SYSTEM ABSTRACTIONS FOR RESOURCE SCALING ON HETEROGENEOUS PLATFORMS

A Thesis
Presented to
The Academic Faculty

by

Vishal Gupta

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
December 2013

SYSTEM ABSTRACTIONS FOR RESOURCE SCALING ON HETEROGENEOUS PLATFORMS

Approved by:

Dr. Karsten Schwan, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Sudhakar Yalamanchilli
School of Electrical & Computer Engineering
Georgia Institute of Technology

Dr. George Cox
School of Computer Science
Georgia Institute of Technology

Dr. Scott Hahn
Systems Architecture Lab
Intel Labs

Dr. Ricardo Bianchini
Department of Computer Science
Rutgers University

Date Approved: September 5, 2013

oṁ ajñāna-timirāndhasya jñānāñjana-śalākayā

cakṣur unmīlitaṁ yena tasmai śrī-gurave namaḥ

ACKNOWLEDGEMENTS

There are many people who have been instrumental in helping me complete my Ph.D. I would like to take this opportunity to formally thank them for their support.

I would like to sincerely thank my thesis advisor Dr. Karsten Schwan for his guidance and support during my doctoral studies. He always encouraged exploring ideas, giving me his feedback, and thus, nudging me along at each step from finding a research topic to shaping this thesis till its completion. I am especially thankful to him for allowing me to change my research area in the second year of my Ph.D. I also thank him for for providing excellent lab facilities, hardware equipments, travel assistance, and support for doing internships which gave me valuable industry exposure.

I feel indebt while expressing my immense gratitude towards Dr. Krishnan for making me understand the purpose behind doing a Ph.D and giving me the right focus. His guidance made my graduate studies a wonderful experience which otherwise felt like a difficult journey at times. His teachings on principle-centered life affected both my professional and personal life deeply. I dedicate this thesis to him without whose well-wishes and encouragement, this work would not have completed.

I owe special thanks to Mr. Ganapati Srinivasa and Dr. Scott Hahn for giving me the opportunity to do internships at Intel Labs which helped me gain momentum in my Ph.D. The projects initiated and the resources made available at Intel provided me the building blocks to develop this thesis further. I am also thankful to other team members including Paul Brett, Dheeraj Reddy, David Koufaty, Eugene Gorbatov, and Karthik Gururaj for all the insightful discussions regarding internals of Linux kernel and Intel architecture. Also, many thanks to Intel for donating the QuickIA system and their financial support through ISTC grants and fellowship award.

I am grateful to my thesis committee members Dr. Sudhakar Yalamanchili, Dr. George Cox, Dr. Scott Hahn, and Dr. Ricardo Bianchini for their feedback which was critical in developing a more coherent thesis. In addition, I wish to thank Dr. Ada Gavrilovska and Dr. Hyesoon Kim for giving their valable guidance at various junctures during Ph.D. I would also like to acknowledge my MS advisor Dr. Montek Singh for supporting my decision to move to Georgia Tech for doctoral studies.

I am thankful to my friends Sanket and Vishal for their association, not letting me lose focus of life beyond academics. I would also like to acknowledge all of my labmates for their assistance in overcoming various obstacles at work. My special thanks to Hrishi and Ripal who have been of great help and good friends to me over the years. I am thankful to my roommates Balaji, Partha, and Pushkar for their companionship making my stay in Atlanta pleasant. I also thank Susie McClain for taking care of all the administration work and the devotees for providing us with 'prasadam' lunch on-campus, relieving me from cooking on time-constrained days.

Foremost and above all, I thank Lord for His grace and blessings for giving me the opportunity and ability to get to this point, and His mercy in the form of countless other things in life for which I feel undeserving. Last but not the least, I would like to express my deep gratitude towards my parents and brother for their constant love, guidance, and support all these years. I owe this degree to them without whose selfless sacrifices, it would not have been possible for me to reach this juncture.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The increasingly diverse nature of modern applications makes it critical for future systems to have dynamic resource scaling capabilities which enable them to adapt their resource usage to meet user requirements. Such mechanisms should be both fine-grained in nature for resource-efficient operation and also provide a high scaling range to support a variety of applications with diverse needs. To this end, heterogeneous platforms, consisting of components with varying characteristics, have been proposed to provide improved performance/efficiency than homogeneous configurations, by making it possible to execute applications on the most suitable component. However, introduction of such heterogeneous architectural components requires system software to embrace complexity associated with heterogeneity for managing them efficiently. Diversity across vendors and rapidly changing hardware make it difficult to incorporate heterogeneity-aware resource management mechanisms into mainstream systems, affecting the widespread adoption of these platforms.

Addressing these issues, this dissertation presents novel abstractions and mechanisms for heterogeneous platforms which decouple heterogeneity from management operations by masking the differences due to heterogeneity from applications. By exporting a homogeneous interface over heterogeneous components, it proposes the scalable 'resource state' abstraction, allowing applications to express their resource requirements which then are dynamically and transparently mapped to heterogeneous resources underneath. The proposed approach is explored for both modern mobile devices where power is a key resource and for cloud computing environments where platform resource usage has monetary implications, resulting in HeteroMates and

HeteroVisor solutions. In addition, it also highlights the need for hardware and system software to consider multiple resources together to obtain desirable gains from such scaling mechanisms. The solutions presented in this dissertation open ways for utilizing future heterogeneous platforms to provide on-demand performance, as well as resource-efficient operation, without disrupting the existing software stack.

CHAPTER I

INTRODUCTION

1.1 Motivation

1.1.1 Resource Scaling

The diversity in the behavior of modern applications, both across applications and within applications, keeps growing. For instance, users perform a wide variety of tasks on mobile devices, ranging from low activity audio playback to compute-intensive gaming and media editing. Concerning server systems, the behavior of various applications can also be highly variable, either due to various phases in the application or variation in input load at different durations. Apart from the applications, the demands from the users of these platforms can be highly variable as well. For instance, a user may desire high energy-efficiency when operating the device on battery which may be less relevant when running on wall-power. Similarly, elastic resource scaling is a core feature for cloud platforms, due to the cost implications of used resources, particularly in the IaaS (infrastructure-as-a-service) environments like Amazon Elastic Compute Cloud (EC2) [5] and Google Compute Engine (GCE) [28].

This diverse nature of applications and user preferences demands systems that support various operating modes in order to meet their dynamic needs, thus, providing both high-performance and resource-efficient operation. However, balancing between these conflicting goals of on-demand performance and resource-efficiency can be challenging. For instance, supporting high levels of performance on a mobile system may affect its battery life negatively. Therefore, these systems should support dynamic resource scaling capabilities to address both of the requirements. Without such capabilities, a system has to either sacrifice performance for under allocation

scenarios or waste resources as in the case of over allocation. Further, it is non-trivial to figure out the right resource allocations statically which may require profiling the application under different configurations.

There are two key features for the resource management methods to be effective. First, they should be *fine-grained* in nature, implying that they should allow scaling resources in small quantities at short timescales for efficient operation. Second, the mechanisms should provide a *large scaling range* to meet the requirements of highly diverse applications. Various scaling mechanisms are already prevalent in existing systems including dynamic voltage scaling for processors [82], ballooning for memory [9], and virtual machine (VM) scaling, i.e., varying the number of VM instances used by an application as done by Amazon EC2 AutoScale [3]. While techniques like voltage scaling are fine-grained in nature but have limited scaling range, VM-level scaling options are rather a coarse-grained and heavy-weight operation.

1.1.2 Platform Heterogeneity

The approach used in this work exploits resource heterogeneity to enhance the scaling capabilities of modern platforms. Heterogeneity can exist in various platform subsystems such as processor, memory, and storage.

Heterogeneous processors, consisting of CPU cores that different in their performance/power capabilities, have been proposed as an energy-efficient alternative to homogeneous configurations [23, 30, 55]. This form of performance heterogeneity can exist at both levels: cores within a socket or across sockets as shown in Figure 1. There are several commercial implementations of such heterogeneous CPU architecture [18, 29, 45, 78]. Several studies have shown that different processor architectures are suited for different applications. For example, prior work has discussed the utility of low-powered cores for the design of datacenters [6, 48] as well as the need for high-performance brawny cores [10, 59]. Various scheduling methods for heterogeneous

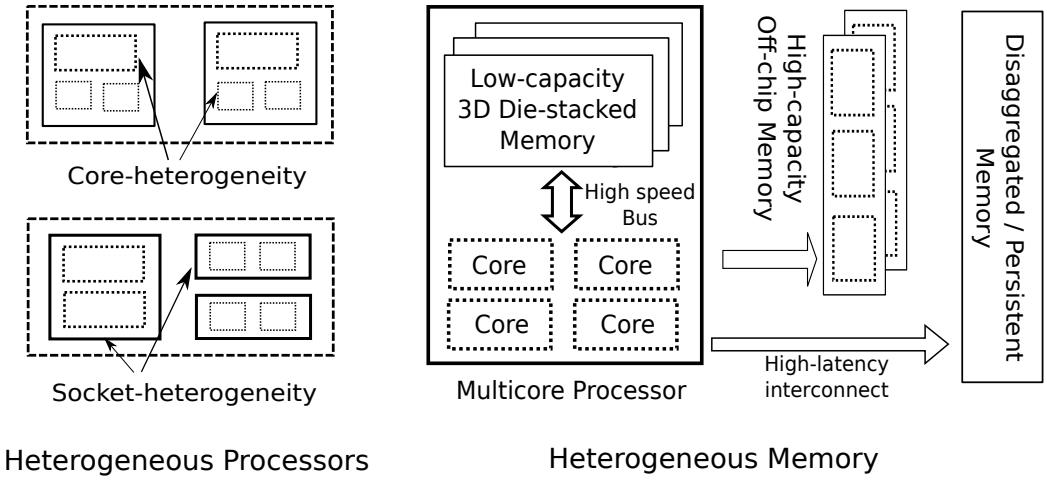


Figure 1: Platforms consisting of heterogeneous resources

cores have also been investigated [32, 54, 64, 92].

Similarly, introduction of new memory technologies such as die-stacked 3D memories, non-volatile memories, in addition to traditional DRAM, can result into a hierarchy of heterogeneous memory organization shown in Figure 1. 3D stacked memories can provide lower latency and higher bandwidth, in comparison to traditional off-chip memories [67]. However, the capacity of such memories is likely to be limited to only a few hundreds of megabytes [69]. Thus, a combination of both fast on-chip memory with additional slower off-chip memory is needed for higher capacity and expansion capabilities, specially for high-end enterprise machines. Further, addition of disaggregated memory or persistent memory technologies can also result in memory heterogeneity [21, 88, 49, 65].

Similar heterogeneity can exist in storage subsystem as well composed of persistent memory, flash memory, and hard disk based components. In this work, we focus on heterogeneous processors and memories, but the approach is applicable to other resources as well.

1.2 Challenges & Approach

The aim of this dissertation is to enable fine-grained scaling mechanisms on such heterogeneous platforms taking user requirements into account for intelligent and efficient allocation. To this end, it provides a scalable resource interface using heterogeneous components such that it uses various heterogeneous components dynamically, according to the scaling requirements expressed by the user. A scale up operation results into using a larger proportion of the faster resource for execution (a thread or a page). Similarly, a scale down operation would imply using the slower resource. This component level scaling enables a fine-grain scaling interface. Moreover, such scaling can be applied to various platform resources such as processor, memory, and storage subsystem to provide a highly scalable platform with large scaling range.

However, introduction of heterogeneity on the platforms raises new resource management challenges regarding the interface for exposing heterogeneity and mechanisms for allocation of heterogeneous resources to applications. There are two ways to approach this problem. For instance, in a virtualized environment, both hypervisor and guest operating system run their resource management operations. One option would be to expose the heterogeneous components and delegate the responsibility of heterogeneity-aware resource management to guest VMs. This choice, though giving more flexibility to applications, can be too disruptive requiring changes across the stack. An alternative approach would be to manage heterogeneous platforms in the hypervisor, thus, hiding heterogeneity for easier adoption of these systems. However, this approach can be too restrictive, not providing user and applications the ability to express their allocation preferences.

The techniques proposed in this dissertation aim to achieve the advantages of both the approaches: having flexibility of resource allocation but not overloading the applications with complexity. The proposed interface, as depicted in Figure 2, is inspired by the P-state (performance state) abstraction used by modern operating systems to

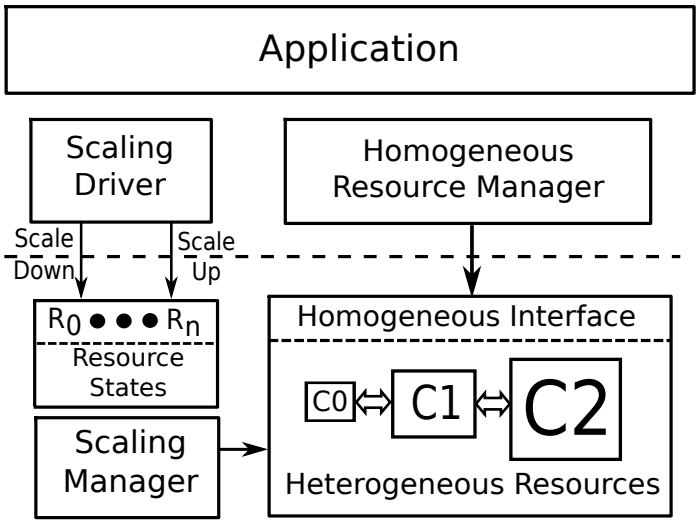


Figure 2: Resource scaling on heterogeneous platforms

perform dynamic voltage and frequency scaling (DVFS). Leveraging this, it presents a 'resource-state' interface, defining multiple-levels of resource allocations which can be requested by the application using a scaling driver, similar to the CPU governor as in the case of P-states. The input from the scaling driver is used by a scaling manager to perform heterogeneity-aware resource allocation. Thus, differences due to component heterogeneity are handled by the manager hidden from the remaining system. There are several advantages to the proposed abstractions:

- The resource state interface decouples heterogeneity management handled by the scaling manager from policy management which is handled by the scaling driver. Thus, it provides a way to hide heterogeneity which is critical to support legacy software and applications for wider adoption of such platforms.

