Boston University

OpenBU | http://open.bu.edu

Boston University Theses & Dissertations | Boston University Theses & Dissertations

2015

Adaptive runtime techniques for power and resource management on multi-core systems

https://hdl.handle.net/2144/13682

"Downloaded from OpenBU. Boston University's institutional repository."

BOSTON UNIVERSITY

COLLEGE OF ENGINEERING

Dissertation

ADAPTIVE RUNTIME TECHNIQUES FOR POWER AND RESOURCE MANAGEMENT ON MULTI-CORE SYSTEMS

by

CAN HANKENDI

B.S., Sabanci University, 2008
M.Sc., University of Southern California, 2010

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2015

© 2015 by
Can Hankendi
All rights reserved.

Approved by

First Reader

Ayse K. Coskun, Ph.D
Associate Professor of Electrical and Computer Engineering

Second Reader

Martin Herbordt, Ph.D
Professor of Electrical and Computer Engineering

Third Reader

Ajay Joshi, Ph.D
Assistant Professor of Electrical and Computer Engineering

Fourth Reader

Sherief Reda, Ph.D
Associate Professor of School of Engineering, Brown University

Two roads diverged in a wood, and I–
I took the one less traveled by,
And that has made all the difference.

Robert Frost

Acknowledgments

First, I would like to thank my advisor, Professor Ayse K. Coskun, for her support and encouragement during my Ph.D studies.

I would like to thank Professor Martin Herbordt for his valuable feedback and guidance that contributed to the first two publications of my graduate student career. I thank Professor Ajay Joshi for his advice and comments during my studies at Boston University. I would like to also thank Professor Sherief Reda for the experience I gained throughout our collaboration, which led to three publications.

I would also like to thank Manish Arora and Dr. Wei Huang for the summer internship opportunity at Advanced Micro Devices, Inc. (AMD) .

I would like to thank my fellow lab mates, co-authors, and friends at Boston University for their friendship and for their encouragement. I would like to send my special thanks to Onur Sahin and Ata Turk for proof-reading the thesis.

I would like to thank my dearest friends, Nuri K., Elliot S., Dave Robert J. I would not be able to overcome the constant stress and pressure without their help and support.

Finally, I would like to give special thanks to my family for their unconditional support.

The research that forms the basis of this dissertation has been partially funded by VMware, Inc., BU College of Engineering Dean's Catalyst Award and Massachusetts Green High-Performance Computing Center (MGHPCC) seed funds.

The content of Chapter 3 is in part a reprint of the material from the papers, *Can Hankendi and Ayse Coskun, "Reducing the Energy Cost of Computing Through Efficient Co-scheduling of Parallel Workloads"*, in Proceedings of Design Automation and Test in Europe Conference (DATE), 2012.

The contents of Chapter 4 are in part reprints of the material from the papers *Can*

Hankendi and Ayse Coskun, "Reducing the Energy Cost of Computing through Efficient Co-scheduling of Parallel Workloads", in Proceedings Design Automation and Test in Europe (DATE), 2012, *Can Hankendi and Ayse Coskun, "Energy-efficient Server Consolidation for Multi-threaded Applications in the Cloud"*, in Proceedings International Green Computing Conference (IGCC), 2013, and, *Can Hankendi and Ayse Coskun, "Autonomous Resource Sharing For Multi-threaded Workloads In Virtualized Servers"*, in VMware Technical Journal, Volume III, 2013.

The contents of Chapter 5 are in part reprints of the material from the papers, *Ryan Cochran, Can Hankendi, Ayse Coskun and Sherief Reda, "Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps"*, in International Symposium on Microarchitecture (MICRO), 2011, *Can Hankendi, Sherief Reda and Ayse Kivilcim Coskun , "vCap: Adaptive Power Capping for Virtualized Servers"*, in IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2013, and *Can Hankendi, Henry Hoffmann, Ayse Coskun, "Adapt&Cap: Coordinating System and Application-level Adaptation for Power Constrained Systems"*, in IEEE Design&Test Magazine, 2015.

ADAPTIVE RUNTIME TECHNIQUES FOR POWER AND RESOURCE MANAGEMENT ON MULTI-CORE SYSTEMS

CAN HANKENDI

Boston University, College of Engineering, 2015

Major Professor: Ayse K. Coskun, PhD, Associate Professor of Electrical and Computer Engineering

ABSTRACT

Energy-related costs are among the major contributors to the total cost of ownership of data centers and high-performance computing (HPC) clusters. As a result, future data centers must be energy-efficient to meet the continuously increasing computational demand. Constraining the power consumption of the servers is a widely used approach for managing energy costs and complying with power delivery limitations. In tandem, virtualization has become a common practice, as virtualization reduces hardware and power requirements by enabling consolidation of multiple applications on to a smaller set of physical resources. However, administration and management of data center resources have become more complex due to the growing number of virtualized servers installed in data centers. Therefore, designing autonomous and adaptive energy efficiency approaches is crucial to achieve sustainable and cost-efficient operation in data centers.

Many modern data centers running enterprise workloads successfully implement energy efficiency approaches today. However, the nature of multi-threaded applications, which are becoming more common in all computing domains, brings additional design and management challenges. Tackling these challenges requires a deeper understanding of the interactions between the applications and the underlying hardware nodes. Although cluster-level management techniques bring significant benefits, node-level techniques provide more visibility into application characteristics, which can then be used to further improve the overall energy efficiency of the data centers.

This thesis proposes adaptive runtime power and resource management techniques on multi-core systems. It demonstrates that taking the multi-threaded workload characteristics into account during management significantly improves the energy efficiency of the server nodes, which are the basic building blocks of data centers. The key distinguishing features of this work are as follows:

We implement the proposed runtime techniques on state-of-the-art commodity multi-core servers and show that their energy efficiency can be significantly improved by (1) taking multi-threaded application specific characteristics into account while making resource allocation decisions, (2) accurately tracking dynamically changing power constraints by using low-overhead application-aware runtime techniques, and (3) coordinating dynamic adaptive decisions at various layers of the computing stack, specifically at system and application levels. Our results show that efficient resource distribution under power constraints yields energy savings of up to 24% compared to existing approaches, along with the ability to meet power constraints 98% of the time for a diverse set of multi-threaded applications.

Contents

1 Introduction **1**

1.1 Challenges for Multi-threaded Application Management 4

1.2 Node-level Analysis and Management 4

1.3 Thesis Contributions 7

2 Background and Related Work **10**

2.1 Workload and VM Placement 10

2.2 Resource Allocation 13

2.3 Power Management 15

2.4 Distinguishing Aspects 17

3 Instrumenting Multi-core Servers **19**

3.1 Overview of the Experimental Setup 19

3.2 Systems Under Test 20

3.3 Hardware Event Measurements 20

3.4 Application-specific Performance Measurement 23

3.5 Virtualized Environment Setup 23

3.6 Benchmarking Methodology 24

3.7 Summary of Experimental Methodology 28

4 Resource Allocation and Consolidation for Energy Efficiency **29**

4.1 Co-scheduling Analysis on Native Environments 29

4.1.1 Multi-level Co-scheduling Policy 33

4.1.2 Experimental Results 35

4.2 Co-scheduling Analysis in Virtualized Environments 38
4.2.1 Application Selection Based Co-scheduling 38
4.2.2 Performance Isolation on Consolidated Environments 41
4.3 Adaptive Resource Sharing for Multi-threaded Workloads 44
4.3.1 Predicting Application Energy Efficiency 45
4.3.2 Autonomous Resource Sharing 50
4.3.3 Runtime Implementation 52
4.3.4 Consolidation with Throughput Constraints 53
4.4 Experimental Results 55
4.4.1 Runtime Behavior 55
4.4.2 Evaluation for Various Cluster Workload Sets 57
4.4.3 Consolidation with a Higher Number of VMs 60
4.5 Chapter Summary 61

5 Dynamic Power Capping **62**

5.1 Power Capping on Native Environments 63
5.1.1 Pack & Cap Methodology 63
5.1.2 Experimental Results 64
5.2 Power Capping on Virtualized Environments 66
5.2.1 Adaptive Power Capping on Virtualized Environments 71
5.2.2 Estimating QoS Degradation Under Power Caps 74
5.2.3 Consolidation Based On Performance Scalability 76
5.2.4 Experimental Results 77
5.3 Scale & Cap: Scaling-Aware Resource Management for Consolidated Multi-threaded Applications 80
5.3.1 LP Solution to Resource Distribution 85

5.3.2 Maximizing Server-QoS with Power Constraints 87
5.3.3 Runtime Implementation of Scale & Cap 90
5.3.4 Evaluating Resource Allocation Techniques 91
5.3.5 The Impact of VM Density on Placement Techniques 95
5.4 Coordinating System and Application-level Adaptations for Power Constrained Systems 99
5.4.1 Benefits of Coordinating System and Application-level Adaptation 100
5.4.2 Adapt & Cap: Unifying System and Application-level Adaptation 102
5.4.3 Power Tracking Performance 107
5.5 Chapter Summary 110

6 Conclusions 112

6.1 Summary of Major Contributions 112
6.2 Open Problems 114
6.2.1 Improving Boosting Algorithms 114
6.2.2 Cluster-level Management 115

References 116

Curriculum Vitae 125

List of Tables

3.1 Our experimental infrastructure consists of three state-of-the-art servers with AMD and Intel multi-core processors. This table summarizes the specifications of the processors. 21
3.2 Summary of characteristics of PARSEC Benchmarks (Bienia et al., 2008). 26

5.1 Definitions of the abbreviations used in the LP solution. 90

List of Figures

1·1 Historical data and future projections for power & cooling expenses, management cost, server costs (left axis) and number of physical and logical servers installed on data centers (right axis). The significant increase in number of logical servers creates the *virtualization management gap*, which refers to the increasing management complexity (IDC, 2011). 3

3·1 Experimental Setup 20

3·2 ROI-Synchronization flow. 27

4·1 Minimum and maximum change in E/w for 6 thread consolidated PARSEC benchmarks w.r.t 12 thread execution on a native environment. 30

4·2 First two principal component coefficients for various performance metrics when running PARSEC benchmarks. 31

4·3 Performance characteristics of randomly generated workload sets and the optimum policy for each workload set. 35

4·4 IPC * CPU Utilization for PARSEC benchmarks running 12 threads. 36

4·5 Average E/w saving improvements for 4 policies w.r.t 12 thread execution on a single node. 37

4·6 Maximum E/w saving improvements due multi-level policy w.r.t previously proposed policies. 37

4·7 Performance comparison when `canneal` is co-scheduled with the other PARSEC benchmarks shown on the x axis. Figure demonstrates `canneal`'s throughput (per core) when each of the two co-scheduled applications runs with 2 threads and when each has 6 threads. Performance impact of co-scheduling is higher at 2 threads due to higher resource contention at the shared caches. 39
4·8 Throughput-per-watt for various benchmark sets that are co-scheduled with various policies. On average, randomly co-scheduling applications provides comparable energy efficiency in comparison to previously proposed based policies. 40
4·9 In this experiment, a PARSEC benchmark is co-scheduled with another benchmark (only two benchmarks at a time) under various CPU binding and NUMA balancing settings. The experiment is repeated to cover all possible application pairings. Figure shows the performance variation (standard deviation/mean) of each benchmark across its co-scheduled runs with the other benchmarks. Smaller bars indicate better performance isolation. 43
4·10 Maximum, minimum and average throughput across all co-scheduled pairs for the native and virtual system. Figure shows the effect of CPU binding and NUMA balancer on the performance. 44
4·11 Maximum, minimum and average errors for predicting energy efficiency of PARSEC benchmarks for candidate metrics. 46
4·12 Correlation between *IPC*CPU Utilization* (right axis) and application energy efficiency (left axis). 47

4·13 Benchmark classification through density based clustering. Class-4 represents the highest level of energy efficiency, where as Class-1 represents the lowest level. 49
4·14 Runtime implementation of the resource allocation technique (Hankendi and Coskun, 2013). 52
4·15 Throughput degradation of 4 PARSEC benchmarks as a function of CPU resource limits. 55
4·16 Runtime behavior of the resource allocation routine for 3 applications pairs. CPU resources are adjusted according to power efficiency classes of the applications to improve the overall efficiency of the server. . . . 57
4·17 Runtime behavior of the resource allocation routine with performance guarantees. 57
4·18 Normalized throughput-per-watt with respect to the baseline case, where each VM is given the maximum resources, for randomly generated 50 workload sets. 58
4·19 Average throughput-per-watt, throughput, power and energy comparison normalized w.r.t baseline case. 59
4·20 Normalized throughput-per-watt with respect to the baseline case, where each VM is given the maximum resources, for varying number of co-scheduled applications. Energy efficiency improvements decrease with increasing number of applications. 61

5·1 Demonstration of DVFS and thread-packing control for `bodytrack` under changing power caps (Cochran et al., 2011). 65
5·2 Performance scaling of some of the PARSEC and benchmarks and `hadoop` as a function of CPU resource limits. 67

5·3 Overall normalized QoS of two distinct co-scheduling cases under various power caps. QoS range for the *scaling VMs* is much smaller than the *non-scaling VMs*. Thus, selecting non-scaling VMs to co-schedule have high potential for energy efficiency improvement. 68

5·4 Memory-boundedness (last level cache misses per cycle) vs. scalability of PARSEC and CloudSuite applications. Scalability is measured as the ability to utilize the 12-core system when running with 12 threads. This experiment shows that memory-boundedness does not capture the scalability characteristics of the applications. 69

5·5 QoS of the VM running `canneal` when consolidating with all the other VMs in pairs of two. Performance of `canneal` is not significantly affected by any of the co-runners. 71

5·6 Performance overhead for running higher number of threads under power caps. Running applications with 12 threads and applying resource limits introduces large overheads for some of the applications. Packing the threads onto smaller number of vCPUs reduces the overhead up to 45%. 73

5·7 QoS as a function of CPU resource limits for `blackscholes` and `dedup`. Degradation estimations are derived by using Equation 5.1. Equation 5.1 provides better QoS prediction without requiring any offline/training phase when compared to polynomial models. 75

5·8 Comparison of QoS, QoS/watt and power consumption of the server with various consolidation techniques. The proposed technique improves the overall QoS by 17% when compared to the baseline case, where VMs have no CPU resource limitations. 78

5·9 Runtime behavior of *vCap* under power caps and QoS constraints for the VM group running `dedup` and `hadoop`. *vCap* adheres to the power cap and ensures that the QoS guarantees are met. 79
5·10 Peak power (left axis, red) and power weight (right axis, blue) values for 4 PARSEC benchmarks and the PARSEC average measured on AMD Opteron 6172. 81
5·11 Total QoS comparison for consolidating two application pairs, `canneal-facesim` (a) and `blackscholes-swaptions` (b) on AMD Opteron 6172 with various power caps for utilization-based approach (baseline), naive approach (only scaling-aware) and power-aware (scaling and power-aware) approach. Power-awareness brings up to 18% and 11% QoS improvements over the scaling-only approach. 84
5·12 LP-solution for resource distribution across two applications ($m = 2$) with various lower and upper bounds for a given amount of resources R_k. 88
5·13 Performance variation for all applications for various VM density cases. Higher VM-density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity. 93
5·14 Performance variation for all applications for various VM density cases. Higher VM-density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity. 93
5·15 Performance variation for all applications for various VM density cases. Higher VM-density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity. 96
5·16 Comparison of placement techniques in terms of performance degradation (i.e., lower is better). 97

5·17 Best performing placement technique for various VM-density and active memory size. Memory-based technique is superior to other techniques with increasing number of VMs consolidated at the same time, which also leads to higher active memory size. 98
5·18 Power and performance tradeoff space for various adaptive techniques on Intel Xeon E5 multicore server when running x264. Proposed coordinated management extends the Pareto-optimal curve to a more efficient operating point. 101
5·19 Adapt & Cap reads heartbeat rates and power measurements, and chooses the amount of CPU resources required and the optimum application state. 104
5·20 Pseudo-code for Adapt & Cap control modules. Adapt &Cap first discovers the higher performance application state (blue box), then periodically checks power (green box) and performance (red box) requirements to adjusts its decisions. 105
5·21 Uncoordinated approach shows oscillatory behavior, as system and application adaptation controls are not aware of other decisions that impact the performance of the system significantly. 107
5·22 Comparison of power consumption for Adapt & Cap and only adaptive application under dynamically changing performance constraints. Adapt & Cap reduces the power consumption up to 27% compared to only application-level adaptation. 109
5·23 Performance results for Adapt & Cap under dynamically changing power constraints. Adapt & Cap improves the performance up to 2.7x compared to vCap under the same power constraints. 110

List of Abbreviations

CRM	....	Customer Relationship Management
DBSCAN	....	Density-based Spatial Clustering of Applications with Noise
DPM	....	Distributed Power Manager
DRS	....	Dynamic Resource Scheduler
DVFS	....	Dynamic Voltage-Frequency Scaling
ED	....	Energy-delay
EDP	....	Energy-delay-product
HPC	....	High-performance Computing
IPC	....	Instructions-per-cycle
ISO	....	Independent Service Operator
KVM	....	Kernel Virtual Machine
LUT	....	Lookup Table
MPC	....	Memory Access-per-cycle
MPI	....	Message-passing Interface
NUMA	....	Non-uniform Memory Access
PCA	....	Principal Component Analysis
QoS	....	Quality-of-Service
RAPL	....	Runtime-average Power Limiter
ROI	....	Region-of-interest
SLA	....	Service-level Agreement
SMP	....	Simultaneous Multiprocessing Processors
VM	....	Virtual Machine

Chapter 1

Introduction

Computer systems have evolved from room-sized machines to single-chip many-core systems throughout the last 60 years. In the last decade, cloud computing has become the new computing paradigm that enables sharing computing resources to provide a variety of services to many users. The user demand on the cloud has inevitably increased in recent years, as vast amount of services are provided through cloud resources (IDC, 2011). It is predicted that 78% of all workloads will be executed on cloud resources by 2018 (Cisco, 2013).

Data centers are the main facilitators of the cloud services. In order to meet the increasing user demand on cloud services, the number of servers in data centers has been tripled in the last decade (IDC, 2009). However, the achievable maximum performance of a data center is determined by not only the amount of available hardware resources (e.g., CPU, memory, I/O), but also by the infrastructural limitations (e.g., power delivery, cooling capacity) and operational costs. In fact, energy-related costs and challenges are the major limiting factors for today's data centers (IDC, 2011). Optimizing the performance under power and cost constraints is critically important to provide reliable operation and reduce the cost of computing in data centers. Therefore, future data centers are required to be energy-efficient to meet the continuously increasing computational demand.

As a result of the power delivery and cost limitations, constraining the power consumption of the servers (i.e., *power capping*) in data centers has become a common

practice (Nathuji and Schwan, 2008). In addition to traditional motivations for power capping, recent trends in energy markets provide significant opportunities in cost savings by offering new pricing mechanisms and advanced power market features for electricity (Chen et al., 2013). For example, in the regulation service reserves program (Chen et al., 2013), independent service operators (ISOs) require the participant data center to closely track the dynamically changing power constraints and offer cost reduction based on power tracking performance of the data center. Aforementioned reasons provide a strong motivation to develop accurate power capping techniques for modern data centers.

In tandem with the growth of data centers, virtualization has become one of the main enablers for designing cost-efficient data centers, as it allows to reduce the number of active servers by enabling consolidation of multiple workloads on a single physical server. In other words, virtualization makes it possible to enclose multiple *isolated execution environments* called *virtual machines* (VMs) on a single physical server. Through VMs, virtual environments provide isolated and secure execution for multiple users on the same underlying physical environment (i.e., physical server or host). Therefore, in recent years, virtualizing the data center resources has also become another common practice.

Virtualization not only simplifies the sharing of the physical resources, but also enables flexible management of the VMs through control knobs (e.g., using resource allocation reserves/limits or VM migration) (Vecchiola, C. and Pandey, S. and Buyya, R., 2009). As a result, the number of virtualized servers started to outnumber the native (not virtualized) servers in recent years (IDC, 2009). Although data center virtualization brings many benefits, it also introduces new challenges to energy-efficient management of power and compute resources. For example, Figure 1·1 shows the financial trends (left axis) and the number of servers (right axis) from 1996 to 2013

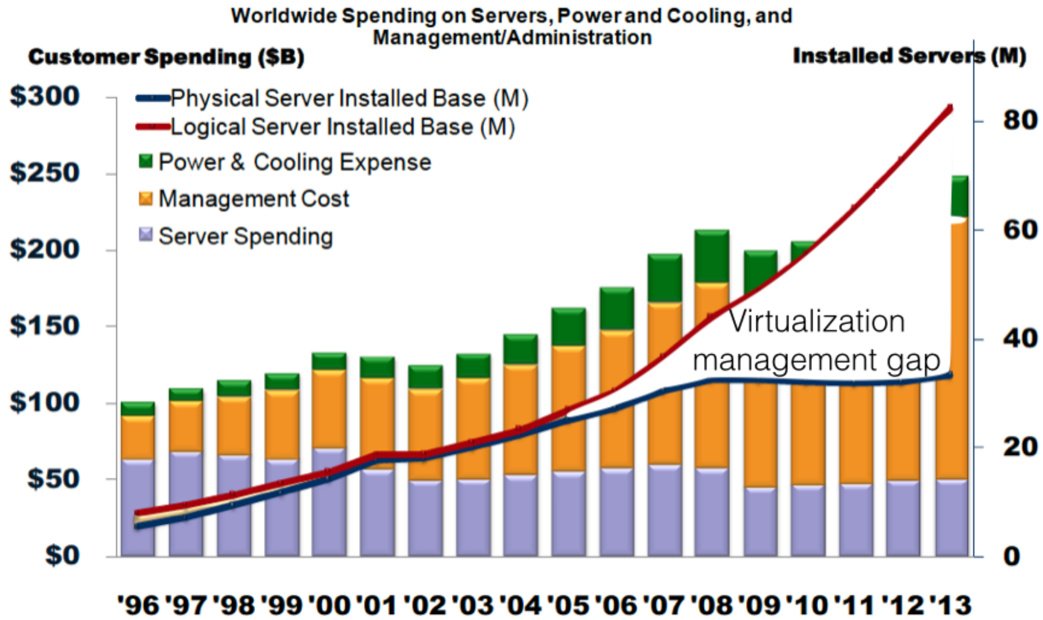

Figure 1·1: Historical data and future projections for power & cooling expenses, management cost, server costs (left axis) and number of physical and logical servers installed on data centers (right axis). The significant increase in number of logical servers creates the *virtualization management gap*, which refers to the increasing management complexity (IDC, 2011).

(IDC, 2011). According to this figure, with the introduction of hardware-assisted virtualization in 2005, the number of *logical servers* has doubled compared to the number of *physical servers*, since each physical server can contain multiple logical (or virtual) servers. The gap between the number of physical servers and virtual servers is called the *virtualization management gap*, which manifests itself as the increasing management costs (orange bar) of the data centers. Thus, as a consequence of virtualization, administration and management of the data center resources have become more complex. Therefore, the ability to manage data center resources autonomously has become a crucial design goal for data centers.

1.1 Challenges for Multi-threaded Application Management

Many modern data centers running enterprise workloads (e.g., transactional workloads, batch processing, mail servers, customer relationship management (CRM) softwares, etc.) successfully implement energy and resource management techniques today. However, many computing domains employ multi-threaded applications to efficiently utilize the underlying hardware parallelism. Although traditionally private clouds and grids used to be preferable than cloud resources for highly computational loads, advanced virtualization techniques and hardware support for virtualization make it possible to provide comparable performance to native (not virtualized) systems for computationally intensive multi-threaded workloads (Macdonell and Lu, 2008). As a result, cloud providers (e.g., Amazon, SGI) have already started providing high performance computing resources for their customers. However, the nature of multi-threaded applications brings additional design and management challenges. For example, multi-threaded applications exhibit varying power and performance requirements with changing amount of resources due to application-specific characteristics (e.g., synchronization/communication overheads), and/or architectural bottlenecks (e.g., bus bandwidth). Therefore, optimizing the performance of consolidated multi-threaded applications under power constraints require a deeper understanding of the interactions between the applications and the underlying hardware nodes.

1.2 Node-level Analysis and Management

Achieving energy efficiency on data centers have been extensively studied through cluster-level management techniques (Anderson and Tucek, 2010) (Jennings and Stadler, 2015). Although cluster-level management techniques can bring significant energy efficiency benefits (Wang and Chen, 2008) (Fan et al., 2007) (Barroso and Hölzle, 2007), cluster or rack-level optimizations lack the ability to pinpoint the underlying

reasons for inefficient use of the individual server-nodes. It is shown that the individual nodes in data centers mostly run below their peak performance, which leads to inefficient data center operation (Rasmussen et al., 2011). **Node-level techniques** (i.e., server-level) provide more visibility into workload characteristics and this visibility can be used to further improve the overall energy efficiency of the data centers (Orgerie et al., 2014). In the era of multi-threaded applications, workload visibility and understanding become even more critical, due to the complex behavior of multi-threaded applications with varying number of threads and/or amount of resources. Furthermore, node-level approaches allow using finer granularity control knobs (e.g., dynamic voltage-frequency scaling (DVFS), core power gating). On the other hand, rack or cluster-level approaches view the server-nodes as black boxes and mainly use coarse granularity control knobs (e.g., turning on/off servers). In fact, finer granularity control knobs at the server-level, such as the ability to turn on/off individual cores, are shown to be critical for energy-efficient execution of multi-threaded workloads (Pusukuri et al., 2011). As a result, processor vendors have already started to develop node-level managers that can be integrated to a larger-scale management scheme (HP-Intel Dynamic Power Capping, 2009) (Intel, 2013). However, most of the existing node-level managers (i) operate based on predefined thresholds, (ii) do not capture the multi-threaded application characteristics and (iii) lack the ability to utilize multi-threaded application specific control knobs to improve the energy efficiency.

Node-level analysis is also needed for better understanding of the challenges that arise from consolidation. In order to operate data centers efficiently, many applications are executed side-by-side with unknown applications (i.e., they are consolidated). However, resource sharing across applications might create additional contention on the CPU, memory, bus and other shared resources. Therefore, it is impor-

tant to take into account the resource requirements of the consolidated applications while making resource management or VM placement decisions. Although there are studies on analyzing the interference impact across consolidated applications (Isci et al., 2010), these studies do not consider the dynamic nature of multi-threaded applications, and assume constant resource allocations. However, changing the amount of resources allocated to each multi-threaded application can also change the interference impact (Bhadauria and McKee, 2010).

Consolidation related challenges, such as performance interference, are even more significant under power constraints. For power constrained environments, distributing the limited amount of resources across the VMs that are consolidated is another important problem, which cannot be solved solely through intelligent VM placement. Resource distribution/allocation techniques target efficiently distributing the total available resources across consolidated VMs. Although there is a significant amount of work in the literature to address the consolidation and resource distribution problem under power constraints (Jerger et al., 2007) (Delimitrou and Kozyrakis, 2013), multi-threaded applications exhibit distinct power/performance requirements with changing amount of resources (or threads) that impacts the efficiency of the resource distribution decisions. Therefore, capturing the distinct power/performance trade-offs of the multi-threaded applications under dynamically changing power and performance constraints is an important problem when consolidating multi-threaded applications.

Finally, single node analysis also enables evaluating adaptive approaches at various layers of the computing stack. Adaptive approaches and techniques are becoming more commonly adopted by cloud administrators to tackle the administration complexity problem and to comply with dynamically changing constraints. However, the lack of coordination across various adaptation techniques makes it even more challenging to accurately meet these varying constraints. Traditional adaptive solutions

employ system-level management knobs to comply with the power and performance requirements. These system-level adaptive solutions use control knobs such as DVFS or turning on/off cores (Reda et al., 2012). However, system-level solutions lack the ability to optimize the performance of the application running on the system depending on the architectural characteristics of the underlying platform. Adaptive applications address the performance optimization problem by dynamically configuring application parameters depending on the hardware properties and the performance goals (Hoffmann, 2014). As application and system-level decisions impact both the performance and the power consumption, uncoordinated decisions at these two levels may lead to unstable and inefficient control (Hankendi et al., 2015). Therefore, coordination across multiple adaptive techniques is a major challenge to overcome, when simultaneously employing adaptive techniques at various levels of the computing stack.

1.3 Thesis Contributions

In this work, we focus on node-level techniques to improve the energy efficiency of data center resources. Our hypothesis is that *future power and resource management techniques should take into account the distinct characteristics of multi-threaded applications (i.e., performance scalability) at the node-level to substantially improve the energy efficiency.* Based on our hypothesis, we focus on multi-threaded applications that are designed for shared-memory architectures (Bienia et al., 2008) and the multi-threaded slave nodes of the scale-out applications (Ferdman et al., 2012) (Wang et al., 2014). Our specific contributions in this work are as follows:

- We provide detailed analyses on consolidation techniques on both virtual and native environments. On native environment, we show that the best performing consolidation strategy is dependent on the overall characteristics of the work-

load sets (Sec. 4.1) (Hankendi and Coskun, 2012). On virtual environments, we present a technique to classify the applications according to their energy efficiency levels (Sec. 4.3) (Hankendi and Coskun, 2013). We demonstrate that allocating resources proportional to the energy efficiency levels of the consolidated applications can significantly improve the energy efficiency.

- We present **Pack & Cap** (Sec. 5.1), a power capping technique to optimize specifically the performance of multi-threaded applications under power constraints (Cochran et al., 2011). **Pack & Cap** achieves accurate power tracking and maximizes the performance by packing the active threads onto a variable number of cores (i.e., *thread packing*), in addition to using DVFS, through a machine learning based mechanism.

- As most servers in data centers are virtualized today, we introduce **vCap**, a virtualized system management framework that can be used with other resource distribution policies depending on the user scenarios and requirements to maximize performance under power constraints (Sec. 5.2) (Hankendi et al., 2013). **vCap** (i) meets the dynamically changing power caps by using virtual environment specific control knobs (i.e., limits on CPU usage), (ii) guides the placement decisions based on the multi-threaded application specific characteristics and (iii) makes resource allocation decisions to distribute a limited amount of compute resources across multiple VMs, each of which is running multi-threaded applications.

- On top of the vCap framework, we design **Scale & Cap**, which incorporates not only the performance scalability characteristics of the multi-threaded applications, but also the power efficiency characteristics while making resource allocation decisions to improve the performance while meeting the power caps

(Sec. 5.3). **Scale & Cap** achieves up to 24% performance improvements by employing a formal linear programming-based solution. We also show that resource distribution techniques provide superior benefits for tight power constraints when compared to placement techniques.

- As adaptive applications that can adjust their parameters at runtime are becoming more common, we introduce the **Adapt & Cap** technique, which coordinates adaptation decisions at application and system-level to improve the performance under power constraints, while providing stable and efficient power and performance control (Sec. 5.4) (Hankendi et al., 2015). We implement **Adapt & Cap** on two state-of-the-art multi-core servers and achieve power savings up to 25% and performance improvements up to 1.7x for a set of adaptive applications.

