WEI, G.-Y., AND BROOKS, D. M. Understanding Voltage Variations in Chip Multiprocessors Using a Distributed Power-Delivery Network. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2007), EDA Consortium, pp. 624–629.

[51] GURUMURTHI, S., SIVASUBRAMANIAM, A., KANDEMIR, M., AND FRANKE, H. Reducing Disk Power Consumption in Servers with DRPM. *IEEE Computer 36*, 12 (2003), 59–66.

[52] HAMILTON, J. Overall Data Center Costs. `http://perspectives.mvdirona.com/2010/09/18/OverallDataCenterCosts.aspx`, 2010.

[53] HATTORI, T., IRITA, T., ITO, M., YAMAMOTO, E., KATO, H., SADO, G., YAMADA, T., NISHIYAMA, K., YAGI, H., KOIKE, T., TSUCHIHASHI, Y., HIGASHIDA, M., ASANO, H., HAYASHIBARA, I., TATEZAWA, K., SHIMAZAKI, Y., MORINO, N., YASU, Y., HOSHI, T., MIYAIRI, Y., YANAGISAWA, K., HIROSE, K., TAMAKI, S., YOSHIOKA, S., ISHII, T., KANNO, Y., MIZUNO, H., YAMADA, T., IRIE, N., TSUCHIHASHI, R., ARAI, N., AKIYAMA, T., AND OHNO, K. Hierarchical Power Distribution and Power Management Scheme for a Single Chip Mobile Processor. In *Proceedings of the 43rd Annual Design Automation Conference* (2006), pp. 292–295.

[54] HAZUCHA, P., KARNIK, T., BLOECHEL, B., PARSONS, C., FINAN, D., AND BORKAR, S. Area-Efficient Linear Regulator with Ultra-Fast Load Regulation. *IEEE Journal of Solid-State Circuits 40*, 4 (2005), 933–940.

[55] HENNESSY, J., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann, 2011.

[56] HENNING, J. L. Performance Counters and Development of SPEC CPU2006. *ACM SIGARCH Computer Architecture News 35*, 1 (2007).

[57] HERDRICH, A., ILLIKKAL, R., IYER, R., SINGHAL, R., MERTEN, M., AND DIXON, M. SMT QoS: Hardware Prototyping of Thread-level Performance Differentiation Mechanisms. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism* (2012).

[58] HEWLETT-PACKARD. HP Power Regulator for Proliant Servers. `http://h18004.www1.hp.com/products/servers/management/ilo/power-regulator.html`, 2011.

[59] HEWLETT-PACKARD CORPORATION, INTEL CORPORATION, MICROSOFT CORPORATION, PHOENIX TECHNOLOGIES LTD., AND TOSHIBA CORPORATION. Advanced Configuration and Power Interface Specification, December 2005.

[60] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A. D., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation* (2011), pp. 22–22.

[61] HO, R., MAI, K., AND HOROWITZ, M. A. The Future of Wires. *Proceedings of the IEEE 89*, 4 (2001).

[62] HUANG, H., PILLAI, P., AND SHIN, K. G. Design and Implementation of Power-Aware Virtual Memory. In *Proceedings of the USENIX Annual Technical Conference* (2003).

[63] HUGHES, C. J., AND ADVE, S. V. A Formal Approach to Frequent Energy Adaptations for Multimedia Applications. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (2004), IEEE, p. 138.

[64] HUR, I., AND LIN, C. A Comprehensive Approach to DRAM Power Management. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture* (2008).

[65] INTEL. Enhanced SpeedStep Technology for the Intel Pentium M Processor. White paper, Intel Corp., 2004. `http://download.intel.com/design/network/papers/30117401.pdf`.

[66] INTEL. Voltage Regulator Module (VRM) and Enterprise Voltage Regulator-Down (EVRD) 10.1. Design guidelines, Intel Corp., 2005. `http://download.intel.com/support/processors/xeon/sb/30273203.pdf`.

[67] INTEL. Intel Core 2 Duo Processors and Intel Core 2 Extreme Processors on 45-nm Process. Datasheet, Intel Corp., 2008. `http://download.intel.com/design/mobile/datashts/31891401.pdf`.