- The scaling driver mechanism provides a way for each application to guide resource allocation to suit its own needs by using a driver customized to its own needs. For example, an application may use a power or cost-driven policy while the other application which is more sensitive to performance can employ a performance-driven policy.

- The interface shown is generic to be used across different components. Thus, it can be used to perform resource scaling across processor, memory, and storage subsystem. Further, it is also applicable to systems involving multiple levels of heterogeneity such as using stacked DRAM, off-chip DRAM, and persistent memory.

In this dissertation, we analyze the impact of heterogeneity by considering several use cases for both server systems and client devices and develop resource management methods to intelligently map heterogeneous resources to different workloads. Approaches, associated methods, and their implementation are evaluated experimentally using representative heterogeneous platforms and workloads from the mobile and the enterprise spaces.

1.3 Thesis Statement and Contributions

This dissertation aims to support the following hypothesis:

Novel resource management abstractions can exploit platform heterogeneity to enhance resource scaling capabilities on future systems, without disrupting the existing software stack.

To this end, this dissertation makes the following specific contributions:

We first perform a comparative analysis of heterogeneous multicores on the performance and energy efficiency of mobile devices and server systems. Using several real-world workloads from both the mobile and enterprise domains, experimental evaluations are carried out on a unique experimental testbed comprised of real heterogeneous CPUs that differ in both their core architecture and cache sizes, comparing the performance and efficiency for these applications. The experimental results presented in this study provide platform and system software designers a perspective on the trade-offs involved with these architectures and thus make optimal design choices.

Extending the analysis on heterogeneous cores, we also consider the 'uncore' subsystem, which in modern platforms, is an increasingly important contributor to total SoC power. Using a unique testbed comprised of heterogeneous cores with a shared uncore, we highlight the need for uncore-awareness and uncore scalability to maximize intended efficiency gains from heterogeneous cores. Next, going beyond the processor by considering the memory subsystem, we present an analysis and description of techniques for managing the heterogeneous memory resources of next generation multicore platforms with fast 3D die-stacked memory and slow off-chip memory. The resulting ability to characterize the memory behavior of representative server workloads demonstrates the feasibility of software-managed heterogeneous memory resources.

We then present **HeteroMates**, a solution that uses heterogeneous processors to extend the dynamic power/performance range of client devices. By using a mix of different processors, HeteroMates offers both high performance and reduced power consumption. The solution uses *core groups* as the abstraction that groups a small number of heterogeneous cores to form a single execution unit. Group heterogeneity is exposed as multiple *heterogeneity (H) states*, an interface similar to the P-state interface already used for frequency scaling. An H-state controller governs H-state transitions based on dynamic policies maximizing performance or minimizing power consumption, while a 'core switcher' transparently migrates tasks to the appropriate core, i.e., the one matching the chosen H-state. Thus, HeteroMates decouples heterogeneity from scheduling and provides a seamless way for adoption of such platforms in mobile devices.

Finally, we present **HeteroVisor**, a heterogeneity-aware hypervisor, that exploits resource heterogeneity to enhance the elasticity of cloud systems. Introducing the notion of 'elasticity' (E) states, HeteroVisor permits applications to manage their changes in resource requirements as state transitions that implicitly move their execution among heterogeneous platform components. Masking the details of platform

heterogeneity from virtual machines, the E-state abstraction allows applications to adapt their resource usage in a *fine-grained* manner via VM-specific 'elasticity drivers' encoding VM-desired policies. The approach is explored for the heterogeneous processors evolving for modern server platforms, leading to mechanisms that can manage these heterogeneous resources dynamically and as required by the different VMs being run. HeteroVisor is implemented for the Xen hypervisor, with mechanisms to perform elastic core scaling. Evaluation on an emulated heterogeneous platform uses workload traces from real-world data, demonstrating the ability to provide high on-demand performance while also reducing resource usage for these workloads.

In addition, we also present the description of a client workload suite used in this work along with its implementation details in Appendix A. These workloads include a diverse set of real-world client applications, representing the usage model of modern client devices.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 begins by providing an overview of processor heterogeneity and presents experimental evaluation of modern client and server workloads on a unique heterogeneous multicore platform. This evaluation is extended beyond CPU cores by analyzing the impact of uncore subsystem and evaluating heterogeneous memory organization in Chapter 3. Next, Chapter 4 describes the HeteroMates solution for mobile platforms to enable extended resource scaling modes. Chapter 5 presents the HeteroVisor system for enhancing the elasticity of cloud platforms. Finally, Chapter 6 summarizes the conclusions from the dissertation, along with several directions for future work.

CHAPTER II

HETEROGENEOUS CORES: BRAWNY VS. WIMPY

2.1 Introduction

Energy efficiency remains a critical concern for both mobile devices and server systems. Since the battery capacities of mobile devices are severely restricted due to constraints on size and weight, energy efficiency is critical to their usability. Similarly, due to cost implications of power and cooling, energy-efficient operation is a core issue for server systems as well. Desired energy efficiency, however, is challenged by ever-increasing demands of high-performance from these platforms. To continue scaling performance, the industry has made a shift towards multicore architectures for both mobile and enterprise platforms. While thus far these architectures have incorporated symmetric computational components, heterogeneous processors have been proposed as a possible alternative to improve power efficiency [23, 42, 55, 73].

This work focuses on heterogeneous processors consisting of a mix of cores that expose the same instruction-set-architecture (ISA), but differ in their power and performance characteristics. Examples of such platforms include Variable SMP from NVIDIA [78], Big.LITTLE from ARM [18, 29], and Xeon Phi architecture from Intel [45]. Such heterogeneous platforms make it possible for different applications within a diverse mix of workloads to be run on the most appropriate cores. For example, applications that do not produce a result that is time critical to the user can be run on low-power wimpy cores, while applications with their output visible to the user can be allocated to high-performance cores. Similar arguments have been made to utilize low-powered cores for the design of datacenters [6, 48], while others have discussed the need for high-performance brawny cores as well [10, 59].

This chapter investigates the opportunities and limitations in using such heterogeneous multicores on the performance and energy efficiency of modern workloads from both the mobile and enterprise domains. Our goal is to better allow system designers to assess the trade-offs and merits of moving from homogeneous systems, which are already well supported, to heterogeneous architectures that require changes across both hardware and software. We begin by providing the motivation for employing fast brawny cores and slow wimpy cores and describe advantages and limitations of using each of them. We then provide a description of the workloads used in our analysis which consists of a diverse mix of server benchmarks and a client benchmark suite targeted towards modern end-user devices like smartphones and tablets. We characterize the behavior of these applications and compare the performance and power trade-offs of using different types of processors. While previous studies either relied on simulators or emulated heterogeneous platforms, this chapter presents real performance and power data from a real heterogeneous platform.

Experimental evaluations are carried out using a unique, experimental heterogeneous multicore platform 'QuickIA', comprised of both high and low power CPUs operating in a coherence domain under shared memory. The processors differ in both their core architecture and last-level-cache (LLC) sizes. A key element of our analysis includes an evaluation of the power overhead of shared system components such as memory on the energy efficiency of heterogeneous cores which have been ignored in prior work. The QuickIA platform allows us to separate the effects of processor heterogeneity from the rest of the system which is shared by both the processors. Experimental results demonstrate that heterogeneous architectures can provide performance improvements while also lowering energy consumption for a diverse set of applications when compared to homogeneous processor configurations. They also indicate the need for novel resource management approaches for heterogeneous CPUs accounting for non-CPU components and user-perceived performance.

2.2 Why heterogeneity?

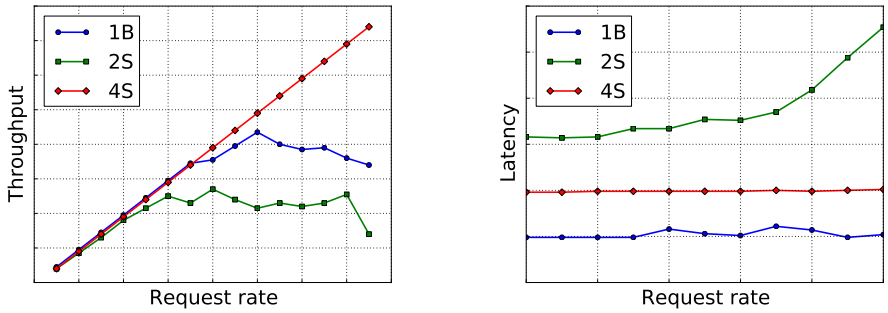
Users perform a wide variety of tasks on mobile devices, resulting in diverse platform demands. Similarly, various applications hosted in a datacenter also exhibit highly diverse behavior in their processor usage and performance requirements. The presence of virtualization technologies and server consolidation only exacerbate such diversity. Therefore, underlying platforms hosting these applications should be designed to accommodate such software diversity. However, modern processors are typically designed to satisfy only one of the two conflicting requirements: performance vs. energy efficiency. This chapter explores whether and to what extent the hardware-based arguments for heterogeneity stated above lead to realistically achievable gains for modern client devices and server systems. The remainder of this section describes various scenarios under which different types of processors can be useful.

2.2.1 Why wimpy cores?

Slow wimpy cores can provide higher energy efficiency than the larger high-performance cores, and thus, they can be used for applications not requiring high performance to save energy. For example, a small core can be used for background tasks like email update checks and normal user operation to ensure longer battery life, while the big core is reserved for performance-critical tasks. Similarly, wimpy cores can be used for I/O bound applications which consume low levels of CPU resources.

Wimpy cores can also be used to improve application throughput. Since a larger number of wimpy cores can be employed under a fixed power envelope in comparison to power-hungry big cores, they can provide higher throughput for parallel applications which can make use of such cores. For example, Figure 3(a) compares the response throughput of a web-server microbenchmark as a function of request rate for three different processor configurations consisting of one big (1B), two small (2S), and four small (4S) cores on an emulated heterogeneous platform. As seen in the figure,

the 4S configuration provides the highest throughput among these configurations at high request rates.



(a) Throughput (higher=better) (b) Latency (lower=better)

Figure 3: Best of both latency and throughput using heterogeneous cores

2.2.2 Why not wimpy cores?

Due to the limited performance of wimpy cores, they may not be suitable for latency-sensitive applications. For example, user-facing tasks which are CPU-intensive such as browsing and gaming may require a high-performance processor. Various data-center applications also have associated latency SLAs (service-level-agreements) and a wimpy core may not be suitable for these applications. For example, Figure 3(b) compares the response time of the web-server microbenchmark for the three CPU configurations (1B, 2S, and 4S) where the big core (1B) provides the lowest latency. Thus, brawny cores may win over wimpy cores when latency matters.

In addition to such latency improvements, brawny cores may also provide higher energy efficiency than their low-power counterparts for certain applications. Due to the power overhead of various system components such as memory, execution on fast cores may allow various platform components to quickly enter low power modes, resulting in lower energy consumption. This phenomenon is also known as 'race-to-idle' [72] and is particularly prominent for modern systems with deeper idle states.

2.3 Workload Description

A diverse set of applications from mobile and enterprise domain are included in this study. This section provides an overview of these workloads.

2.3.1 Client Workload Suite

Table 1: Client workload suite

Category	Workload	Description	Metric
Browser	browse	Web-page rendering	Load time
	javascript	Javascript operations	Load time
	palbum	Photo album application	Load time
Gaming	chess	2D chess game	Time
	strike	2D browser gaming	FPS
Multimedia	animate	Image sequence animation	Time
	convert	Image resize	Time
	mencoder	Video encoding	Time
	mplayer	Video playback	FPS
Productivity	calc	Spreadsheet operations	Time
	impress	Power-point slideshow	Time
	writer	Document editing	Time
Utility	7zip	File compression	Time
	diskscan	Disk I/O operations	Time
	gtkperf	GUI operations	Load time
	pguard	File encryption	Time
	sqlite	Database access	Time
	wget	File download	Time

To assess the viability of using heterogeneity on client devices, it is useful to refer to prior server-centric research on heterogeneous processors [8, 54, 64, 92], but such server-centric investigations do not directly address the needs and processor usage models seen on typical client devices. This section presents representative and typical client workloads used in our analysis and summarized in Table 1, along with relevant performance metrics. The benchmarks consists of the following components: browser, gaming, multimedia, productivity, and utility which we briefly describe below. All of these workloads are implemented in Linux and completely automated using scripts.

Browser workloads are run using the open-source Chromium browser. A more detailed description along with relevant implementation details is provided in Appendix A.