The rest of the thesis is organized as follows. In Section II, we discuss the significance of our approach to power capping and consolidation in comparison to state-of-the-art techniques. In Section III, we present our experimental setup, instrumentation techniques and benchmarking methodology. In Section IV, we present our consolidation and resource management techniques. In Section V, we present our adaptive power capping techniques on both virtual and native environments and our results based on experiments on real-life multi-core servers. Section VI discusses our future research directions and Section VI concludes the thesis.

Chapter 2

Background and Related Work

In this section, we summarize the prior work in resource and power management techniques on multi-core systems. We mainly focus on three main categories: placement techniques, resource allocation techniques, and power management/capping techniques.

2.1 Workload and VM Placement

Placement techniques mainly target reducing the performance degradation due to interference across consolidated applications and VMs that are sharing the same physical resources. In order to reduce the performance degradation on consolidated environments, placement techniques find the best matching applications to consolidate together.

One of the main sources of contention is the rate of cache accesses (Dhiman and Rosing, 2007). For instance, co-locating (i.e., placing) multiple memory-bound applications together degrades the performance due to increased amount of cache contention. In order to tackle the increased contention problem, Bhadauria *et al.* (Bhadauria and McKee, 2010) propose co-scheduling techniques for multi-core systems to improve energy-delay (ED) by making scheduling decisions based on bus contention, last level cache miss rates, and thread counts. Proposed co-schedulers use offline lookup tables for these three metrics to guide the co-scheduling decisions. Dhiman *et al.* propose a VM scheduling technique that estimates VM-level CPU and

memory usage based on system-level metrics to guide co-scheduling and migration decisions (Dhiman et al., 2009). Their proposed technique consolidates the applications that have complementary resource usage characteristics to reduce the performance degradation. Dey *et al.* propose a methodology to characterize the shared-resource contention of parallel applications (Dey et al., 2011). They provide experimental analysis on inter and intra-thread dependencies for PARSEC benchmarks. Meng *et al.* propose a joint VM provisioning technique based on workload pattern analysis (Meng et al., 2010). Their technique selects VM combinations with complementary workload patterns (e.g., high vs. low cache-miss) to improve the energy efficiency. Zhang *et al.* propose a methodology to predict performance degradation due to interference on both memory and CPU by offline characterization (Zhang et al., 2014).

Another line of work uses statistical techniques to capture the interference impact of consolidation (Eyerman and Eeckhout, 2010) (Kim et al., 2013). Delimitrou *et al.* propose machine-learning-based recommendation strategy that identifies the best VM groups to minimize the interference across consolidated applications (Delimitrou and Kozyrakis, 2013). The proposed technique improves the utilization of the system and prevents wasting idle power on underutilized nodes.

There are works that target improving the performance of consolidated environments through OS/hypervisor or microarchitectural modifications (Gupta et al., 2006) (Porterfield et al., 2008) (Sanchez and Kozyrakis, 2011). One line of work focuses on improving performance isolation through memory scheduling on virtualized environments to reduce the performance degradation and to provide predictable performance for consolidated applications (Fedorova et al., 2007). For fair resource distribution, Kim *et al.* propose cache sharing techniques to provide fair performance across consolidated threads through dynamic cache partitioning (Kim et al., 2004).

At the cluster-level, VM placement techniques target to solve a global optimization

problem through control knobs, such as VM migration, which refers moving VMs from one server to another (Ahmad et al., 2015) (Zhu et al., 2014). Hermenier *et al.* propose a framework to find a globally optimal solution for VM scheduling by using constraint programming (Hermenier et al., 2009). In order to improve the data center utilization, Liu *et al.* propose a consolidation framework that schedules VMs based on the CPU utilization (Liu et al., 2009).

For data intensive workloads, contention at the disk is another important factor that determines the efficiency (Korupolu et al., 2009) (Srikantaiah et al., 2008). Romosan *et al.* propose algorithms for co-scheduling computation and data on clusters by load balancing frequently used files across multiple cluster nodes (Romosan et al., 2005). For distributed parallel applications, Frachtenberg *et al.* (Frachtenberg et al., 2005) propose a co-scheduling technique through monitoring message-passing interface (MPI) calls of the parallel applications. Their proposed co-scheduler identifies processes that communicate frequently through an MPI monitoring layer to make co-scheduling decisions. McGregor *et al.* (McGregor and Antonopoulos, 2005) present scheduling algorithms to improve performance by determining the best thread mixes. They monitor workload behavior through performance counters and propose scheduling policies to reduce the resource contention by using bus transaction rate, stall cycle rate and last level cache miss rate.

Although placement techniques are reported to significantly improve the performance of consolidated applications, their benefits are limited with reducing the contention on shared resources and do not optimize the resource distribution across consolidated applications.

2.2 Resource Allocation

Resource allocation is distributing the available amount of resources across multiple entities (i.e., across applications or VMs). The common goal of resource allocation techniques is improving the performance of the servers through efficiently distributing the available resources (i.e., compute and/or power resources).

For determining the amount of resources to allocate to consolidated VMs, Kusic *et al.* propose a dynamic resource provisioning framework based on look-ahead control (Kusic et al., 2008). Vasic *et al.* propose a framework that makes resource allocation decisions based on the history of the VMs to reduce the resource management overhead (Vasić et al., 2012). Zheng *et al.* present an empirical infrastructure for data center management (Zheng et al., 2009). Their proposed infrastructure allocates a server node (i.e., sandbox) to experimentally derive the energy/performance tradeoffs.

Modern virtualization environments such as Xen, KVM and vSphere provide resource management mechanisms to improve the efficiency of the server nodes. Xen and KVM mainly rely on the default Linux credit scheduler to minimize the number of idle cycles (KVM, 2008) (Xen, 2009). The credit scheduler automatically load balances the processes (i.e., vCPUs) across all available physical cores. As vCPUs take time on the physical cores, they consume their *credits*, and the hypervisor keeps track of the consumed credits to provide fair resource allocation across all vCPUs/VMs. Xen and KVM also provides additional control knobs, such as resource reserves, shares and limits, to guide the scheduler decisions depending on the application requirements. However, the resource allocation decisions through VM-level control knobs are mostly left to the user, and the default managers are agnostic to the dynamically changing application and user requirements. Therefore, Xen and KVM managers are not *workload-optimized* to improve the energy efficiency.

vSphere's Distributed Resource Scheduler (DRS) provides balanced load distribu-

tion across server nodes based on user-selected policies (VMware DRS, 2009). On vSphere, user can request five levels of automation policies, which determines the frequency of VM migrations (e.g., aggressive or naive). Automated load balancing monitors the activity on CPU and memory and uses static thresholds (e.g., minimum of 81% CPU utilization to consider migration) to make migration decisions. However, using the same static thresholds for dynamic and/or unknown applications leads to inefficient resource management decisions (Beloglazov and Buyya, 2010). In addition to DRS, VMware's Distributed Power Management (DPM) tool aims to dynamically reduce the number of active server nodes through aggressive consolidation (VMware DPM, 2010). In case of changes in user demand, DPM can turn on/off server nodes, and migrate VMs to reduce the power consumption without compromising the performance. However, both DPM and DRS lack the ability to adapt their decisions for dynamically changing performance and power requirements.

Providing performance and availability guarantees to the users is an important feature for users and cloud providers. One line of work targets providing service-level agreement (SLA) guarantees for consolidated applications (Van et al., 2009) (Wu et al., 2011). Beloglazov *et al.* propose an adaptive threshold-based dynamic consolidation technique, which provides SLA guarantees by selecting VMs to migrate to different physical nodes based on the resource utilization (Beloglazov and Buyya, 2010). In order to satisfy the changing user requirements and SLAs, adjusting the size/type of the VMs (e.g., right-sizing) is also a crucial mechanism (AWS, 2013) (Lin et al., 2013). It is possible to meet the SLA and availability guarantees by removing or adding resources (i.e., allocation of cores or entire new server nodes) (Bonvin et al., 2011). There are also works that allow user-specified workload provisioning policies to optimize energy efficiency on clusters (Wang et al., 2012).

2.3 Power Management

As power is one of the main limiting factors of the data center performance, power management techniques have been extensively studied in the context of data center management to improve the energy efficiency. Dynamic power and energy management techniques such as controlling idle-power modes and voltage-frequency scaling are well studied research areas (Meisner et al., 2009) (Benini et al., 2000). For power management at the cluster-level, Fan *et al.* study power provisioning strategies to improve the overall utilization of data center and to reduce the power consumption by turning off the underutilized servers (Fan et al., 2007). Other techniques aim to coordinate node-level managers with global manager through iterative feedback-based techniques (Wang and Chen, 2008) (Kumar et al., 2009) (Raghavendra et al., 2008) (Wang et al., 2009). Some of the prior work focus on the total wait time of the workloads (i.e., queue wait time + completion time) to optimize the cluster-level performance (Gandhi et al., 2009) (Urgaonkar et al., 2010).

At the node-level, both software and hardware-level power management strategies have been proposed in recent years. Most of the modern processor cores support DVFS and power gating capabilities. Therefore, DVFS and core power gating are commonly used power management knobs for node-level techniques (Li and Martinez, 2006). Kim *et al.* study on-chip regulators to achieve DVFS at a finer granularity reaching nanosecond range (Kim et al., 2008). Shin *et al.* propose intra-task voltage scheduling through compiler level static timing analysis (Shin et al., 2001). For multi-threaded applications, Rangan *et al.* propose a thread scheduling policy that maps the threads to various voltage domains to optimize performance under power constraints (Rangan et al., 2009). For a mixture of single and multi-threaded application, Ma *et al.* propose a power capping technique by power-gating the cores and applying per-core DVFS through application monitoring (Ma and Wang, 2012).

Recent commercial servers also provide power capping capabilities (Intel, 2013). AMD processors are equipped with internal power estimation capabilities at the firmware-level and allow power capping based on predetermined threshold values (Samson, 2009). Starting with the Sandybridge architecture, Intel started to provide a power consumption estimator and a runtime average power limiter (RAPL) (David et al., 2010). RAPL allows fine-grained power control at various component levels, including package, DRAM controller, CPU and graphics processor. Using the internal power estimation, RAPL enables capping the average power over a predefined measurement window, but lacks the ability to cap the peak power.

For capping the power in virtualized environments, Kansal *et al.* propose a VM-level power estimation technique that correlates the system-level power measurements with VM-level resource usage. They use the VM-level power estimation to show that accurate VM-level power information can be used to improve the performance under power caps (Kansal et al., 2010). As efficiently distributing the available power capacity heavily depend on the workload characteristics, Govindan *et al.* propose a statistical technique to predict the power efficiency of the workloads to make efficient power allocation across consolidated VMs. Nathuji *et al.* design a power allocation technique for VMs to improve the performance for a given power budget by allocating power budgets proportionally across VMs according to the SLA requirements of the VMs (Nathuji and Schwan, 2008). Their technique uses CPU utilization to proportionally distribute the available power budget. Hwang *et al.* study the impact of consolidation in virtualized multi-core environments (Hwang et al., 2012). Their study investigates finding the optimum VM-density for multi-core processors for single-threaded applications that have distinct characteristics (i.e., memory/CPU-bounded) and they propose a consolidation policy that uses DVFS and power gating.

2.4 Distinguishing Aspects

The key distinguishing aspects of this thesis compared to related work are:

- Our proposed solutions exclusively consider multi-threaded application characteristics and make use of multi-threaded specific control dimensions (i.e., performance and power scalability) while making resource distribution decisions.

- We implement the proposed resource and power management techniques on state-of-the-art real-systems and demonstrate the benefits through an extensive set of experiments, which show the practical value of the proposed solutions in this thesis.

- We propose an online workload demand estimation technique (integrated in vCap) to identify the applications that benefit substantially from increasing their CPU resources. In contrast to previous work, our technique dynamically makes resource allocation decisions without requiring offline analysis.

- We show that our resource allocation and power capping techniques can be jointly used with existing placement policies to further improve the energy efficiency of multi-core servers (Hankendi and Coskun, 2013) (Hankendi et al., 2015).

- We propose power capping techniques on virtualized environments and show that our proposed technique achieves finer granularity power tracking compared to existing DVFS or clock gating based methods, making it a promising technique for future data centers with dynamic power regulation capabilities (Hankendi et al., 2013).

- We show that coordinating dynamic adaptations at different layers (i.e., system and application-level adaptations) improves the power tracking accuracy and

overall system efficiency significantly (Hankendi et al., 2015).

- We demonstrate that performing *application and QoS-aware* power and resource provisioning brings significant energy efficiency benefits under user-defined QoS requirements and power constraints.

Chapter 3

Instrumenting Multi-core Servers

In this work, we evaluate and test our power and resource management techniques on multiple state-of-the-art commodity servers, which we explain in-depth in this section. Our instrumentation infrastructure includes both power and performance measurement capabilities, as well as a workload execution framework to emulate real use-case scenarios. Although simulation techniques provide valuable information when evaluating a new idea, it is essential to demonstrate the benefits and shortcomings of a proposed idea on real-system as a proof-of-concept.

3.1 Overview of the Experimental Setup

Our experimental setup includes system and processor-level power measurements, as well as performance measurements on both native and virtualized environments. We store and utilize the collected data on a separate logger machine in real-time to make management decisions, as well as for offline benchmark analysis. We measure the system-level power by using a *Wattsup PRO* power meter. In order to identify the chip power, we measure the current flow on 12V inputs of the voltage regulator with an Agilent 31134A hall-effect clamp ammeter. We use the Agilent 34410A digital multimeter to log the current measurements from the ammeter. Figure 3·1 shows the overall experimental setup. Performance measurement includes both polling hardware performance counters and other system-level metrics, as explained in Section 3.3.

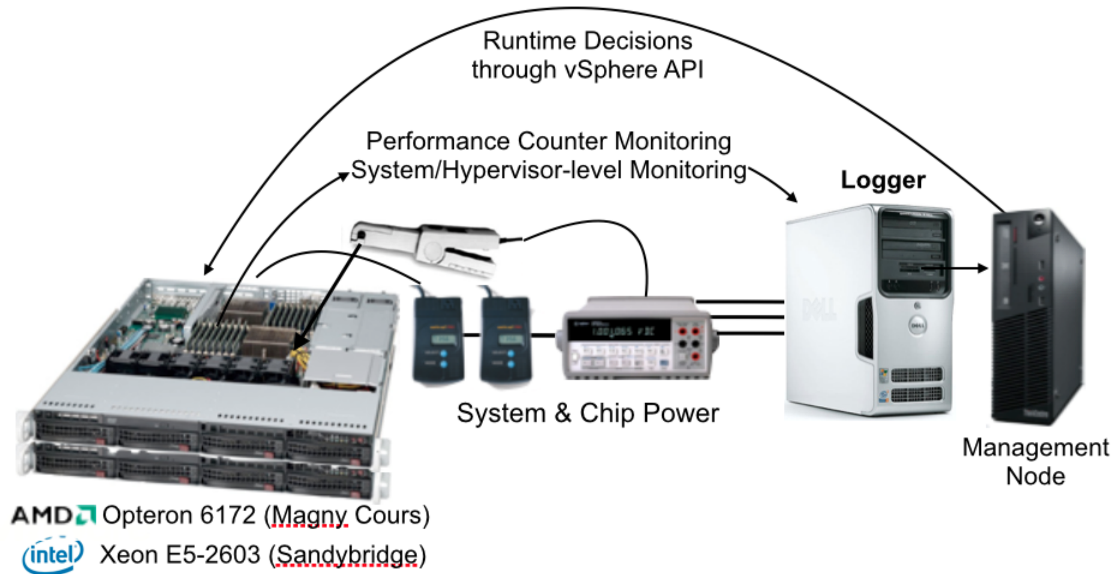

Figure 3·1: Experimental Setup

3.2 Systems Under Test

As our main target is managing multi-core environments, our experimental setup includes three state-of-the-art servers: one with Intel Xeon E5, one with Intel i7 and the other with AMD Magny Cours (Opteron 6172) multi-core processors. Intel Xeon E5-2603 processors consist of 8 cores. Each core has 32 KB of private L1 and 256 KB of private L2 cache. All 8 cores share 10 MB of L3 cache and 32 GB memory. Magny Cours consists of two 6-core dies attached together on a single chip. Each 6-core die includes a 12 MB shared L3 cache, and each core has a 512 KB private L2 cache. All cores also share a 16 GB off-chip memory. We use an 4-core Intel i7 processor based multi-core system and collect data at Brown University (Cochran et al., 2011).

3.3 Hardware Event Measurements

On the **native environment**, which runs Ubuntu Server 12.04 as the OS, we collect data from the performance counters to characterize applications and to identify possible bottlenecks for multi-threaded applications such as cache misses, stalls, memory

	AMD Opteron 6172	**Intel Xeon E5**	**Intel i7**
# of Cores	12	8	4
Frequency Range	2.1 - 0.8 GHz	1.8 - 0.8 GHz	1.6 - 2.67 GHz
I-cache	64 KB	32 KB	32 KB
D-cache	64 KB	32 KB	32 KB
L2-cache (Private)	512 KB	256 KB	256 KB
L3-cache (Shared)	12 MB	10 MB	8 MB
RAM	16 GB	32 GB	8 GB

Table 3.1: Our experimental infrastructure consists of three state-of-the-art servers with AMD and Intel multi-core processors. This table summarizes the specifications of the processors.

accesses, etc. Although the naming conventions for performance counters vary across different processor vendors, we choose similar sets of counters for all processors. We use the *perf* utility tool to collect the following core-level performance counters at a 100ms sampling interval: μ-OPs retired, unhalted CPU cycles, data cache misses, L2-cache misses, executed lock operations, mispredicted branch instructions, dispatch stalls, dispatched FP operations on the native system. In addition to per-core measurements, we collect system-level L3-cache misses.

Performance counters provide a wealth of data on the interaction of the workload with the processor and the memory hierarchy. In this subsection, we provide the details of the collected hardware event: data cache misses, instruction cache misses, resource stalls, load-lock operations, and IPC.

- **Data cache misses:** Data cache misses are commonly used to measure the memory dependency of the applications. Applications that have frequent data cache misses are expected to benefit less from increasing the number of active threads and/or the frequency setting. Therefore, analyzing the data cache miss events is crucial to understand the performance and power tradeoffs. We analyze all levels of data caches because depending on the application and the cache sizes, each level of cache can potentially introduce performance limitations to the application.

- **Instruction cache misses:** Instruction cache misses are among the sources of performance degradation as each instruction miss causes extra stall cycles. For memory-bounded applications, effect of instruction misses is minor as the primary bottleneck is the data cache misses that occur at the higher level of the cache hierarchy. On the other hand, for CPU-bounded applications instruction misses might be a significant factor in performance degradation.

- **Resource stalls:** In our analysis, we also investigate the effects of resource stalls on performance. There are various sources for stalls. Some of the most important causes for resource stalls are full load-store buffer, full reorder buffer, branch misprediction recovery. We measure all resource stalls through the corresponding performance counter.

- **Load-lock operations:** For parallel workloads, thread synchronization and resolving data dependencies impact the performance. There are a number of mechanisms to handle communication across threads, depending on the parallelization model of the application (i.e., locks, barriers, or semaphores). Load-locks ensure that only one thread can modify the shared data to maintain the data consistency across threads. Therefore, number of locks in an application provides information on the inter-thread communication characteristics of workloads.

- **IPC:** While IPC is a common metric for performance evaluation of single-threaded workloads, note that it is not a robust metric to evaluate the performance of multi-threaded workloads (Alameldeen and Wood, 2006). We compute application IPC here as the total number of instructions executed by all threads divided by the number of maximum cycles among threads because the longest thread determines the length of execution.

3.4 Application-specific Performance Measurement

As each application performs a different task, the application-specific performance metric varies depending on the application type. Therefore, we measure the performance with runtime, throughput, as well as application-specific metrics. We track the application-specific performance metrics for the PARSEC benchmarks by utilizing the Application Heartbeats framework (Hoffmann et al., 2011). CloudSuite applications report application-specific performance without requiring a modification to the source code. We derive the quality-of-service (QoS) of the applications using the application-specific metrics. For instance, for image processing applications (e.g., `bodytrack`) the QoS metric is *frames-per-second* (FPS), whereas the QoS metric for the option trading application (e.g., `blackscholes`) is the *number of options*. Instead of such metrics, application-specific performance (e.g., frames per second, requests serviced per second, etc.) can be tracked at the cost of minimal modifications to the application source code. We report the relative QoS for each application, where we define the maximum QoS of an application (i.e., QoS=1) as the case where the application is running alone with the maximum amount of available CPU resources (e.g., maximum number of cores).

3.5 Virtualized Environment Setup

As most data centers are virtualized, we conduct experiments on virtualized environments in part of the work in this thesis. We virtualize our systems by the VMware vSphere 5.1 ESXi hypervisor. We create VMs with multiple vCPUs (SMP VMs), such that each VM accommodates a multi-threaded application. Each VM runs Ubuntu Server 12.04 or 14.04 as the guest OS.

We use the default `vmkperf` utility to poll the following performance counter data from the physical CPUs at every 2 seconds: CPU cycles, retired instructions, and L3-

cache misses. These metrics determine the performance and power characteristics of the applications, as shown in previous work (Khan et al., 2011). Each vCPU runs as a process (i.e., world in ESXi) on the hypervisor, thus it is possible to derive VM-level performance measurements through configuring `vmkperf` to poll performance counter readings for vCPUs. In addition to `vmkperf`, we use the recently introduced *virtualized performance counters* (VPC) for some of our experiments. Based on our analysis and reported numbers from other work, VPCs provide accurate measurements of the hardware events that we focus on in this work (Serebrin and Hecht, 2009). `vmkperf` and VPC measurements show between 2-4% difference when running the same set of applications. We also use `esxtop` utility to collect VM-level metrics, such as CPU utilization, READY%, RUN% at every 2 seconds. These metrics provide additional insight into resource usage.

3.6 Benchmarking Methodology

Target workload type in this thesis is multi-threaded applications. We run applications from the Princeton Application Repository for Shared-Memory Computers (PARSEC) (Bienia et al., 2008) multi-threaded benchmark suite (Bienia et al., 2008), CloudSuite (Ferdman et al., 2012), and BigDataBench benchmark suite (Wang et al., 2014) in our experiments as a representative set of multi-threaded workloads on the cloud resources.

PARSEC (Bienia et al., 2008) is a benchmark suite consisting of multi-threaded applications targeting future general purpose architectures and data centers as well as the HPC domain. On the other hand, CloudSuite and BigDataBench benchmark suites include scale-out applications, which are already occupying significant amount of resources on the cloud resources today. As our techniques target node-level optimizations, we evaluate the slave-nodes of the scale-out applications, rather than

evaluating the cluster-level challenges that arise from the data-intensive nature of the scale-out applications.

Bienia *et al.* provide various categories and characteristics for the PARSEC benchmarks such as parallelization model, data usage/sharing, and working set size (Bienia et al., 2008). Although these categories provide important insights about the software structure, they do not quantify the application performance bottlenecks. Table 3.2 summarizes the main characteristics of the PARSEC benchmarks based on our categorization. Data sharing and data exchange characteristics are expected to impact the performance of the parallel workloads. However, some benchmarks with high data sharing/exchange do not have any notable bottleneck according to our measurements. For example, `freqmine` is reported to have high data sharing and medium data exchange; however, our analysis shows that `freqmine` has no significant performance bottlenecks. This implies that having high data exchange or high data sharing do not always cause performance bottlenecks in practice, and motivates a closer look at the performance counter data.

In order to accurately measure the multi-threaded specific characteristics through the performance counters and power measurements, we only evaluate the parallel phases (i.e., region-of-interest (ROI)) of the applications. The parallel phase of multi-threaded applications dominate the application execution time in real-life clusters. Therefore, we collect performance and power data only for the parallel phases (Hankendi and Coskun, 2012).

As time-spent in ROI and serial phases show significant variation depending on the workload type, we implement a consolidation management interface, `consolmgmt`, on top of the default PARSEC benchmark management interface `parsecmgmt` to align the parallel phases of the consolidated applications. `consolmgmt` interface manages thread affinity settings to assign each thread to one core and the ROI-Synchronization

	Bienia et al. (Bienia et al., 2008)			Measured				
	Working Set	**Data Sharing**	**Data Exchange**	**D-cache Misses**	**I-cache Misses**	**Locks**	**Bottleneck**	**IPC**
blackscholes	small	low	low	low	low	low	none	low
bodytrack	medium	high	medium	low	medium	low	i-cache	high
canneal	unbounded	high	high	high	low	high	d-cache	low
dedup	unbounded	high	high	low	low	medium	locks	medium
facesim	large	low	medium	low	medium	low	i-cache	medium
ferret	unbounded	high	high	low	low	low	none	medium
fluidanimate	large	low	medium	low	low	medium	locks	medium
freqmine	unbounded	high	medium	low	low	low	none	high
raytrace	unbounded	high	low	low	low	low	none	medium
streamcluster	medium	low	medium	high	low	low	d-cache	low
swaptions	medium	low	low	low	low	low	none	medium
vips	medium	low	medium	low	low	low	none	high
x264	medium	high	high	medium	high	low	d-cache	medium

Table 3.2: Summary of characteristics of PARSEC Benchmarks (Bienia et al., 2008).

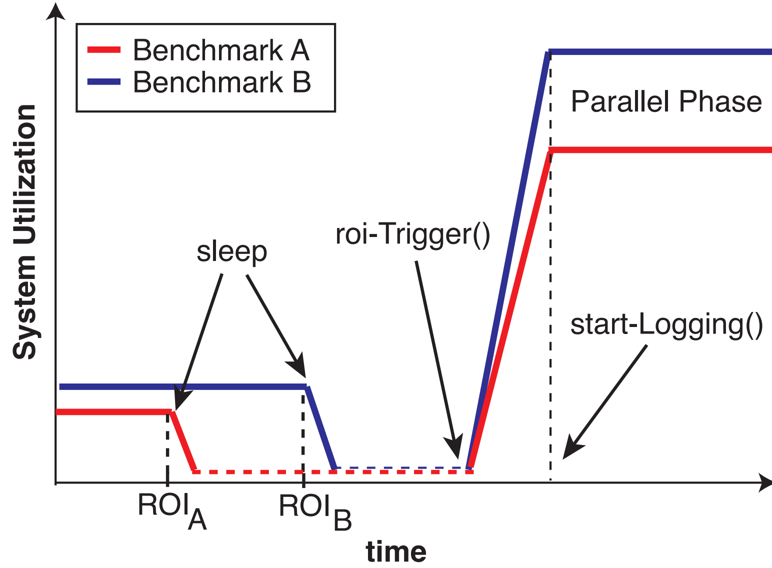

Figure 3·2: ROI-Synchronization flow.

routine (ROI-Synch) to synchronize the ROI of multiple workloads. We implemented the ROI-Synch inside the `HOOKS` library routines of the default PARSEC package. In Figure 3·2, we depict the synchronization flow for a hypothetical 2 benchmark consolidation case. ROI-Synch ensures that all benchmarks wait at the ROI checkpoints (ROI_A, ROI_B) for other benchmarks to reach their own ROI checkpoints. After all benchmarks reach the ROI checkpoint, ROI-Synch calls *roi-Trigger()* function to synchronously trigger the benchmarks to continue their execution. ROI-Synch also synchronizes power and performance measurement infrastructure with the application ROI via communicating with the logger computer. After an initial cold start phase, data loggers are triggered by the *start-Logging()* function to collect the ROI performance and power characteristics.

3.7 Summary of Experimental Methodology

We design an infrastructure to conduct experiments on real-life systems. Our experimental infrastructure consists of system and processor-level power measurements, as well as performance counter measurements together with system and application-level performance measurements. In order to accurately evaluate the multi-threaded characteristics of the consolidated applications, we also develop a framework called `consolmgmt` that synchronizes the parallel phases of the multi-threaded applications.

Chapter 4

Resource Allocation and Consolidation for Energy Efficiency

Resource management on consolidated environments is a major challenge, as it strongly impacts resource contention, workload interference, and performance scalability of the multi-threaded applications. In this work, our goal is to optimally manage available hardware resources to improve energy efficiency, while satisfying user requirements. In this section, we first present analysis on co-scheduling on native environments and evaluate previously proposed co-scheduling policies. Second, we analyze performance isolation on virtual environments and based on our analysis, we develop an autonomous resource management technique on consolidated VMs. The proposed technique first classifies applications according to their energy efficiency levels through runtime monitoring. Based on the energy efficiency levels of the co-scheduled applications, our technique adaptively adjusts the resources allocated for each application at runtime.

4.1 Co-scheduling Analysis on Native Environments

Co-scheduling of parallel applications is a promising method to improve the energy efficiency of computing systems by taking advantage of contrasting resource requirements of different parallel applications. Nonetheless, energy savings due to co-scheduling varies significantly depending on the characteristics of the applications that are being consolidated. In Figure 4·1, we evaluate the range of energy-per-

work (E/w) savings due to consolidation of PARSEC parallel benchmarks (Bienia, 2011). We show minimum and maximum savings when 2 PARSEC benchmarks are co-scheduled with 6 threads on the same node, with respect to 12 thread execution on separate nodes. Although maximum E/w savings reach up to 40%, majority of the benchmarks exhibit increased E/w due to increased resource contention when co-scheduled with other benchmarks. This implies that, to improve the energy efficiency through co-scheduling, it is important to consider characteristics of parallel applications. This analysis also shows that depending on the set of workloads that are co-scheduled, E/w savings show significant variations.

Energy consumption of a multi-core system varies as a function of the characteristics of the parallel workloads running on the system. As a result, the range of potential E/w savings show significant variations across workloads. To understand

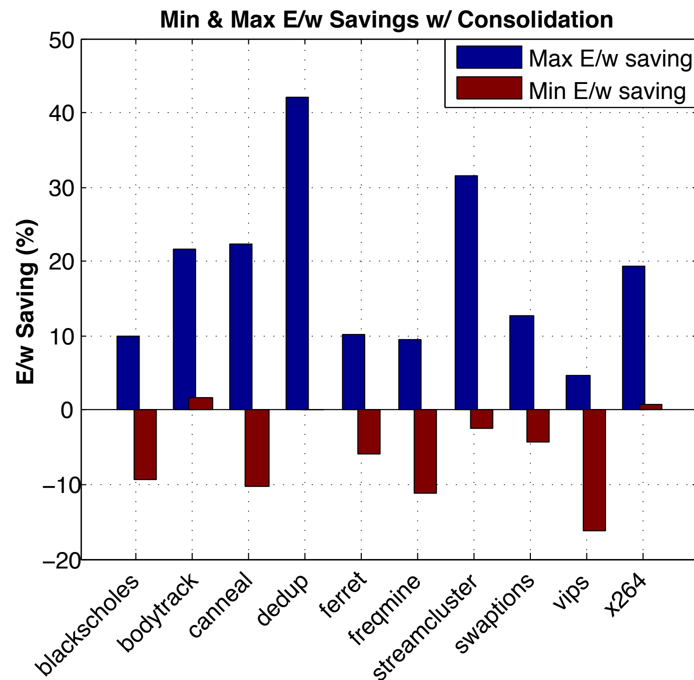

Figure 4·1: Minimum and maximum change in E/w for 6 thread consolidated PARSEC benchmarks w.r.t 12 thread execution on a native environment.

these variations, we analyze the energy tradeoffs for a set of performance events for the PARSEC benchmarks.