[68] ISCI, C., BUYUKTOSUNOGLU, A., CHER, C.-Y., BOSE, P., AND MARTONOSI, M. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th Annual International Symposium on Microarchitecture* (2006), IEEE, pp. 347–358.

[69] JACOB, B., NG, S. W., AND WANG, D. T. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2007.

[70] JAMES, N., RESTLE, P., FRIEDRICH, J., HUOTT, B., AND MCCREDIE, B. Comparison of Split-Versus Connected-Core Supplies in the POWER6 Microprocessor. In *Proceedings of International Solid-State Circuits Conference* (2007), IEEE, pp. 298–299.

[71] JEDEC. DDR3 SDRAM Specification. JESD79-3B, 2007.

[72] JEYAKUMAR, V., ALIZADEH, M., MAZIERES, D., PRABHAKAR, B., KIM, C., AND AZURE, W. EyeQ: Practical Network Performance Isolation for the Multi-Tenant Cloud. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing* (2012), pp. 8–8.

[73] JOHNSON, T., AND NAWATHE, U. An 8-core, 64-thread, 64-bit Power Efficient SPARC SoC (Niagara2). In *Proceedings of the International Symposium on Physical Design* (2007).

[74] KAPLAN, J. M., FORREST, W., AND KINDLER, N. Revolutionizing Data Center Energy Efficiency. Technical report, McKinsey & Company, 2008.

[75] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012).

[76] KAUSHIK, R. T., AND BHANDARKAR, M. GreenHDFS: Towards an Energy-Conserving Storage-Efficient, Hybrid Hadoop Compute Cluster. In *Proceedings of the USENIX Annual Technical Conference* (2010).

[77] KELTCHER, C., MCGRATH, K., AHMED, A., AND CONWAY, P. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro 23*, 2 (2003).

[78] KENDALL, D. G. Stochastic Processes Occurring in the Theory of Queues and Their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics* (1953), 338–354.

[79] KIM, W., GUPTA, M., WEI, G.-Y., AND BROOKS, D. System Level Analysis of Fast, Per-Core DVFS using On-Chip Switching Regulators. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture* (2008).

[80] KNICKERBOCKER, J. U., ANDRY, P. S., BUCHWALTER, L. P., DEUTSCH, A., HORTON, R. R., JENKINS, K. A., KWARK, Y. H., MCVICKER, G., PATEL, C. S., POLASTRE, R. J., SCHUSTER, C., SHARMA, A., SRI-JAYANTHA, S. M., SUROVIC, C. W., TSANG, C. K., WEBB, B. C., WRIGHT, S. L., MCKNIGHT, S. R., SPROGIS, E. J., AND DANG, B. Development of Next-Generation System-on-Package (SOP) Technology based on Silicon Carriers With Fine-Pitch Chip Interconnection. *IBM Journal of Research and Development 49*, 4.5 (2005), 725–753.

[81] KONGETIRA, P., AINGARAN, K., AND OLUKOTUN, K. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE Micro 25*, 2 (2005), 21–29.

[82] KOOMEY, J. G. Estimating Total Power Consumption by Servers in the US and the World. Technical report, Lawrence Berkeley National Laboratory, Feb 2007.

[83] KOSONOCKY, S. Practical Power Gating and Dynamic Voltage/Frequency Scaling. In *Proceedings of Hot Chips: A Symposium on High Performance Chips* (2011).

[84] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. H. NapSAC: Design and Implementation of a Power-proportional Web Cluster. In *Proceedings of the First ACM SIGCOMM Workshop on Green Networking* (2010), ACM.

[85] LE SUEUR, E., AND HEISER, G. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the International Conference on Power Aware Computing and Systems* (2010), USENIX.

[86] LEBECK, A. R., FAN, X., ZENG, H., AND ELLIS, C. Power Aware Page Allocation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2000).

[87] LEVERICH, J., MONCHIERO, M., TALWAR, V., RANGANATHAN, P., AND KOZYRAKIS, C. Power Management of Datacenter Workloads Using Per-Core Power Gating. *IEEE Computer Architecture Letters 8*, 2 (2009), 48–51.