2.3.2 Server Workloads

Table 2: Server workload summary

Category	Workload	Description
Transaction processing (OLTP)	lusearch	Text search against Lucene search engine
	tomcat	Webpage retrieval using Tomcat server
	tradebeans	Online stock trading system (Java Beans)
	tradesoap	Online stock trading system (SOAP)
	hsqldb2	Transactions against a banking application
Data processing (MapReduce)	histogram	RGB histogram in a set of images
	linreg	Compute the best fit line from points
	revindex	Build reverse index from HTML files
	strmatch	Search word in a file with keys
	wordcount	Determine frequency of words in a file
Analytics	kmeans	Clustering algorithm for classification
	matrix	Dense integer matrix multiplication
	pca	Principal components analysis on a matrix
Other benchmarks		
ST-CPU	SPECCPU	Single-threaded CPU benchmarks
MT-CPU	PARSEC	Multi-threaded application kernels

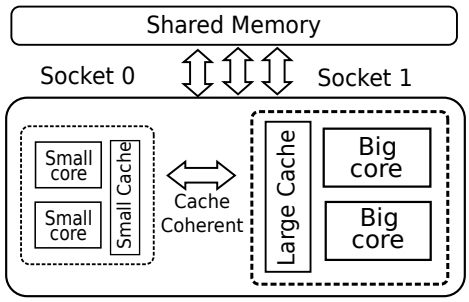
A large body of prior work on heterogeneity has relied on high-performance benchmarks such as SPEC CPU2006 and NAS parallel benchmarks for evaluation [55, 54, 64, 92]. However, applications running on modern servers are more sophisticated and diverse in their characteristics such as search engines, MapReduce, key-value stores, etc. In order to evaluate the impact of heterogeneity on the server workloads, therefore, a diverse set of server-centric workloads are included in the analysis which are summarized in Table 2. These workloads include several transactional applications such as the Lucene search engine, the Tomcat application server, an online trading system, several MapReduce data processing benchmarks (reverse index, word count,

etc.), and data analytics kernels. In addition, CPU-intensive SPEC CPU2006 benchmarks [41] and multi-threaded PARSEC benchmarks [11] are also evaluated. MapReduce and analytics benchmarks use the shared-memory Phoenix implementation of MapReduce [89].

2.4 Evaluation

2.4.1 Experimental Platform

The QuickIA heterogeneous multicore platform is used for experimental evaluation [16]. The QuickIA platform is based on a dual socket Intel Xeon 5400 series platform that has a real Xeon 5450 CPU in one socket and a real Atom N330 CPU in the other socket (see Figure 4). Both the sockets are fully cache coherent with full access to the shared platform services like memory and I/O. The processors differ in their core architecture (in-order vs. out-of-order) as well as LLC sizes, making it a unique experimental platform for evaluating the impact of CPU heterogeneity. Since various platform components such as motherboard, memory, disks, etc. are common, it allows us to isolate the effect of differences in CPU power/performance. the system is configured to run with 4GB of DRAM for client workloads and 16GB for server workloads. Figure 9(a) shows a performance comparison of the two processors for SPEC CPU2006 workloads showing an average performance difference of 2.27x.



Processor	Atom N330	Xeon 5450
Cores	2	2
H/W Threads	OFF	N/A
CPU Frequency	1.60 GHz	1.60 GHz
L1 Inst Cache	32KB	32KB
L1 Data Cache	24KB	32KB
L2 size	512KB/core	2MB/2 cores
C states	N/A	ON

Figure 4: QuickIA heterogeneous multicore platform

2.4.2 Performance Monitoring

We modified the Linux kernel to add performance monitoring support for heterogeneous cores. Since the QuickIA system contains heterogeneous cores with different CPUIDs, standard performance monitoring tools available do not work on this platform. We added a kernel module which periodically reads appropriate performance monitor counters from the system, taking into account the differences in core architectures.

2.4.3 Power Measurement

The Wattsup power meter is used to obtain system power/energy consumption data. It provides instantaneous voltage and current data with a measurement accuracy of 1.5% of reading values. Data is logged to disk using the USB interface available on the power meter with the help of a Linux driver. Since this work focuses on analyzing the impact of processor heterogeneity, we use a difference of total system power and system idle power to obtain active power used by the workload and report in the results.

2.4.4 Methodology

Experimental evaluation and analysis are carried out as the multiple steps summarized below.

- Each workload is first evaluated on a system configured to use only Xeon cores. Multi-threaded applications are configured for a one to one mapping of threads to cores used.
- Next, the same workloads are run using only Atom processor.
- The metrics collected include: application performance, power, and various performance counters including instructions retired, unhalted core cycles, LLC misses, MPERF, and timestamp counters.

- With the help of data collected in previous steps, we calculate the performance improvement provided by Xeon over Atom cores, and the energy savings or performance/watt that can be obtained by using these processors.

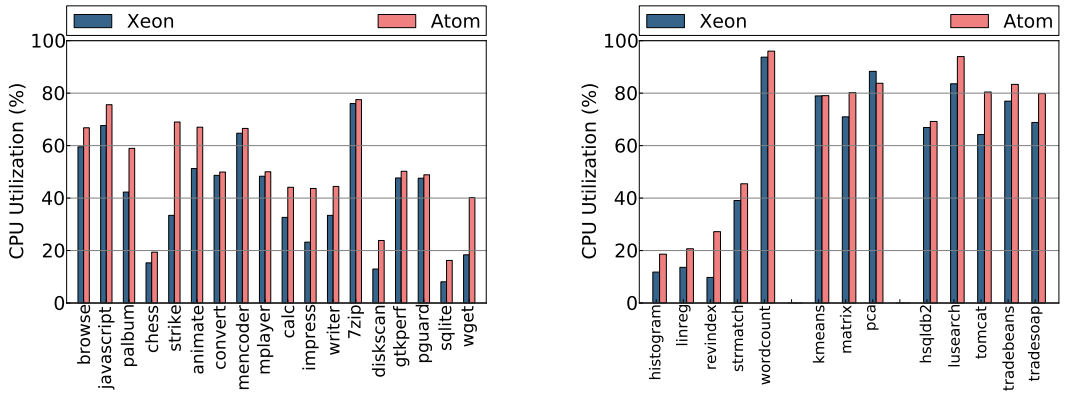
The analysis currently uses Xeon or Atom cores for the entire execution of the application. In practice, an application can dynamically switch between different types of cores and achieve higher gains, but the implementation and evaluation of a dynamic scheduling algorithm remains part of our future work.

2.4.5 Limitations

There are few limitations to the study performed in this work. First, the processors available on the QuickIA platform may not represent the latest high-performance and low-power CPUs available in the market. However, these two processors belong to the same generation. The performance/power profiles of both Intel Xeon and Atom processors have improved so we expect the relative trends to be comparable to the results obtained from the QuickIA platform. Second, the network connection used in the experiments is through the ethernet port available on QuickIA machine. However, mobile devices commonly use wireless connections which can affect the results for browser workloads.

2.5 Experimental Results

This section presents experimental results and the analysis of heterogeneous multi-cores for all of the client and server workloads described in Section 2.3. Results for the client workloads are presented first in Section 2.5.1, followed by the server workload evaluation (Section 2.5.2).



(a) CPU usage of client workloads

(b) CPU usage of server workloads

Figure 5: A comparison of CPU usage profile of client vs. server workloads

2.5.1 Client Workload Evaluation

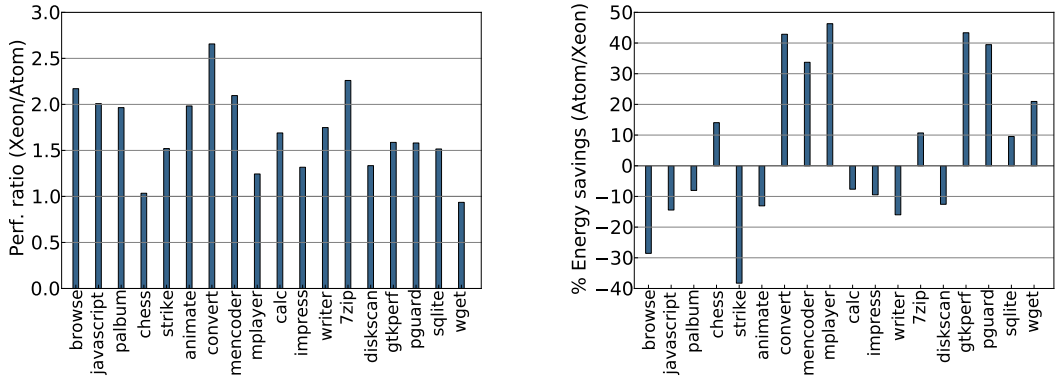
2.5.1.1 Application Behavior

The results shown in Figure 5(a) show the average CPU utilization of all of the client applications in Table 1 for execution on Xeon and Atom CPUs. As seen in the figure, applications exhibit diverse behavior in their CPU usage. 7zip, mencoder, and javascript have high CPU utilization, while other applications like productivity apps (calc, impress, writer), chess, wget make light use of CPU resources. This behavior is in contrast to that of typical server applications used by earlier work on heterogeneity [54, 55, 64, 92] which exhaust the CPU. It can also be noticed that the average CPU utilization is higher on small cores due to their simpler core architecture, requiring more processing time for the same task.

For comparison, Figure 5(b) shows the CPU utilization profile of the transactional, MapReduce, and analytics workloads in Table 2 which we collectively call 'DATACTR' workloads. These workloads either almost saturate the CPU (transactional applications and analytics kernels) or have much lower CPU-usage as in the case of MapReduce workloads due to their I/O-bound nature. In comparison, client applications exhibit much more diverse behavior.

2.5.1.2 Performance Analysis

Figure 6(a) evaluates the impact on client application performance of using heterogeneous processors. Specifically, it shows the performance ratio of using only Xeon CPU over using only small Atom for these applications. As evident from the figure, Xeon provides significant performance improvement for several applications like 7zip, convert, javascript, browse, etc. Application convert shows the highest gain of 2.67x. On the other hand, wget, diskscan, mplayer, impress, and chess show only small gains. The average performance gain for all the applications is observed to be 1.7x.



(a) Performance impact

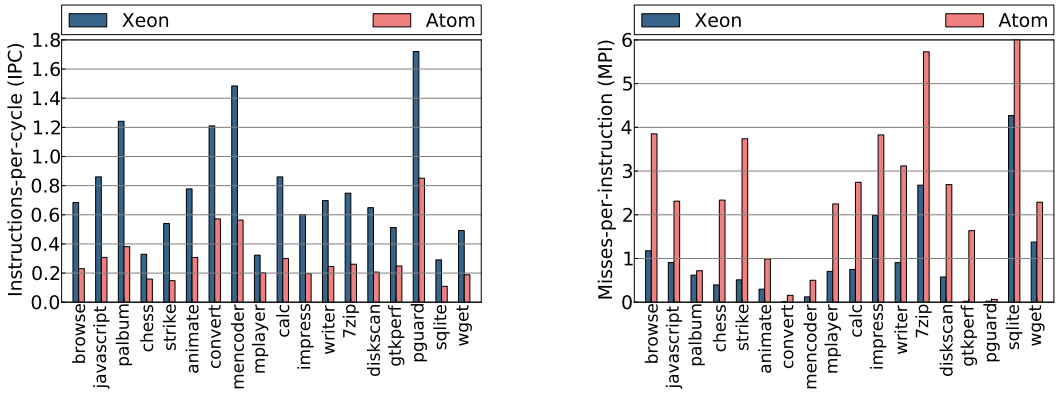
(b) Energy savings

Figure 6: Performance and energy impact of using small vs. big cores for client workloads

2.5.1.3 Power Analysis

Results comparing energy consumption on the big and small cores are shown in Figure 6(b). The results provide energy savings (%) of using Atom cores over Xeon cores. These results are particularly interesting. Several workloads like pguard, gtkperf, mplayer, convert, etc. show significant savings by using Atom CPU (maximum 46% for mplayer). However, many other applications show negative savings during execution on Atom cores. For example, strike game consumes 38% more energy when running on small cores while also providing lower performance, in comparison to big

core execution. This implies that Xeon provides both higher performance and energy efficiency for these applications.



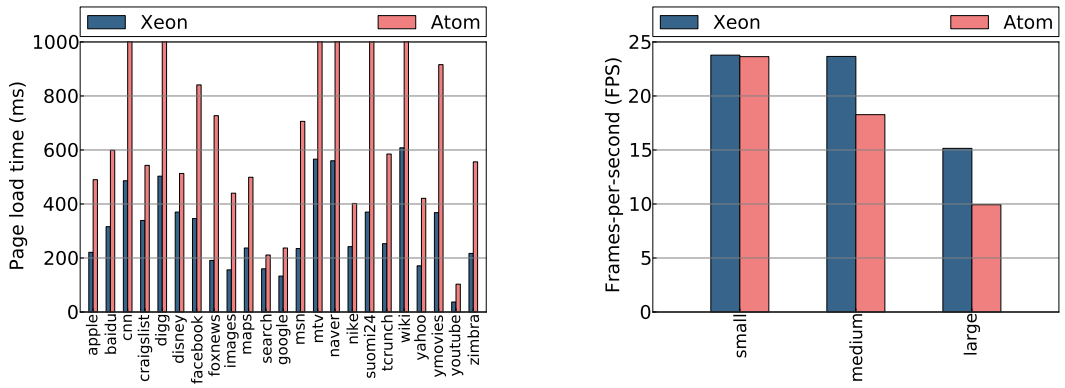
(a) Instructions-per-cycle

(b) Misses-per-instruction

Figure 7: A comparison of the behavior of client workloads on big vs. small cores

This behavior is contrary to popular understanding that small cores are more energy-efficient and can be attributed to the increased execution time and increased power of non-CPU components. It can be verified using Figure 7(a) and 7(b) which respectively compare average IPC (instructions-per-cycle) and MPI (misses-per-instruction) for these workloads on two types of CPUs. Most of the applications observe a significant decrease in their IPC when running on the small core as compared to the big core. This reduction in IPC results in the small core being active for longer durations, thereby either causing an increase in core utilization or longer execution time. Further, Figure 7(b) shows a significant increase in cache miss rate (MPI) for several applications when run on the small cores, indicating a large increase in the memory access rate and thus memory power consumption. For example, application browse and strike have the worst energy impact in Figure 6(b), while they also have a large increment in their MPI rate in Figure 7(b). The overall behavior is a combination of several factors including big/small core power ratio, performance difference, and the application behavior.