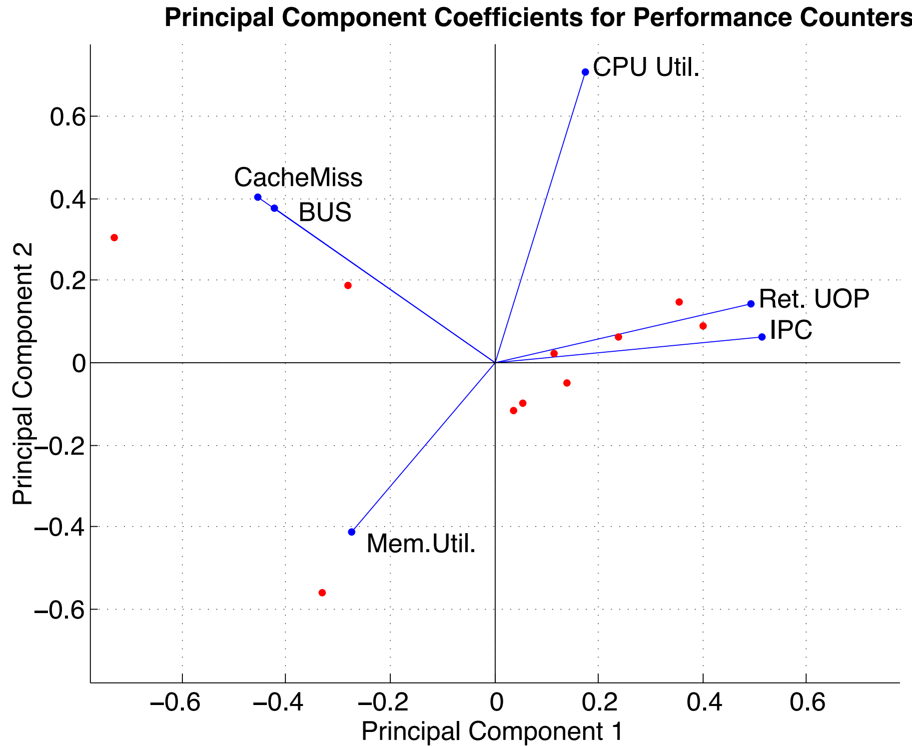

Figure 4·2: First two principal component coefficients for various performance metrics when running PARSEC benchmarks.

We use Principal Component Analysis (PCA) to determine which performance events vary considerably across the benchmark suite. Figure 4·2 shows the coefficients for various performance events for their first two principal components (PCs). First two PCs together explain more than 85% of the overall variations. Figure 4·2 demonstrates that performance events cover the PC space in four distinct directions. Cache misses and bus accesses almost have the same coefficients. Similarly, retired μOPs and IPC are closely related. These two groups of events cover distinctive features on the x axis of the PC space. On the other hand, CPU and memory utilization cover other distinct features of the applications on the y-axis of the PC space. Note

that memory utilization is located at a different quadrant than the cache misses, motivating investigating the impact of both cache misses and memory utilization metrics separately.

Memory utilization exhibits a distinctive variation in comparison to cache misses and bus accesses. This implies that, using memory utilization as a metric would help to identify another dimension of bottlenecks that is different than cache misses and bus accesses. This is due to varying workings set sizes of the applications in PARSEC suite. On the other hand CPU utilization have positive coefficients on both x and y axis unlike the memory utilization, cache misses and bus accesses. Thus, in order to identify the distinctive features of workloads, we focus on the following performance events:

CPU utilization: CPU utilization measures percentage of time-spent for doing useful work by CPU as opposed to being idle. Performance scaling of the applications that fully utilize the CPU resources can be categorized as CPU-bounded as their performance improvement is only limited by the CPU resources. Thus, CPU-bounded applications are expected to benefit less from co-scheduling as they already spent most of their execution time doing useful work. On the contrary, applications that are not CPU-bounded are expected to benefit most from co-scheduling. Applications such as `dedup` and `bodytrack` poorly utilize the CPU resources and they are the only PARSEC benchmarks that consistently benefit from consolidation as Figure 4·1 implies.

Cache misses: Cache miss rates measure the memory dependencies of the applications. Since different levels of cache cause different performance penalties, we evaluate the weighted cache misses by using the miss penalty for each level of cache. As cache misses cause extra stall cycles, they can become the main reason for lower performance scaling of the applications. Thus, applications that have high cache miss

rates would benefit more from co-scheduling in comparison to running with higher number threads.

Memory utilization: Memory utilization percentage shows the utilization of the DRAM modules. Memory and CPU are two of the main bottlenecks that affect the performance of applications. When consolidating multiple benchmarks on a multi-core system with shared resources, it is important to consider the degree of memory utilization, as memory is a shared resource across applications. `canneal`, `dedup` and `freqmine` utilize the memory significantly higher than the rest of the benchmarks. Despite having low cache miss rates, `freqmine` and `dedup` utilizes the memory heavily. On the other hand, `streamcluster` does not utilize the memory heavily, although it generates high cache miss rates.

This analysis on performance events show that, considering only one performance event would not be sufficient to make the optimum decisions to improve energy-efficiency when consolidating multiple workloads. As shown above, there is not a single trend across benchmarks in terms of different performance events. In order to make optimum decisions for co-scheduling, it is important to consider various performance events interchangeably according to the overall characteristics of the workload sets.

4.1.1 Multi-level Co-scheduling Policy

Following our analysis on the factors that affect the overall energy-efficiency of the multi-core systems, we propose a novel multi-level co-scheduling policy to improve the energy-efficiency. Our multi-level co-scheduling policy utilizes two previously proposed co-scheduling policies: (1) Cache Miss based (Bhadauria and McKee, 2010), (2) IPC*CPU-utilization based (Dhiman et al., 2009). Our proposed policy comprises the following two main steps:

1) Computation-to-communication ratio of the benchmarks is an important met-

ric to identify the CPU and bus requirements of the benchmarks. Main resource limitation for workload sets that have higher computation-to-communication ratio is the cache misses. We experimentally derived a threshold (Th_1) of 5000 and classify the workload sets according to their sum of $IPC * CPU_{Util}/BUS_{acc}$ values. Workload sets that are computationally intensive and have high bus accesses are co-scheduled using the $IPC * CPU_{Util}$ metric to balance the computation-to-communication ratio of the benchmarks. We use cache misses as the co-scheduling policy to balance the applications' memory requirements if they don't require heavy communication.

2) Benchmarks that are computationally intensive might suffer from high memory utilization or bus accesses. This step can be called as a fine-tuning step, as it identifies whether there is stronger bottleneck than the cache misses for computationally intensive benchmarks. We evaluate the memory utilization (MEM_{Util}) and BUS_{acc} for the computationally intensive benchmarks. Computationally intensive workload sets that have high memory utilization ($Th_2 = 25$) or high bus accesses ($Th_3 = 0.1$) are co-scheduled to balance $IPC * CPU_{Util}$ values, rather than the cache misses. Rest of the workload sets would have only cache misses as their major bottlenecks. Thus we co-schedule them through the cache miss based policy. Algorithm 1 demonstrates the details of the policy selection algorithm.

ALGORITHM 1: Policy Selection

```
if IPC * CPU_Util / BUS_acc > Th_1 then
  Balance: CacheMiss
  if IPC * CPU_Util > Th_2 and (BUS_acc > Th_3 or MEM_Util > Th_4) then
    Balance: IPC * CPU_Util
  end if
else
  Balance: IPC * CPU_Util
end if
```

4.1.2 Experimental Results

We implemented bus access, cache miss and IPC*CPU-Utilization based policies that are previously proposed. We compare our multi-level co-scheduling policy with previously proposed approaches and demonstrate E/w saving improvements over randomly generated 50 workload sets that are shown in 4·3. We also show the optimum policies for each workload set. Proposed multi-level policy chooses the best performing policy with an accuracy of 86%.

In order to compare our experimental results with previous co-scheduling techniques, we implement three naive co-scheduling algorithms that are based on previous approaches (Dhiman et al., 2009) (Bhadauria and McKee, 2010). These two approaches focus on stalls based on memory and bus accesses as well as computa-

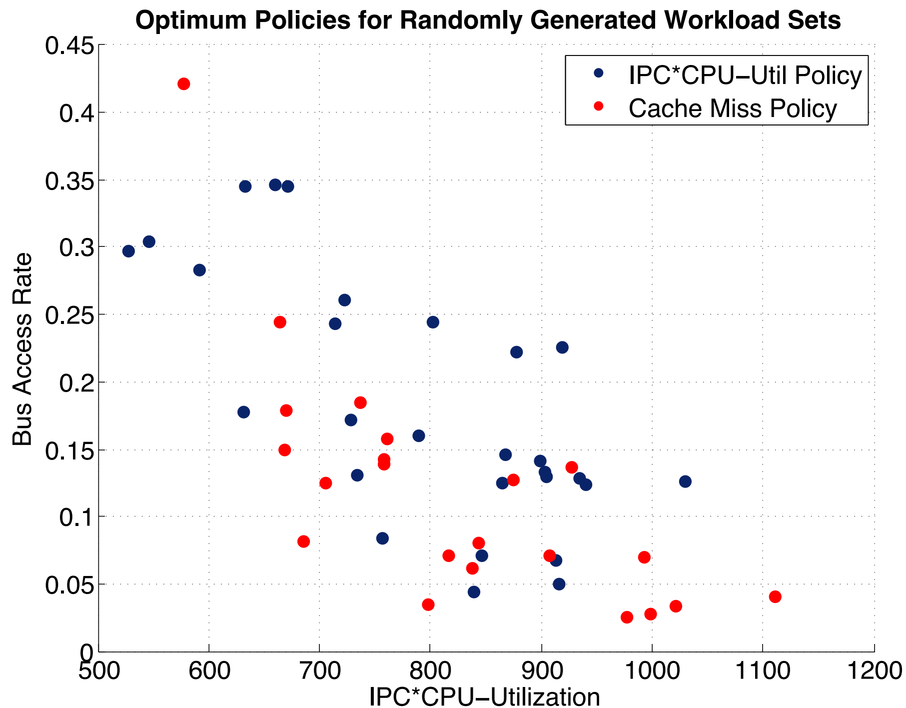

Figure 4·3: Performance characteristics of randomly generated workload sets and the optimum policy for each workload set.

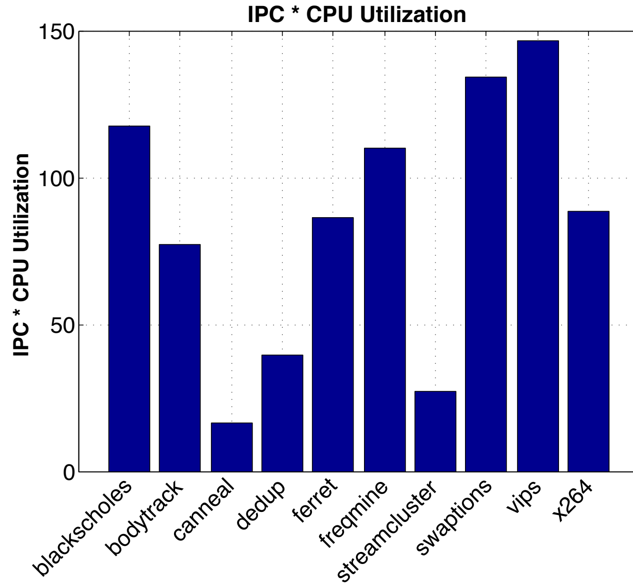

Figure 4·4: IPC * CPU Utilization for PARSEC benchmarks running 12 threads.

tional intensity of the benchmarks to make co-scheduling decisions to achieve better energy efficiency.

We use the cache-miss rate measurements to balance the cache misses across various consolidation combinations. As a naive implementation, we co-schedule the benchmarks starting from the benchmarks that has the highest and the lowest cache misses. Figure 4·4 shows $IPC{*}CPU_{Util}$ values of 10 PARSEC benchmarks. Similar to cache miss balancing approach, this time we balance bus accesses and $IPC * CPU_{Util}$ values across benchmarks as the co-scheduling policy.

In Figure 4·5, we show average E/w savings for 50 workload sets with respect to 12 thread execution of the workloads running on a single node. Bus access based policy performs worst among all policies. Our proposed multi-level co-scheduling policy provides 31.8% savings, which is 9.1% higher than the best performing previous co-scheduling policy.

In Figure 4·6, we show maximum E/w saving improvements due to multi-level

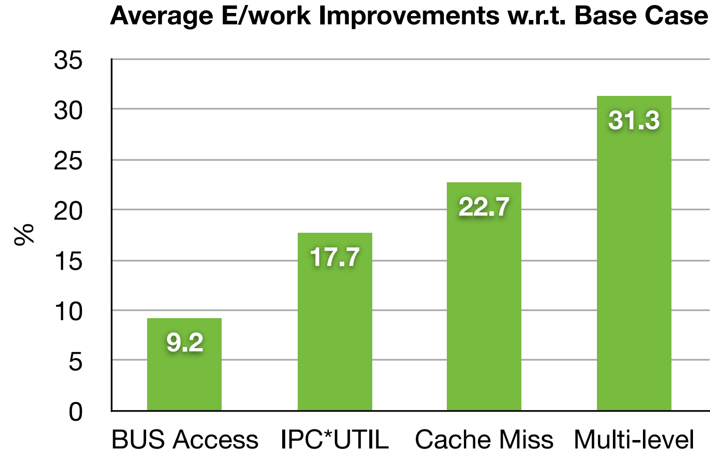

Figure 4·5: Average E/w saving improvements for 4 policies w.r.t 12 thread execution on a single node.

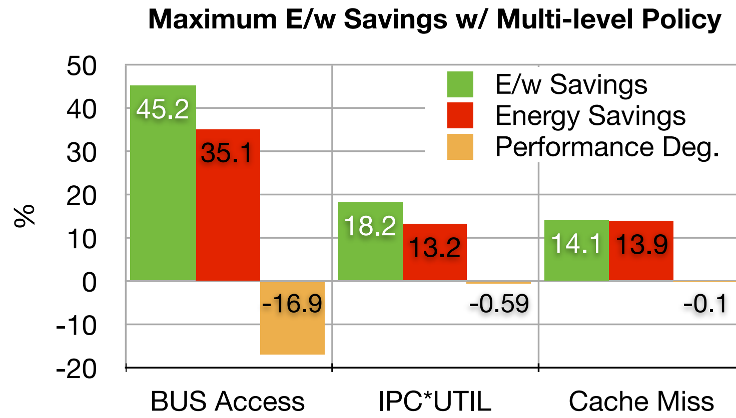

Figure 4·6: Maximum E/w saving improvements due multi-level policy w.r.t previously proposed policies.

policy. We also report energy savings and performance degradation for the maximum saving case. Our proposed approach improves the E/w savings by 14.1% with respect to the best performing cache miss based policy with 0.1% performance degradation. E/w saving improvements reach up to 45.2% with respect to the worst performing policy with a performance degradation of 16.9%.

4.2 Co-scheduling Analysis in Virtualized Environments

This section investigates the impact of co-scheduling on application performance under various system settings and policies. The goal is to develop an understanding of the tradeoffs and constraints in virtualized environments to enable better policy design for runtime management. We first evaluate whether selecting which applications to co-schedule together on the same resources changes the overall energy efficiency. We then evaluate the effect of CPU binding and NUMA scheduling on virtualized and native environments in terms of throughput and performance isolation.

4.2.1 Application Selection Based Co-scheduling

Co-scheduling the application pairs that have contrasting performance characteristics, such as high IPC and low IPC, is expected to improve the energy efficiency significantly, as it leads to more balanced resource usage (Dhiman et al., 2009; Bhadauria and McKee, 2010).

We first investigate the performance impact of co-scheduling as the number of cores/threads per application changes. In the first case, we run each application with 2 threads and bind both applications on the same chip. In the second case, each application runs with 6 threads and the applications are located on separate dies. Two applications share an L3 cache in the first case and have their own L3-cache in the second case. Figure 4·7 shows the per-core throughput for `canneal` when it is co-scheduled with the other benchmarks (2 application at a time). We choose `canneal` as it has the highest amount of L2 and L3 cache misses. As we have experimentally determined that throughput is a meaningful indicator of the application progress in PARSEC, we use throughput as our performance metric. For a system with low performance isolation, the performance of `canneal` is expected to vary significantly depending on the co-runner application (x-axis). On contrary, for a system

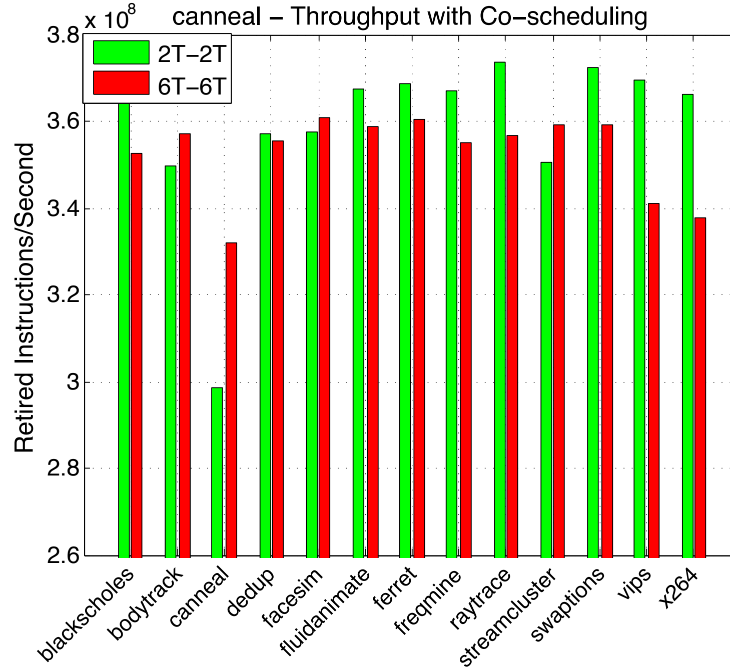

Figure 4·7: Performance comparison when `canneal` is co-scheduled with the other PARSEC benchmarks shown on the x axis. Figure demonstrates `canneal`'s throughput (per core) when each of the two co-scheduled applications runs with 2 threads and when each has 6 threads. Performance impact of co-scheduling is higher at 2 threads due to higher resource contention at the shared caches.

with higher performance isolation, the performance of `canneal` is expected to show little variation regardless of the co-runner application. Co-scheduling `canneal` with another instance of `canneal` significantly affects the throughput due to high resource contention on the last level cache. As Figure 4·7 shows, maximum throughput degradation (`canneal - canneal` pair) with respect to the average throughput is 18% for the 2 thread case and 6% for the 6 thread case. In other words, increasing the thread count and separating the last level caches reduce the performance impact resulting from the specific characteristics of the co-runner application.

In order to further quantify our observation, we evaluate the *throughput-per-watt* of for three distinct workload sets (i.e., memory-bound, cpu-bound, mixed) under var-

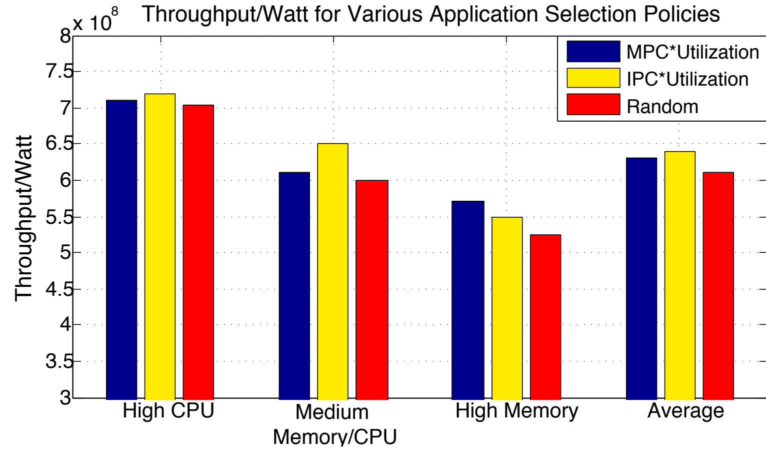

Figure 4·8: Throughput-per-watt for various benchmark sets that are co-scheduled with various policies. On average, randomly co-scheduling applications provides comparable energy efficiency in comparison to previously proposed based policies.

ious co-scheduling (i.e., application selection) policies. We use throughput-per-watt as a metric of energy efficiency, as it captures the useful work done per watt consumed (Bhadauria and McKee, 2010). We implement co-scheduling policies that are based on determining the best application pairs to co-schedule together (Dhiman et al., 2009; Bhadauria and McKee, 2010). These techniques first rank the applications according to a selected metric and then co-schedule the highest ranked benchmark with the lowest one, and proceed through the ranked list in a similar fashion. In this way, these policies try to balance the resource usage by co-scheduling applications that have contrasting characteristics. We evaluate MPC*CPU Utilization and IPC*CPU Utilization as the metrics used while ranking the applications' resource usage. These metrics are proposed previously to derive vCPU-level IPC and MPC by using the CPU utilization of each vCPU as the weight factor (Dhiman et al., 2009). We also evaluate the throughput-per-watt of the workload sets when applications to be co-scheduled are selected randomly.

Figure 4·8 compares the energy efficiency of three distinct benchmark sets under

various co-scheduling policies. IPC* Utilization performs best for medium and high CPU benchmark sets, whereas MPC*Utilization is the best policy for highly memory intensive benchmark set. However, on average, best performing policy improves the energy efficiency by 4% in comparison to the random policy.

This observation implies that the benefits from application policies are limited for a system with high performance isolation. Thus, we need other mechanisms in addition to application selection to continue to improve the energy efficiency of virtualized servers. Next, we investigate the system configuration that provides the best performance isolation.

4.2.2 Performance Isolation on Consolidated Environments

In this section, our goal is to find the optimum system configuration that provides the highest performance isolation. We investigate the effect of CPU binding and NUMA scheduling on the performance in virtualized (ESXi) and native OS environments (Ubuntu Server 12.04). We use CPU binding and NUMA scheduling as they are inexpensive, commonly available, yet powerful knobs for improving performance isolation.

CPU Binding: For multi-threaded workloads, CPU binding refers to pinning threads of applications to a specific set of cores (i.e., affinities). CPU binding prevents OS to migrate threads outside the defined affinity set. Binding typically reduces the contention on NUMA nodes by prioritizing the use of local NUMA nodes. Performance of the NUMA systems is sensitive to memory layout and thread affinities as access latency depends on the distance to remote and local nodes. For instance, AMD Magny Cours has two NUMA nodes and each NUMA node is a local node to one of the 6-core dies, while the other is a remote NUMA node. Binding applications to one of the 6-core dies ensures that threads will prioritize their local NUMA node for

data accesses. For co-scheduling two applications, binding each application to one of the 6-core dies reduces the contention on the NUMA nodes. Therefore, CPU binding improves the performance isolation by ensuring data locality.

NUMA Balancing: Balancing the accesses across the NUMA nodes reduces the performance cost of resource contention on the NUMA nodes. ESXi hypervisor includes a NUMA balancer to reduce the data access latency by monitoring the data access patterns of the VMs. NUMA balancing across VMs reduces the contention by assigning a different NUMA node to each VM.

Figure 4·9 shows the performance variation for all the PARSEC benchmarks during co-scheduled execution under both virtualized and native environments on our server. Higher performance variation implies that the application performance is affected more significantly by the co-runner application, which means poor performance isolation. For all experiments on the native OS, we co-schedule 2 applications at a time, each of them running with 6 threads. We test the native OS with and without CPU binding. For the virtual system, we test 3 different cases. We create 12 vCPUs (12 threads) per VM for the *"VM w/o binding w/ NUMA Bal."* configuration, and 6 vCPUs (6 threads) per VM for the other configurations.

As Figure 4·9 shows, both virtual and native environments have significantly higher performance variations in absence of CPU affinities. Also, disabling the NUMA balancer on the virtual system results in higher performance variation for the virtual environment. This is because NUMA balancer assigns each VM to a specific NUMA node, even when we do not set CPU affinities. As a result, virtual systems with NUMA balancing consistently provide lower performance variation. The VM with 12 vCPUs, where only NUMA balancer is enabled, has slightly worse performance isolation compared to the VM with 6 vCPUs with both CPU binding and NUMA balancer enabled.

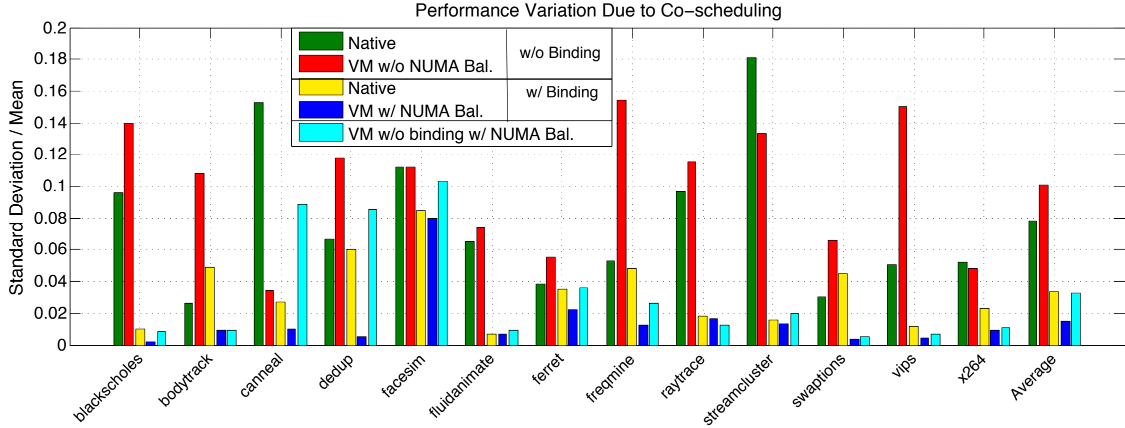

Figure 4·9: In this experiment, a PARSEC benchmark is co-scheduled with another benchmark (only two benchmarks at a time) under various CPU binding and NUMA balancing settings. The experiment is repeated to cover all possible application pairings. Figure shows the performance variation (standard deviation/mean) of each benchmark across its co-scheduled runs with the other benchmarks. Smaller bars indicate better performance isolation.

Figure 4·10 evaluates the maximum, minimum and average throughput across all application pairs that are co-scheduled, comparing various binding and NUMA balancing configurations. Ideally, we would like to choose the configuration that provides high performance isolation and comparable performance to the native system. Native and virtual systems with CPU binding provide higher average throughput compared to the systems without CPU binding or NUMA balancing. As expected, native system provides slightly higher throughput compared to the virtual system because of the virtualization overhead, which is limited to 2%.

We choose the *VM without binding, with NUMA balancer*, as our co-scheduling environment due to its comparable performance to *VM with binding*, high performance isolation and its advantages in practical implementation. For clarity purposes, we leave further reasoning of this design choice to Section 4.3.1.

Our analysis on performance isolation shows that application performance can be affected by the co-runner application, especially when the performance isolation is

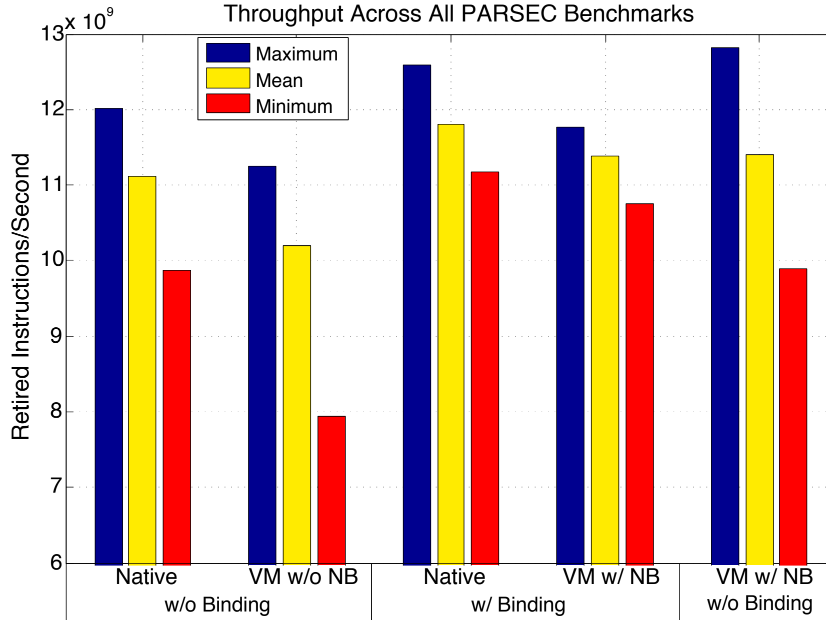

Figure 4·10: Maximum, minimum and average throughput across all co-scheduled pairs for the native and virtual system. Figure shows the effect of CPU binding and NUMA balancer on the performance.

poorer. Although application selection policies improves the energy efficiency of the consolidated servers, these improvements are limited for a system with high performance isolation. We observe that it is possible to reduce the performance variation resulting from co-runner application selection by optimizing CPU and NUMA node affinities. We next describe our autonomous resource allocation policy for multi-threaded workloads, which aims to improve the energy efficiency through energy proportional resource allocation.

4.3 Adaptive Resource Sharing for Multi-threaded Workloads

This section presents our adaptive resource sharing technique for multi-threaded workloads. Our technique maximizes the energy efficiency of a server node by allocating

resources to VMs based on application-specific power and performance characteristics. The goal is to provide more resources to energy-proportional applications. We first present a clustering-based application classification technique that identifies the energy proportionality of the applications. Our resource sharing technique then uses this classification for dynamically changing the CPU resources allocated for each VM on the server.

4.3.1 Predicting Application Energy Efficiency

To improve the energy efficiency of the system, we aim to proportionally allocate the resources depending on the energy efficiency levels of each co-scheduled application. Therefore, the first step is to identify a metric that reflects the energy efficiency of the applications accurately.