[88] LI, J., AND MARTINEZ, J. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture* (2006).

[89] LI, S., AHN, J., STRONG, R. D., BROCKMAN, J. B., TULLSEN, D. M., AND JOUPPI, N. P. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (Dec 2009).

[90] LIANG, X., WEI, G.-Y., AND BROOKS, D. ReVIVaL: A Variation-Tolerant Architecture Using Voltage Interpolation and Variable Latency. In *Proceedings of the 35th International Symposium on Computer Architecture* (2008).

[91] LIM, K., RANGANATHAN, P., CHANG, J., PATEL, C., MUDGE, T., AND REINHARDT, S. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the 35th International Symposium on Computer Architecture* (2008).

[92] LIU, C. L., AND LAYLAND, J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[93] LIU, H. A Measurement Study of Server Utilization in Public Clouds. In *Proceedings of the 9th International Conference on Dependable, Autonomic and Secure Computing* (2011), IEEE.

[94] LORCH, J. R., AND SMITH, A. J. Improving Dynamic Voltage Scaling Algorithms with PACE. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* (2001).

[95] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the Conference on Programming Language Design and Implementation* (2005), ACM SIGPLAN.

[96] MA, L., AMALI, A., KIYAWAT, S., MIRCHANDANI, A., HE, D., THAPAR, N., SODHI, R., SPRING, K., AND KINZER, D. New Trench MOSFET Technology for DC-DC Converter Applications. In *Proceedings of the 15th International Symposium on Power Semiconductor Devices and ICs* (2003).

[97] MARKOFF, J., AND HANSELL, S. Hiding in Plain Sight, Google Seeks More Power. *New York Times* (June 8, 2006).

[98] MARS, J., TANG, L., AND HUNDT, R. Heterogeneity in "Homogeneous" Warehouse-Scale Computers: A Performance Opportunity. *IEEE Computer Architecture Letters 10*, 2 (July 2011), 29–32.

[99] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture* (2011).

[100] MARTIN, M. M. K., SORIN, D. J., BECKMANN, B. M., MARTY, M. R., XU, M., ALAMELDEEN, A. R., MOORE, K. E., HILL, M. D., AND WOOD, D. A. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News 33*, 4 (2005).

[101] MATHEW, B. K., MCKEE, S. A., CARTER, J. B., AND DAVIS, A. Design of a Parallel Vector Access Unit for SDRAM Memory Systems. In *Proceedings of the 6th IEEE International Symposium on High Performance Computer Architecture* (2000).

[102] MCGHAN, H. SPEC CPU2006 Benchmark Suite. In *Microprocessor Report* (Oct 2006).

[103] MEISNER, D., GOLD, B. T., AND WENISCH, T. F. PowerNap: Eliminating Server Idle Power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (2009), ACM.

[104] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power Management of Online Data-intensive Services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (2011), ACM.

[105] MEISNER, D., WU, J., AND WENISCH, T. F. BigHouse: A Simulation Infrastructure for Data Center Systems. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software* (2012), IEEE.

[106] METZ, C. Facebook Lets You Spy on Its Data Centers. *Wired Enterprise* (Apr 18, 2013). `http://www.wired.com/wiredenterprise/2013/04/facebook-data-center-dashboard/`.

[107] MICRON TECHNOLOGY INC. *DDR3 SDRAM Datasheet*, 2006. Rev. K 04/10 EN, `http://www.micron.com/products/dram/ddr3/`.

[108] MICRON TECHNOLOGY INC. Calculating Memory System Power for DDR3. Technical report TN-41-01, Micron, 2007.

[109] MICRON TECHNOLOGY INC. *RLDRAM Datasheet*, 2008. `http://www.micron.com/products/dram/rldram/`.

[110] MILLER, R. Facebook Has Spent $210 Million on Oregon Data Center. *Data Center Knowledge* (Jan 30, 2012). `http://www.datacenterknowledge.com/archives/2012/01/30/facebook-has-spent-210-million-on-oregon-data-center/`.

[111] MINAS, L. The Problem of Power Consumption in Servers. Technical report, Intel, 2009.

[112] MOORE, G. E. Cramming More Components onto Integrated Circuits. *Electronics 38*, 8 (Apr 19, 1965).