2.5.1.4 User-perceived Performance



(a) browse (b) mplayer

Figure 8: User-perceived performance for client applications

In comparison to server systems where typically total work done is used as a performance metric, user-perceived performance counts on consumer devices. If an application performs better than what a user wants or can notice, it may not be useful work. For example, Figure 8(a) shows the average load-time for various web-pages within the *browse* workload. It can be seen that the page-load latency is significantly decreased for these applications when using a big core. For example, the average page-load time for facebook page is decreased from 841ms to 346ms on the big core. However, a user may or may not perceive such a change in load-time. If 500ms is considered as the load-time threshold for the user, various sites like apple, google, yahoo can be rendered using a small core as well without exceeding the tolerance limit. On the other hand, other websites like cnn, digg, mtv, etc. strictly require a big core to be used to meet the desired performance requirement.

Similarly, Figure 8(b) shows the frames-per-second (FPS) metric for the mplayer workload when playing videos with different resolution quality (480p, 720p, 1080p). In the case of low resolution 480p video, the small core is able to perform comparable to the big core by sustaining the 23 FPS rate. Therefore, it can be run on a small core,

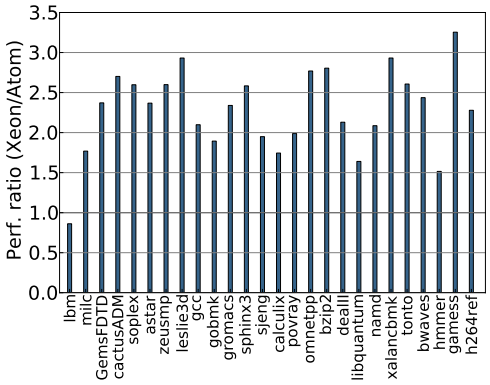
with only minor performance loss and a decrease in energy consumption. However, the playback quality degrades for higher resolution videos, demanding a big core for maintaining the desired quality. Thus, both these examples highlight the challenge of user-perceived performance associated with client applications which need to be taken into account for scheduling operations on heterogeneous processors.

2.5.2 Server Workload Analysis

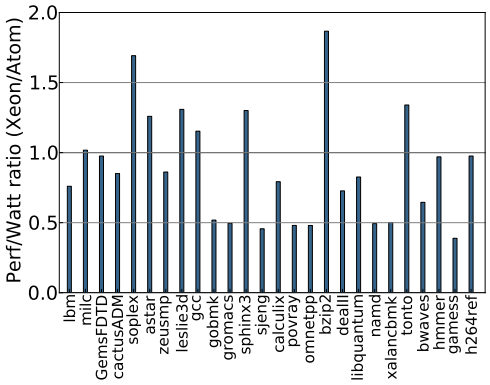
Under the server workload analysis, we first present results for SPEC CPU2006 and PARSEC applications followed by the DATACTR workloads.

Figure 9(a) shows the performance ratio for SPEC CPU2006 workloads on two QuickIA processors. The benchmarks are sorted in the order of increasing IPC (left to right), with average performance gain of 2.27x. The corresponding performance/watt ratio of Xeon over Atom is shown in Figure 9(b). A ratio greater than one signifies that the Xeon consumes less energy for the same amount of work, while a ratio lower than one implies Atom consuming less energy. As evident in the figure, different applications show affinity towards different processor for energy-efficient execution. Several applications like bzip2, soplex, sphinx3 etc. take less energy on the big core, while gamess, namd, sjeng, etc. run more efficiently on the big core. These results show the need for heterogeneity for these CPU-bound workloads to maximize system performance/watt.

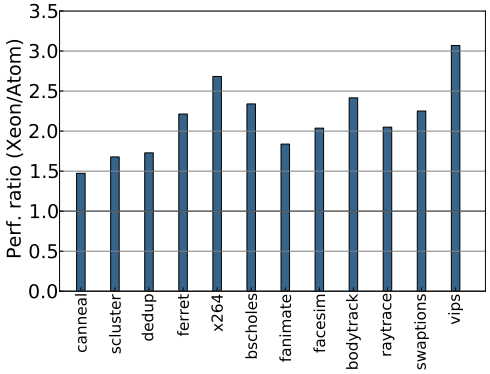
Further, many applications with low IPC (on left) such as soplex, astar, gcc run more efficiently on the Xeon core, as opposed to typical understanding that memory-intensive applications with low IPC can be offloaded to smaller cores. However, these memory-intensive applications perform better on the Xeon core with larger cache size. Our ongoing work is further investigating the performance and power predictors which can be used to make optimal scheduling decisions for such heterogeneous multicores.



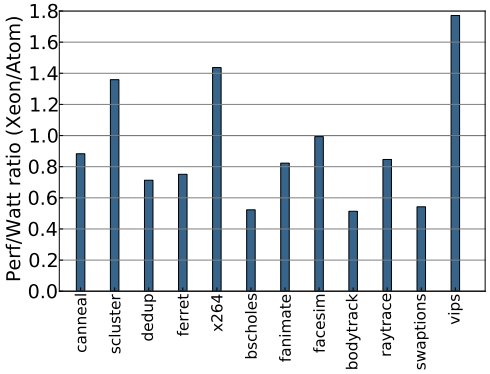
(a) CPU2006



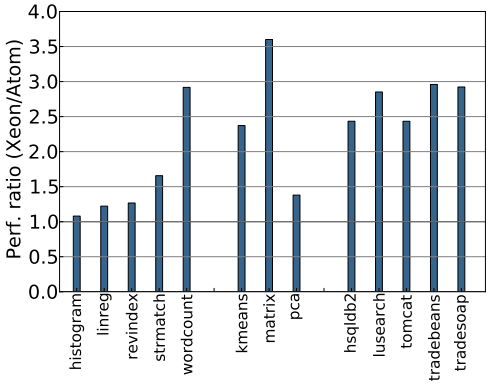
(b) CPU2006



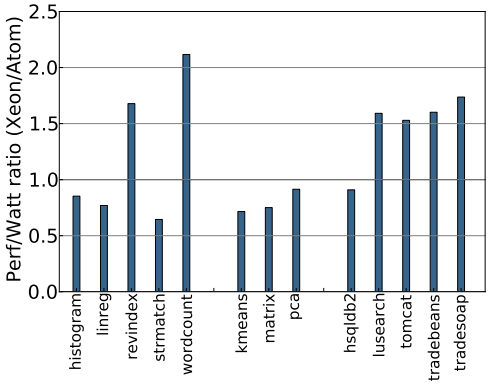
(c) PARSEC



(d) PARSEC



(e) DATACTR



(f) DATACTR

Figure 9: Performance and Performance/Watt comparison of server workloads on Xeon vs. Atom CPUs.

Similar results are obtained for several multi-threaded applications from the PARSEC benchmark suite, as shown in Figure 9(c) and 9(d). The performance improvement of using the big cores for these applications is 2.15x. The energy profile of these workloads shows that most of the applications (except fluidanimate, vips, and x264) consume less energy when run on the Atom cores.

Finally, results comparing the performance and performance-per-watt of the DATACTR workloads on Xeon vs. Atom processors are presented in Figure 9(e) and 9(f) respectively. Applications under the transactional and analytics category see large performance gains from the faster CPUs. The kmeans application kernel observes a sharp decline in its IPC when run on small cores causing a performance degradation. Similarly, performance gains for applications like wordcount, tomcat, tradesoap, tradebeans and matrix kernel can be attributed to high increase in the LLC miss rate on small cores. In comparison, data processing MapReduce applications (except wordcount) observe only minor performance gains from large Xeon cores, due to their low CPU requirement.

Comparing the performance/watt ratio of these workloads in Figure 9(f), almost all the transactional applications show lower energy consumption on the Xeon cores, thus, favoring them for both performance and energy-efficiency. On the contrary, analytics applications and many data processing MapReduce applications (except revindex and wordcount) take less energy when run on small cores. These results again confirm the original hypothesis that small cores are not always most efficient and by provisioning the appropriate server configuration for each application, a datacenter can be optimized for energy-efficient operation.

2.5.3 Opportunity Analysis

A summary of the experimental results for all the workloads is presented in Table 3, showing the relative performance and power on Xeon and Atom configurations. The

results are categorized by workload suites namely SPECCPU, PARSEC, DATACTR, and CLIENT. We observe that the Xeon processor provides average performance gains of 2.28x and 2.15x over Atom for SPECCPU and PARSEC workloads. The corresponding increase in energy consumption is 1.35x and 1.25x. On the other hand, DATACTR workloads exhibit a performance gain of 2.24x as well as a reduction in energy consumption (4%) from Xeon over Atom cores. Finally, workloads from the mobile domain show an average 68% performance gain on the faster CPU, with a 17% increase in energy consumption. The average performance improvement from big cores for all the workload categories is 2.09x with 18% higher energy consumption.

Algorithm 1: Hetero-A Policy

```
1: if Energy gain > 10% (wrt. Xeon) then
2:     CPU ← Atom
3: else
4:     CPU ← Xeon
5: end if
```

Algorithm 2: Hetero-B Policy

```
1: if Energy gain > 10% AND Perf. loss < 50% (wrt. Xeon) then
2:     CPU ← Atom
3: else
4:     CPU ← Xeon
5: end if
```

An opportunity analysis is performed to estimate the benefits of using heterogeneous processors for these workloads. The analysis assumes knowledge of relative performance/power of each workload on two processors. Two CPU assignment policies are evaluated as shown in Algorithm 1 and 2. Hetero-A policy is an energy-centric policy which selects the CPU for each workload that is more energy efficient, irrespective of its performance impact. On the other hand, Hetero-B policy is also sensitive to performance and selects Atom only if it is energy efficient and does not degrade

performance by more than 50% when compared to Xeon.

Results from the analysis are shown in Table 3 with several observations. First, Hetero-A policy provides large energy savings when compared to homogeneous configurations. When compared to Atom-only execution, it provides both higher performance (average 47%) as well as power savings (11%) by opportunistically using the big core when it is more efficient for all the workload categories. In comparison to Xeon only execution, it reduces both energy consumption and performance of the system. On the other hand, Hetero-B policy provides high performance at the cost of increased energy consumption. Overall, it provides 2x performance with only a 8% increased energy with respect to Atom, and 8.5% energy savings with a performance degradation of 4.3% with respect to Xeon-only results.

Table 3: Performance and power comparison for Xeon, Atom, and Heterogeneous configurations

Workload suite	**Performance**				**Energy**			
	Atom	Xeon	Hetero-A	Hetero-B	Atom	Xeon	Hetero-A	Hetero-B
SPECCPU	1.00x	2.28x	1.42x	2.15x	1.00x	1.35x	0.93x	1.20x
PARSEC	1.00x	2.15x	1.37x	2.09x	1.00x	1.25x	0.92x	1.21x
DATACTR	1.00x	2.24x	1.72x	2.17x	1.00x	0.96x	0.81x	0.90x
CLIENT	1.00x	1.68x	1.35x	1.60x	1.00x	1.16x	0.93x	1.02x
Average	1.00x	2.09x	1.47x	2.00x	1.00x	1.18x	0.89x	1.08x

These results show that heterogeneous multicores can be employed to improve energy-efficiency of mobile devices and server systems.

2.6 Related Work

Heterogeneous processors have been proposed to provide higher energy-efficiency than symmetric multicore processors. Using a mix of different types of cores, different phases within an application can be mapped to the core which can run them most efficiently [23, 42, 55, 56, 73]. Experimental studies have been performed to analyze

the impact of such performance asymmetry on several server workloads [8, 14]. Similar studies also exist for the mobile domain, characterizing the behavior of several client applications on heterogeneous multicore platform [32, 35]. However, earlier work relied on either simulators or emulated heterogeneous platforms for their evaluation using mechanisms like throttling (T) states, dynamic voltage and frequency scaling (DVFS), or proprietary techniques like core de-featuring. Such emulations do not realistically represent the behavior of a real heterogeneous system [54]. Also, the power data reported was obtained from models instead of real measurements. In comparison, QuickIA system used in our work contains real heterogeneous processors, allowing us to obtain real power data.

Several scheduling algorithms for efficient execution of applications on asymmetric multicore processors have also been proposed [33, 54, 58, 63, 64, 92, 99]. In addition, several techniques have been described to accelerate critical sections [42, 97] and virtual machine monitors [52, 57] using heterogeneous cores. Similarly, prior work has also investigated asymmetric cache-aware scheduling algorithms [50]. A study of several server workloads was performed by using processors with different cache sizes [7]. In comparison, our work involves processors with different core architecture in addition to cache asymmetry. Further, previous work has relied on throughput-oriented server workloads for evaluating the impact of heterogeneity, while this study targets server domain as well as client devices where energy is a premium resource, application behavior and performance metrics are diverse. We also motivate the need for novel energy-aware scheduling approaches for heterogeneous multicores.

Various benchmark suites for embedded devices are available including MiBench [38] and EEMBC [85]. However, MiBench was developed during a different era of embedding computing. Modern CPUs found in consumer devices like smartphones and tablets and the applications run on these platforms have become quite sophisticated. EEMBC suite is proprietary and is not freely available to academics. Recent work

has characterized the behavior of several Android applications on ARM-based mobile devices [39, 60]. However, their work deals with only homogeneous multicore platforms.