IPC and CPU utilization are commonly used metrics to evaluate the performance and power characteristics of applications (Dhiman et al., 2009) (Isci et al., 2006). CPU utilization measures the busy cycles of a processor and reports the percentage of the time that the CPU is actually processing instructions (i.e., CPU is not idle). However, none of these two metrics alone capture the overall energy efficiency characteristics of the application. A high-IPC application might still utilize the CPU at lower rates and similarly, an application that highly utilizes the CPU might have lower IPC rates due to factors such as cycle stalls, data access latencies and communication. While CPU utilization is measuring how often the CPU is busy (i.e., % of unhalted state), IPC measures the activity only when the CPU is in the unhalted state. Thus, combining these two metrics provides a good estimate of the activity of the CPU over time. Moreover, for cases such as spin-loops or high cache misses evaluating only IPC or utilization would likely be misleading. We measure the IPC and utilization for the spin loop example on our system. Although IPC for the spin-loop example is around 1, utilization goes down to 1-2%. Therefore, we observe that combining IPC

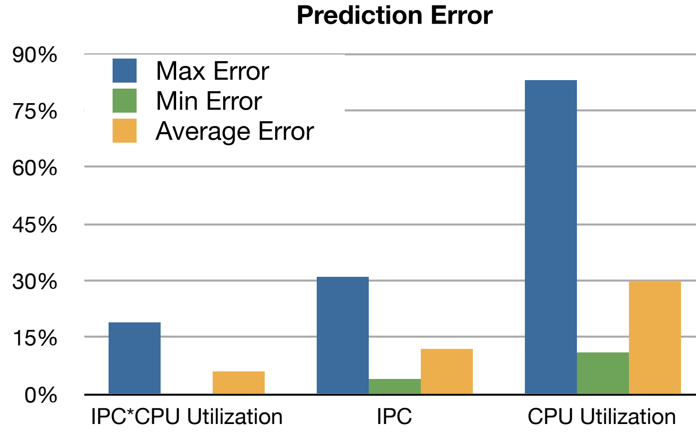

Figure 4·11: Maximum, minimum and average errors for predicting energy efficiency of PARSEC benchmarks for candidate metrics.

and utilization captures the real performance more accurately compared to observing solely the IPC or utilization.

In order to quantify the accuracy of each metric (i.e., IPC and CPU utilization), we perform linear regression analysis. Figure 4·11 shows the average error for each metric to predict the energy efficiency of all the 13 PARSEC benchmarks. *IPC*CPU Utilization* outperforms both IPC and CPU utilization metrics with only a 6% average error rate in predicting the energy efficiency of the applications.

Figure 4·12 demonstrates the correlation between the application energy efficiency and the *IPC*CPU Utilization* metric across the PARSEC suite. Pearson correlation coefficient between *IPC*CPU Utilization* and throughput-per-watt is 0.95 and the p-value (significance test) of the correlation is less than 0.01, demonstrating high relevance. These values quantify the strong correlation between *IPC*CPU Utilization* and throughput-per-watt. As a result, we use *IPC*CPU Utilization* as an accurate measure of application energy efficiency while characterizing applications.

We design an application classification scheme that applies density-based clustering (DBSCAN) (Ester et al., 1996) using the chosen metric, *IPC * CPU Utilization*. Clustering here refers to separating applications of different power/performance char-

acteristics into different classes. We use DBSCAN clustering algorithm to classify benchmarks, which does not require *a priori* knowledge of number of clusters. DBSCAN discovers the clusters automatically based on a density reachability threshold, ε. Let us assume each data point on an IPC vs. utilization plot (e.g., see Figure 4·13) is a *node*. Thus, each node represents a benchmark. A neighbor node, q, is *density reachable* from node p, if the distance between q and p is less than the density reachability threshold, ε. *Density reachability test* essentially determines whether two nodes belong to the same cluster based on their distance.

DBSCAN starts from an arbitrary point, p, and discovers all neighbor nodes that are density-reachable. Distance between clusters S_1 and S_2 (i.e., set of points) is given as the minimum distance across all member points, p, q, where $\forall p \in S_1$, $\forall q \in S_2$.

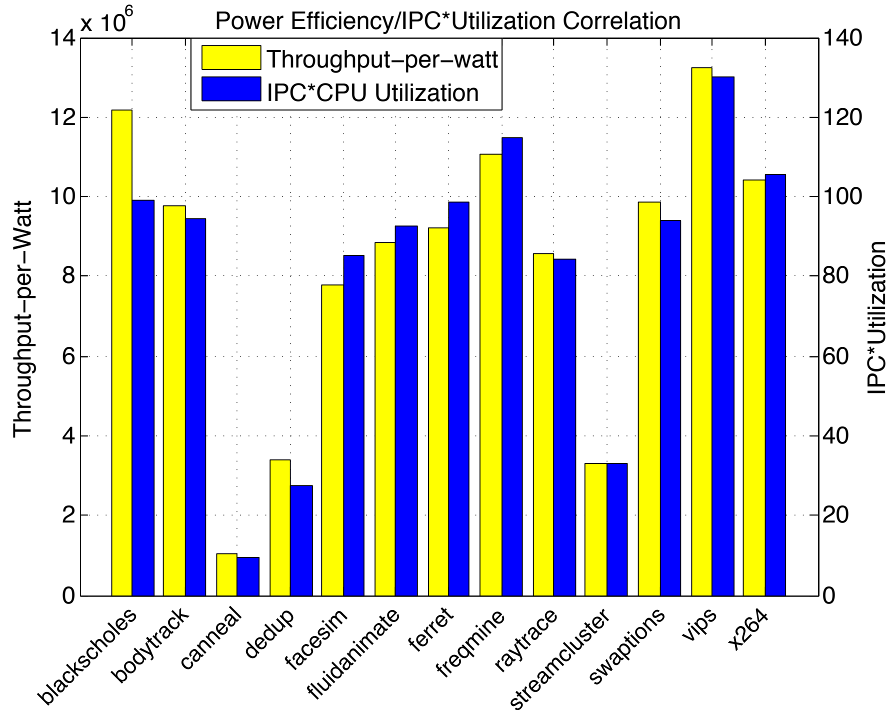

Figure 4·12: Correlation between *IPC*CPU Utilization* (right axis) and application energy efficiency (left axis).

Clusters are expanded or merged only if: $\forall p, q : dist(p, q) < \varepsilon$

Based on our experimental analysis, we choose ε=20 as the minimum distance between two clusters. We set the ε value such that it is close to the standard deviation of the IPC*CPU utilization of the PARSEC benchmarks, which is 19.2. Therefore, we aim to capture the energy efficiency deviations across benchmarks. Choosing a smaller ε will result in a higher number of clusters, which creates a finer granularity classification at the cost of a larger runtime for running DBSCAN. On the other hand, choosing a larger ε will reduce the number of clusters, and therefore may not have a granularity that is sufficient to capture the various energy efficiency classes. In Figure 4·13, we show the cluster classes (i.e., *Class-1* to *Class-4*) and the members (i.e., benchmarks) of each cluster class. Benchmarks that belong to *Class-4* are the most energy-efficient benchmarks, where as the *Class-1* corresponds to the lowest energy efficiency class. Although the clustering algorithm utilizes a single metric (IPC*CPU utilization), in Figure 4·13 we show applications on a 2D plot in order to illustrate the distribution of applications more clearly. We use the average IPC*CPU utilization data for each application to create a representative training data set for our model. Therefore, each benchmark in Figure 4·13 basically represents the center data point among all phases of the application, as it is the mean of all phase data points. We empirically observe that using average benchmark characteristics during classification is sufficient for our purposes. It is also possible to sample phase data and use a larger number of input data points in DBSCAN. However, using average characteristics provide a more light-weight implementation. As long as the initial training set covers a representative set of application characteristics, the offline classification scheme will work for *unknown* applications. For applications that do not fit within the current classification scheme (i.e., outliers), we compute a new ε for the new data set and rerun the classification. As the DBSCAN algorithm has $O(nlogn)$ average

complexity, runtime overhead for re-classification is low (Ester et al., 1996). Running the DBSCAN classifier adds a 0.1 to 0.2 second latency to the resource allocation decisions without affecting the application performances.

VM Reconfiguration

For server-level resource management, modern hypervisors provide resource control knobs to the administrators to manage the resources allocated for VMs (KVM, 2008) (Xen, 2009). These resources (e.g., CPU, memory, disk) can be reconfigured through the hypervisor during the runtime, without any need for restarting the VMs. ESXi hypervisor provides various resource control knobs such as vCPUs hot plugging, and adjusting resource reservations, limits and shares.

Hot plugging feature of the ESXi hypervisor enables adding vCPUs or increasing the size of the virtual memory of the VMs during runtime. However, these features require support from the guest OS and some of the OSes today do not support

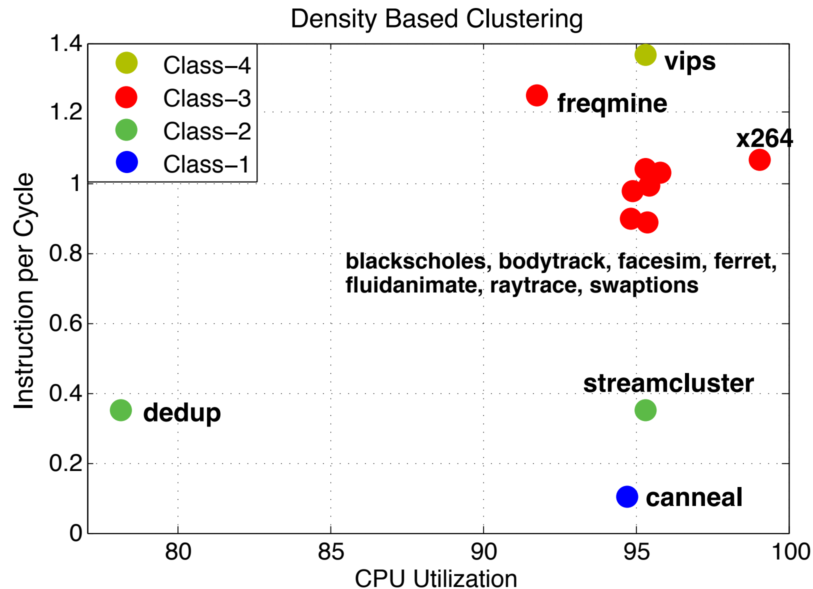

Figure 4·13: Benchmark classification through density based clustering. Class-4 represents the highest level of energy efficiency, where as Class-1 represents the lowest level.

unplugging of the resources. An alternate technique for resource management on ESXi is adjusting the limits of the allocated resources (i.e., CPU and memory). Resource limits restrict the resource usage of the VMs. By adjusting the limits, it is possible to control the resource usage of individual VMs to optimize the performance and power tradeoffs. In this work, we propose running each VM with 12 vCPUs and using the limits settings for resource allocation.

Running a larger number of vCPUs might introduce higher overhead on the hypervisor side, as the hypervisor needs to handle higher number of vCPUs and multiplex them to run on the physical CPUs(pCPUs). In order to quantify the overhead of running a higher number vCPUs on the performance of the applications, we compare two scenarios. In the first scenario we create 2 VMs with 6 vCPUs each with CPU binding and the NUMA balancer. In the second scenario we create 2 VMs with 12 vCPUs each and we limit the resource allocation of each VM to the level equal to running 6 vCPUs. The overhead of 12-vCPU case is less than 2% with respect to 6-vCPU case.

On the other hand, applications such as `bodytrack`, `dedup` do not exhibit performance benefits from increasing amount of resources, due to software characteristics (Khan et al., 2011). We see similar performance scaling behavior on both native and virtual environments (Hankendi and Coskun, 2013). We use the default ESXi scheduler tick (30ms), which provides comparable performance to the native environment.

4.3.2 Autonomous Resource Sharing

By utilizing the benchmark classes that are derived from the DBSCAN algorithm (Ester et al., 1996), we allocate CPU resources to each benchmark by assigning more resources to the more energy-efficient ones. Based on the benchmarks classes, we compute a weight, w_i, for each class, where $\sum_{i=1}^{n} w_i = 1$ for all the n applications. We use average IPC*CPU utilization of each class, u_i, to distribute the resources

proportionally with the energy efficiency classes of the applications. We compute the weight of each class as $w_i = u_i/U$, where $U = \sum_{i=1}^{n} u_i$. We then use w_i to compute the amount to resources allocated (r_i) to each application, as $r_i = w_i * R$, where R is the total amount of available resources. On the ESXi environment, available resources are represented in units of frequency, f (MHz). We allocate CPU resources (r_i) for VMs, such that $\sum_{i=1}^{n} r_i = 23940MHz$, where 23940 MHz (12 pCPUs) is the maximum available CPU resources (R) for the VMs on our system. For the rest of the section, we will represent CPU resources in terms of the number of pCPUs rather than MHz.

We adjust the resources in a granularity that is equal to the compute resources of a single pCPU. Initially each co-scheduled application is executed with equal resources, and we start monitoring the IPC and CPU utilization. We dynamically store the application throughput, when both applications have equal resources, as the reference throughput to later calculate the gains and loss due to changing resource allocations. We then utilize the benchmark classification and allocate resources (r_i) to VMs proportional to the energy efficiency classes by using the *weight* of each class, w_i, as explained in Section 4.3.1. While tracking application phases, we also monitor the throughput gain and losses. If the throughput gains for the higher class benchmark is higher than the throughput loss for the lower class benchmark, we continue to increase the resources for the higher class benchmark and decrease resources for the lower class. If the classes change due to application variation, we restart the algorithm with equal resources.

Offline benchmark classification is used as a lookup table (LUT) to adaptively adjust the resource sharing across VMs during runtime. LUT stores the IPC and CPU utilization values and the corresponding energy efficiency classes, and the maximum, minimum and mean IPC*CPU utilization values and the *weight* for each class.

After we sample the IPC and CPU utilization of the applications, we access the LUT to determine the class of the current phase of the application. Thus, our runtime implementation is able to capture the potential phases that might occur during the execution of an application. If IPC*CPU utilization reading falls beyond the maximum threshold of the highest class, or falls below the minimum threshold of the lowest class, we rerun the DBSCAN to include the outlier workload phases.

4.3.3 Runtime Implementation

We implement our autonomous resource sharing technique on a real-life multi-core server. Our architecture consists of a management node (vCenter terminal) and virtualized server(s). Figure 5·9 shows the architecture of our implementation. Utilizing a centralized management node is a common practice on VM environments (e.g., VMware‘s DRS (VMware DRS, 2009)). DRS can be utilized to manage the resources through a centralized controller (e.g., vCenter). The default VM management framework of VMware uses SDKs and APIs, some of which are leveraged in our implementation. In general, data center administrators do not always have access to the hypervisor code and management through a centralized node brings ease of implementation. Our technique, however, could be implemented within the hyper-

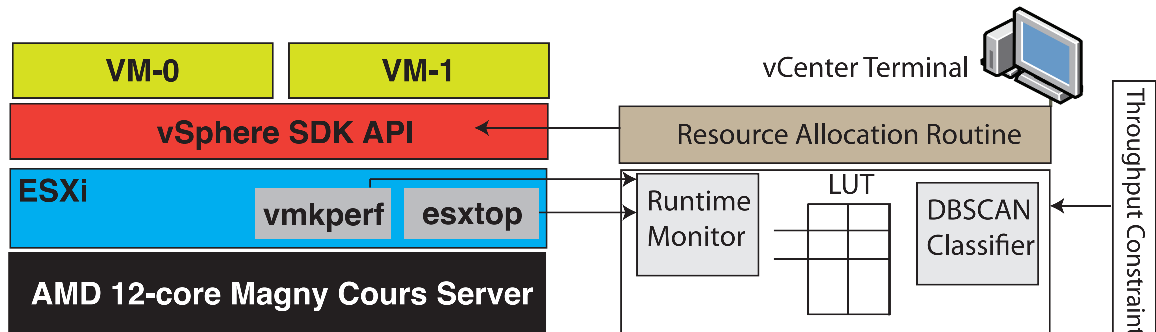

Figure 4·14: Runtime implementation of the resource allocation technique (Hankendi and Coskun, 2013).

visor as well for open-source hypervisors. In a multiple node scenario, management node still serves as the centralized resource manager. Each host (server node) is then interfaced to the management node through the default vSphere SDK. Thus, extension to multiple nodes and/or to multi-socket chips requires no major modification to the implementation. In this work, we demonstrate the capabilities of our runtime implementation for a single server node.

We use an Intel i3 dual-core processor based machine as our management node, which runs Ubuntu 12.04 as its OS. In the training phase, we collect IPC and CPU utilization for each PARSEC benchmark and run the DBSCAN classifier, which we implement as a Python script. Management node periodically collects runtime performance statistics from the ESXi hypervisor then utilizes the LUT to classify each sampling phase. The runtime monitor polls VM-level performance counter readings (i.e., retired instructions, clock cycles) and CPU utilization from the ESXi every 2.5 seconds, which is the minimum achievable sampling period. Data processing/polling VM-level performance counter readings adds 0.3 to 0.4 second additional latency to the sampling process. The resource allocation routine communicates with the ESXi through the vSphere SDK that performs administrative tasks (i.e., VM reconfiguration) on the ESXi host and makes the resource allocation decisions based on the energy efficiency class of the co-scheduled applications

4.3.4 Consolidation with Throughput Constraints

Allocating fewer resources to the applications that are not energy-efficient may have negative impact on the throughput of some workloads. To provide the capability of controlling the maximum throughput degradation arising from resource sharing decisions, we implement a feedback mechanism into the resource allocation routine. The routine continuously monitors the throughput changes on each application. The feedback mechanism takes maximum performance degradation as a user input, and

compares the current performance of the application with respect to the baseline case, in which all VMs are allocated equal resources. If the maximum performance degradation limit is exceeded, feedback mechanism asserts a signal to the resource allocation routine. Feedback mechanism can send signals specific to each VM (i.e., *VM0-alert*, *VM1-alert*). When the resource allocation routine receives a feedback signal, it increases the resources for the VM that has the throughput violation and decreases the resources for the other VM by one step (1 pCPU). Implementing a feedback mechanism extends the capabilities of our system in two ways:

(1) Throughput degradation of the applications varies depending on the application performance characteristics. Figure 4·15 shows the throughput degradation as a function of CPU resource limits of 4 PARSEC benchmarks. The throughput of each application in the figure is normalized with respect to the highest CPU resource limit of 12 pCPUs. Throughput of `dedup` is minimally affected up to 6-pCPU resource limit. However, throughput of `blackscholes` shows almost linear correlation with the CPU resource limits. As `blackscholes` highly utilizes the CPU resources, it is more sensitive to the changes in the CPU resource limits. On the contrary, `dedup` has the lowest CPU utilization rate, thus the throughput is consistent between 12 and 6-pCPU resource limits. Our feedback mechanism allows us to capture such application characteristics and tune the resource allocation decisions.

(2) By utilizing the feedback mechanism, we can provide the user an option to limit the maximum throughput degradation on a specific application. Our resource allocation routine takes the maximum performance degradation as an input and makes resource allocation decisions accordingly. Users can also set application-specific performance limits (e.g., minimum frames per second in `bodytrack`) by utilizing frameworks such as Heartbeats (Hoffmann et al., 2011).

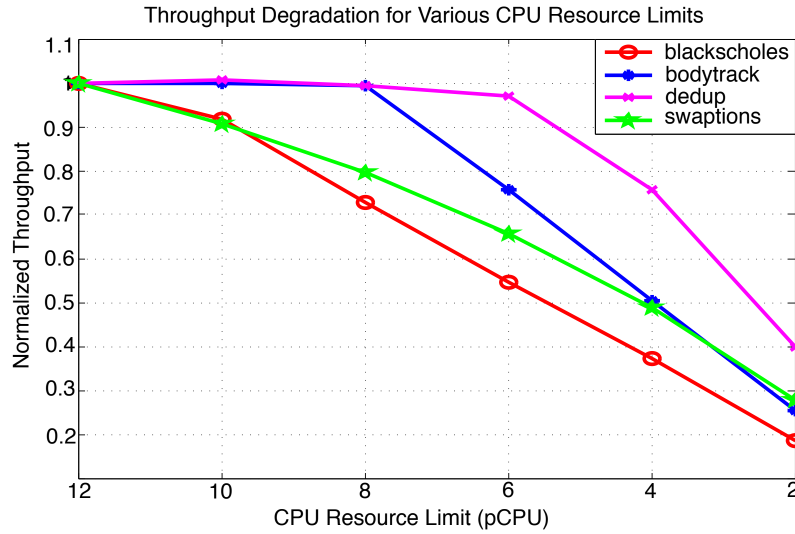

Figure 4·15: Throughput degradation of 4 PARSEC benchmarks as a function of CPU resource limits.

4.4 Experimental Results

In this section, we first present the runtime behavior of the resource allocation technique on a real-life server. We then show the capabilities of the feedback mechanism that enables user to enforce throughput guarantees to selected applications. Finally, we report average throughput-per-watt improvements over the existing co-scheduling policies.

4.4.1 Runtime Behavior

Figure 4·16 demonstrates the runtime behavior of our technique for 3 application pairs without any throughput constraints from the user. We show the throughput of each VM after both applications reach their parallel phases (at the top), and the CPU resource limits (at the bottom), which are imposed by our method. We first co-schedule `blackscholes` and `swaptions`, which are both in the same power-efficiency class (Class-3). Although they are in the same class, our resource allocation routine can adapt to application variations and allocate more resources to `blackscholes` at

the 20th second. For the `canneal-freq-mine` pair, resource allocations are adjusted 4 times, at t=50, 65, 70 and 75s, due to application variations. Between t=87.5s and t=102.5s, `bodytrack` is allowed to use CPU resources that is equal to 8 pCPUs, and `dedup` is allowed 4 pCPUs. After t=100, our technique favors `bodytrack` and increases its pCPU resources to 9 pCPUs, and reduces the resource limits for `dedup` to 3 pCPUs.

In Figure 4·17, we demonstrate the runtime behavior under user-defined throughput constraints for the same application pairs. For 3 application pairs, we choose 20%,15%, and 10% maximum throughput degradation limit as examples to demonstrate the runtime behavior when there is a performance constraint on individual applications. We show the throughput constraints with the horizontal dash lines. As soon as the throughput constraints are violated, feedback mechanism forces the resource allocation routine to increase the resources for the application that falls below the minimum throughput limit. In the figure, both applications have equal resources until t=22.5s. At t=22.5s, throughput for `swaptions` (VM1), falls below the maximum throughput degradation limit due to the resource allocation decision. Therefore, feedback mechanism signals the resource allocation routine to increase the resources of VM0, and to reduce the resource of VM1. Similarly at t=45s, throughput constraints are violated and resource allocation routine performs similar actions.

In the results presented so far, we synchronize the parallel phases of the applications through the `consolmgmt` interface as explained in Section 3.6. However, our implementation works seamlessly as applications go in and out of parallel phases and does not disrupt the default scheduler decisions. If we consider the entire execution of the applications, throughput-per-watt gains are expected to be much higher in comparison to the ROI execution, as there will be higher gains when one application is in serial and the other is in its parallel phase. For instance, throughput-per-watt

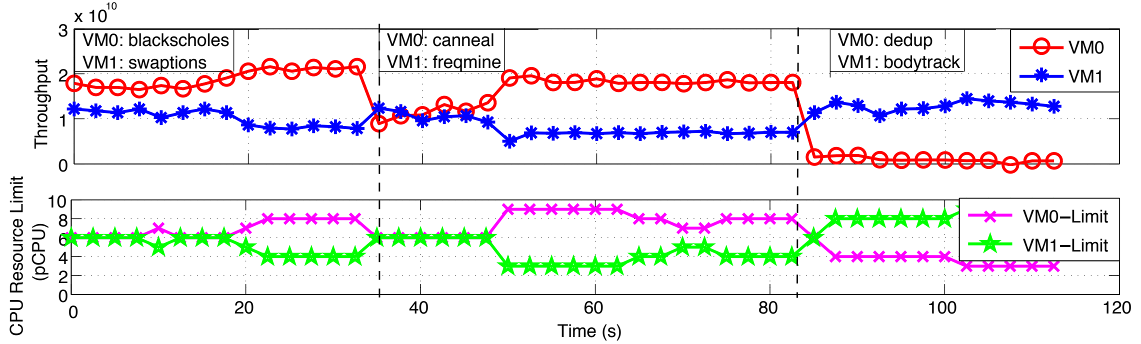

Figure 4·16: Runtime behavior of the resource allocation routine for 3 applications pairs. CPU resources are adjusted according to power efficiency classes of the applications to improve the overall efficiency of the server.

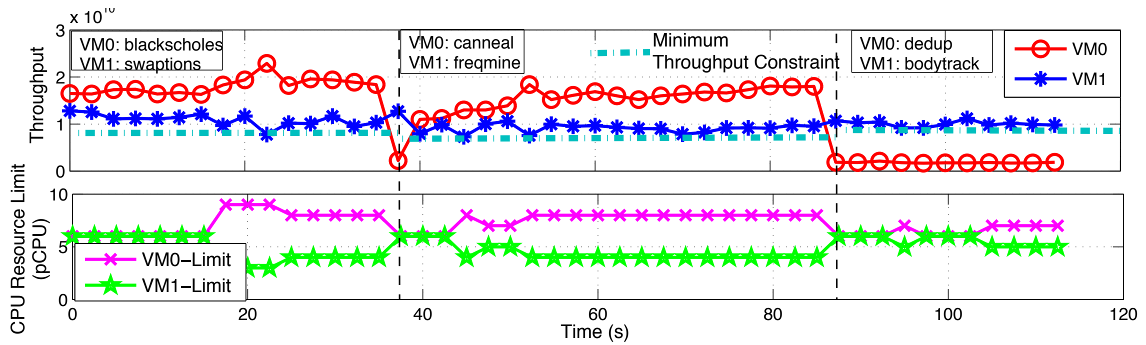

Figure 4·17: Runtime behavior of the resource allocation routine with performance guarantees.

improvements for `blackscholes-raytrace` pair reaches to 22% for the entire execution and 9% for the ROI execution. For all of the other results, we report energy efficiency improvements for the ROI only, as energy gains during parallel phases are more valuable in real-life settings.

4.4.2 Evaluation for Various Cluster Workload Sets

In order to evaluate impact of our technique on the overall energy efficiency of a cluster in a real-life scenario, we generate 50 random workload sets, each with 10 benchmarks as in the three workload sets described in Section 4.2. For each workload set, we evaluate the application selection based co-scheduling policies (i.e., us-

ing memory-per-cycle (MPC)*Utilization and IPC*Utilization metrics), the proposed technique and the combination of application selection policies and the proposed approach (i.e., *Proposed+IPC*, *Proposed+MPC*). Figure 5·1 compares the throughput-per-watt of each technique with respect to the baseline case, where each VM is given maximum resources of 12 pCPUs (Base). We report maximum, minimum and average improvements of the 50 workload sets. The proposed technique together with MPC*Utilization application selection policy improves the energy efficiency by 21% on average, whereas *MPC*Utilization* policy alone improves the energy efficiency by only 4% with respect to the baseline.

Figure 4·19 compares the average throughput-per-watt, throughput, power and energy consumption of the workload sets for various techniques. We normalize the values with respect to the baseline case. As Figure 4·19 shows, throughput-per-watt improvements are due to achieving increased throughput for a marginal power

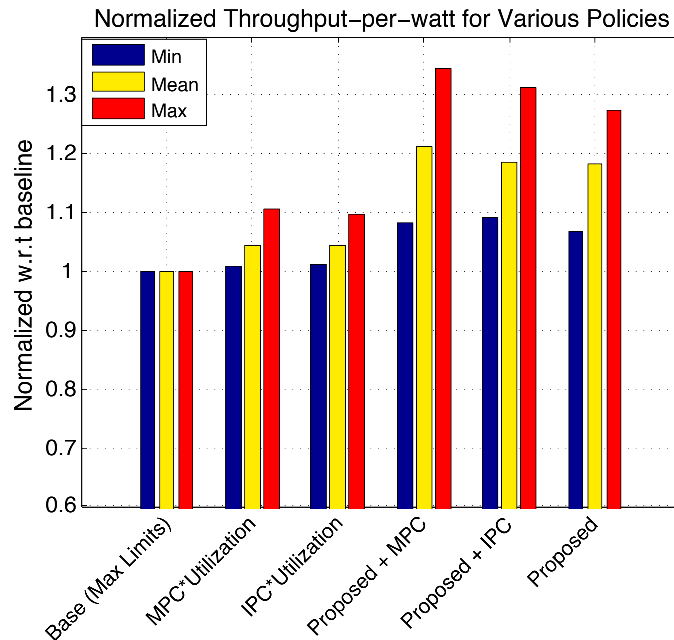

Figure 4·18: Normalized throughput-per-watt with respect to the baseline case, where each VM is given the maximum resources, for randomly generated 50 workload sets.

increase. The proposed technique alone improves the throughput by 21% with 3% increase in power consumption, which leads to 16% lower energy consumption with respect to the baseline case.

We also test our research allocation technique for co-scheduling 3 applications on 3 VMs on the same server. As a case study, we randomly select 12 applications (i.e., 4 application triples) and co-schedule 3 of them at a time. We compare the energy efficiency improvements with the baseline case and the 2-VM case, where we co-schedule 2 applications at a time (i.e., 6 application pairs). 2-VM case achieves 19% energy efficiency improvements on average with respect to the baseline case, whereas 3-VM case improves the energy efficiency by 11%. Increasing the VM density for CPU-bounded multi-threaded applications limits the potential improvements, as the system is expected to be already fully utilized, which leaves less room for tradeoff management through resource adjustments. Our method works with an arbitrary number of VMs and server nodes; however, in an HPC environment each application should be allocated sufficient CPU resources to ensure high performance.

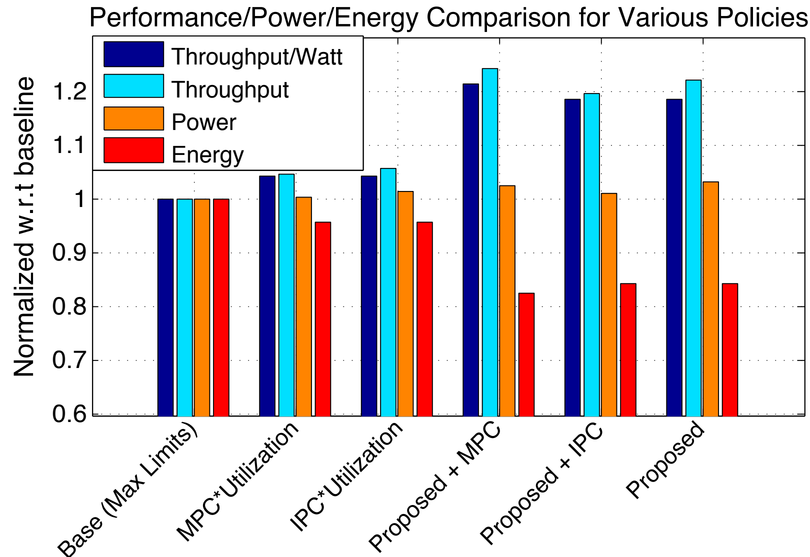

Figure 4·19: Average throughput-per-watt, throughput, power and energy comparison normalized w.r.t baseline case.