[113] MOORE, J. Gamut v0.7.0. `http://issg.cs.duke.edu/cod/`.

[114] MUTLU, O., AND MOSCIBRODA, T. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2007).

[115] MUTLU, O., AND MOSCIBRODA, T. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Proceedings of the 35th International Symposium on Computer Architecture* (2008).

[116] NAFFZIGER, S., STACKHOUSE, B., GRUTKOWSKI, T., JOSEPHSON, D., DESAI, J., ALON, E., AND HOROWITZ, M. The Implementation of a 2-Core, Multi-Threaded Itanium Family Processor. *IEEE Journal of Solid-State Circuits 41*, 1 (2006).

[117] NARENDRA, S. G., AND CHANDRAKASAN, A. *Leakage in Nanometer CMOS Technologies*. Springer-Verlag, 2005.

[118] NELSON, M., LIM, B.-H., AND HUTCHINS, G. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference* (2005).

[119] NESBIT, K. J., AGGARWAL, N., LAUDON, J., AND SMITH, J. E. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006).

[120] NOMURA, S., TACHIBANA, F., FUJITA, T., TEH, C. K., USUI, H., YAMANE, F., MIYAMOTO, Y., KUMTORNKITTIKUL, C., HARA, H., YAMASHITA, T., TANABE, J., UCHIYAMA, M., TSUBOI, Y., MIYAMORI, T., KITAHARA, T., SATO, H., HOMMA, Y., MATSUMOTO, S., SEKI, K., WATANABE, Y., HAMADA, M., AND TAKAHASHI, M. A 9.7mW AAC-Decoding, 620mW H.264 720p 60fps Decoding, 8-Core Media Processor with Embedded Forward-Body-Biasing and Power-Gating Circuit in 65nm CMOS Technology. In *Proceedings of the International Solid-State Circuits Conference* (2008).

[121] OLUKOTUN, K., AND HAMMOND, L. The Future of Microprocessors. *ACM Queue 3*, 7 (2005).

[122] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., PARULKAR, G., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *ACM SIGOPS Operating Systems Review 43*, 4 (Jan. 2010).

[123] OWEN, J., AND STEINMAN, M. Northbridge Architecture of AMD's Griffin Microprocessor Family. *IEEE Micro 28*, 2 (2008).

[124] PABLA, C. S. Completely Fair Scheduler. *Linux Journal*, 184 (2009).

[125] PACHAURI, R. K. Climate Change 2007: Synthesis Report, 2008. Intergovernmental Panel on Climate Change, Geneva (Switzerland).

[126] PAN, H., ASANOVIĆ, K., COHN, R., AND LUK, C.-K. Controlling Program Execution through Binary Instrumentation. *SIGARCH Computer Architecture News 33*, 5 (2005).

[127] PATEL, C. D., AND SHAH, A. J. Cost Model for Planning, Development and Operation of a Data Center. Technical report HPL-2005-107R1, Hewlett-Packard Labs, 2005.

[128] PESTEREV, A., STRAUSS, J., ZELDOVICH, N., AND MORRIS, R. T. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012).

[129] PETERSON, W. W., AND E. J. WELDON, J. *Error-Correction Codes*, 2nd ed. MIT Press, 1972.

[130] QURESHI, M. K., AND PATT, Y. N. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006).

[131] RAGHAVENDRA, R., RANGANATHAN, P., TALWAR, V., WANG, Z., AND ZHU, X. No "Power" Struggles: Coordinated Multi-Level Power Management for the Data Center. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[132] RAMOS, L., AND BIANCHINI, R. C-Oracle: Predictive Thermal Management for Data Centers. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture* (2008).

[133] REISS, C., TUMANOV, A., GANGER, G. R., KATZ, R. H., AND KOZUCH, M. A. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012).

[134] RHODEN, B., KLUES, K., ZHU, D., AND BREWER, E. Improving Per-Node Efficiency in the Datacenter with New OS Abstractions. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011).

[135] RIVOIRE, S., RANGANATHAN, P., AND KOZYRAKIS, C. A Comparison of High-Level Full-System Power Models. In *Proceedings of the Workshop on Power Aware Computing and Systems* (2008), USENIX.