Concerning the datacenter environment, arguments have been made in favor of low-powered cores for the design of datacenters (e.g., FAWN [6]), while others have questioned the wisdom of using such cores for server systems [10]. Specific applications like database and web-search have been analyzed and compared on Atom and Xeon-based platforms [48, 59]. In this work, we look at a wider range of server applications from the point of view of server consolidation. Also, we use an evaluation platform which differs only in CPU configuration, with other components being shared, thus, allowing us to quantify the effect of processor heterogeneity and isolate it from differences in other components. Further, Polfliet et al. performed a cost analysis for provisioning datacenters with heterogeneous servers [84]. However, their work relied on models and simulators. Finally, several techniques have been proposed to optimize execution of datacenter applications on inherent heterogeneity due to servers from multiple generations [2, 74, 107]. Our results support their conclusions stressing the importance of employing heterogeneity for datacenter applications.

2.7 Summary

This chapter investigates the use of heterogeneous, i.e., low-power wimpy and high performance brawny, processors in modern mobile devices and server platforms. A unique experimental heterogeneous platform consisting of real Xeon and Atom processors with shared system resources is used to study and analyze a diverse mix of real-world applications from mobile and enterprise domain. Client applications represent the typical usage of end-user devices such as smartphones, tablets, while server applications include transactional applications, data processing MapReduce benchmarks, and data analytics kernel. The behavior of these applications is characterized

on heterogeneous CPUs and a power-performance analysis is carried out to quantify the benefits of using heterogeneity for these applications. Results show that heterogeneous CPUs can be used to provide a superior solution for these platforms by enabling energy-efficient execution of various applications. The importance of power consumed by non-CPU components, the challenges of user-perceived performance for mobile devices, and its implication on the energy-efficiency of heterogeneous processors are also highlighted.

CHAPTER III

BEYOND CORE: UNCORE & MEMORY SUBSYSTEM

This chapter extends the analysis performed in Chapter 2 on heterogeneous cores by including the uncore and memory subsystem. A large fraction of CPU resources is dedicated to uncore on modern multicore platforms which is shared by all the cores. In this work, we first describe the relevance of uncore in the context of heterogeneous processors and study the effect of uncore using an experimental heterogeneous platform. Going beyond the processor, we also analyze heterogeneity in the memory subsystem consisting of fast on-chip and slow off-chip memory and discuss mechanisms required to support them. Specific contributions include hypervisor-level mechanisms to detect guest memory access patterns using access bit information and transparency support for managing heterogeneous memory for virtual machines, implemented by the hypervisor. We also present an evaluation of the sensitivity of several server workloads to the performance of heterogeneous memory subsystems from an emulated heterogeneous platform.

In the remaining chapter, Sections 3.1 and 3.2 provide details on discussions related to uncore, followed by mechanisms for heterogeneous memory management in Sections 3.3 – 3.5. Finally, related work and conclusions are described in Section 3.6 and 3.7 respectively.

3.1 Uncore subsystem

Energy efficiency remains a critical concern for both mobile devices and server systems. To improve energy efficiency while providing high performance, chip vendors have adopted heterogeneous multicore processors. Previous work on heterogeneous

processors has primarily focused on core power [42, 55], but modern multicore processors also contain an *uncore* subsystem (see Figure 10), with components like the last level cache, integrated memory controllers, etc. With growing cache sizes, increasing complexity of the interconnection network, various core power optimizations, and the integration of SoC (system-on-a-chip) components on the CPU die, the uncore is becoming a significant power component in total SoC power [68]. For energy efficient operation, therefore, it becomes increasingly important to account for uncore while executing on heterogeneous cores.

This work investigates the importance of uncore power on the energy efficiency of heterogeneous multicore platforms. Unlike previous work on heterogeneous processors focused on server workloads [23, 54, 55], it targets client devices where energy is a premium resource and workload profiles are diverse. Since server workloads are not representative of the usage model of client devices, it characterizes the behavior of a diverse set of real-world client applications which are typical of end-user mobile devices and describes different ways in which they can exploit heterogeneity. Using these workloads, it further analyzes the impact of heterogeneity on workload performance and energy-efficiency, including both core and uncore components.

Experimental evaluations use a unique, experimental, heterogeneous multicore platform, comprised of both high and low power cores operating in a shared coherence domain. Results demonstrate that heterogeneous core architectures can provide significant performance improvements while also lowering energy consumption for a diverse set of applications when compared to homogeneous processor configurations. They also demonstrate that potential savings are strongly affected by the 'uncore' contribution, which motivates the need for uncore-awareness in managing heterogeneous multicore platforms and a scalable uncore design to completely realize the intended gains.

3.1.1 What is uncore?

The uncore is a collection of components of a processor not in the core but essential for core performance. The CPU core contains components involved in executing instructions, including execution units, L1 and L2 cache, branch prediction logic, etc. Uncore functions include the last level cache (LLC), integrated memory controllers (IMC), on-chip interconnect (OCI), power control logic (PWR), etc. as shown in Figure 10. With growing cache sizes and the integration of various SoC components on CPU die, the uncore is becoming an increasingly important contributor to total SoC power.

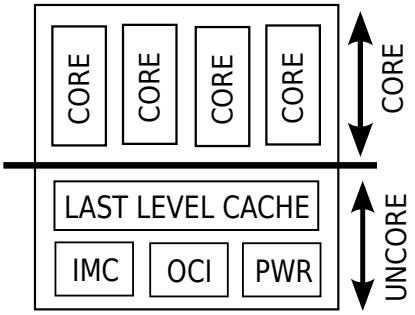


Figure 10: Core and uncore in multicore processors

3.1.2 Idle State Coordination

Modern multicore processors contain core idle states (C-states) to progressively turn off components in order to conserve power. These C-states are denoted as Cx, where x is a digit. C0 is the active C-state when the processor is executing instructions, while a higher numbered C-state (e.g., C2) is a *deeper sleep* state consuming lower power.

In addition to core C-states, processors also contain package idle states (PCx states) that govern uncore power consumption. These package C-states are related to core C-states in that the processor can only enter a low-power package C-state when all of the cores are ready to enter that same core C-state. Table 4 shows this

Table 4: Core and package idle state coordination

Package PCx		Core 1 C0	Core 1 C1	Core 1 C2
Core 0	C0	PC0	PC0	PC0
	C1	PC0	PC1	PC1
	C2	PC0	PC1	PC2

coordination of core and package idle states for a two-core system with three idle states. The resultant package C-state is always the lower of the two core C-states. Thus, the uncore subsystem remains active and consumes power as long as there is any active core on the CPU.

3.1.3 Impact of uncore

Figure 11 illustrates the impact of uncore power on the energy consumption of an application executing on heterogeneous cores. A big core running an application finishes its execution faster and enters a low-power idle state. The same application when executed on a small core takes longer (t_{small}) to finish, which also keeps the uncore active for a longer period of time. If uncore power is substantial in comparison to core power, then the energy gains from running on a small core can be strongly affected by the uncore power. For such a system, energy-efficiency gains from small core execution may be offset by the increase in uncore energy consumption due to longer execution time. This observation is in line with prior work that highlights the tradeoff between CPU and system-level power reduction in the context of frequency scaling [72].

Energy consumption for big and small core execution for such platforms can be modeled using Equations 1 and 2, respectively. Here, E refers to the energy consumed, t denotes execution time, and P_{core} and P_{uncore} represent average core and uncore power, respectively. P_{idle} is the idle platform power, and t_{idle} is the corresponding

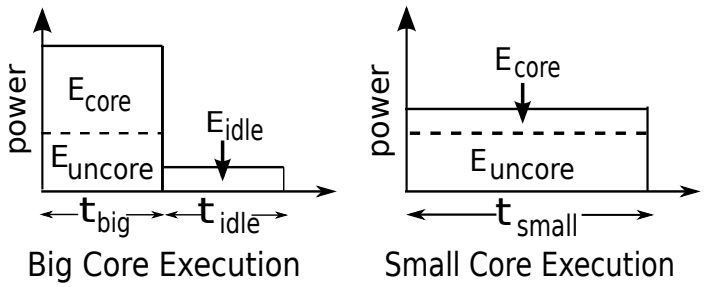


Figure 11: Effect of uncore power on the energy-efficiency of heterogeneous cores

idle time, as shown in the figure.

$$E_{big} = t_{big} * (P^{big}_{core} + P^{big}_{uncore}) + P_{idle} * t_{idle} \quad (1)$$

$$E_{small} = t_{small} * (P^{small}_{core} + P^{small}_{uncore}) \quad (2)$$

To understand the impact of uncore power, the analysis in Section 3.2 considers two uncore configurations: fixed and scalable. The fixed uncore configuration uses the same uncore subsystem when executing on either big or small cores. The scalable uncore scenario models an uncore where certain uncore components are turned off or powered down as we move to the small core. For example, fewer memory channels, memory controllers, or a smaller cache can be used with a slow small core that imposes smaller resource requirement on the cache and memory subsystem. Hence, in this case, the uncore power scales along with core power when a workload moves to a different core.

3.2 Experimental Evaluation

3.2.1 Testbed

Our experimental platform consists of a quad-core Intel i7-2600 client processor. To create heterogeneity, we use proprietary Intel tools to defeature a subset of the cores in order to emulate the performance of low-powered small cores [54]. A block diagram of the platform configuration is shown in Figure 12. An on-die graphics processor is used to accelerate graphics workloads. All of the cores operate at a frequency of

2.6 GHz and share an LLC of size 8 MB. All the workloads are run using the Linux kernel 3.0 and automated using scripts.

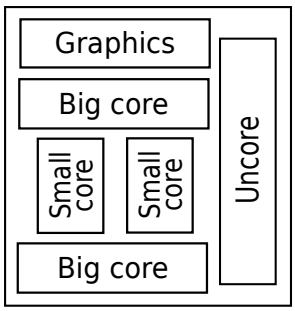


Figure 12: Experimental heterogeneous platform

3.2.2 Client Workloads

Table 5: Client workload summary

Workload	Description	Metric
browse	loads a set of web-pages periodically emulating user's think time	Load time
javascript	Javascript benchmark performs standard browser operations	Load time
palbum	photo-album application that flips through photographs	Load time
mplayer	a H/W accelerated version of mplayer plays an HD movie clip	FPS
mytube	plays an H.264 video inside the browser for 120 seconds	FPS
openarena	plays a benchmarking demo from a 3D first-person-shooter game	FPS
strike	replays a demo session of a web-based 2D game (120 sec.)	FPS
7zip	a parallelized version of 7zip compress a text file	Time
eclipse	Java based benchmark runs performance tests for the Eclipse IDE	Time
filescan	I/O intensive workload that scans through the Linux source tree	Time
gmagick	GraphicsMagick application is used to resize a set of images	Time
x264	x264 media encoder is used to encode a media file	Time

To assess the viability of using heterogeneity for client systems, we choose a diverse set of real-world applications which are typical of modern end-user devices since prior server-centric research on heterogeneous processors [23, 54, 55] does not directly address the needs and processor usage models seen on client devices. Table 5 provides a summary of the applications used in our analysis and relevant performance metrics. This section categorizes these applications based on their behavior and discusses opportunities for exploiting heterogeneity.

3.2.3 Methodology

Experimental evaluation and analysis are carried out as the multiple steps summarized below.

- Each workload is first evaluated on a system configured to use only big cores. Multi-threaded applications are configured for a one to one mapping of threads to big cores.
- Next, the workloads are run using only small cores.
- The metrics collected include: application performance, IPC, LLC accesses, and various core and package C-state residencies.
- With the help of data collected in the previous steps and the power models described in Section 3.2.4, we calculate the performance improvement and the energy savings of using small vs. big cores.

Our analysis assumes the use of big or small cores for the entire application run. The implementation and evaluation of a dynamic scheduling algorithm for client devices remains part of our future work.

3.2.4 Power Model

The emulated heterogeneous platform mimics the performance of small cores. However, it does not match the power characteristics of an actual small core built using a different process technology for low power consumption. We, therefore, rely on power models to obtain core and uncore energy consumption.

3.2.4.1 Core Power

The average power consumption of a CPU core can be modeled using the following equations:

$$P_{core} = R_{active} * P^{core}_{active} + R_{idle} * P^{core}_{idle} \quad (3)$$

$$P^{core}_{active} = C_{dyn} * V^2 * f \quad (4)$$

Here, R_{active} and R_{idle} denote core active and idle state residencies (%), and P^{core}_{active} and P^{core}_{idle} are the corresponding power values. C_{dyn} is the dynamic capacitance, V denotes the operating voltage, and f represents the switching frequency. Big core C_{dyn} is modeled as a function of IPC in Equation 5, as shown and validated by other researchers [95]. Similarly, Equation 6 models the capacitance for a small core having three-times smaller area than that of the big core.

$$C_{big} = 0.499 * ipc_{big} + 0.841 \quad (5)$$

$$C_{small} = 0.472 * ipc_{small} + 0.176 \quad (6)$$

3.2.4.2 Uncore Power

Similar to core power, uncore power is modeled using package idle state residencies (U_x) as shown below:

$$P_{uncore} = U_{active} * P^{uncore}_{active} + U_{idle} * P^{uncore}_{idle} \quad (7)$$

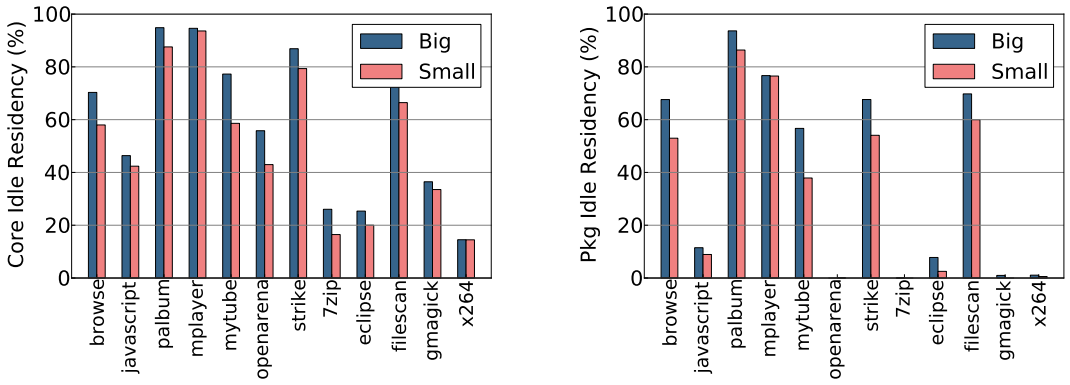
$$P^{uncore}_{active} = P_{wake} + P_{activity} * LLC_{rate} \quad (8)$$

Further, uncore active power (P^{uncore}_{active}) is modeled as a function of LLC activity in Equation 8 where P_{wake} is the fixed power cost of waking up various uncore components, while the $P_{activity}$ component scales with the LLC access rate LLC_{rate} (relative to peak access rate including both cache hits and misses).