4.4.3 Consolidation with a Higher Number of VMs

We test our research allocation technique for co-scheduling 3, 4 and 6 applications (i.e., each application on a separate VM) on the same server. As a case study, we first create a set of 12 applications (2x `blackscholes`, 2x `dedup`, 2x `vips`, `bodytrack`, `canneal`, `facesim`, `swaptions`, `streamcluster`, `x264`). In each experiment, we co-schedule the applications in groups of 2, 3, 4 or 6 applications at a time. We evaluate our resource sharing policy with various numbers of VMs by allocating the CPU resources proportionally to the throughput of the applications. We compare the energy efficiency improvements with respect to the baseline case (i.e., without any limits on CPU resources). Figure 5·1 shows that the 2-VM case achieves 16% energy efficiency improvements on average with respect to the baseline case, whereas the 3-VM case improves the energy efficiency by 9%. The energy efficiency improvements decrease with increasing the number of applications co-scheduled at a time. As CPU-bounded multi-threaded applications already utilize the resources at high levels, increasing the VM density diminishes the energy efficiency improvements, while leaving less room for managing the performance/energy tradeoffs. Moreover, resource contention at lower levels of the memory and cache is expected to be higher, since larger number of VMs (i.e., vCPUs) will be sharing the hardware resources. In our experiments, we test our technique under fixed amount of CPU resources (i.e., single-node). Therefore, it is expected to have lower gains with increasing VM number, as the performance of the PARSEC applications already scale well up to 4 threads. Although our method works with an arbitrary number of VMs and server nodes, in an HPC environment each application should be allocated a sufficient amount of CPU resources to ensure high performance.

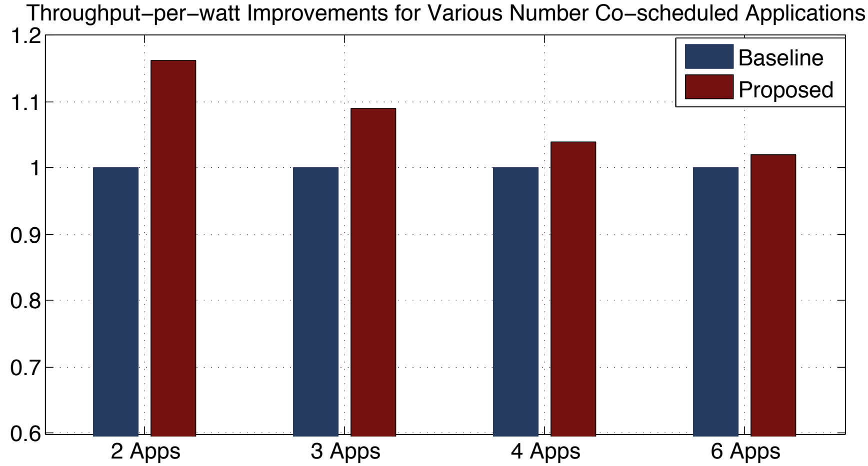

Figure 4·20: Normalized throughput-per-watt with respect to the baseline case, where each VM is given the maximum resources, for varying number of co-scheduled applications. Energy efficiency improvements decrease with increasing number of applications.

4.5 Chapter Summary

In this chapter, we focus on consolidation and resource management techniques for multi-threaded workloads running on multi-core servers. We first show that the best performing consolidation policy varies depending on the overall characteristics of the workload set and propose a selection algorithm that finds the best performing consolidation policy. Second, we propose resource allocation technique on virtualized environment. Our proposed technique classifies the multi-threaded applications depending on their energy efficiency levels, and favors the more energy-efficient ones while making resource allocation decisions. We show that our techniques improve the energy efficiency by 16% to 24% on real-life experiments.

Chapter 5

Dynamic Power Capping

Power capping is the ability to control the power consumption of the server nodes, in order to comply with power delivery limitations and control the energy costs. In this section, we present various dynamic power capping techniques targeting native and virtualized environments. We first introduce the details of our power capping technique on a native environment (i.e., **Pack & Cap**) for multi-threaded applications. **Pack & Cap** controls DVFS choices and number of active cores to optimize performance under power constraints. Secondly, we introduce a management framework for virtualized systems (i.e., **vCap**, where multiple multi-threaded applications are consolidated on a single multi-core server. **vCap** achieves fine-granularity power capping on virtualized systems, which optimizes the overall performance of the multi-core server by dynamically allocating resources for consolidated VMs. Third, we introduce **Scale & Cap**, which considers the power and performance scaling characteristics of multi-threaded applications when distributing the available resources through a linear programming-based solution to distribute the available resources. Finally, we present **Adapt & Cap**, a framework to coordinate and unify application and system-level adaptations to (1) improve the performance under power constraints and (2) reduce the power consumption under performance constraints. We implement and evaluate all of our techniques on real-life commercial servers with multi-core processors.

5.1 Power Capping on Native Environments

5.1.1 Pack & Cap Methodology

Our approach for runtime thread packing and DVFS control requires an offline step to train the logistic regression model. In the offline step, we use an extensive set of performance data collected for the parallel workloads in the PARSEC benchmark suite (Bienia et al., 2008) to train the classifiers. Each classifier takes performance counter and per-core temperature measurements as inputs, and outputs the system operating point with the highest probability of maximizing performance within a given power constraint. A classifier instance is trained for each desired power constraint. During runtime, we recall the model associated with the desired power constraint using a lookup table. Then, the control unit sets the system operating point with the highest probability of being optimal. In this way, our model is able to constrain the system power under a power cap at runtime without using expensive power measurement devices.

The offline characterization step makes use of an L_1-regularized multinomial logistic regression (MLR) classifier (Alpaydin, 2004). While previous techniques require manual inputs to select the performance metrics that are most relevant to energy, power and delay optimization, we use L_1-regularization to systematically find the relevant inputs and mask irrelevant ones. In the offline characterization step, we use an extensive set of power, temperature and performance counter data collected for each PARSEC benchmark at all feasible system operating points (V-F and thread packing combinations). For each workload's parallel phase (region of interest, ROI), we divide the data into 100 billion μ-op execution intervals. We then train an MLR classifier for each desired power constraint.

Previous modeling techniques (Lee and Brooks, 2006) rely on linear regression for estimating power and performance. Regression models predict continuous values

representing power and performance as a function of performance counter inputs. By modeling the boundaries between the discrete decision outputs directly, we observe the MLR classifier to be a more stable predictor of optimal outcomes compared to linear regression techniques, particularly when the test data differs considerably from the training data.

The inputs to the MLR classifier include a set of workload metrics, which are functions of the system performance-counter values (e.g., *μ-ops retired, load locks, cache misses, resource stalls*, etc.), per-core temperatures, and the current operating point. Given the inputs during runtime, the logistic regression calculates the probability of each candidate operating point being optimal under power caps. The output with the highest probability is then chosen as the current operating point. At runtime, the system logs performance counter and temperature data, and calculates the probability of each operating point being optimal using the set of weights derived from the MLR classifier corresponding to the current power constraint p_c. The runtime overhead of the proposed technique is minimal, as the model weights are accessed in the form of a lookup table.

5.1.2 Experimental Results

All experiments are performed on a server including an Intel Core i7 940 45nm quad-core processor, running the 2.6.10.8 Linux kernel OS. We control the system operating points (V-F settings and thread-packing combinations) using Linux C library interfaces. To implement data collection and runtime control, we interface our data measurement and control apparatus to a MATLAB module compiled as a C-shared library. This module is configured to read lookup tables generated offline, buffer incoming performance counter and temperature data, and periodically output control decisions to a control unit. The runtime overhead for each runtime activation of the control algorithm is in the range of 10-50ms.

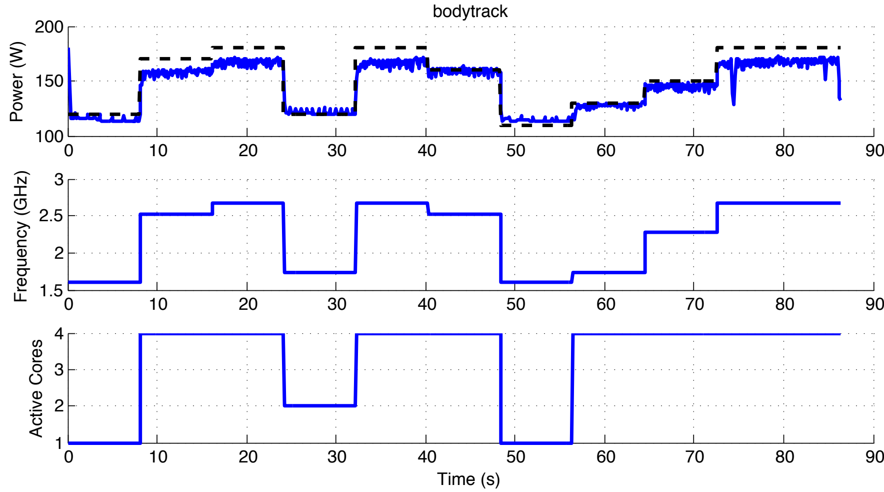

Figure 5·1: Demonstration of DVFS and thread-packing control for `bodytrack` under changing power caps (Cochran et al., 2011).

We demonstrate that our adaptive runtime policy, **Pack & Cap** consistently obeys a wide range of power constraints, regardless of workload behavior or physical operating conditions. During the execution of each parallel workload, we periodically change the power constraint to a random value in the 110W - 180W range, and measure the percentage of the execution time for which the power is within a tolerance of the cap value. We do not utilize any power measurements during runtime control. Overall, we are able to constrain the power consumption within the given cap 96% of the time within a 5W margin beyond the power cap. In Figure 5·1, we demonstrate the adaptive power capping capabilities of our approach. We also compare thread packing with using a fixed smaller number of threads, i.e., thread reduction. While 1-thread fixed or 2-thread fixed corresponds to running 1 or 2 threads, thread packing corresponds to executing applications with 4 threads *packed* on 1 or 2 cores on a quad-core machine. Thread packing is capable of matching the lower bound on the power cap associated with the 1-thread case, but achieves an average of 51.6% reduction in energy. When compared to the 2-thread case, thread packing is able to achieve a better power range, and an average of 15.6% reduction in energy.

5.2 Power Capping on Virtualized Environments

As virtualized cloud environments provide comparable performance to native execution, multi-threaded workloads from various applications domains (e.g., high performance computing and scale-out applications) are commonly deployed in virtualized data centers. VMs that run multi-threaded workloads (Symmetric Multi-Processing (SMP) VMs) exhibit significantly different power-performance tradeoffs compared to the VMs that run traditional enterprise loads with low system utilization. In addition, the energy efficiency of SMP VMs varies because of multi-threaded application characteristics (i.e., inter-thread communication and performance scaling). Therefore, optimizing the performance of the virtualized servers under power constraints requires sufficient understanding of the multi-threaded application characteristics.

The proposed technique, **vCap**, adheres to the power caps by dynamically adjusting the total amount of CPU resources that is utilized by the SMP VMs. At runtime, **vCap** monitors the performance characteristics of the VMs adjusts the resource allocation decisions to improve the energy efficiency of the virtualized server nodes. Our specific contributions are as follows:

- We propose a co-scheduling technique that finds the best VM groups to consolidate together based on the scalability of the multi-threaded applications to maximize the achievable performance. We show that scalability-based co-scheduling outperforms prior co-scheduling methods that solely consider application resource use.
- We propose a fine-grained power capping technique, *vCap*, that is able to meet the performance objectives such as maximizing total QoS or achieving a minimum QoS level for specific VMs. We also propose a fast and accurate runtime QoS estimation methodology for VMs that does not require any offline training phase. Based on the QoS estimations, *vCap* distributes the resources among VMs to optimize the energy efficiency of the server node.

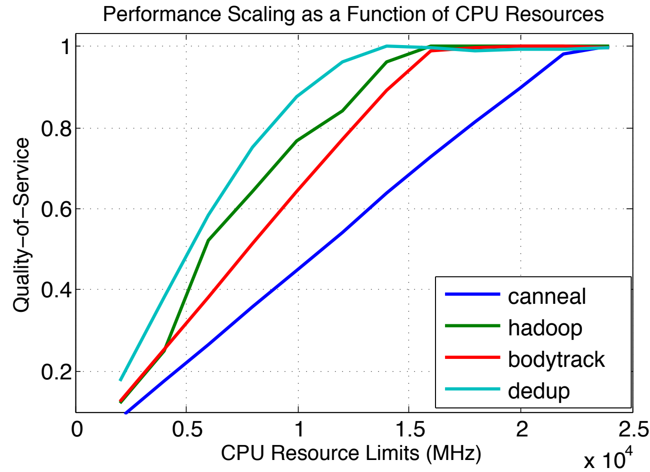

Figure 5·2: Performance scaling of some of the PARSEC and benchmarks and `hadoop` as a function of CPU resource limits.

- We demonstrate the benefits of vCap on a real-life multi-core server. vCap is able to meet dynamically changing power constraints with high accuracy while improving the overall QoS provided to the user by 17% and the QoS/watt by 12% for workload sets created out of PARSEC (Bienia et al., 2008) and CloudSuite (Ferdman et al., 2012) benchmarks.

Performance Scalability of Multi-threaded Applications

Ideally, multi-threaded applications are designed to efficiently utilize an arbitrary number of cores. However, application characteristics such as inter-thread communication and architectural bottlenecks such as the off-chip bus bandwidth cause sub-linear performance improvements when the amount of CPU resources are increased. SMP VMs that run *poorly scaling* applications are not able to utilize all the available CPU resources of a multi-core system; hence, such VMs are good candidates for consolidation. Although consolidation might degrade the performance of the individual VMs, the aggregate performance of the server node and the energy efficiency improve significantly.

On the ESXi hypervisor, the total available computational capacity of a server

node is represented in MHz, where the total amount of CPU resources, R, is equal to the number of physical CPUs multiplied by the maximum core frequency. CPU resource usage of a VM can be constrained by adjusting the *CPU resource limits* on the ESXi hypervisor. *CPU resource limits* put upper-bounds on the time, that VM gets scheduled. If the total CPU usage exceeds the limit, ESXi de-schedules the vCPUs. Figure 5·2 shows the QoS scaling of 4 applications from PARSEC and CloudSuite as a function of the CPU resources (in MHz). As Figure 5·2 shows, `bodytrack` cannot utilize all the available hardware resources, due to increased amount of data dependencies and communication overhead with increased number of threads. Therefore the QoS does not improve beyond a certain amount of CPU resources (i.e., 15970 MHz). In addition, reducing the CPU resources has a larger performance impact on the poorly scaling VMs (e.g., `canneal`, `bodytrack`) at lower CPU resource limits.

Total QoS of a consolidated server depends on the specific VMs that are co-scheduled. Figure 5·3 shows the QoS breakdown of two systems, each of which are running two distinct VMs under various power caps. We observe that the overall

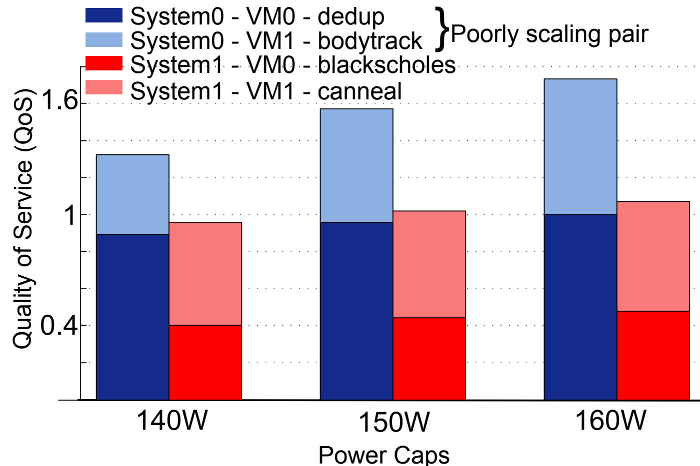

Figure 5·3: Overall normalized QoS of two distinct co-scheduling cases under various power caps. QoS range for the *scaling VMs* is much smaller than the *non-scaling VMs*. Thus, selecting non-scaling VMs to co-schedule have high potential for energy efficiency improvement.

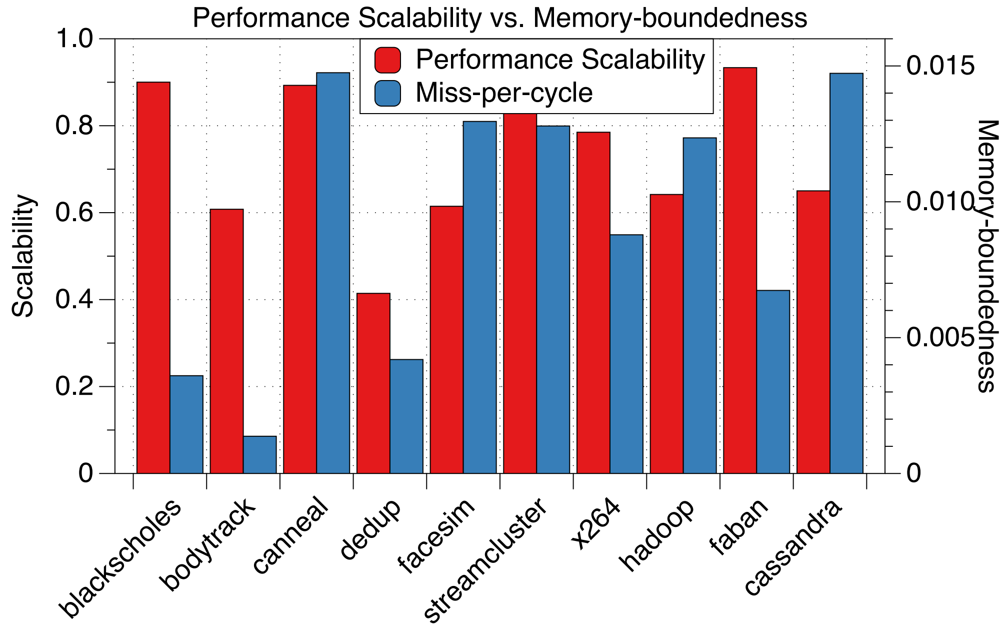

Figure 5·4: Memory-boundedness (last level cache misses per cycle) vs. scalability of PARSEC and CloudSuite applications. Scalability is measured as the ability to utilize the 12-core system when running with 12 threads. This experiment shows that memory-boundedness does not capture the scalability characteristics of the applications.

QoS improvement of consolidation is much larger for the VMs running poorly scaling applications (i.e., blue bars), as these VMs are not able to fully utilize the system when running alone. For the VMs running `dedup` and `bodytrack`, consolidation improves the overall QoS by 73% and QoS/watt by 68%. Consolidating the SMP VMs that have near-linear performance improvements as the amount of resources grow (i.e., VMs running `blackscholes` and `canneal`) provides 7% higher total QoS and only 4% higher QoS/watt compared to running them alone. This observation motivates the design of a co-scheduling policy that takes the scalability characteristics of the applications into account while making co-scheduling and resource allocation decisions.

Consolidating VMs that have complementary resource usage characteristics is ex-

pected to reduce contention. For example, memory-boundedness can be used as a metric to choose which VMs to co-schedule (Dhiman et al., 2009). Depending on the requirements of the applications, it is known that consolidation causes performance degradation due to increased contention on bus and caches (Dhiman et al., 2009). However, on virtual environments it is possible to minimize the impact of the co-runner application through isolating and balancing the memory accesses. In order demonstrate that, we evaluate the impact of the co-runner VMs on performance of the most memory-intensive application (i.e., `canneal`. Figure 5·5 shows the QoS of the VM running `canneal` alone and consolidating with all other VMs in pairs of two VMs at a time. Consolidating two instances of `canneal` has the highest negative impact on the performance. However, in all other cases co-runner applications does not introduce significant performance degradation. Our experimental results discussed above imply that for multi-threaded applications, performance scalability has a more dominant impact on the energy efficiency of the server. Figure 5·4 shows the scalability and the memory-boundedness metrics (i.e., cache miss-per-cycle) for a selection of PARSEC and CloudSuite. We use cache miss-per-cycle rather than cache miss-per-instructions, as miss-per-cycle reflects the cache-miss rate over time. As Figure 5·4 shows, memory-boundedness and scalability have significantly different trends (i.e., 0.34 Pearson coefficient with 0.26 confidence level) across benchmarks. Therefore, co-scheduling decisions based on memory-boundedness would differ from decisions that are based on the application scalability. Based on our analysis, we make the following observations:

- When consolidating multiple VMs, it is beneficial to allocate only the necessary resources to poorly scaling VMs that reach nearly their maximal QoS with a relatively smaller amount of resources (e.g., dedup), and reserve the remaining resources for the co-runner VMs.

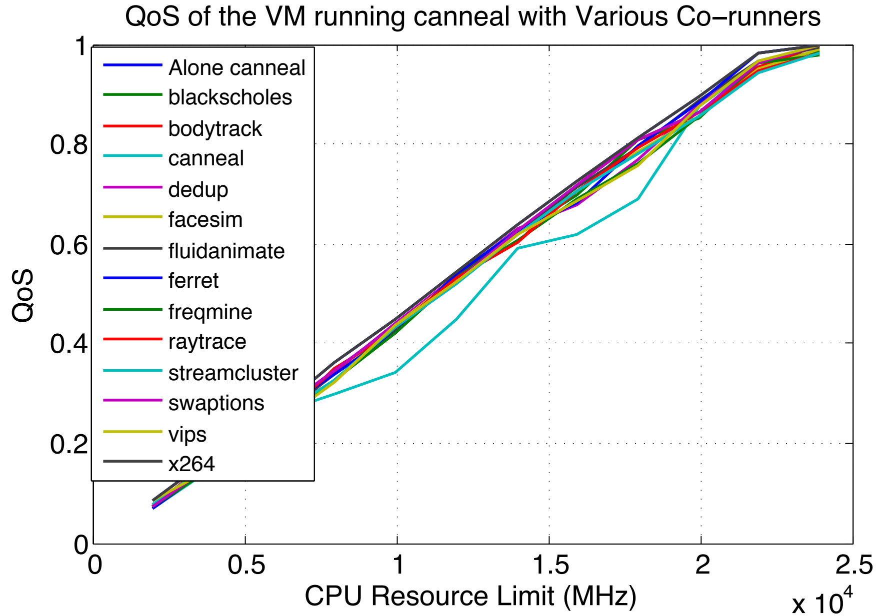

Figure 5·5: QoS of the VM running `canneal` when consolidating with all the other VMs in pairs of two. Performance of `canneal` is not significantly affected by any of the co-runners.

- As the performance of the VMs is minimally affected by the co-runner VMs, consolidation decisions based on resource usage (e.g., memory-boundedness) are not sufficient to improve the energy efficiency. Scalability of a VM is a more important factor while making consolidation decision when compared to resource usage metrics. The energy efficiency of a server node, QoS/watt, can be maximized by co-scheduling poorly scaling VMs together (e.g., `bodytrack-dedup`).

5.2.1 Adaptive Power Capping on Virtualized Environments

Virtual environments provide additional control knobs to manage the amount of resources allocated for each VMs. ESXi hypervisor allows administrator to limit the maximum amount of CPU resources allocated to a specific VM by adjusting the *CPU resource limit* knob, which restricts the resource usage of the VMs. By adjusting the CPU resource limits, it is possible to cap the peak power consumption of the server node. In this section, we discuss the design of *vCap* and provide an overview of our

implementation. We first discuss our methodology to track the power caps using the CPU resource limits and our VM-level QoS estimation technique under power caps. We then present our consolidation and the resource distribution algorithms that maximize the energy efficiency under power constraints.

Tracking Power Caps

By utilizing the CPU resource limits, it is possible to adjust resources with a MHz granularity, instead of adjusting the number of active cores (i.e., core power gating (CPG)) and/or DVFS. Therefore, CPU resource limit knob enables us to control the performance/power tradeoffs with a finer granularity, which provides up to 14% higher QoS in comparison to power capping with DVFS+CPG granularity.

Thread Packing Overhead

In this work, we propose to execute applications with the maximum number of threads (i.e., number of threads = number of cores) and then applying CPU resource limits, as dynamically changing the thread number requires modification to the original code. However, running higher number of threads may introduce performance overheads due to increased contention and communication among threads. In order to quantify the negative impact of running a higher number of threads when running at CPU resources that is equal to 4 cores, we compare the normalized runtime of the applications in three cases. In the baseline case, we execute the applications with 4 threads, no that there is no overhead due to higher number of threads. We then test two techniques: (1) *Resource Limits* represents the case of running the application with 12 threads and limiting CPU resources that will be equal to compute power of 4 cores, (2) *GuestPacking* represents the case where VMs run with 12 threads, which are packed onto 4 vCPUs at the guest OS level. Figure 5·6 shows the normalized runtime for running the applications with the configurations explained above. Guest-

Packing reduces the number of vCPUs that simultaneously access the pCPUs. As a result, GuestPacking allows us to reduce the overhead by 45% for `dedup`. Therefore, we implement GuestPacking together with CPU resource limit adjustments to minimize the negative impact of running higher number of threads under power caps.

In order to meet the power caps, our proposed technique utilizes runtime power monitoring as the feedback. We separate the total power consumption P_{tot} as the sum of the dynamic (P_{dyn}) and idle (P_{idle}) power. At runtime, **vCap** estimates the total amount of available CPU resources (R_{cap}), that meets the given power cap. For n number of VMs consolidated, $R_{cap}(MHz) = Utilization * P_{cap}/P_{dyn}$. Based on the calculated R_{cap}, we first derive the number of active vCPUs, such that

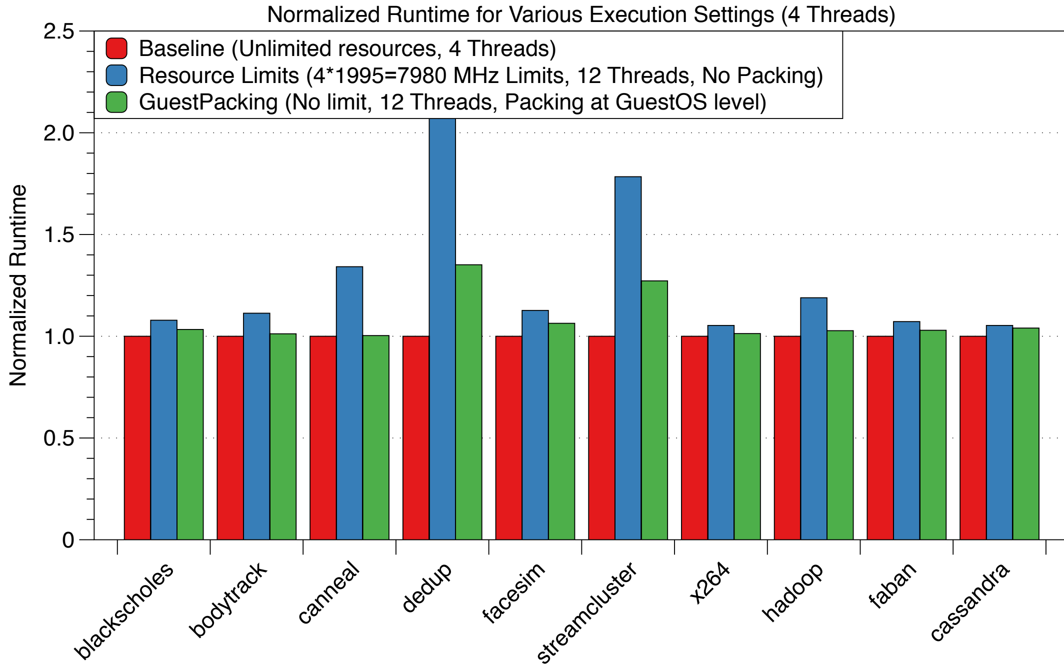

Figure 5·6: Performance overhead for running higher number of threads under power caps. Running applications with 12 threads and applying resource limits introduces large overheads for some of the applications. Packing the threads onto smaller number of vCPUs reduces the overhead up to 45%.

$\#ofvCPUs * CoreFreq > R_{cap}$. We then fine tune the CPU resource limits to match the R_{cap} value.

5.2.2 Estimating QoS Degradation Under Power Caps

ESXi hypervisor provides VM-level metrics to pinpoint the CPU resource usage and bottlenecks. The sum of these 4 performance metrics is equal to the total amount of available CPU resources of a system with, which is $100\% * \#ofvCPUs =$ ($\texttt{RUN\%} + \texttt{READY\%} + \texttt{COSTOP\%} + \texttt{WAIT\%}$). In this work, we focus on `READY` and `RUN` to identify the scalability and the QoS of the application at various CPU resource limits.
`RUN`: The percentage of total schedule time of the VM, which excludes the system time ($\%\texttt{UTIL} = \%\texttt{RUN} + \%\texttt{SYS}$).
`READY`: The percentage of time that the VM is ready to run, but not scheduled. This metric implies that the application will be able to utilize the CPU, if more resources were allocated to the VM. Therefore, `READY` metric can be utilized to estimate maximum utilization level, which reflects the performance scalability characteristics of the applications.

`COSTOP`: The percentage of time the VM spent in a rescheduled state, where ESX CPU scheduler deliberately puts the vCPU into a sleep mode, if a certain vCPU advances much farther than other vCPUs. `COSTOP` is expected to high for multi-threaded workloads that have load-balancing issues that originate at the input distribution/preparation stage (i.e., input/start phase) of the application.
`WAIT`: The total wait time where the VM is waiting for some VMKernel resource. `WAIT` includes I/O wait time, idle time and among other resources of stalls.