[136] RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P. R., AND OWENS, J. D. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture* (2000).

[137] ROGERS, A., KAPLAN, D., QUINNELL, E., AND KWAN, B. The Core-C6 (CC6) Sleep State of the AMD Bobcat x86 Microprocessor. In *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design* (2012).

[138] ROTEM, E., NAVEH, A., RAJWAN, D., ANANTHAKRISHNAN, A., AND WEISSMANN, E. Power-Management Architecture of the Intel Microarchitecture Code-named Sandy Bridge. *IEEE Micro 32*, 2 (2012).

[139] RUSU, S., TAM, S., MULJONO, H., AYERS, D., CHANG, J., CHERKAUER, B., STINSON, J., BENOIT, J., VARADA, R., LEUNG, J., LIMAYE, R., AND VORA, S. A 65-nm Dual-Core Multithreaded Xeon Processor With 16-MB L3 Cache. *IEEE Journal of Solid-State Circuits 42*, 1 (2007).

[140] SAAB, P. Scaling Memcached at Facebook. `https://www.facebook.com/note.php?note_id=39391378919`, December 2008.

[141] SANCHEZ, D., AND KOZYRAKIS, C. Scalable and Efficient Fine-Grained Cache Partitioning with Vantage. *IEEE Micro 32*, 3 (2012).

[142] SARWATE, D. V., AND SHANBHAG, N. R. High-Speed Architectures for Reed-Solomon Decoders. *IEEE Transactions on VLSI Systems 9*, 5 (2001).

[143] SCHALLER, R. R. Moore's law: Past, present and future. *IEEE Spectrum 34*, 6 (1997).

[144] SCHROEDER, B., PINHEIRO, E., AND WEBER, W.-D. DRAM Errors in the Wild: A Large-Scale Field Study. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems* (2009), ACM.

[145] SCHWAN, P. Lustre: Building a File System for 1000-Node Clusters. In *Proceedings of the Linux Symposium* (2003).

[146] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013).

[147] SHEN, J. P., AND LIPASTI, M. H. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, 2005.

[148] SHERWOOD, T., PERELMAN, E., HAMERLY, G., AND CALDER, B. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2002).

[149] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design & Implementation* (2012).

[150] SINGHAL, R. Inside Intel Core Microarchitecture (Nehalem). In *Proceedings of Hot Chips: A Symposium on High Performance Chips* (2008).

[151] STANSBERRY, M., AND KUDRITZKI, J. Uptime Institute 2012 Data Center Industry Survey. White paper, Uptime Institute, 2012.

[152] STINSON, J., AND RUSU, S. A 1.5GHz Third Generation Itanium Processor. In *Proceedings of the International Solid-State Circuits Conference* (2003), IEEE.

[153] SUDAN, K., CHATTERJEE, N., NELLANS, D., AWASTHI, M., BALASUBRAMONIAN, R., AND DAVIS, A. Micro-Pages: Increasing DRAM Efficiency with Locality-Aware Data Placement. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2010).

[154] THOZIYOOR, S., AHN, J., MONCHIERO, M., BROCKMAN, J. B., AND JOUPPI, N. P. A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies. In *Proceedings of the 35th International Symposium on Computer Architecture* (2008).

[155] THOZIYOOR, S., MURALIMANOHAR, N., AHN, J., AND JOUPPI, N. P. CACTI 5.1. Technical Report HPL-2008-20, Hewlett-Packard Labs, 2008.

[156] TOLIA, N., WANG, Z., MARWAH, M., BASH, C., RANGANATHAN, P., AND ZHU, X. Delivering Energy Proportionality with Non Energy-Proportional Systems – Optimizing the Ensemble. In *Proceedings of the 1st Workshop on Power Aware Computing and Systems* (2008), USENIX.

[157] TSCHANZ, J., NARENDRA, S., YE, Y., BLOECHEL, B., BORKAR, S., AND DE, V. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE Journal of Solid-State Circuits 38*, 11 (2003).

[158] TURNER, P., RAO, B. B., AND RAO, N. CPU Bandwidth Control for CFS. In *Proceedings of the Linux Symposium* (2010).