The analysis uses a value of 0.9 V for the voltage (V), and frequency (f) is kept at 2.6 GHz. For this platform, the average big core and small core power for all our workloads is obtained to be 2.37 W and 0.95 W respectively. A comparable uncore is modeled using a value of 1.2 W for P_{wake} and $P_{activity}$ in case of a fixed uncore and scaled down to half for a scalable uncore. Core and uncore idle power are assumed to be 0.1 W and a 1.5 W power component is attributed to the on-die graphics processor which also scales with the LLC activity.

3.2.5 Results

The results shown in Figure 13 provide a comparison of application behavior on heterogeneous cores. Specifically, they compare core and package idle state residency for all of the workloads in Table 5, for big and small core execution. As evident from the figure, most of the applications observe a decrease in their idle residency when running on the small vs. big cores due to the small cores being active for longer durations. Further, many applications are seen to have almost negligible package idle residency. These applications either heavily use the graphics processor (e.g., openarena), or they always keep one of the cores busy (e.g., 7zip, gmagick, x264), and thus do not allow the uncore to enter into an idle state.

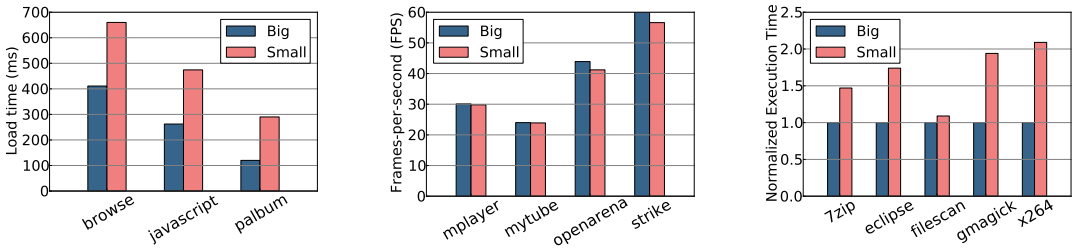


(a) Core Idle Residency

(b) Package Idle Residency

Figure 13: A comparison of the behavior of several client workloads on big vs. small cores

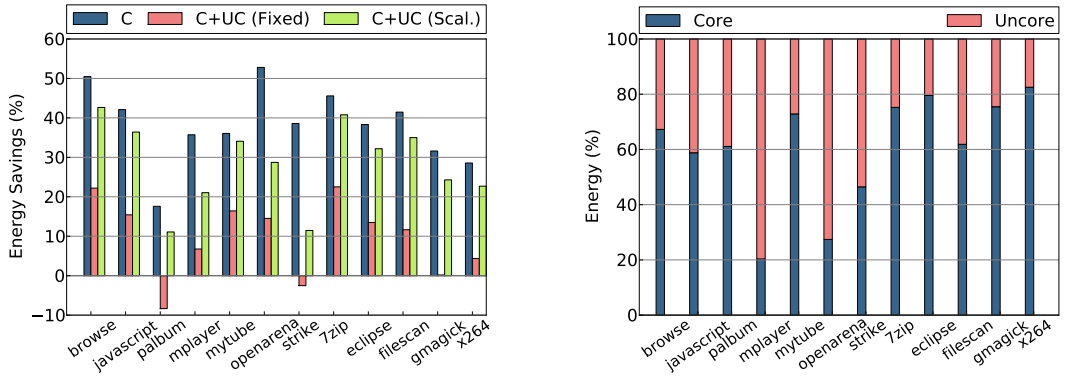
The results shown in Figure 14 evaluate the impact on performance of using heterogeneous processors for various client applications in Table 5, categorized by the respective performance metrics. Figure 14(a) compares the average load-time for the browse, javascript, and palbum workloads. We see that the latency is significantly decreased for these applications when using a big core. Thus, a big core provides a notable performance boost for such intermittent applications. In contrast and as



(a) Load time (ms) (b) Frames-per-second (FPS) (c) Normalized Execution Time

Figure 14: Application performance comparison on big and small cores

depicted in Figure 14(b), when considering the frames-per-second (FPS) metric for various graphics and media applications, we see only minor performance degradation on a small core, at levels not perceivable to end-users. Therefore, they can be run on a small core, to gain potential decreases in energy consumption (discussed further below). The last graph (see Figure 14(c)) compares the normalized execution times for various applications. All of the applications except filescan in this category show a significant improvement in performance with the big core.



(a) Energy Savings (%) (b) Energy Distribution

Figure 15: Uncore evaluation showing energy savings and energy distribution

Energy savings results computed based on our power models are shown in Figure 15(a). The figure shows savings for three configurations: core-only savings (C), total SoC-wide savings (C+UC) with a fixed uncore, and with a scalable uncore.

As seen in the figure, all of the applications show significant gains on a small core in terms of core energy savings. The palbum application has the lowest savings of 17.58%, while openarena has the largest savings of 52.79%. However, these savings are strongly affected when the power consumption of the uncore is taken into account. Some applications even exhibit negative energy savings. On the other hand, when a scalable uncore is used, these savings increase and become comparable to core-only energy savings. Further, Figure 15(b) shows the relative contribution of core and uncore energy consumption for all the applications during big core execution, on a fixed uncore configuration. These results include graphics power in the uncore component. As evident, CPU-intensive applications (e.g., 7zip, gmagick, x264) show a significant core power component, while the uncore fraction dominates for other applications like openarena and mplayer. These results not only demonstrate the importance of taking uncore power into account for scheduling operations, but they also motivate the need for a scalable uncore design to obtain large gains from heterogeneous multicores.

Going beyond the processor (core and uncore), heterogeneity can also exist in memory susbystem using a combination of fast on-chip memory and slow off-chip memories. In the following sections, we study the effect of such memory architectures on modern applications and describe mechanisms for managing them efficiently.

3.3 Heterogeneous Memory Organization

Die-stacked memories can provide lower access latency and higher bandwidths at lower power levels, in comparison to traditional off-chip memories [67]. However, such die-stacked memories are likely to be constrained in size, i.e., they are projected to have capacities ranging only to a few hundreds of megabytes [69]. This suggests a usage model in which they are combined with off-chip memory to provide higher capacity and low latency capabilities. For enterprise-class or high-performance machines combining a limited amount of fast on-chip memory with additional slower off-chip

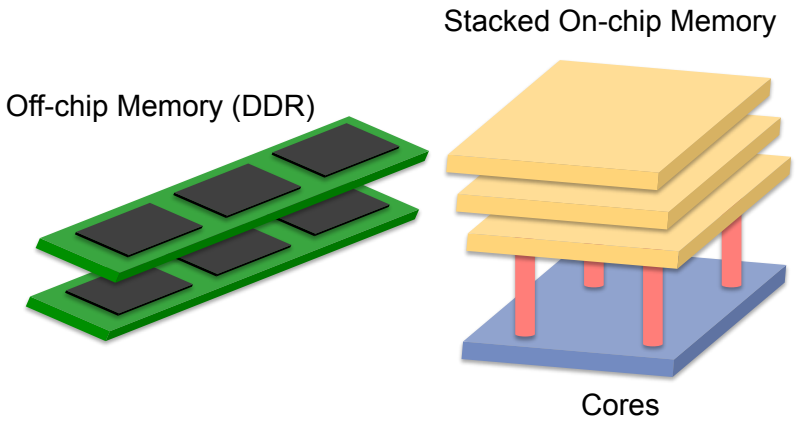


Figure 16: Heterogeneous memory organization consisting of a combination of on-chip and off-chip memories.

memory, will therefore, result in the hybrid or heterogeneous memory systems shown in Figure 16.

Die-stacked DRAM can be utilized as (i) hardware-managed cache or (ii) software-managed memory. The former approach has the advantage of being able to quickly react to changing memory access patterns, and it provides a transparent way to incorporate such memory architecture in ways that support legacy applications. Potential drawbacks and challenges of this approach are that first, it can result in high overhead for managing the tags of such large sized caches, and second, it would require extended coherency support. An additional issue is the consequent lack of software control over memory placement.

Alternative ways to manage stacked DRAM are actively being investigated in the architecture community, but in this work, we explore how an operating system or hypervisor can use its information about application behavior to manage the heterogeneous memory resources of future multicore platforms. Specifically, we investigate an approach in which stacked DRAM is explicitly exposed as system-visible memory, and we then evaluate the feasibility of software-based memory management for the resulting heterogeneous memory platforms.

In particular, given the increasing use of virtualization in server systems, several challenges need to be investigated for managing heterogeneous memory resources. (1) Hardware provides only limited visibility into the memory access behavior of guest virtual machines (VMs), e.g. x86 provides only one-bit information such as access bit in the page tables. Therefore, efficient methods are required to detect which pages are critical for a guest's performance based on such limited information from hardware. (2) The hypervisor should implement its management scheme transparently to the guest OSes. This may be challenging since the page tables are owned by the guest in a paravirtualized environment, thus making it difficult to migrate its pages between memories transparently without guest involvement. Even with hardware virtualization support, such multiple mappings should be handled properly. This also involves TLB management across cores to prevent stale mappings.

This chapter presents techniques to address these issues. First, we enhance the hypervisor to build an access-bit history for each VM, by periodically scanning the access bits available in page tables. This 'a-bit history' is then used to detect the guest's 'hot' pages and determine the guest VM's page working set. Since hot page and working set detection requires periodically scanning page tables, which can incur high overhead, we maintain additional data structures for quickly accessing page table entries. In addition, scans are done in the virtual time of guest virtual machines, for accurate accounting. Finally, the hypervisor mirrors guest page tables and transparently uses these mirrors, which allows the hypervisor to manipulate guest page mappings by simply changing their mirror page tables, without requiring guest operating systems to be altered in any way, i.e., transparently to guests.

Page access tracking, hot page detection, and mirroring are fully implemented in the Xen open-source hypervisor, thereby enabling experimental evaluation of overheads in realistic server platforms. To emulate such platforms' future memory heterogeneity, we use a multi-socket Intel Westmere platform in which one of its memory

controllers is throttled, resulting in the presence of both 'fast' (regular DRAM, emulating future 3D stacked DRAM) and 'slow' (throttled DRAM, emulating future off-chip DRAM) memory in the system. Experimental results obtained on this machine and memory configuration characterize the memory behavior of standard server workloads, in terms of their working set sizes and the performance impact of memory heterogeneity. The page migration mechanism is evaluated with micro-benchmarks, to show the feasibility of software management for future heterogeneous memory systems.

3.4 Implementation

To leverage die-stacked low latency DRAM to reduce an application's overall memory access latencies, it is important to detect and then manage its 'hot' pages. This requires efficient methods for memory access tracking, described next.

3.4.1 Memory Access Tracking

Current multicore platforms provide limited support for detecting applications' memory access patterns. Specifically, each entry in the page table is associated with an *access bit*. This bit is set by the hardware when the corresponding page is accessed. Software is provided control to reset this bit. This single-bit information is used in our work to determine a VM's memory access pattern, leveraging earlier work on cache management [62]. Specifically, we periodically scan and collect the access bits in guest page tables, to form a bitmap termed as 'A-bit history' (access-bit history). If a 32-bit word and a 100ms time interval are used, one word amounts to roughly 3.2 seconds of virtual time. Therefore, a dense A-bit history (i.e., many 1's) would indicate the presence of hot pages. Several optimizations are used to minimize overheads, discussed later in this section.

To capture an accurate A-bit history, a process's virtual time rather than wall-clock time is used. This avoids unnecessary page table scans and a more accurate

detection of hot pages. The hypervisor is extended to track processes' virtual time across various events, and each time the 100ms boundary is crossed, its page table is scanned for A-bit collection.

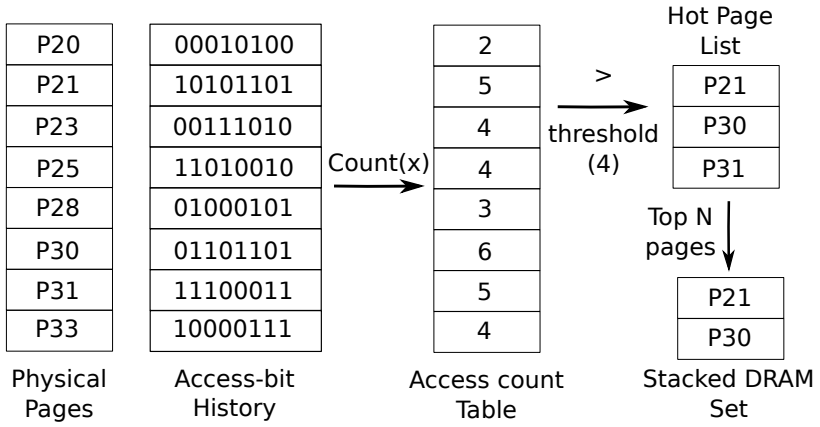


Figure 17: Hot page detection using a-bit history

An implementation in the Xen open-source hypervisor obtains and maintains A-bit histories for arbitrary guest VMs. Since Xen employs frame tables for memory management – large tables in which each entry corresponds to some physical page – we extend this data structure to embed our A-bit history and other information, as shown in Figure 17. The A-bit history is used to hold each frame's access bit history. Next/Prev pointers help form linked lists of pages for efficient access.