By using the above metrics, it is possible to estimate the total CPU requirements of a VM by using the `RUN` and `READY` metrics. Although `RUN` metric is similar to the CPU utilization metric, the additional `READY` metric captures the potential performance improvements of the VM, if more CPU resources were allocated. For instance, when

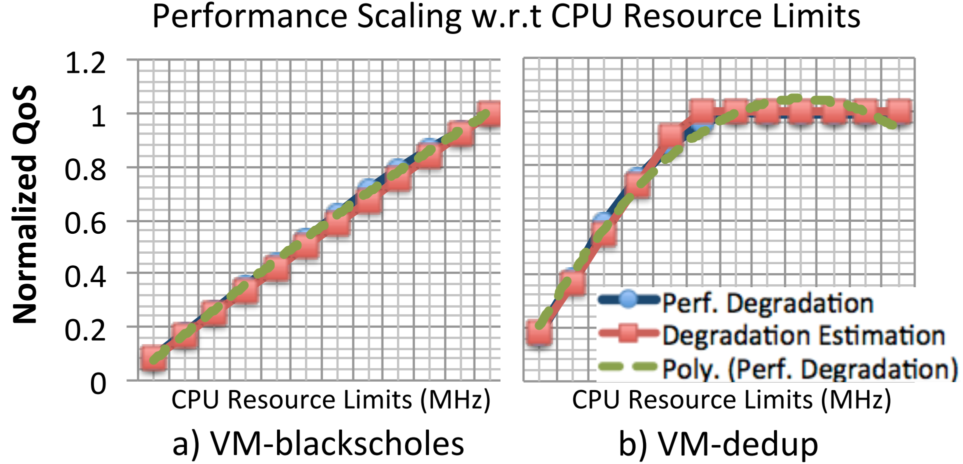

Figure 5·7: QoS as a function of CPU resource limits for `blackscholes` and `dedup`. Degradation estimations are derived by using Equation 5.1. Equation 5.1 provides better QoS prediction without requiring any offline/training phase when compared to polynomial models.

running a single application with no restrictions, `READY` metric is 0%. In case there are threads that cannot be scheduled due to CPU usage restrictions, `READY` metric will reflect the amount of CPU resources that is needed to achieve the maximum the performance. Based on our observations on these metric, we conclude that the total CPU requirement of an application (C_{req}) can be estimated by $C_{req}(MHz) = (\texttt{RUN\%} + \texttt{READY\%}) * R$, where R is the total amount of resources in MHz. For a given CPU resource limit (C_{lim}), performance degradation of an application with respect to the highest CPU resource limit (i.e., QoS=1) can be calculated as,

$$QoS(C_{lim}) = \begin{cases} 1 - \frac{(C_{req} - C_{lim})}{C_{req}} & C_{req} > C_{lim} \\ 1 & C_{req} < C_{lim} \end{cases} \tag{5.1}$$

It is also possible to train a workload specific model for QoS estimation, however this requires additional training and offline data collection time for each new workload. Considering the vast number of diverse application sets that are running on the cloud, offline training becomes a bigger overhead. Figure 5·7 shows actual and predicted QoS for `blackscholes` and `dedup`, as a function of CPU resource limits. *Degradation*

estimation reflects the QoS estimations using Equation 5.1, while we use second order polynomial fit as the baseline estimation technique that requires an offline training phase for each application. We evaluate our proposed QoS estimation methodology that is explained in Equation 5.1 for all PARSEC applications when running with $n-$cores, where $0 \leq n \leq 12$, on a 12-core system. Our proposed technique can estimate the QoS of all PARSEC applications with an average error of 3%, while polynomial fit provides 5% error on average. These results show that the proposed QoS estimation technique provides better accuracy even without any training phase, which eliminates the workload characterization overhead.

Providing QoS Guarantees: As decreasing the amount CPU resources allocated to a VM causes degradation on the application performance, it is essential to provide lower-bounds for the QoS of the individual VMs. In a real-life scenario, users might request minimum QoS guarantees (QoS_{req}) for time-sensitive applications. In order to provide QoS guarantees for VMs, we can use the Equation 5.1 to estimate the C_{lim}, such that $1 - \frac{(C_{req} - C_{lim})}{C_{req}} = QoS_{req}$.

5.2.3 Consolidation Based On Performance Scalability

Consolidating applications that do not scale linearly improves the energy efficiency by improving the overall utilization of the system, while consolidating scaling applications do not bring significant improvements as the system is already utilized by the application. Therefore, we run VMs that have $C_{req} > 11 * CoreFreq$ alone, as those VMs already utilize the hardware resources. In order to choose the VMs to consolidate together, we rank the VMs according to their C_{req} values and pair the VMs starting from the bottom of the list, with the constraint $\sum_{i=1}^{n} C_{req_i} < R_{cap}$. In our experiments, the maximum value of n (i.e., number of VMs) that do not exceed the R_{cap} is 2 for our 12 core system. This observation shows that consolidating more than two VMs is not feasible for the system under test.

In order further improve the energy efficiency, we propose to distribute the CPU resources to VMs by prioritizing the ones that reaches to its maximum QoS with the smallest amount of CPU resources (i.e., poor scaling VMs). Therefore, the aggregate QoS of the system can be maximized by allocating the rest of the CPU resources to the co-runner VM. We set the C_{lim}=C_{req} for the VMs that have lower C_{req} than its co-runner and allocate the rest of the CPU resources (i.e., $R_{cap} - C_{lim}$) to the co-runner VM which has a higher C_{req}.

5.2.4 Experimental Results

We implement the *vCap* on an AMD Magny Cours multi-core server. We deploy a management node to perform administrative tasks on VMs. Management node collects runtime performance statistics from the ESXi hypervisor and power readings from the power meter. QoS guarantees and the power constraints are implemented user-defined input parameters of the runtime routine. We keep the track of power estimation errors and the runtime routine continue to adjust the CPU resource limits by recalculating the R_{cap}, till the tracking error is smaller than 2W. QoS and R_{cap} estimation modules are implemented as Phyton modules on the management node. In order to adjust the CPU resource limits, we created Perl modules using the vSphere SDK for Perl. Perl module communicates with the ESXi and reconfigures the VMs CPU resource limits based on the input from the estimation modules in every 2 seconds, which is equal to the minimum sampling rate of the performance monitoring tool (i.e., `esxtop`). To implement the *GuestPacking* technique, we use the default `taskset` tool at the guest OS level to pack the threads onto smaller number of vCPUs. As vCap modules are implemented on a separate management node, the overhead on the server side is negligible.

In order to evaluate the success of our technique, we randomly generate 10 workload sets, each of which consists of 10 applications selected among PARSEC and

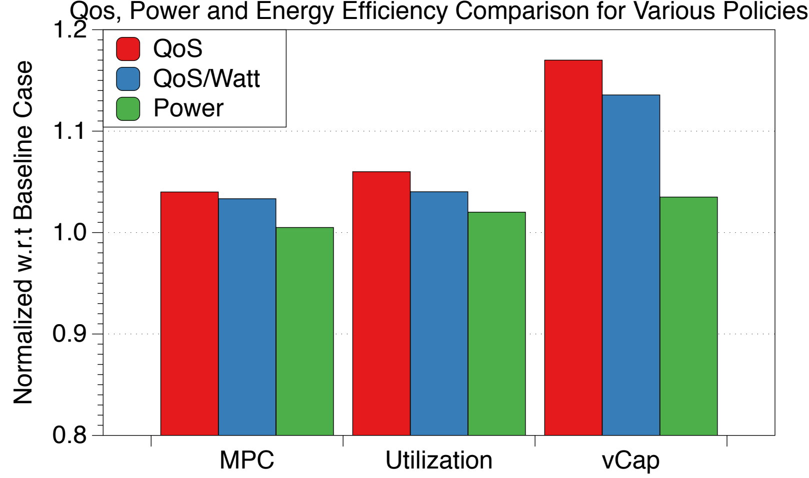

Figure 5·8: Comparison of QoS, QoS/watt and power consumption of the server with various consolidation techniques. The proposed technique improves the overall QoS by 17% when compared to the baseline case, where VMs have no CPU resource limitations.

CloudSuite applications. In Figure 5·8, we evaluate both the overall QoS of the system, as well as the QoS/watt metric to measure the energy efficiency. We compare our technique with previously proposed consolidation techniques that are based on CPU utilization and MPC metrics. For CPU utilization and MPC based policies, we first rank the applications according to the selected metric. We then pair the highest ranked VM with the lowest ranked one and progress through the list and allocate equal resources to all VMs. We normalize each value with respect to the baseline case, where we pair the VMs randomly and do not impose any CPU resource limits. The proposed technique improves the QoS by 11% on average in comparison to the best performing previous policy. The energy efficiency of the server node also improves by 9% in comparison to the most energy efficient previous policy.

We also test our technique under dynamically changing power caps to evaluate the power cap tracking accuracy. We change the power caps in every 8 seconds between 100W and 150W, similar to the real-life power regulation signals that are sent in every 4/8 seconds. We compare the overall QoS for the proposed technique and the

baseline case. *vCap* is able to adhere to the power cap 92% of the time within the 2W error margin, and 98% of the time within the 5W error margin.

Figure 5·9 shows the runtime behavior of *vCap* for the VM pair that is running `dedup` and `hadoop`. We test our technique under the minimum QoS requirement of 70% for VM0 (i.e., `dedup`) and dynamically change the power caps between 130W and 160W. *vCap* accurately tracks the power cap accurately and satisfies the minimum QoS requirement that is required for VM0 by adaptively adjusting the resources in case of a QoS violation (e.g., t=10).

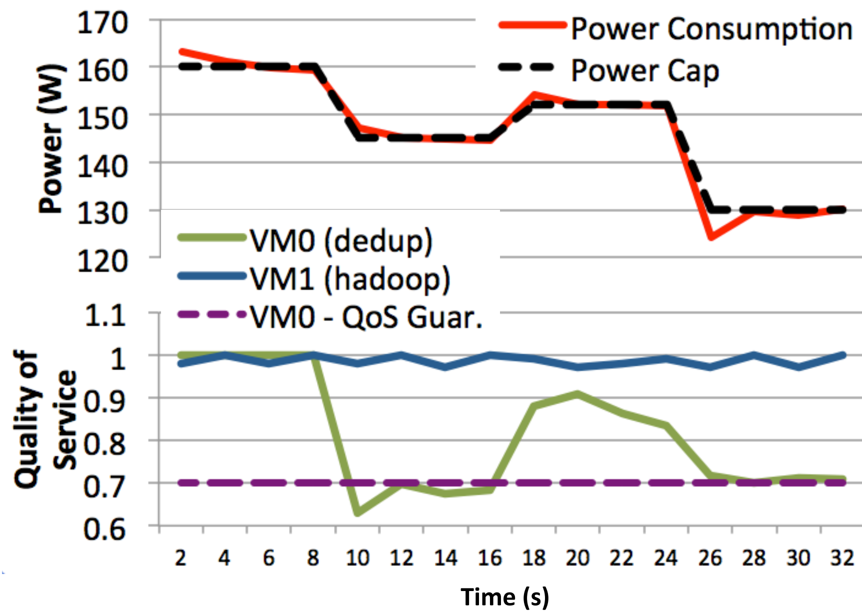

Figure 5·9: Runtime behavior of *vCap* under power caps and QoS constraints for the VM group running `dedup` and `hadoop`. *vCap* adheres to the power cap and ensures that the QoS guarantees are met.

5.3 Scale & Cap: Scaling-Aware Resource Management for Consolidated Multi-threaded Applications

The total amount of available compute resources on a server varies over time due to the power cap given to the server, thermal emergencies, user demands and application types. Traditionally, CPU utilization has been used as the metric to determine the resource distribution ratio across single-threaded applications (Nathuji et al., 2009). By using the CPU utilization metric to distribute the resources proportionally, these techniques aim to minimize the performance degradation, while maximizing the server utilization. However, with the emergence of multi-threaded applications, using a single dimensional metric, such as CPU utilization, become inefficient to capture the multi-threaded specific characteristics. In this section, we first present two motivational examples to show the need for a novel approach to the resource allocation problem for multi-threaded applications. We then introduce our dynamic resource allocation technique that jointly uses the power and performance characteristics to capture the multi-threaded characteristics.

Is CPU Utilization Enough?

CPU utilization metric measures the percentage of busy cycles of a specific period of time. Traditional resource allocation techniques distribute the available compute resources proportional to the CPU utilization levels of the consolidated applications (Xen, 2009) (VMware DRS, 2009). Although for single-threaded applications, CPU utilization can capture all CPU requirements of an application; for multi-threaded applications we need to incorporate an additional dimension, which is the performance scalability. Performance scalability can be defined as the characteristics of a multi-threaded application that reflects how the performance of the application is increasing (i.e., scaling) with increased amount of resources. Ideally, all multi-threaded applications are designed to scale perfectly with increasing amount of resources (i.e.,

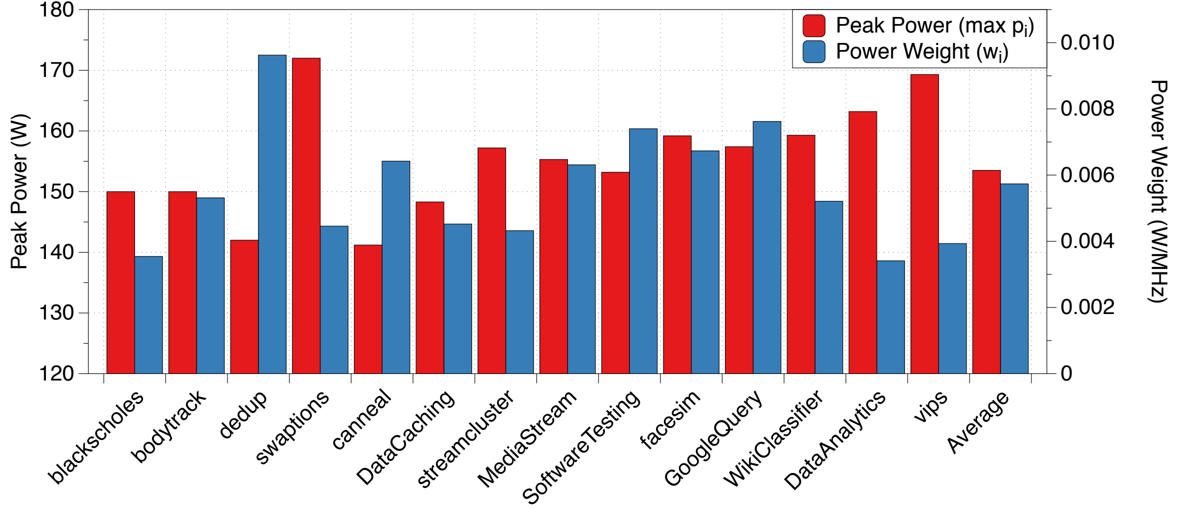

Figure 5·10: Peak power (left axis, red) and power weight (right axis, blue) values for 4 PARSEC benchmarks and the PARSEC average measured on AMD Opteron 6172.

$2x$ performance improvement for $2x$ increase in allocated resources). However, due to various multi-threaded application bottlenecks, such as communication, synchronization, serial code, etc. most of the multi-threaded applications do not scale linearly with increasing amount of resources. Therefore, each multi-threaded application has a specific performance/resource curve that reflects its performance scalability.

ALGORITHM 2: Utilization-aware Resource Distribution

Inputs:

$U[n]$ // *Utilization Array*

R_k // *Total amount of available resources*

Output: r_i

$W = sum(U[n]);$

for i = 1 to n ;

$w_i = U[i]/W$

$r_i = R_k * w_i$

end

ALGORITHM 3: Scalability-aware Resource Distribution

Inputs:

$C[n]$ // *CPU Demand Array*

R_k // *Total amount of available resources*

Output: r_i

sort: C[n] **for** i = 1 to n ;

if $R_k > 0$ **then**

  $r_i = C[i]$;

  $R_k = R_k - r_i$;

else

  return

end

end

Similar to the performance scalability characteristics, each application has a distinct peak power consumption and a power weight, w_i, which represents the power consumed at one unit of computing capacity (i.e., MHz in the virtual environment), while running application i. In Figure 5·10, we show the peak power consumption and the power weights of 14 applications. In order to obtain the peak power values, we run the applications alone with maximum amount of thread count that is equal to the total core count for each server (i.e., 12 threads for the AMD-based and 8 threads for the Intel-based server) with the default resource manager. As figure shows, both peak power and power weight values show significant variation, due to the software characteristics. For instance, vips heavily utilizes the floating point units, which is a more power hungry component than integer units. Therefore, the peak power consumption of vips is much higher when compared to blackscholes, which mostly utilizes the integer units. Similar to our argument for performance scalability, resource distribution without considering the power weights of the individual applications can

lead to inefficient resource distribution by not being able to favor the more power-efficient applications. In order to better illustrate our observations, we present two case studies that demonstrate the benefits of considering performance scalability and power weights while making resource distribution decisions.

Case Study #1: Applications with Distinct Performance and Power Scaling

In this example, we compare the benefits of various resource distribution approaches on two applications that do not exhibit only distinct resource requirements (i.e., performance scaling), but also distinct power characteristics. Such an application-pair from PARSEC suite is `canneal` and `facesim`. The performance of `canneal` can scale almost up to 12-threads, while `facesim`'s performance increase saturates beyond 8-threads. On the contrary, `facesim` is a more power hungry application when compared to `canneal` due to its higher IPC (Hankendi and Coskun, 2012). These contradicting properties play a significant role while making resource distribution decisions. In order to illustrate this, we compare three algorithms: (1) utilization-aware (Nathuji and Schwan, 2008), (2) only performance scalability-aware (Hankendi et al., 2013) and (3) power and performance scalability-aware in Figure 5·11a. Utilization-aware approach proportionally distributes the total available amount of resources based on the CPU utilization of each VM by calculating a weight value (w_i) as shown in Algorithm 2. Scaling-aware resource distribution first estimates the CPU requirements of each VM, and then prioritizes the ones with smaller requirements to maximize the overall QoS of the server, as shown in Algorithm 3. On the other hand, power and performance scaling-aware approach maximizes the QoS for given power weights, CPU demands, power constraints and total amount of available resources using a linear programming-based solver, which is explained in detail in Section 4. As power-aware approach uses power weights as an additional metric, it

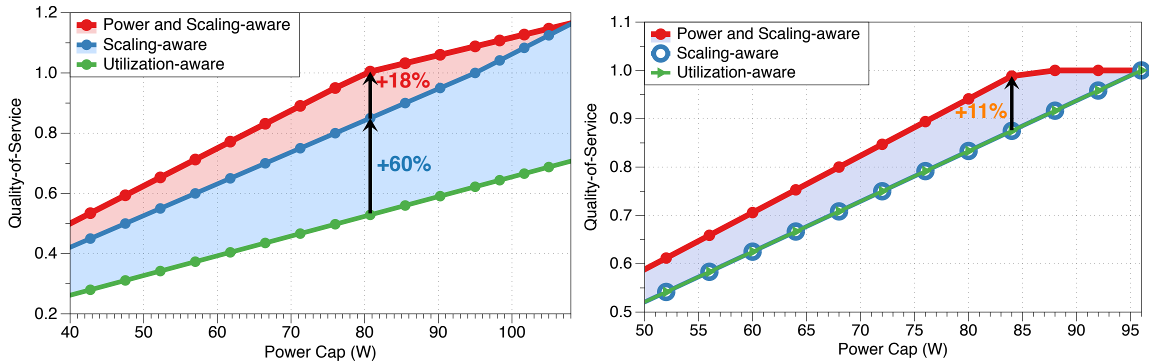

Figure 5·11: Total QoS comparison for consolidating two application pairs, `canneal-facesim` (a) and `blackscholes-swaptions` (b) on AMD Opteron 6172 with various power caps for utilization-based approach (baseline), naive approach (only scaling-aware) and power-aware (scaling and power-aware) approach. Power-awareness brings up to 18% and 11% QoS improvements over the scaling-only approach.

further improves the overall QoS by guiding its decision based on both resource and power constraints.

Case Study #2: Applications with Similar Performance but Distinct Power Scaling

In order to show the benefits of adding power considerations to the resource distribution technique, we evaluate two applications from PARSEC suite that exhibit similar performance scalability characteristics, but distinct power requirements, such as `blackscholes` and `swaptions`. Although both of these applications scale up to 12-threads, `swaptions` consumes significantly higher power than `blackscholes` (e.g., 151W vs. 172W). We use the same baselines as the previous case study.

Figure 5·11 shows the total QoS of consolidating `blackscholes` and `swaptions` under various power caps for three approaches: (1) power and scaling-aware, (2) only scaling-aware, (3) utilization-based (baseline). As Figure 5·11 (right) shows, Utilization-aware and Scaling-aware techniques perform exactly the same, as the performance scalability of these two applications cannot be distinguished. Therefore,

only scaling-aware resource distribution will be equally favoring these two applications, as they have similar performance scaling capacity). On the other hand, power and scaling-aware approach will favor the one with lower power weight value. Power and scaling-aware approach improves the total QoS up to 11% in comparison to only scaling-aware and utilization-based approaches. As `blackscholes` and `swaptions` have very similar performance scaling behavior, scaling-awareness does not bring any benefits over the utilization-based approach. This also demonstrates that the 11% performance improvement is solely due to power-awareness.

5.3.1 LP Solution to Resource Distribution

Based on our observations in the previous section, we conclude that the power efficiency and performance scaling characteristics of parallel applications play a significant role. In order to incorporate our findings in a formal solution, we formulate the problem as a linear programming problem. The main goal of the solution provided here is to maximize the total QoS of consolidated applications without violating the individual QoS requirements of the applications under power constraints through resource allocation. We first formulate the problem of maximizing QoS for n number of VMs, then explain how to incorporate power efficiencies of individual applications into our linear programming solution.

We define the maximum QoS as the performance (i.e., runtime) of an application when running alone on the target system. This measurement gives us the upper-bound for the performance of each application, which we use as the maximum QoS of 1. Any performance loss is reflected as the percentage of the maximum QoS performance. The problem of maximizing QoS of a server when consolidating m applications with QoS values on a single physical server can be represented as follows.

$$\begin{aligned} &\text{maximize} \quad \sum_{i=1}^{m} q_i \\ &\text{subject to} \quad 0 \leq q_i \leq 1,\ i = 1, \ldots, m. \end{aligned} \tag{5.2}$$

For an application i, the achievable maximum QoS (q_i) is a function of the resource demand, (d_i), and the total amount of resources allocated,(s_i). In order to achieve the maximum QoS, the amount of supplied resources (s_i) should not be lower than the resource demand of the application (d_i). Therefore, we can define the QoS of an application, i, as a function of resource demand (d_i) and the allocated/supplied resources (s_i). For the case, where $d_i \leq s_i$, QoS is 1, as the demand is met by the supply. Therefore, our focus is to solve the resource distribution problem, where resources are limited ($d_i \geq s_i$). For such cases, QoS of the application i, q_i is described as follows:

$$\begin{aligned} &q_i = s_i/d_i \\ &\text{subject to} \quad 0 \leq s_i \leq d_i \leq R_k. \end{aligned} \tag{5.3}$$

R_k is the total resource capacity of a server, k. For consolidating $i = 1, \ldots, m$ number of applications on a server, k, Equation (5.2) becomes

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{m} s_i/d_i \\ &\text{subject to} \quad 0 \leq s_i \leq d_i \leq R_k. \end{aligned} \tag{5.4}$$

In order to be able to guarantee certain performance requirements, we need to be able to enforce lower bounds for the QoS of each application. We can use *lower-bound* for minimum performance guarantees and *upper-bound* for maximum performance limitations, as data centers may provide incentives to users for bounding the maximum

performance. Therefore, our problem becomes a constrained QoS maximization problem, which can be represented by putting lower and upper bound constraints on q_i (i.e., s_i/d_i) of the applications, such that

$$\begin{aligned} \text{maximize} \sum_{i=1}^{m} s_i/d_i \\ \text{subject to} \quad 0 \le s_i \le d_i \le R_k \\ l_i \le q_i \le u_i. \end{aligned} \tag{5.5}$$

Based on the Equations above, we can convert the problem into a linear-programming problem as follows:

Find **q** that maximizes

$$\begin{aligned} f(q) = q_1 + q_2 + \dots q_m \\ \text{subject to} \quad \sum_{i=1}^{m} d_i q_i \le R_{max} \\ l_i \le q_i \le u_i. \end{aligned} \tag{5.6}$$

Solution of this LP-problem provides us the **q** vector, which consists of set of possible q_n values, $q_1, q_2, \dots q_m$ to assign for each application, i, to maximize the $f(q)$ (i.e., total QoS) under a total resource capacity constraint, R_{max}. For an unlimited amount of resources, R_{max}, the maximum value of $f(q)$ would be m, as $q_1, q_2, \dots, q_m = 1$. For an application, i, with a total resource demand d_i, the total amount of resources required to provide a QoS of q_i is, $s_i = d_i q_i$. From the LP solution, we can derive the necessary amount of resources, s_i that maximizes the total QoS of the system.

5.3.2 Maximizing Server-QoS with Power Constraints

For the case where there are power constraints, the amount of resources need to be reduced to R_k, where $R_k \le R_{max}$. In order to determine the value of R_k, our run-

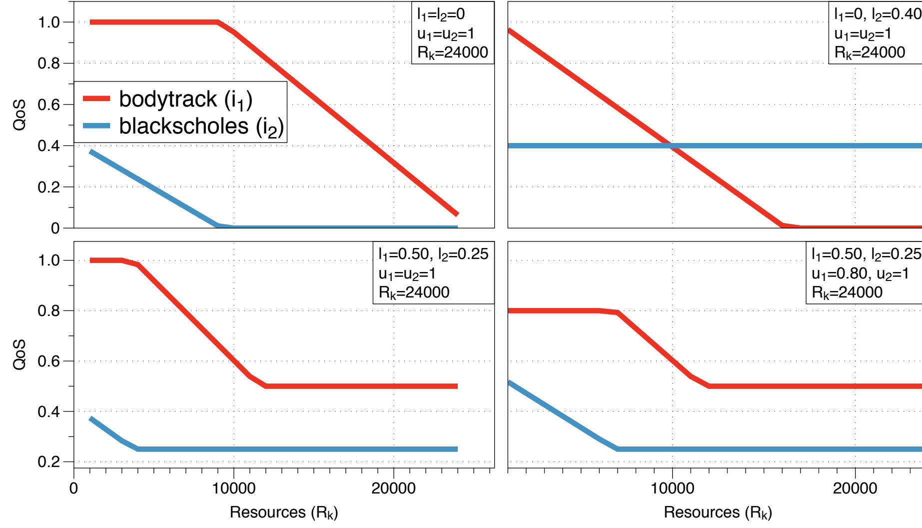

Figure 5·12: LP-solution for resource distribution across two applications ($m = 2$) with various lower and upper bounds for a given amount of resources R_k.

time system utilizes the power feedback, P_t, and the system utilization measurement window, $R_(t)$, which is the running average of the last 4 resource demand estimates, $R_{[t_{n-3},\ldots,t_n]}$. For a given time, t, and power constraint at time t, $P_{cap}(t)$, it is possible to derive the $R_k(t)$, by using the linear correlation between $R(t)$ and $P(t)$. Based on our experimental results and reported results from prior work, we assume that power constraints on the system are linear and can be derived at runtime through power measurement feedback. As a straightforward approach that meets power constraints through modifying R_k, we simply compute $R_k(t)$ based on the $P(t)$ for a given power constraint, $P_{cap}(t)$ by

$$R_k(t) = P_{c,k}(t)/P_k(t-1) * R_k(t-1) \tag{5.7}$$

By using the most recent power weight $W_k(t-1)$, it is possible to find an $R_k(t)$ that consumes $P_k(t) \leq P_{c,k}(t)$. After deriving the $R_k(t)$ value to be enforced on the server, QoS maximization problem can be solved by solving the problem represented

in (5.6). However, this approach simply uses a lumped value, W_k, for modeling the power/performance relation of a set of applications, where each individual application has a distinct power weight value, w_i. Therefore, any change on r_i has a different impact on p_i, and therefore P_k, where, $P_k = \sum_{i=1}^{m} p_i$. Ignoring the power weight differences across applications lead to inefficient resource distribution as illustrated in Section 3.3. In order to derive the power weights of the applications, we use VM-level power estimations on the Intel-based server. As VM-level power estimations are not available for the AMD-based server, we use offline data for the AMD-based server. For the Intel-based server, power weights (w_i) values can be derived at runtime by $P_{VM_i}(t)/R_{VM_i}$. We incorporate the power weight (w_i) information into the LP solution as shown in Equation (5.8).

$$
\begin{aligned}
& f(q) = q_1 + q_2 + \dots q_m \\
\text{subject to} \quad & \sum_{i=1}^{m} s_i \leq R_k \\
& \sum_{i=1}^{m} s_i * w_i \leq P_{cap} \\
& l_i \leq q_i \leq u_i.
\end{aligned}
\tag{5.8}
$$

As the power weight value (w_i) represents the power consumed-per-MHz computation, we estimate the power consumption by summing up the $s_i * w_i$ multiplication in Equation (5.8). By enforcing an additional constraint in the form of power constraints, we target to use the available resources and power as efficiently as possible. We list all variables and their definitions in Table 5.1.

Table 5.1: Definitions of the abbreviations used in the LP solution.

Abbreviation	Definition
P_k	Power consumption of a server, k.
p_i	Power consumption estimation of an application, i.
R_k	Total amount of resources allocated to a server.
r_i	Total amount of resources allocated to an application, i.
W_k	Power weight of a consolidated application set, i.
w_i	Power weight of an individual application, i.
u_i	Performance upper bound for application, i.
l_i	Performance lower bound for application, i.
k	Server index
i	Workload index

5.3.3 Runtime Implementation of Scale & Cap

We implement the LP solution in MATLAB and compile it as an executable file. In order to simplify the implementation, we convert the QoS maximization problem ($maxf(n)$) to a minimization problem ($minf(-n)$). We then use *linprog* MATLAB routine to solve the minimization problem to find the q. The inputs to the MATLAB routine are the user constraints for upper and lower-bounds, respectively u_i and l_i, power constraints (P_{cap}), power measurements from the power meter (P_k) and VM-level metrics from the hypervisor to derive w_i and r_i values.

We compiled the MATLAB module as a C library to call at runtime on a seperate management node that communicates with the host through the VMware vCLI Perl API to change the allocated resources for each VM. The main control loop of the implementation runs every 2 seconds, which is the same as the esxtop sampling rate. Changing resource limits at this granularity imposes a negligible performance overhead on the applications. The LP solution routine takes between 0.2 to 0.4 seconds to return the updated r_i values. The API-based communication with the hypervisor completes its function within the range of 0.1 to 0.3 seconds. Therefore, within 2 seconds window, the runtime implementation can finalize its decision and action.

For faster control, it is also possible to implement the LP solution with alternative libraries and/or languages. However, note that for large scale systems, monitoring period is reported to be over 20 seconds (VMware DRS, 2009).

5.3.4 Evaluating Resource Allocation Techniques

In this section, we quantify the benefits of power and performance scaling-aware resource distribution across multiple VMs. We compare our technique with previously proposed resource distribution policies and present the performance improvements under various power caps. In addition to comparison among resource distribution techniques, we also compare the benefits from placement and resource distribution techniques to gain insights about the interaction as well as to provide quantitative comparisons between two resource management schemes.

In order to quantify the benefits of our resource distribution technique, we choose three approaches that are already implemented or proposed in prior work, namely the default ESXi manager, demand proportional distribution (Isci et al., 2010) and performance scaling proportional resource distribution (Hankendi et al., 2013). We briefly explain each approach as follows:

Baseline: Our baseline case for all experiments is the policy where we allocate equal number of cores to number of threads requested by the user. We set hard limits across VMs by using CPU resource limits, which prevents dynamic adjustment of underutilized resources.