[159] TURNER IV, W. P., AND BRILL, K. G. Cost Model: Dollars per kW Plus Dollars per Square Foot of Computer Floor. White paper, Uptime Institute, 2008.

[160] TURNER IV, W. P., SEADER, J. H., RENAUD, V., AND BRILL, K. G. Tier Classifications Define Site Infrastructure Performance. White paper, Uptime Institute, 2008.

[161] UDIPI, A. N., MURALIMANOHAR, N., CHATTERJEE, N., BALASUBRAMONIAN, R., DAVIS, A., AND JOUPPI, N. P. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *Proceedings of the 37th International Symposium on Computer Architecture* (2010).

[162] U.S. ENERGY INFORMATION ADMINISTRATION. Electric Power Monthly with Data for September 2012. `http://www.eia.gov/electricity/monthly/`.

[163] VAID, K. Datacenter Power Efficiency: Separating Fact From Fiction. In *Proceedings of the Workshop on Power Aware Computing and Systems* (2010), USENIX. Invited talk.

[164] VASAN, A., SIVASUBRAMANIAM, A., SHIMPI, V., AND SIVABAL, T. Worth Their Watts? An Empirical Study of Datacenter Servers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture* (2010).

[165] VMWARE. VMware Infrastructure: Resource Management with VMware DRS. White paper, VMware, 2006.

[166] WARE, F. A., AND HAMPEL, C. Improving Power and Data Efficiency with Threaded Memory Modules. In *Proceedings of the International Conference on Computer Design* (2006).

[167] WEDDLE, C., OLDHAM, M., QIAN, J., WANG, A.-I. A., REIHER, P., AND KUENNING, G. PARAID: A Gear-Shifting Power-Aware RAID. *ACM Transactions on Storage 3*, 3 (2007).

[168] WHITE, T. *Hadoop: The Definitive Guide*. O'Reilly, 2012.

[169] WOO, S. C., OHARA, M., TORRIE, E., SINGH, J. P., AND GUPTA, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture* (1995).

[170] WU, Q., MARTONOSI, M., CLARK, D. W., REDDI, V. J., CONNORS, D., WU, Y., LEE, J., AND BROOKS, D. Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance. *IEEE Micro 26*, 1 (2006).

[171] WU, W., DEMAR, P., AND CRAWFORD, M. Why Can Some Advanced Ethernet NICs Cause Packet Reordering? *IEEE Communications Letters 15*, 2 (2011).

[172] XAMBO-DESCAMPS, S. *Block Error-Correcting Codes: A Computational Primer*. Springer, 2003.

[173] XIE, F., MARTONOSI, M., AND MALIK, S. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (2003).

[174] XIE, F., MARTONOSI, M., AND MALIK, S. Bounds on Power Savings Using Runtime Dynamic Voltage Scaling: An Exact Algorithm and a Linear-Time Heuristic Approximation. In *Proceedings of the International Symposium on Low Power Electronics and Design* (2005).

[175] XU, Y., MUSGRAVE, Z., NOBLE, B., AND BAILEY, M. Bobtail: avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation* (2013).

[176] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th International Symposium on Computer Architecture* (2013).

[177] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster Computing With Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (2010).

[178] ZHANG, K., BHATTACHARYA, U., CHEN, Z., HAMZAOGLU, F., MURRAY, D., VALLEPALLI, N., WANG, Y., ZHENG, B., AND BOHR, M. SRAM Design on 65-nm CMOS Technology With Dynamic Sleep Transistor for Leakage Reduction. *IEEE Journal of Solid-State Circuits 40*, 4 (2005).

[179] ZHANG, X., TUNE, E., HAGMANN, R., JNAGAL, R., GOKHALE, V., AND WILKES, J. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013).

[180] ZHANG, Z., ZHU, Z., AND ZHANG, X. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture* (2000).

[181] ZHAO, W., AND CAO, Y. Predictive Technology Model for Nano-CMOS Design Exploration. *ACM Journal on Emerging Technologies in Computing Systems 3*, 1 (2007).

[182] ZHENG, H., LIN, J., ZHANG, Z., GORBATOV, E., DAVID, H., AND ZHU, Z. Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* (2008).