In addition, an Rmap structure is used to store reverse mapping information to make it easy to unmap and map some given page. Each physical page (mfn) has one Rmap_list, which is list of Rmap_set. Rmap_set is a fixed size array containing pointers to page table (PT) and page table index (PTI). Therefore by iterating Rmap_list and Rmap_set, all mappings to the given page can be found and changed. Without this Rmap structure it would be too expensive to find mappings to a given physical page, which is needed to change mappings for page migration.

Further, for guest transparency, the hypervisor mirrors each guest's page table which is installed in the hardware base register (CR3). This is very similar to shadow

page table, and this allows us to change virtual-to-physical mappings without changing guest OS. By this, page migrations are transparent to the guest OS.

3.4.2 Memory Allocation Policy

Stacked DRAM memory management is concerned with both intra-VM and inter-VM memory allocation policies.

Our intra-VM page placement policy aims to utilize a limited allocation of fast stacked DRAM for a VM. Pages with the highest hit rate are moved to stacked DRAM. For hot read-only pages, two copies are maintained: a home copy and a satellite copy. The home copy resides in off-chip DRAM, while the satellite copy resides in stacked DRAM. When such read-only pages need to be migrated back to off-chip memory, the satellite copy for these pages is simply discarded, and the home copy is used for accesses thereafter. This saves a page copy for moving data back to off-chip memory. For read-write pages, only a single copy is maintained, and a copy is performed each time when a page is moved back and forth between memories.

In a manner similar to allocating constrained physical memory resources across VMs using memory ballooning, the inter-VM allocation policy aims to distribute stacked memory across applications based upon their activity. We consider two policies in this work.

Share-based allocation: this policy uses pre-defined shares, e.g., set by the administrator or a cloud allocator, to divide stacked DRAM capacity among VMs. Memory is distributed as a weighted sum of these shares as shown in Equation 9.

$$mem(vm_i) = mem_{total} * \frac{share(vm_i)}{\sum_{i=1}^{n} share(vm_i)} \tag{9}$$

WSS-based allocation: this policy uses the working set size (WSS) information for each VM to control memory allocation. The allocation is performed by using WSS as the share value in Equation 9.

3.5 Experimental Evaluation

3.5.1 Heterogeneous Memory Emulation

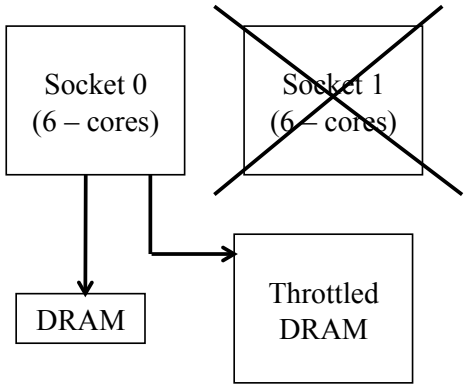


Figure 18: Emulated heterogeneous memory platform

Earlier work on stacked DRAM in the architecture community has relied on architectural simulators. In order to conduct heterogeneous memory research on actual systems with realistic server workloads, we take the alternative approach of emulating heterogeneous memory on a multi-socket platform.

Our experimental platform consists of a dual-socket 12 core Westmere X5650 server with 12GB DDR3 memory. As shown in Figure 18, cores from the first socket are used for running programs that can access memory from both sockets, i.e., cores from the second socket are kept idle. This NUMA configuration provides an approximate 1.5x difference in memory latency between the two nodes. In order to emulate more heterogeneous configurations, however, we use memory controller throttling on the remote node to slow it down further.

Memory throttling is enabled by writing to the PCI registers (Integrated Memory Controller Channel Thermal Control). By applying different amount of throttling, varied memory configurations can be emulated for emerging memory technologies [86, 106]. Figure 19 shows a comparison of normalized bandwidth and latency for three memory configurations for the memory-intensive Stream benchmark. The M0

memory configuration corresponds to no throttling, while M1 implies small throttling, and M2 implies higher throttling. As expected, we see progressively lower bandwidth and higher latency with M1 (2.5x) and M2 (5x) configurations. M0 is used as the base configuration for evaluation.

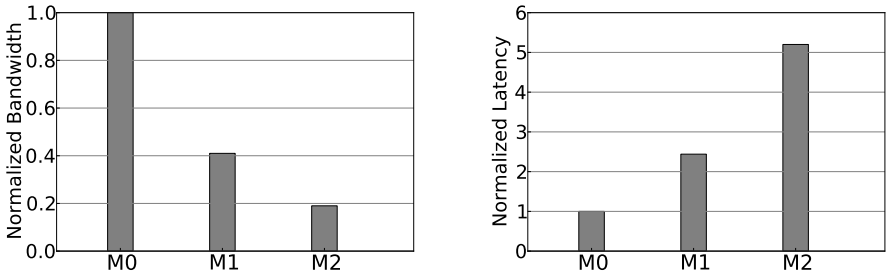


Figure 19: Bandwidth and latency comparison for different memory configurations

3.5.2 Workloads

We evaluate the impact of heterogeneity on server workloads by using a diverse set of server-centric workloads summarized in Table 6. These workloads include CPU-intensive SPEC CPU2006 benchmarks, multi-threaded PARSEC benchmarks, and several MapReduce data processing benchmarks and with data analytics kernels. The MapReduce benchmarks use the shared-memory Phoenix implementation of MapReduce [89], where input datasets are cached in memory.

Table 6: Workload summary

Workloads	Description
SPECCPU	Single-threaded CPU-intensive benchmarks
PARSEC	Multi-threaded application kernels
Phoenix	Shared-memory MapReduce kernels

3.5.3 Results

The experimental data shown in Figure 20 depicts working-set size (WSS) graphs as a function of time for several SPEC CPU2006 workloads. As seen in the figure, several CPU-bound applications have very small WSS, e.g., 0.8 MB for namd. In comparison, memory-intensive workloads like mcf and lbm have much larger working-sets of size 200 MB and 400 MB respectively. Further, WSS dynamically changes over time for these applications, thereby showing the need for dynamic memory management. These results highlight the fact that only a fraction of the total memory region is actively used by the application which is critical for its performance. These pages should be retained in fast memory, while the remaining pages can be allocated from slow memory.

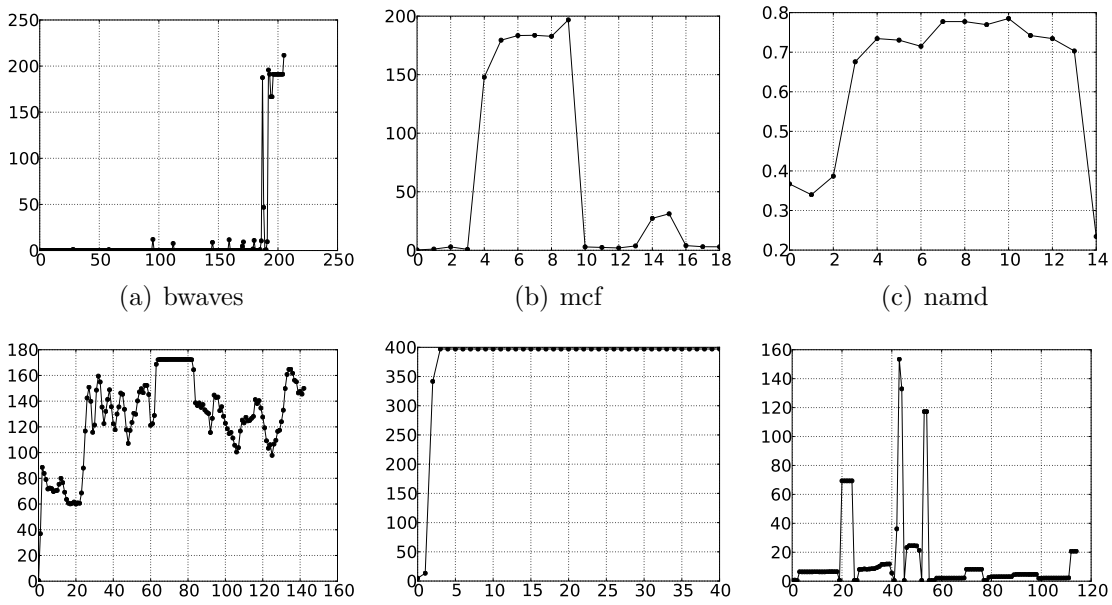


Figure 20: WSS curve for SPEC CPU2006 applications (x-axis = time (s), y-axis = WSS (MB)).

Our next results evaluate the performance impact of memory slowdown for all of the workloads in Table 6. These applications are executed with different memory

configurations, by varying the amount of throttling applied to the memory controller. Figure 21 shows the performance loss (%) for the applications for two memory configurations (M1 and M2) as compared to no throttling (M0) as described in Section 3.5.1. As we see in the figure, several applications suffer from high performance loss due to memory slowdown, while many others see small impact. Particularly, the mcf, milc, GemsFDTD, and lbm workloads from SPEC CPU2006; the facesim, canneal, and streamcluster benchmarks from PARSEC, and the pca kernel from the Phoenix suite observe severe degradation. As expected, the performance degradation becomes smaller with faster M1 memory configurations. The highest impact is observed for the mcf workloads to be 1431% (15x) and 537% (6x) for the two configurations. Thus, by managing the active memory pages for these applications in the stacked DRAM, substantial performance gains can be achieved. These experiments were also performed with different CPU frequencies, to analyze the correlation between processor speed and memory slowdown on the performance. We observe similar trends for these applications, but with a smaller magnitude due to a slower CPU.

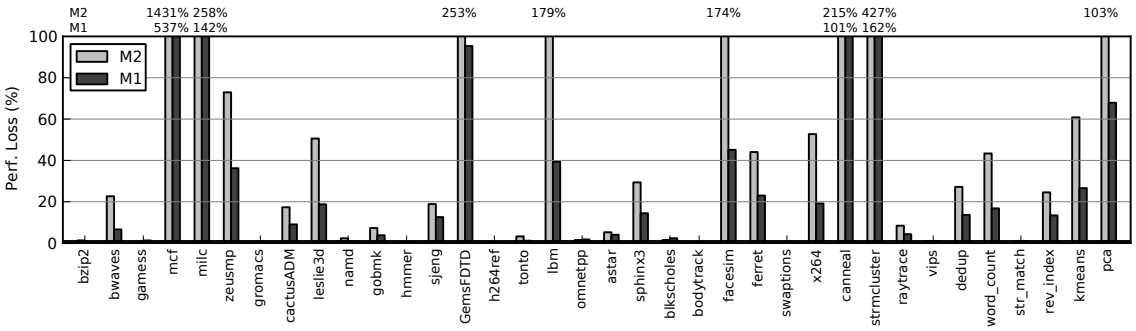


Figure 21: Comparison of performance impact of memory slow down with different memory configurations

We evaluate our page migration mechanisms using a micro-benchmark *memlat*, which allocates a large region of memory and randomly accesses it. The benchmark runs for 30 seconds and reports average access latency for each second. Figure 22 shows the experimental results for two scenarios: when memory is statically allocated

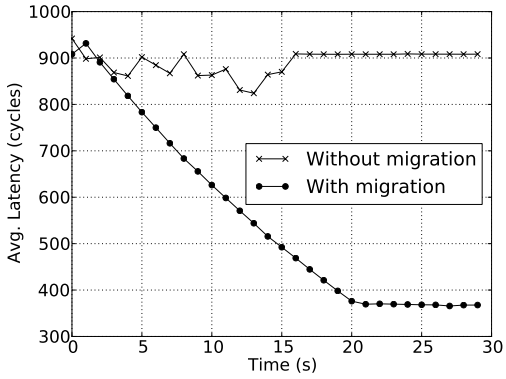


Figure 22: Micro-benchmark results: Memory access latency with and without hot-page migration

from slow memory and when migrations are enabled to dynamically move hot-pages to fast memory. When migration is disabled, the access latency remains high throughout the execution, with an average value of 891 cycles. On the other hand, when hot-page migration is enabled, latency starts to decrease until it reaches a value of 367 cycles and then remains fixed. Pages are initially gradually moved to fast memory, thus the memlat partially accesses pages from fast and slow memory. At t=21s, when all of the memory has been moved to fast memory, the latency reaches a stable value. These results show the feasibility and effectiveness of our software-controlled approach for managing heterogeneous memory resources.

3.6 Related Work

Substantial prior work has proposed the use of heterogeneous processors to improve the energy efficiency of multicore platforms [23, 42, 55]. Researchers have developed appropriate scheduling algorithms to efficiently run applications on heterogeneous cores [23, 54]. However, the previous work has focused only on cores within the CPU, ignoring the uncore part which accounts for a large fraction of die resources in modern CPUs. Thus, we focus on the uncore subsystem and highlight the significance of its power in total SoC power and analyze its impact on the energy efficiency of several

real-world client applications.