Default Manager (ESXi): The default manager allocates CPU resources based on the requested resource limits or reservations. Resource limits are hard constraints that can not be modified by the manager. On the other hand, resource reservations are soft constraints, such that the resources that are not utilized can be lended to other VMs by the ESXi manager. Therefore, we reserve n number of vCPUs for n number of threads requested for each VM. In this scenario, the default manager can

borrow any unused CPU resources to other VMs, but never limits the VM the usage.

Demand proportional: Demand metric for a workload has been defined as the maximum amount of utilization of the system for a given number of threads. Demand proportional policy distributes the available resources across VMs proportional to their demand estimations.

Scaling priority: Similar to the demand proportional approach, scaling priority approach uses the demand metric for all VMs to make resouce distribution decisions. However, this technique favors the higher demand workloads (i.e., better scaling ones) over the low demand ones, rather than proportionally distributing the available resources.

Proposed: The proposed technique incorporates both the scalability estimations through demand metric and the power efficiency through MHz/W metric. The proposed approach utilizes these two measurements to solve a maximization problem using linear programming, as explained in Section 5.3.1.

In order to compare and evaluate the aforementioned resource distribution techniques, we create workload sets out of the benchmarks explained in Section 3. Each workload set consists of applications with various thread counts. We generate 100 workload sets such that the each workload set consists of total of 12 threads or 8 threads for AMD and Intel-based servers respectively. Therefore, we aim to create a data center scenario, in which the servers are not overutilized. To fairly compare the resource distribution policies, we use the same default placement scenario for all policies. The default placement policy is a *first-fit bin packing* algorithm run on the list of applications with thread numbers requested.

Placement Algorithms

We focus on three different placement techniques that are previously proposed to improve performance of consolidated environments, namely memory-based, similarity-

based and demand-based placement algorithms. The main idea behind all three placement algorithms is reducing the contention that might occur when consolidating

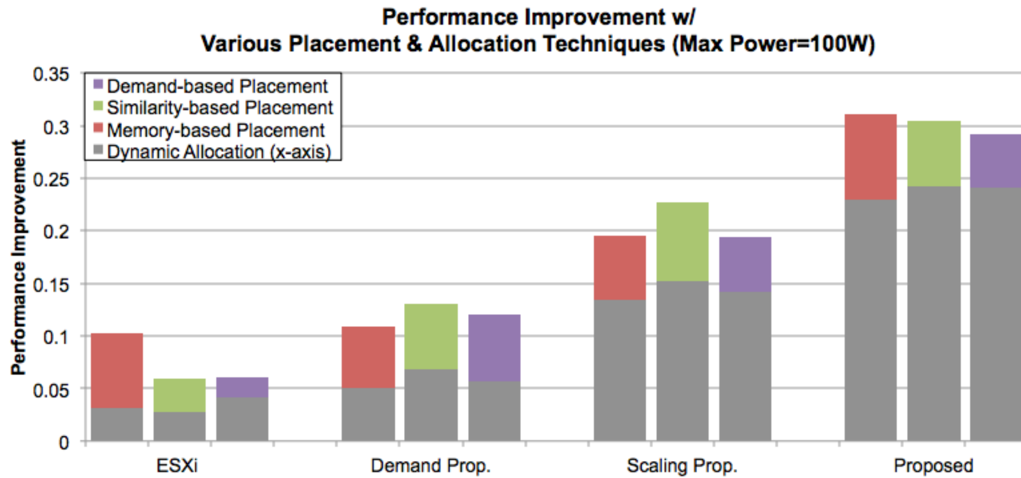

Figure 5·13: Performance variation for all applications for various VM density cases. Higher VM-density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity.

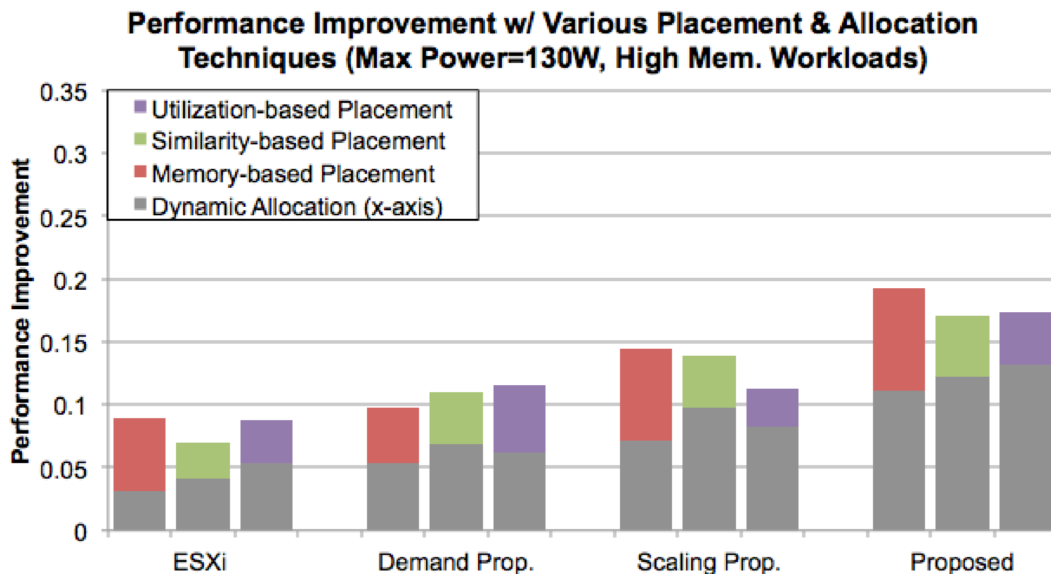

Figure 5·14: Performance variation for all applications for various VM density cases. Higher VM-density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity.

multiple workloads, thus improving the performance. Each of these algorithms use a different metric as a proxy to evaluate the potential contention. The goal is to create a balanced resource consumption across all VMs to improve the overall performance.

For evaluating the placement algorithms, we first collect the necessary measurements for each benchmark when they are executed alone and use offline data while making placement decisions. In Algorithm 4, we show the pseudo-code for the memory-based placement algorithm. Algorithm 4 first sorts all the benchmarks in the workload queue based on the last-level cache miss rates, which can be used as an indicator of the memory accesses. As a next step, Algorithm 4 starts grouping the benchmarks starting from top then the bottom of the list and progresses through the list, until the total number of threads or the total utilization of the benchmark group do not exceed predetermined threshold values.

ALGORITHM 4: Balanced Memory Placement

Input: W_{ij} // *Workload Matrix*

Output: VM mapping

initialization;

k = 1;

sort($W_{ij}.memaccess$) // *Sort based on memory accesses*

for each vm in sorted.W_{ij} ;

if $S_i.util < U_{max}$ ***and*** $S_i.thread < T_{max}$ **then**

  add sorted.$W_{ij}(k)$ to S_i ;

  k = i - j - 1; // *Reverse list index to continue from bottom*

else

  k = k +1; // *Continue from the list*

end

Similarly, demand-based algorithm applies the same idea using the `demand` metric, which is a virtualized environment specific metric. ESXi hypervisor provides VM-level

metrics to pinpoint the CPU resource usage and bottlenecks. Resource demand of a VM can be estimated by adding these two metrics from esxtop.

`RUN`: The percentage of total schedule time of the VM, which excludes the system time (%`UTIL` = %`RUN` + %`SYS`).

`READY`: The percentage of time that the VM is ready to run, but not scheduled. This metric implies that the application will utilize the CPU, if more resources were allocated to the VM. Therefore, `READY` metric can be utilized to estimate maximum utilization level, which reflects the performance scalability of the applications.

By balancing the demand across VM group, it is possible to reduce performance degradation due to CPU contention. These two placement techniques, memory and demand-based, focus on either memory or the CPU as the main source of contention. In order to be able to capture characteristics in other dimensions, similarity-based approaches are recently proposed. Similarity-based techniques evaluate a set of performance counters and derive a euclidean distance based on the similarities of benchmark in each dimension. The final result of this evaluation a single score of similarity, which is then used as the main metric for sorting and grouping the applications.

5.3.5 The Impact of VM Density on Placement Techniques

In order to evaluate the impact of VM density of the benefits due to placement techniques, we create three workload set scenarios with various average utilization values. Low Load represents a case where the majority of the workloads are utilizing the system around 20%. We call this case as the Low Load, since the average utilization of the server will be low when there is no consolidation. Consolidating a Low Load workload set lead to higher number of VMs to be consolidated. This is expected to have implications when choosing a placement algorithm.

The benefit of choosing a good placement algorithm is due to reducing the potential resource interference across multiple applications (or VMs). In Figure 5·15,

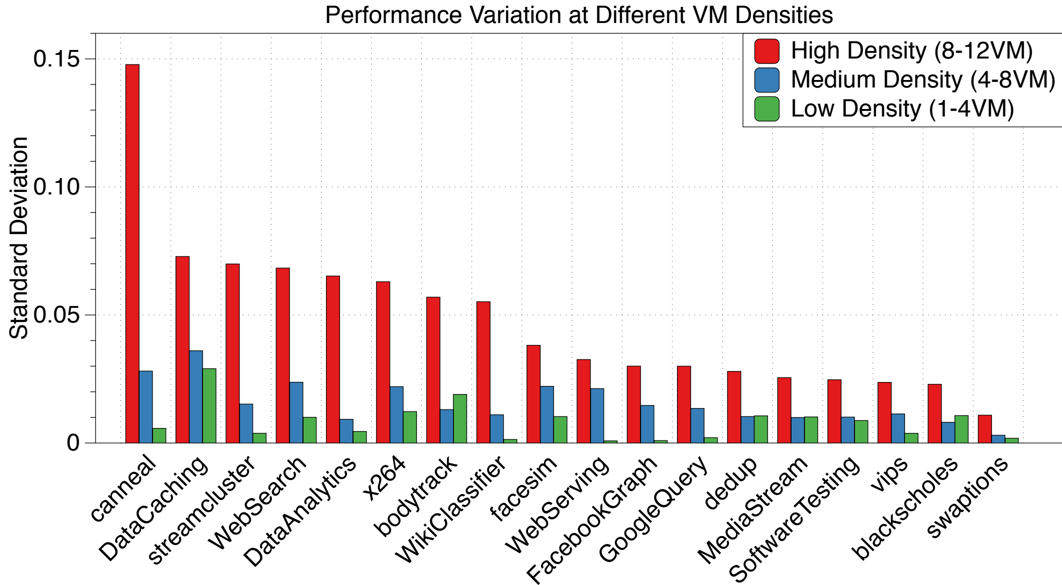

Figure 5·15: Performance variation for all applications for various VM density cases. Higher VM-density leads to higher variation, and the memory-bounded applications have the highest variation due to higher cache sensitivity.

we show the performance degradation due to consolidation for various placement algorithms at three different VM density scenarios. High VM Density represents the case that has the highest number of VMs consolidated at a particular time period. By using the Low Load workload set, we can create consolidation cases where higher number of VMs are consolidated at the same time. Similarly, by using the High Load set, we end up with consolidation cases with low VM densities. As the placement algorithms become more critical when there is more contention, High VM Density case is expected to have the most benefits from placement algorithms.

In Figure 5·16, we show the performance degradation at different VM densities for 4 different placement algorithms. Depending on the placement algorithm, the degradation ranges from 24% to 1%, where the High VM Density case causes the highest degradation. For Medium and Low VM densities, there is minimal differences across different placement algorithms. However, at High VM densities, choosing

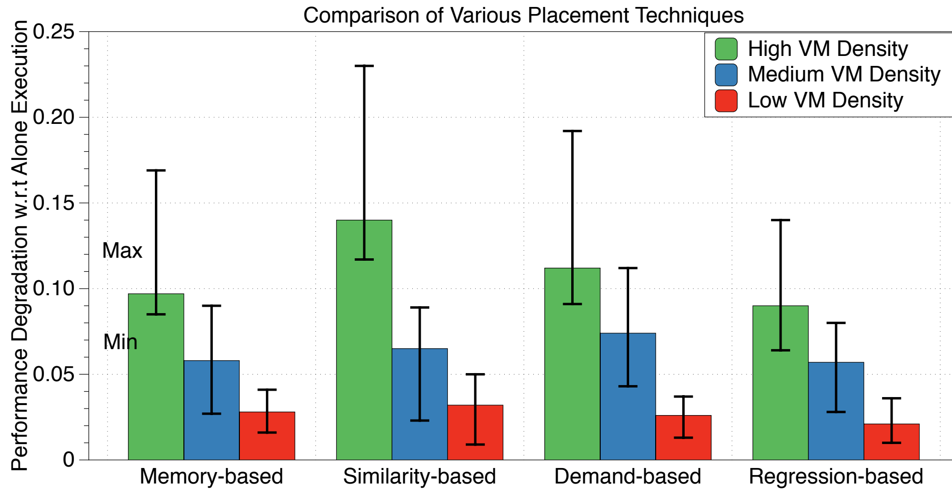

Figure 5·16: Comparison of placement techniques in terms of performance degradation (i.e., lower is better).

a memory-based placement algorithm reduces the degradation by up to 11%. For higher VM densities, the memory overhead for VM creation causes additional memory contention. Therefore, memory-sensitive approach brings up to 7% with respect to other placement techniques.

In order to look at the impact of increased VM density and the higher memory stress, we compared 3 placement techniques for the same 100 consolidation sets and report the best performing placement techniques for each of these distinct workload sets. In Figure 5·17, we color code the best performing placement technique for varying VM densities and active memory sizes. As Figure shows, memory-based placement needs to be favored for high memory and high VM density consolidation scenarios, due to the aforementioned reasons. On the other hand, similarity-based placement technique starts to perform better for lower VM density and lower memory sizes, as the CPU resources become more critical for CPU-heavy workload sets and similarity-based favors the CPU resource demand metric when making placement decisions, as also explained in Section 5.3.

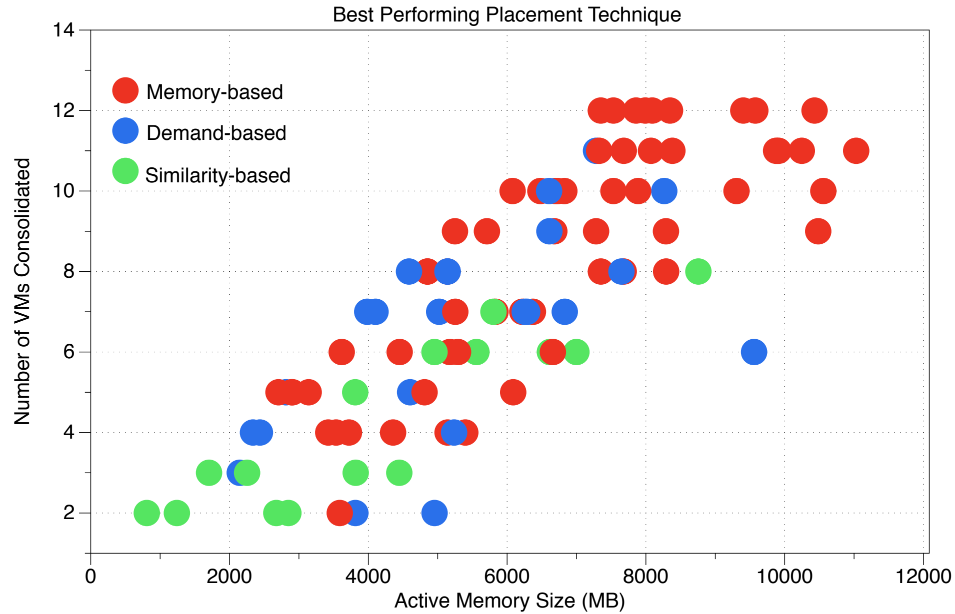

Figure 5·17: Best performing placement technique for various VM-density and active memory size. Memory-based technique is superior to other techniques with increasing number of VMs consolidated at the same time, which also leads to higher active memory size.

To summarize, in this work, we present a resource allocation technique that incorporates the multi-threaded specific performance scalability and power efficiency characteristics to distribute the available resources across multiple VMs running heterogeneous applications. We formulate our solution as linear programming-based algorithm and implement our technique on two multi-core servers. We evaluate various resource allocation and placement techniques together and provide in-sights regarding the interaction between placement and resource allocation techniques. Our results show that for tight power budgets, resource allocation brings up to 22% performance improvements in comparison to only using a placement algorithm.

5.4 Coordinating System and Application-level Adaptations for Power Constrained Systems

The aforementioned interplay between power and performance requirements and constraints adds to the complexity of data center management. In order to reduce the administration and management costs, designing adaptive solutions has become necessary. Traditional adaptive solutions employ system-level management knobs to comply with the power and performance requirements. These system-level adaptive solutions use control knobs such as DVFS or turning on/off cores. However, system-level solutions lack the ability to optimize the performance of the application. Adaptive applications address the performance optimization problem by dynamically configuring application parameters depending on the hardware properties and the performance goals (Hoffmann, 2014). As application and system-level decisions impact both the performance and the power consumption, uncoordinated decisions at these two levels can significantly hurt the overall energy efficiency of the system.

In this work, we propose a unified framework that takes advantage of both system and application-level adaptability to (1) improve performance under power caps, and (2) reduce power consumption under performance constraints. Adapt & Cap priorities improving the energy efficiency through maximizing the performance first, then using the system-level adaptations to reduce the power while meeting the user requirements. Our specific contributions in this work are as follows:

- We first demonstrate how to improve the power/performance trade-off space by unifying system and application-level adaptation.
- We propose a unified framework, Adapt & Cap, which combines system and application-level adaptations to improve performance while reducing the power consumption.

- We implement Adapt & Cap on real servers and demonstrate up to 27 % power reduction and 2.7x performance improvement compared to system or application-level only adaptation.

5.4.1 Benefits of Coordinating System and Application-level Adaptation

Controlling the tradeoff between an the accuracy and performance of an application is a widely studied area (Hoffmann et al., 2011). Therefore, design of applications that can expose various control knobs to provide control over accuracy and performance targets has become an emerging area of study. Adaptive applications enable dynamic reconfiguration of execution parameters to meet user-defined performance and accuracy constraints.

On a cloud environment, where resources are limited, power, performance and accuracy constraints are expected to be dynamically changing due to changing user requirements, energy pricing and cost management policies. Adaptive applications can meet these dynamically changing performance or accuracy targets by modifying a set of selected application parameters at runtime. Application parameters vary depending on the type of the application. For instance, for an image processing application, these parameters can be block sizes, motion search ranges, or color matrices. An adaptive application iteratively modifies its parameters (i.e., application control knobs) until the user-defined constraints are met.

Although adjusting application parameters can be utilized to meet the performance and accuracy constraints, the impact of modifying the application parameters on the power consumption is limited. In order to meet the power constraints, system-level management techniques are necessary. Various system-level power management techniques have been proposed that utilize control knobs such as DVFS or adjusting the number of active cores. However, these power management techniques are

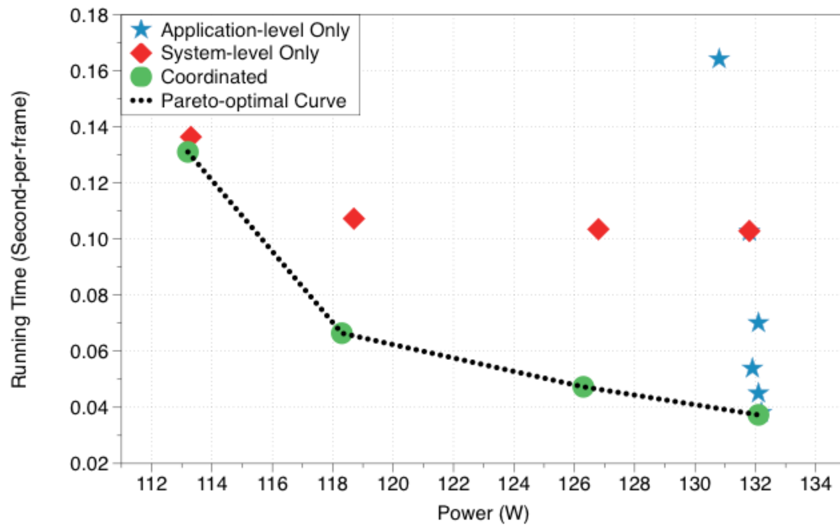

Figure 5·18: Power and performance tradeoff space for various adaptive techniques on Intel Xeon E5 multicore server when running x264. Proposed coordinated management extends the Pareto-optimal curve to a more efficient operating point.

agnostic about the potential adaptive capabilities of the applications. Independently managing system and application adaptation leads to uncoordinated management of power and performance requirements. This interaction may lead to destructive interference, where the application and system behavior oscillate and fail to meet both the performance constraint and the power cap with the desired accuracy, as we show in Section 5.4.3.

In Figure 5·18, we show the power and performance (i.e., seconds elapsed processing each frame) for x264 from the PARSEC suite for three cases: (1) where we use adaptive capabilities only at the application-level (Application-level Only), (2) only at the system-level (System-level Only), and (3) at both application and system level (Coordinated). As Figure 5·18 shows, application-level decisions have minimal impact on the power, while providing the ability to adjust the performance for a wide range of targets. On the other hand, system-level decisions have a significant impact on power consumption, while providing a narrower performance range with respect

to the adaptive application. Unifying the system and application level adaptability provides the most efficient trade-off curve for the power and performance space. This result in Figure 5·18 motivates the idea of using adaptive capabilities of applications to push the performance while reducing the power consumption through system-level control.

5.4.2 Adapt & Cap: Unifying System and Application-level Adaptation

In this section, we present the details of the proposed adaptive framework, Adapt & Cap. Adapt & Cap combines an application-level adaptive framework (i.e., Heartbeats) with a system-level adaptive power management framework to (1) maximize the performance under power constraints and (2) minimize the power consumption under performance constraints. We first discuss the details of the application-level framework for adapting to changing performance and accuracy targets. We then introduce the system-level adaptation framework that adjusts the level of resource usage to closely follow the power constraints.

In order to create adaptive applications, we use the previously proposed PowerDial framework (Hoffmann et al., 2011). PowerDial first identifies the application parameters, then uses these parameters as runtime control knobs to adjust the tradeoffs between performance and accuracy. In order to find the control variables, Powerdial employs influence tracing for the configuration parameters. PowerDial executes the application with varying configuration parameters, and records their influence on the application performance. In order to determine the control variables for the parameters, PowerDial generates a state table that stores various configuration parameters to create the adaptive version of a statically configured application. PowerDial generates the state table at compile time by profiling the applications on representative inputs provided by the user. For each combination of parameter settings, PowerDial profiles all representative inputs and records the speedup and accuracy loss. It then

computes the average speedup and accuracy loss across all inputs, and sorts these average values into the set of Pareto optimal states. By convention, PowerDial stores configurations so that the slowest configuration is the first entry in the state table and the fastest configuration is the last.

In this work, we use two types of adaptive applications that are created with PowerDial and with a loop-perforation technique (Sidiroglou-Douskos et al., 2011) to show the applicability of our technique to virtually any software that has adjustable parameters. Other application domains that rely on iterative algorithms, such as graph applications are also good candidates to create adaptive versions.

Adapt & Cap maximizes performance through utilizing adaptive applications and minimizes the power consumption by employing system-level adaptations (i.e., management). Adapt & Cap is built on top of the vCap framework and extends the capabilities of vCap by taking advantage of the performance optimization capabilities of the adaptive-applications. In Figure 5·19, we illustrate the overall flow of the Adapt & Cap framework. Our framework accepts two types of constraints either from the user for performance (i.e., heartbeat rate) or the system administrator for power (i.e., power cap). Both power consumption and the heartbeat rates are periodically fed to the closed-loop controller to adjust and tune its decisions.

Figure 5·20 provides the pseudo-code for the Adapt & Cap framework that consists of three major steps, which are: (1) configuring the adaptive application (Configure), (2) controlling the power consumption (PowerControl) and (3) meeting performance constraints (HBControl). Each adaptive application comes with built-in state tables, which include various combinations of the application parameters. As a first step, Adapt & Cap discovers the adaptive states of the application within the code and chooses the state that achieves the highest performance. It then measures the performance and power consumption at the highest state (i.e., n). Based on these measure-

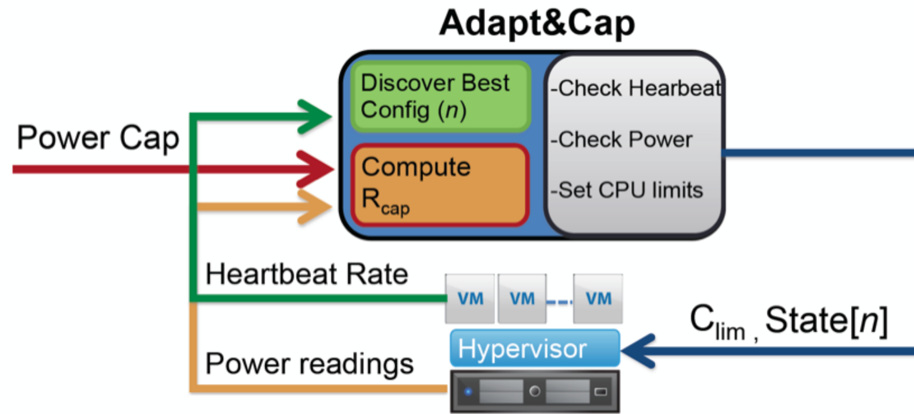

Figure 5·19: Adapt & Cap reads heartbeat rates and power measurements, and chooses the amount of CPU resources required and the optimum application state.

ments at the highest performance state, we derive the dynamic power consumed-per heartbeat (P_{HB}), and the CPU resource used-per heartbeat (CPU_{HB}) assuming that the power and performance are linearly correlated. A later fine-tuning stage enables compensating the potential inaccuracies caused by the linearity assumption. Power and performance characteristics can be accurately estimated at runtime due to the linear relationship, we rely on the feedback mechanism, rather than designing a complex controller.

After deriving the power/performance relationship of an application at its best performing configuration, Adapt & Cap checks the power and performance constraints to adjust the CPU usage limits (CPU_{limit}) and to make thread packing decisions. For a given power cap, Adapt & Cap first computes the maximum achievable performance (HB_{cap}), then it computes the maximum amount of CPU resources that will not violate the power constraints (CPU_{limit}). Based on the CPU_{limit}, we derive the minimum number of active cores that can provide enough CPU resources to meet the computed CPU_{limit}.

In order to compensate for the potential inaccuracies in the power and performance

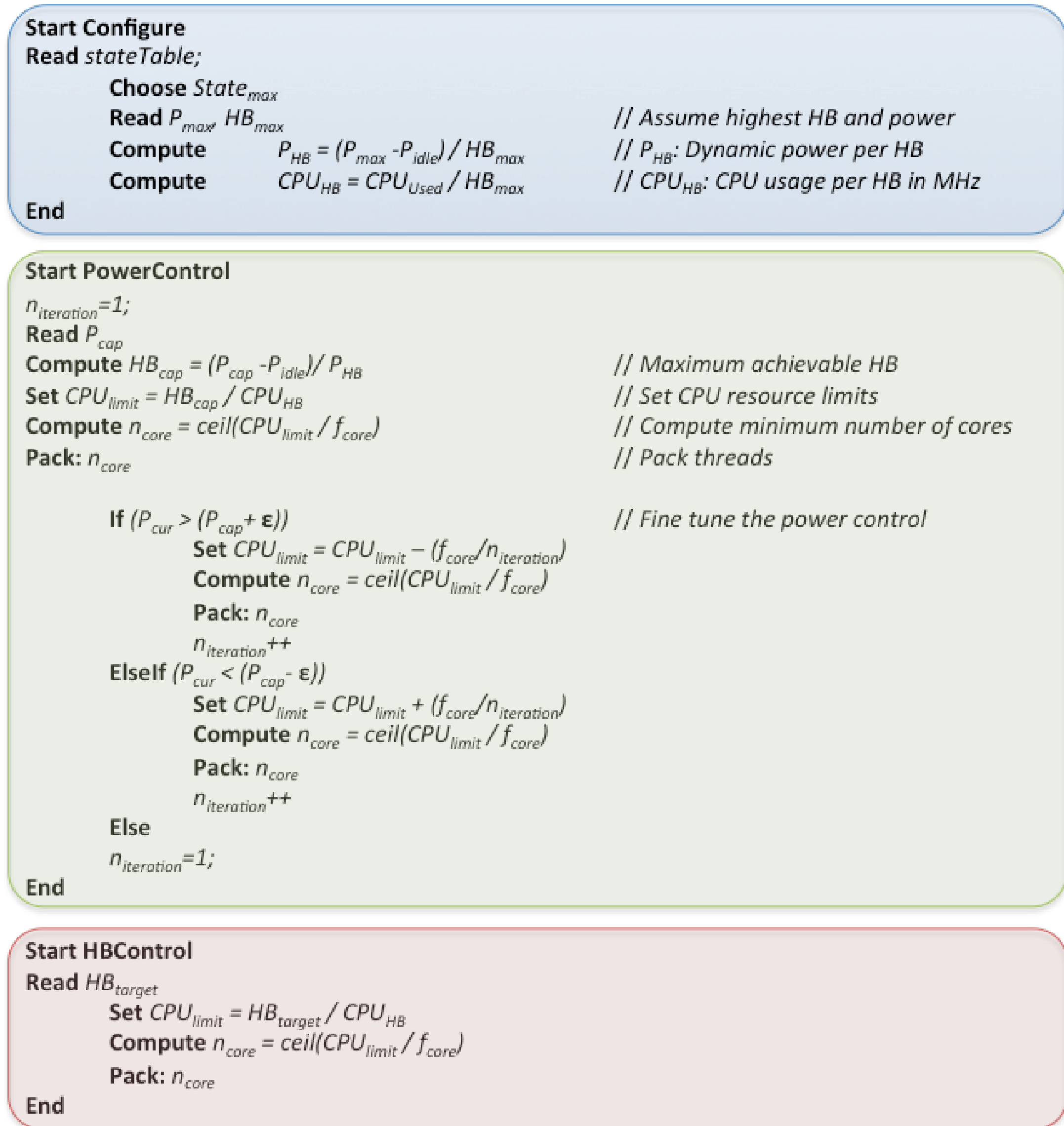

Figure 5·20: Pseudo-code for Adapt & Cap control modules. Adapt &Cap first discovers the higher performance application state (blue box), then periodically checks power (green box) and performance (red box) requirements to adjusts its decisions.

estimation, Adapt & Cap performs fine-tuning on its decisions. In case of a tracking error that is larger than ϵ, Adapt & Cap iteratively adjusts the CPU_{limit}. We start with a granularity of 1-core (i.e., resource limits corresponding to 1-core) to increase or decrease the CPU resources allocated. Until the tracking error is within the range, we increase the granularity of fine-tuning by dividing the adjustment range with the number of iterations. We use $\epsilon = 2W$ in our experiments. As a result, in each iteration, we achieve a finer control on the power consumption. Similarly, for the performance control, Adapt & Cap gets the performance requirement as an input (HB_{target}), and computes the necessary amount of CPU resources that will meet the performance constraints. The overhead of monitoring and management is around %1.

The power and performance control mechanism of Adapt & Cap is implemented to prioritize the hard constraints (i.e., power) over soft constraints (i.e., performance). Therefore, decisions to improve the performance are overwritten in case of a power violation to obey the power constraints. As oppose to disjoint management schemes, Adapt & Cap is an opportunistic approach that does not solely meet the requirements in one dimension, but also targets to improve the efficiency in both power and performance dimensions.

In this section, we present the benefits of the Adapt & Cap framework on real-life servers. We test our framework under two scenarios that are (1) dynamically changing performance constraints and (2) dynamically changing power caps. First, we show that Adapt & Cap provides lower power under dynamically changing performance constraints. We then show that Adapt & Cap can provide higher performance under the same power constraints when compared to algorithms that do not leverage the adaptive features of the applications (i.e., vCap).

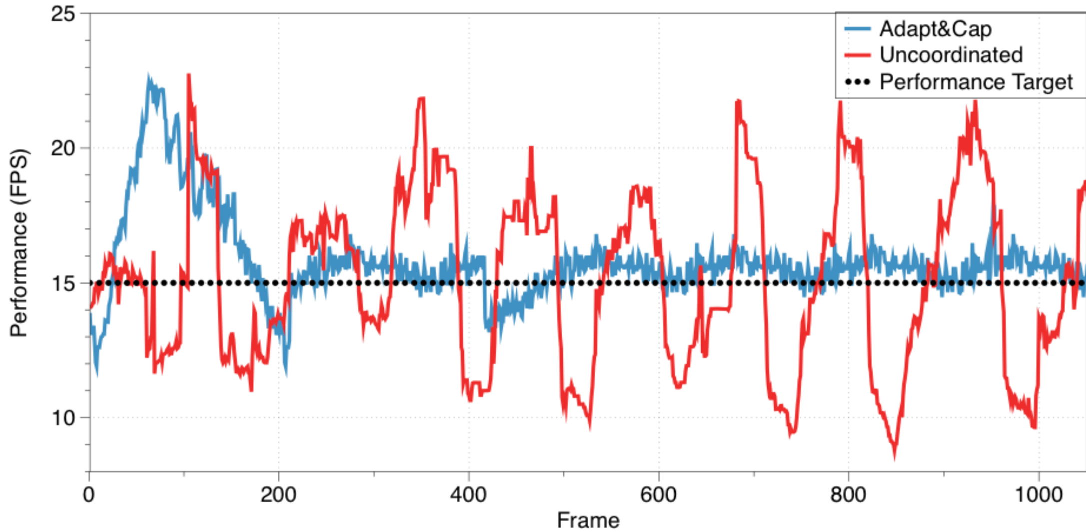

Figure 5·21: Uncoordinated approach shows oscillatory behavior, as system and application adaptation controls are not aware of other decisions that impact the performance of the system significantly.

5.4.3 Power Tracking Performance

In Figure 5·21, we show the runtime behavior of Adapt & Cap under dynamically changing power caps on the Intel Xeon server when running the adaptive version of x264. We randomly change the power cap in every 8 seconds between 115W-135W for the Intel server, and between 85W-165W for the AMD server based on the power dynamics of these two different systems. Adapt & Cap reacts to dynamically changing power constraints by tuning the CPU resource usage and the number of active cores and, in this way, it accurately tracks the power caps. The absolute value of the average tracking error is 1.8W and 1.2W for the Intel and AMD machines respectively.

In order to illustrate the benefits of coordinated approach of Adapt & Cap, we show the performance trace for two approaches under the same performance constraints in Figure 5. In both cases, we run x264 under a constant performance target of 15 FPS. For the uncoordinated case, we independently activate application-level and system-level control, whereas Adapt & Cap simply uses the best application con-

figuration together with system-level control. As Figure 5·21 shows, uncoordinated approach creates oscillatory behavior, as system and application-level adaptation continuously adjusts their decisions for satisfying the same goal, whereas Adapt & Cap control stabilizes after a few iterations. Overall, for 161 experiments with various power and performance constraints, Adapt & Cap reaches to a stable control point after the third iteration 91% of the time. Coordinating system and application-level adaptation also achieves better power tracking accuracy due to reduced oscillation. Uncoordinated approach increases the power tracking error by 3.7W when compared to the coordinated approach.

We next test Adapt & Cap under dynamically changing performance constraints. We compare the benefits of Adapt & Cap with the adaptive versions of the applications that can track the performance requirements with its internal control through parameter adjustments (i.e., AdaptiveOnly). We only evaluate the parallel portions (regions of interest) of the PARSEC benchmarks (i.e., x264, bodytrack, swaptions, streamcluster) and the whole execution of jacobi. The range between maximum and minimum performance varies among applications; therefore we randomly change the performance requirements within the predetermined maximum and minimum ranges for each application. We dynamically change the performance requirement of the applications in every 8 seconds. We use the same performance traces for both techniques.

In Figure 5·22, we report the average system-level power consumption of two real servers. Both approaches (i.e., AdaptiveOnly and Adapt & Cap) meet the performance requirements within 2%. However, in both systems, Adapt & Cap significantly reduces the power consumption by utilizing system-level control knobs. Although adaptive capabilities of the applications are useful to meet the performance requirements, AdaptiveOnly consumes more or less the same amount of power regardless

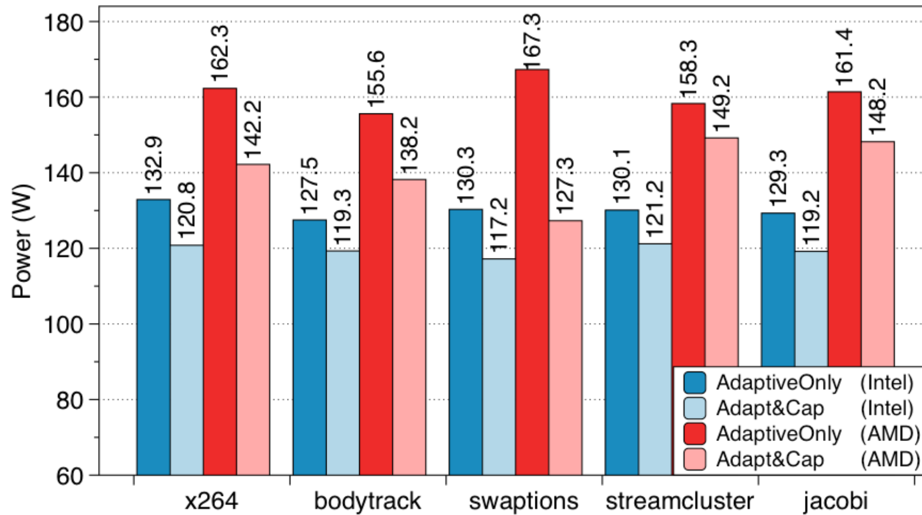

Figure 5·22: Comparison of power consumption for Adapt & Cap and only adaptive application under dynamically changing performance constraints. Adapt & Cap reduces the power consumption up to 27% compared to only application-level adaptation.

of the performance targets. The underlying reason is that the modifications to the application parameters have negligible impact on the power consumption. Therefore, Adapt & Cap can achieve the same performance at a much lower power cost by utilizing additional system-level management knobs. Furthermore, Adapt & Cap provides more efficient execution for multi-threaded applications regardless of the application characteristics. Our application set covers both highly memory-bound applications (i.e., streamcluster), as well as CPU-bound applications, such as x264 and swaptions. On average, Adapt & Cap achieves up to 27% power reduction when compared to AdaptiveOnly approach, and consistently provides lower power for all applications.

In the second set of experiments, we evaluate Adapt & Cap under dynamically changing power caps and compare the performance of Adapt & Cap with vCap, which is an adaptive yet application agnostic power management technique and runs the default versions of the applications. For each system (i.e., Intel, AMD), we create

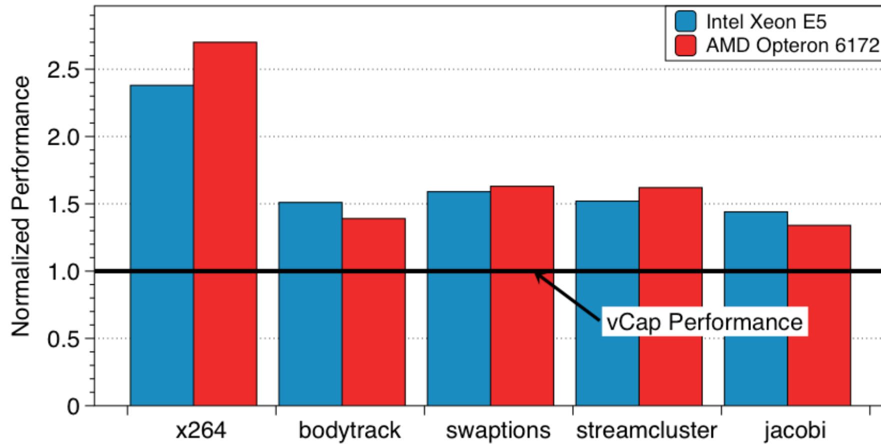

Figure 5·23: Performance results for Adapt & Cap under dynamically changing power constraints. Adapt & Cap improves the performance up to 2.7x compared to vCap under the same power constraints.

separate power cap traces, as the power ranges of these two systems vary significantly. Utilizing adaptive applications, Adapt & Cap improves the performance up to 2.7x when compared to running the default version of the application with vCap. In Figure 5·23, we show the performance improvements on Intel and AMD servers, where the default application performance is normalized to 1.

Depending on the default configuration parameters, achievable performance range, and the underlying platform, performance improvements show significant variation. However, **Adapt & Cap** consistently outperforms the vCap and provides 1.68x performance improvements on average.

5.5 Chapter Summary

In this chapter, we present our dynamic power capping techniques on both native and virtualized environments. First, we present **Pack & Cap**, a novel technique for maximizing the performance of multi-threaded workloads on a multi-core processor by jointly using *thread packing* and DVFS. Next, we introduce **vCap**, our power

capping framework for virtualized environments. **vCap** accurately tracks dynamically changing power constraints, while optimizing the overall QoS through intelligent VM placement and resource distribution across consolidated VMs. On top of the vCap framework, we design **Scale & Cap**, which improves the QoS by considering the power and performance scalability characteristics of multi-threaded applications. We introduce a formal LP-based solution to make resource distribution decisions based on the varying power and performance behavior of the consolidated applications. Finally, we present **Adapt & Cap**, which coordinates adaptive decisions at various layers of the computing stack, specifically system and application-level adaptations. Our techniques can improve the performance by 9% to 24% while meeting the dynamically changing power constraints.

Chapter 6

Conclusions

6.1 Summary of Major Contributions

This thesis has presented resource and power management techniques that target multi-threaded applications to improve the energy efficiency of multi-core server nodes. In this thesis, we targeted multi-threaded applications that are fundamentally different than single-threaded applications, and developed energy efficiency approaches that consider multi-threaded specific characteristics to make adaptive runtime power and resource management decisions.

As a part of this thesis, we have evaluated existing co-scheduling techniques that are based on co-runner application selection. Our work shows that in the case of multi-threaded loads running on multi-core systems, it is more important to adjust the allocated resources depending on the power efficiency of the applications compared to solely selecting which applications to co-schedule. This result is mainly due to the performance isolation advantages of the virtualized environments. Following our analysis, we have presented a novel policy for autonomous resource allocation for multi-threaded loads. Our policy proportionally allocates the resources according to energy efficiency of the applications to efficiently utilize the server node. Our technique includes a feedback mechanism to set user-defined performance targets per application. Based on our experiments on a real-life server, our policy achieves 17% higher throughput-per-watt on average compared to the state-of-the-art co-scheduling techniques.

Second, we have proposed power capping techniques on native and virtualized environments. Our power capping technique on native environments, **Pack & Cap**, is a novel technique for maximizing the performance of multi-threaded workloads on a multi-core processor within an arbitrary power cap. We introduce thread packing as a control knob that can be used in conjunction with DVFS to manage the power-performance tradeoff. We demonstrate that thread packing expands the range of feasible power caps, and it enables fine-grained dynamic control of power consumption. In devising a MLR classifier approach to identifying optimal operating points, we demonstrate that it is possible to automatically select Pareto-optimal DVFS and thread packing combinations during runtime. For virtualized environments, we develop and implement the **vCap**, a virtualized system management framework for multi-core servers that improves the energy efficiency of the server node by taking the applications characteristics into account. **vCap** identifies the VMs that exhibits poor performance scalability and consolidates them together. At runtime, **vCap** first estimates the total amount of CPU resources that meet the power caps. **vCap** then distributes the CPU resources among VMs according to the performance scalability of the VMs. We implemented **vCap** on a real-life multi-core server and show that it provides 12% higher energy efficiency in comparison to the state-of-the-art policies.

On top of the **vCap** implementation, we present **Scale & Cap** that incorporates the multi-threaded specific performance scalability and power efficiency characteristics to distribute the available resources across multiple VMs running heterogeneous applications. We formulate our solution as a linear programming-based algorithm and implement our technique on two multi-core servers. We evaluate various resource allocation and placement techniques together and provide insights regarding the interaction between placement and resource allocation techniques. Our results show that for tight power budgets, **Scale & Cap** brings up to 24% performance improvements

in comparison to only using a placement algorithm.

In order to improve the efficiency adaptations at multiple layers of the computing systems, we developed **Adapt & Cap**, which combines application and system-level adaptation to improve the energy efficiency. We implement **Adapt & Cap** on two real multi-core servers and show that unifying system and application-level adaptations improves the performance by 1.68x and reduces the power by 22% on average, when compared to system-only or application-only adaptations.

6.2 Open Problems

6.2.1 Improving Boosting Algorithms

The state-of-the-art performance boosting algorithms are built to opportunistically utilize the available thermal headroom whenever possible (Intel Turboboost, 2012) (AMD BAPM, 2013). Greedy approaches of performance boosting algorithms provide burst of computation for a short amount of time, therefore processor temperature quickly hits to the thermal limits. It is possible to utilize the thermal time constant phenomena to increase the amount of time spent at the highest performance state (i.e., boost state) by deploying a P-state switching algorithm. The main idea relies on the observation that the faster we switch between high and low P-states, the lower the peak temperature is due to the thermal time constant impact. Therefore, faster switching creates additional thermal headroom that can be utilized to improve the performance or to provide a sustainable boost performance.

On the other hand, the benefits due to boosting algorithms heavily depend on the application execution time. For short execution durations, existing boosting algorithms provide significant performance improvements. However, as the execution time gets longer, thermal throttling mechanism is activated to keep the processor running at a safe and reliable temperature. Throttling mechanisms diminish the

performance benefits of the boosting algorithms. Therefore, designing boosting algorithms that is aware of the execution time of the application might allow us to improve the performance even for long-running applications.

6.2.2 Cluster-level Management

Most of the power and resource management strategies are agnostic about the heterogeneous structure of the data center. Due to significant differences in the dynamic power range of the servers, the impact of the power constraints on these two servers will have different performance costs. In order to be able to optimize the performance of a cluster, it is critical to distribute the available power based on the individual power dynamics of the servers that constitutes the cluster. Therefore, future power and resource management techniques need to be aware of the heterogeneity and adapt its decisions accordingly.

Our presented work on energy efficiency has utilized various control knobs, such as DVFS, number of threads, consolidation, and resource allocation strategies at the single-node level. It is essential to consider various execution parameters to optimally manage limited amount of computational and power resources. However, the interplay across various control knobs increases the complexity of the problem to find the optimum operating point for multi-threaded workloads. Although there are many efforts to address energy efficiency challenges from various design perspectives, incorporating extensive workload analyses and runtime techniques to efficiently manage multi-threaded workload execution strategies through utilizing various control knobs might bring additional benefits to energy-efficient control strategies.

References

Ahmad, R. W., Gani, A., Hamid, S. H. A., Shiraz, M., Xia, F., and Madani, S. A. (2015). Virtual Machine Migration in Cloud Data Centers: A Review, Taxonomy, and Open Research Issues. *The Journal of Supercomputing*, 71(7):2473–2515.

Alameldeen, A. R. and Wood, D. A. (2006). IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17.

Alpaydin, E. (2004). *Introduction to Machine Learning*. The MIT Press.

AMD BAPM (2013). Bios and kernel developers guide (bkdg) for amd family 15h. http://support.amd.com/TechDocs/.

Anderson, E. and Tucek, J. (2010). Efficiency Matters! *SIGOPS Operating Systems Review*, 44(1):40–45.

AWS (2013). AWS Auto Scaling. `https://aws.amazon.com/autoscaling/`.

Barroso, L. A. and Hölzle, U. (2007). The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37.

Beloglazov, A. and Buyya, R. (2010). Adaptive Threshold-based Approach for Energy-efficient Consolidation of Virtual Machines in Cloud Data Centers. In *International Workshop on Middleware for Grids, Clouds and e-Science*, pages 1–6.

Benini, L., Bogliolo, A., and De Micheli, G. (2000). A Survey of Design Techniques for System-level Dynamic Power Management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(3):299 –316.

Bhadauria, M. and McKee, S. A. (2010). An Approach to Resource-aware Co-scheduling for CMPs. In *ACM International Conference on Supercomputing*, pages 189–199.

Bienia, C. (2011). *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University.

Bienia, C., Kumar, S., Singh, J. P., and Li, K. (2008). The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

Bonvin, N., Papaioannou, T., and Aberer, K. (2011). Autonomic SLA-Driven Provisioning for Cloud Applications. In *Cluster, Cloud and Grid Computing (CCGrid)*, pages 434 –443.

Chen, H., Hankendi, C., Caramanis, M. C., and Coskun, A. K. (2013). Dynamic Server Power Capping for Enabling Data Center Participation in Power Markets. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 122–129.

Cisco (2013). Cisco Global Cloud Index: Forecast and Methodology 2013-2018. `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/Cloud_Index_White_Paper.pdf`.

Cochran, R., Hankendi, C., Coskun, A. K., and Reda, S. (2011). Pack & Cap: Adaptive DVFS and Thread Packing Under Power Caps. In *International Symposium on Microarchitecture (MICRO)*, pages 175–185.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). RAPL: Memory Power Estimation and Capping. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, pages 189–194.

Delimitrou, C. and Kozyrakis, C. (2013). Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 77–88.

Dey, T., Wang, W., Davidson, J. W., and Soffa, M. L. (2011). Characterizing multithreaded applications based on shared-resource contention. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 76–86.

Dhiman, G., Marchetti, G., and Rosing, T. (2009). vGreen: A System for Energy-efficient Computing in Virtualized Environments. In *Proceedings of International Symposium on Low PowerElectronics and Design (ISLPED)*, pages 243–248.

Dhiman, G. and Rosing, T. S. (2007). Dynamic Voltage Frequency Scaling for Multi-tasking Systems Using Online Learning. In *Proceedings of International Symposium on Low PowerElectronics and Design (ISLPED)*, pages 207–212.

Ester, M., peter Kriegel, H., S, J., and Xu, X. (1996). A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *International Conference on Knowledge Discovery and Data Mining*, pages 226–231.

Eyerman, S. and Eeckhout, L. (2010). Probabilistic job symbiosis modeling for smt processor scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 91–102.

Fan, X., dietrich Weber, W., and Barroso, L. A. (2007). Power Provisioning for a Warehouse-sized Computer. In *In Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 13–23.

Fedorova, A., Seltzer, M., and Smith, M. D. (2007). Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 25–38.

Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B. (2012). Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 37–48.

Frachtenberg, E., Feitelson, D. G., Petrini, F., and Fern, J. (2005). Adaptive Parallel Job Scheduling with Flexible Coscheduling. *Parallel and Distributed Systems*, pages 1066–1077.

Gandhi, A., Harchol-Balter, M., Das, R., and Lefurgy, C. (2009). Optimal power allocation in server farms. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 157–168.

Gupta, D., Cherkasova, L., Gardner, R., and Vahdat, A. (2006). Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proceedings of the ACM International Conference on Middleware*, Middleware '06, pages 342–362.

Hankendi, C. and Coskun, A. (2012). Reducing the energy cost of computing through efficient co-scheduling of parallel workloads. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 994–999. IEEE.

Hankendi, C. and Coskun, A. (2013). Energy-efficient Server Consolidation for Multithreaded Applications In the Cloud. In *International Green Computing Conference (IGCC)*, pages 1–8.

Hankendi, C., Hoffmann, H., and Coskun, A. (2015). Adapt & Cap: Coordinating System and Application-level Adaptation for Power Constrained Systems. *Accepted for publication in IEEE Design & Test.*

Hankendi, C., Reda, S., and Coskun, A. K. (2013). vCap: Adaptive Power Capping for Virtualized Servers. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 415–420.

Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., and Lawall, J. (2009). Entropy: A consolidation manager for clusters. In *Proceedings of the 2009 ACM*

SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '09, pages 41–50.

Hoffmann, H. (2014). CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems. In *2014 26th Euromicro Conference onReal-Time Systems (ECRTS)*, pages 223–232. IEEE.

Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. (2011). Dynamic Knobs for Responsive Power-aware Computing. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 199–212.

HP-Intel Dynamic Power Capping (2009). Hp-intel dynamic power capping. http://www.hpintelco.net/pdf/solutions/SB_HP_Intel_Dynamic_Power_Capping.pdf.

Hwang, I., Kam, T., and Pedram, M. (2012). A Study of the Effectiveness of CPU Consolidation in a Virtualized Multi-core Server System. In *Proceedings of International Symposium on Low PowerElectronics and Design (ISLPED)*, pages 339–344.

IDC (2009). The Economics of Virtualization: Moving Toward an Application-Based Cost Model. `http://www.vmware.com/files/pdf/Virtualization-application-based-cost-model-WP-EN.pdf`.

IDC (2011). The Benefits of a Virtualized Approach to Advanced-Level Network Services. `http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/unified-network-services-uns/IDC-UNS-Feb11.pdf`.

Intel (2013). Intel Node Manager. `http://www.intel.com/content/www/us/en/data-center/data-center-management/node-manager-general.html`.

Intel Turboboost (2012). Intel turbo boost technology 2.0. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

Isci, C., Buyuktosunoglu, A., Cher, C.-Y., Bose, P., and Martonosi, M. (2006). An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 347–358.

Isci, C., Hanson, J., Whalley, I., Steinder, M., and Kephart, J. (2010). Runtime demand estimation for effective dynamic resource management. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 381–388.

Jennings, B. and Stadler, R. (2015). Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management*, 23(3):567–619.

Jerger, N. E., Vantrease, D., and Lipasti, M. (2007). An evaluation of server consolidation workloads for multi-core designs. In *IEEE 10th International Symposium on Workload Characterization (IISWC), 2007.*, pages 47–56.

Kansal, A., Zhao, F., Liu, J., Kothari, N., and Bhattacharya, A. A. (2010). Virtual machine power metering and provisioning. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 39–50.

Khan, M., Hankendi, C., Coskun, A., and Herbordt, M. (2011). Software Optimization for Performance, Energy, and Thermal Distribution: Initial Case Studies. In *Green Computing Conference and Workshops (IGCC)*, pages 1 –6.

Kim, J., Ruggiero, M., Atienza, D., and Lederberger, M. (2013). Correlation-aware virtual machine allocation for energy-efficient datacenters. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 1345–1350.

Kim, S., Chandra, D., and Solihin, Y. (2004). Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 111–122.

Kim, W., Gupta, M. S., Wei, G. Y., and Brooks, D. (2008). System level analysis of fast, per-core dvfs using on-chip switching regulators. In *International Symposium on High-Performance Computer Architecture*.

Korupolu, M., Singh, A., and Bamba, B. (2009). Coupled placement in modern data centers. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–12.

Kumar, S., Talwar, V., Kumar, V., Ranganathan, P., and Schwan, K. (2009). vmanage: Loosely coupled platform and virtualization management in data centers. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 127–136.

Kusic, D., Kephart, J., Hanson, J., Kandasamy, N., and Jiang, G. (2008). Power and Performance Management of Virtualized Computing Environments Via Lookahead Control. In *International Conference on Autonomic Computing (ICAC)*, pages 3 –12.

KVM (2008). Kernel-based Virtual Machine Management Tools. `http://www.linux-kvm.org/page/Management_Tools`.

Lee, B. and Brooks, D. (2006). Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 185–194.

Li and Martinez, J. (2006). Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *International Symposium on High-Performance Computer Architecture*.

Lin, M., Wierman, A., Andrew, L. L. H., and Thereska, E. (2013). Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Transactions on Networking*, 21(5):1378–1391.

Liu, L., Wang, H., Liu, X., Jin, X., He, W. B., Wang, Q. B., and Chen, Y. (2009). Greencloud: a new architecture for green data center. In *Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*, pages 29–38. ACM.

Ma, K. and Wang, X. (2012). PGCapping: Exploiting Power Gating for Power Capping and Core Lifetime Balancing in CMPs. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 13–22.

Macdonell, C. and Lu, P. (2008). Pragmatics of Virtual Machines for High-performance Computing: A Quantitative Study of Basic Overheads. In *International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, pages 141–152.

McGregor, R. L. and Antonopoulos, C. D. (2005). Scheduling Algorithms for Effective Thread Pairing on Hybrid Multiprocessors. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 28.

Meisner, D., Gold, B. T., and Wenisch, T. F. (2009). Powernap: eliminating server idle power. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09.

Meng, X., Isci, C., Kephart, J., Zhang, L., Bouillet, E., and Pendarakis, D. (2010). Efficient Resource Provisioning in Compute Clouds via VM Multiplexing. In *International Conference on Autonomic Computing (ICAC)*, pages 11–20.

Nathuji, R. and Schwan, K. (2008). VPM Tokens: Virtual Machine-aware Power Budgeting in Datacenters. In *International symposium on High Performance Distributed Computing (HPDC)*, pages 119–128.

Nathuji, R., Schwan, K., Somani, A., and Joshi, Y. (2009). VPM Tokens: Virtual Machine-aware Power Budgeting in Datacenters. *Cluster Computing*, pages 189–203.

Orgerie, A.-C., Assuncao, M. D. d., and Lefevre, L. (2014). A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems. *ACM Computing Surveys*, 46(4):47:1–47:31.

Porterfield, A., Fowler, R., Anirban, M., and Yeol Lim, M. (2008). Performance Consistency on Multi-socket AMD Opteron Systems. In *RENCI Technical Report TR-08-07*.

Pusukuri, K. K., Gupta, R., and Bhuyan, L. N. (2011). Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125.

Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z., and Zhu, X. (2008). No "power" struggles: coordinated multi-level power management for the data center. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 48–59.

Rangan, K. K., Wei, G.-Y., and Brooks, D. (2009). Thread motion: fine-grained power management for multi-core systems. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 302–313. ACM.

Rasmussen, E., Porter, G., Conley, M., Madhyastha, H. V., Mysore, R. N., Pucher, E., and Vahdat, A. (2011). Tritonsort: A balanced large-scale sorting system. In *In USENIX NSDI11*.

Reda, S., Cochran, R., and Coskun, A. (2012). Adaptive power capping for servers with multithreaded workloads. *IEEE Micro*, 32(5):64–75.

Romosan, R., Rotem, D., Shoshani, A., and Wright, D. (2005). Co-scheduling of Computation and Data on Computer Clusters. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 103–112.

Samson, T. (2009). AMD Brings Power Capping to New 45nm Opteron Line. http://www.infoworld.com/d/green-it/amd-brings-power-capping-new-45nm-opteron-line-906.

Sanchez, D. and Kozyrakis, C. (2011). Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 57–68.

Serebrin, B. and Hecht, D. (2009). Virtualizing performance counters. https://labs.vmware.com/download/143/.

Shin, D., Kim, J., and Lee, S. (2001). Low-energy intra-task voltage scheduling using static timing analysis. In *Proceedings of the 38th Conference on Design Automation*.

Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 124–134.

Srikantaiah, S., the, A., and Zhao, F. (2008). Energy aware consolidation for cloud computing. In *Proceedings of the 2008 conference on Power aware computing and systems*, HotPower'08, pages 10–10.

Urgaonkar, R., Kozat, U., Igarashi, K., and Neely, M. (2010). Dynamic resource allocation and power management in virtualized data centers. In *IEEE Network Operations and Management Symposium (NOMS)*, pages 479–486.

Van, H. N., Tran, F., and Menaud, J.-M. (2009). Sla-aware virtual resource management for cloud infrastructures. In *Ninth IEEE International Conference on Computer and Information Technology*, pages 357–362.

Vasić, N., Novaković, D., Miučin, S., Kostić, D., and Bianchini, R. (2012). Dejavu: Accelerating resource allocation in virtualized environments. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 423–436.

Vecchiola, C. and Pandey, S. and Buyya, R. (2009). High-performance Cloud Computing: A View of Scientific Applications. In *International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*, pages 4–16. IEEE.

VMware DPM (2010). VMware Distributed Power Management Concepts and Use. http://www.vmware.com/files/pdf/Distributed-Power-Management-vSphere.pdf.

VMware DRS (2009). Resource Management with VMware DRS. http://www.vmware.com/pdf/vmware_drs_wp.pdf.

Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zheng, C., Lu, G., Zhan, K., Li, X., and Qiu, B. (2014). Bigdatabench: A big data benchmark suite from internet services. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499.

Wang, W., Dey, T., Moore, R. W., Aktasoglu, M., Childers, B. R., Davidson, J. W., Irwin, M. J., Kandemir, M., and Soffa, M. L. (2012). Reeact: A customizable virtual execution manager for multicore platforms. In *Virtual Execution Environments*, pages 27–38.

Wang, X. and Chen, M. (2008). Cluster-level Feedback Power Control for Performance Optimization. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 101–110.

Wang, X., Chen, M., Lefurgy, C., and Keller, T. (2009). SHIP: Scalable Hierarchical Power Control for Large-Scale Data Centers. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 91–100.

Wu, L., Garg, S., and Buyya, R. (2011). Sla-based resource allocation for software as a service provider (saas) in cloud computing environments. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 195–204.

Xen (2009). Xen Management Tools. `http://wiki.xen.org/wiki/Xen_Management_Tools`.

Zhang, Y., Laurenzano, M., Mars, J., and Tang, L. (2014). SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 406–418.

Zheng, W., Bianchini, R., Janakiraman, G. J., Santos, J. R., and Turner, Y. (2009). JustRunIt: Experiment-based Management of Virtualized Data Centers. In *USENIX Annual Technical Conference*, pages 18–18.

Zhu, X., Yang, L., Chen, H., Wang, J., Yin, S., and Liu, X. (2014). Real-Time Tasks Oriented Energy-Aware Scheduling in Virtualized Clouds. *IEEE Transactions on Cloud Computing*, 2(2):168–180.

CURRICULUM VITAE