In addition, arguments have been made in favor of low-powered cores for the design of datacenters [6, 48] as well as high-performance brawny cores for other applications [10, 59]. The same argument regarding the impact of uncore on the energy efficiency of cores applies to these systems as well. Further, the cost of uncore resources in many-core processors has been analytically modeled and analyzed [68], however, this work offers an experimental evaluation in this regard.

Concerning memory management in virtualized systems, the VMware ESX server uses a sampling approach to detect working set sizes and manage allocation of system memory among virtual machines using shares [102]. Similarly, Geiger explores mechanisms to monitor the virtual MMU and storage hardware of a VM to provide meaningful information about the usage of buffer cache and virtual memory subsystems [51], while Hypervisor-exclusive cache uses a ghost buffer based approach to predict page miss rates for virtual machines [70]. In comparison, our work uses page-table access bits to detect not only working set size of virtual machines, but also provides 'hotness' information of each page to guide page placement.

Several architectural solutions have also been proposed for tracking memory access patterns and page placement strategies for hybrid memory systems containing traditional DRAM and other memory technologies such as non-volatile memories [21, 88]. Similarly, hardware approaches for managing DRAM caches have also been investigated [49, 86]. Further, efforts have been made to investigate page replacement policies in the context of disaggregated memory platforms, allowing a large pool of memory to be shared by multiple servers [65]. In comparison, our work focuses on system software control on memory management for more efficient utilization of the stacked DRAM. Techniques using sophisticated LRU heuristics for balancing memory across several virtual machines have also been proposed [108]. This work is complementary to our work as similar policies can be used with our approach.

3.7 Summary

Going beyond CPU cores, this chapter investigates uncore and memory subsystem in the context of platform heterogeneity. Specifically, it analyzes the impact of uncore power on the energy-efficiency of heterogeneous multicore processors for client devices. Using a diverse mix of emerging client applications and an experimental heterogeneous platform, we highlight the growing importance of uncore power with respect to total platform power consumption, thereby motivating the need for uncore-awareness and a scalable uncore design for energy-efficient execution on heterogeneous multicore platforms. Further, it considers a software-managed approach for heterogeneous memory resources that consist of a combination of fast 3D die-stacked DRAM and off-chip DRAM. We believe that such stacked DRAM should be managed by software rather than by hardware (hardware managed cache) for flexible management. To this end, we propose and evaluate mechanisms for tracking the memory behavior of virtual machines and managing memory mappings, in a guest-transparent manner. We conduct basic research and evaluation on an emulated heterogeneous memory platform. Preliminary results show the effect of memory heterogeneity on various workloads and our ability to track guest memory access patterns and improve performance by managing how stacked DRAM is used by applications.

CHAPTER IV

HETEROMATES: PROVIDING HIGH DYNAMIC RANGE ON MOBILE PLATFORMS

4.1 Introduction

The ubiquity of handhelds is causing an unprecedented increase in the range of performance demands imposed on mobile platforms, and at the same time, battery life and energy efficiency remain critical concerns. Yet modern processors are typically designed to meet only one, not both, of these two conflicting goals of performance vs. efficiency. In response, chip vendors have adopted heterogeneous multicore processors (HMPs) as their platforms of choice, which consist of cores with different performance/power characteristics. Examples include Variable SMP from NVIDIA [78] and Big.LITTLE processing from ARM [29, 18]. HMPs make it possible for different applications within a diverse mix of workloads to be run on the 'most appropriate' cores [42, 54, 55, 92]. For example, applications not time critical to the user can be run on low-power small cores, while applications with their outputs visible to the user can be allocated to high-performance big cores.

This chapter presents the *HeteroMates* system, which uses heterogeneous cores to provide a wider *dynamic power range* for client devices, to meet both their high-performance and low-power demands. Specifically, HeteroMates forms execution units from *core groups*, where each group consists of a small number of (e.g., 2-4) heterogeneous cores. Cores within a core group are exposed to the system as multiple *heterogeneity (H) states*, similar to the P-states used for voltage and frequency scaling. An *H-state controller* module performs H-state transitions based upon workload behavior and user-defined policies. Depending on the selected H-state, the workload

is transparently migrated to the appropriate core by a *core switcher*.

H-state abstraction decouples heterogeneity from scheduling such that the scheduler perceives only homogeneous cores. The performance/power differences among cores are transparently handled by a separate H-state driver. H-states can be implemented in hardware, firmware, or software, thereby providing a way to hide heterogeneity from the operating system to support legacy software for wider adoption. Further, core groups allow the system to easily accommodate a variable number of different heterogeneous cores, by adding an H-state for each core. Finally, core groups can also be useful in thermal-constrained scenarios (also known as *dark silicon* [22]) which allow only a fraction of the chip components to be active simultaneously.

HeteroMates is implemented on top of the Linux kernel. Experimental evaluations use a unique, experimental heterogeneous multicore platform comprised of both high and low power cores, along with client applications typically seen in modern end-user devices. Two different usage policies are compared: a performance-driven policy favors high performance for user-facing applications, whereas a power-driven policy favors reduced power consumption and longer battery life. Experimental results demonstrate that by opportunistically utilizing heterogeneous cores, HeteroMates can provide both improved performance and lowered energy consumption for various client applications when compared to homogeneous cores. They also highlight the need for a scalable uncore in order to fully realize the potential gains obtained from the use of heterogeneity.

4.2 Motivation

Users perform a wide variety of tasks on mobile devices, resulting in diverse platform demands. Since their battery capacities are severely restricted due to constraints on size and weight, energy efficiency is critical to their usability. To provide extended

battery life and at the same time, meet the rapidly increasing demands of high performance mobile use cases, a client device must offer a wide *dynamic power range* – it must be able to operate both in high-performance and in power-savings modes. As explained in detail in Section 4.3, heterogeneous cores can be used to extend the dynamic power range offered by homogeneous processor configurations. In that context, this section presents examples of client workloads and the usage patterns of client devices that motivate the need for a wide dynamic power range and discusses opportunities for exploiting heterogeneous cores.

4.2.1 Client Workloads

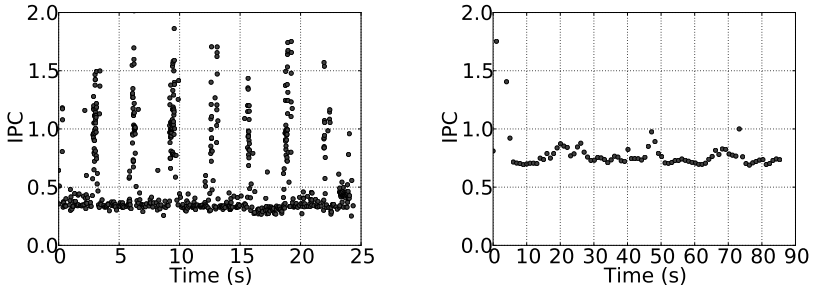
Client applications exhibit highly diverse behavior in their processor usage and performance requirements. These applications can be categorized based on their behavior as described below.

4.2.1.1 Intermittent Workloads

Client devices like cell phones and tablets are typically powered-on for long periods of time, but often perform their heavy processing in short bursts. Web-browsing is an example of such usage, and workloads browse and palbum in Table 7 belong to this category. A timeline trace of IPC (instructions-per-cycle) for the browse workload is shown in Figure 23(a). Idle periods are marked by low IPC periods, while page-loads correspond to spikes in the graph. Since page-loads generate high IPC activity, a big core can be used for rendering the pages and improving page-load performance, while resorting to a small core during low activity periods to conserve power.

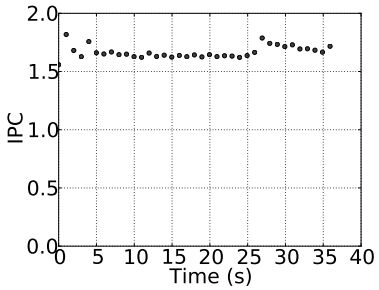
4.2.1.2 Sustained Workloads

These differ from intermittent workloads in that their behavior is uniform over a longer duration. They can be further classified into two sub-categories: sustained-high and sustained-low.

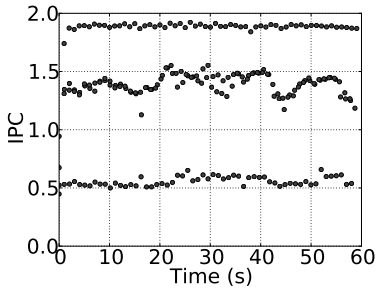


(a) Intermittent (browse)

(b) Sustained-low (openarena)



(c) Sustained-high (x264)



(d) Multi-threaded (mytube)

Figure 23: Diverse client workload profiles (IPC vs. Time)

Sustained-low: Client applications like gaming and media playback typically run for a long duration (a few minutes to hours). Moreover, the wide adoption of accelerators allows them to offload significant portions of their computation to accelerators. Figure 23(b) shows the IPC trace of the openarena gaming benchmark. As the observed IPC is low for the application, it can be run on a small core without significant degradation in performance and at lower power.

Sustained-high: Mobile devices are also used for compute-intensive tasks such as media encoding, video editing etc. These applications typically have a high IPC (e.g., see x264 encoder in Figure 23(c)), and their performance scales well on a big core. This makes big cores suitable for these applications when they require high performance, e.g., when they are user-facing, while a small core may provide higher energy-efficiency when they run in background mode (e.g, virus-scan).

4.2.1.3 Multi-threaded Workloads

With increasing numbers of cores on mobile devices, parallelization of client applications is key to further performance enhancement. Such multi-threaded applications also present opportunities for exploiting heterogeneity. 7zip, gmagick, and eclipse are examples of parallel applications. The mytube workload also uses multiple threads for audio, video decoding, and rendering, for instance. Since such threads differ in behavior and needs, their performance will be affected by how they are mapped to different heterogeneous cores. For example, Figure 23(d) shows that various threads within the mytube workload differ significantly in their IPC, which can be leveraged by task mapping and scheduling methods.

4.2.2 Client Devices

4.2.2.1 Mobility Constraints

Mobility means that client devices will either be powered via wall-power or by battery. Wall-power usage does not impose energy constraints, so that big cores can provide desired high levels of performance. During battery-driven operation, however, a user may be willing to accept lower performance at the benefit of higher battery life. Low-powered energy-efficient small cores may be more suitable under such conditions.

4.2.2.2 Thermal Constraints

Client devices like cell phones and tablets rely on natural cooling. Therefore, these devices are quite sensitive to platform thermal constraints that impose limits on the extent to which it is possible to use power-hungry big cores for sustained periods of time. A small core can be used for moving the execution away from a big core when thermal constraints are violated.

4.3 Dynamic Power Range

This section describes the use of heterogeneous cores to enable a wide dynamic power range, and the role of the uncore subsystem in achieving the same.

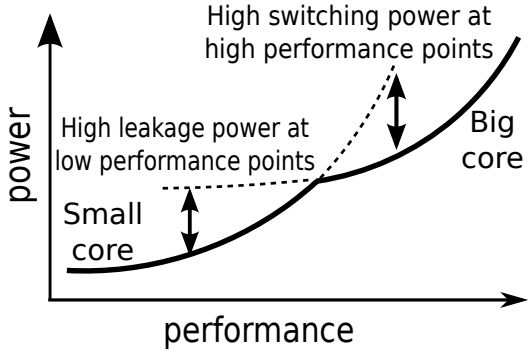


Figure 24: Using a heterogeneous processor provides a wide dynamic power range.

Modern processors are typically designed to satisfy only one of the two conflicting requirements: high-performance and energy-efficiency. Current low-power cores (e.g., Intel's Atom processor) are energy efficient, but their performance is limited. More powerful big cores like Intel Core® processors provide high performance, but at the cost of higher levels of power consumption. The technological reason for this is the fact that the power consumption of a processor core consists of static (leakage) power and dynamic (switching) power. During high activity periods, the total power consumption of the device is dominated by dynamic power consumption, while during low activity periods, leakage power becomes a significant component of the total power consumption. Current high performance cores are built from transistors on fast process technologies that have high leakage power and very fast switching times [78]. Such big cores, therefore, consume high leakage power under idle or near-idle conditions, but can provide high performance without significant increase in dynamic power, as shown in Figure 24. Conversely, low power small cores are built from low power process technologies with low leakage power but slower switching times [78]. Such processors consume small amounts of leakage power, but significantly increase

dynamic power consumption to provide a high-performance mode (see Figure 24).

The intuitive outcome is that by using both types of cores, a single platform can be optimized for both high performance and low power consumption. The objective of such a system would be to always use its most efficient cores for the tasks at hand (shown by the solid line in Figure 24). Such a heterogeneous platform exhibits a higher power-performance range than individual big or small cores. This chapter explores whether and to what extent the hardware-based arguments for heterogeneity stated above lead to realistically achievable gains on client devices.

4.4 HeteroMates Design

HeteroMates enables a wide dynamic power range using heterogeneous cores. This section describes its key components and concepts.

4.4.1 Core Groups

A heterogeneous *core group* is a collection of a small number of (e.g., 2-4) heterogeneous cores that are grouped together to form a single execution unit. For example, Figure 25 shows a core group consisting of three heterogeneous cores: a big (B), a small (S), and a tiny (T) core. The core group appears as a single execution unit with multiple performance/power levels. Depending on application behavior and user-defined policies, an appropriate core is dynamically chosen to run the user task in question, by transparently moving the task's execution to that core, and by placing the other inactive cores into a low power idle state to conserve power. For example, the tiny core can be used for background tasks like email update checks, the small core for normal user operation, and the big core is reserved for performance-critical tasks.

Different cores within a core group are exposed using *heterogeneity-states* (H-states), an interface similar to the P-state (performance-states) interface defined by the ACPI standard and used by operating systems to scale the frequency and voltage

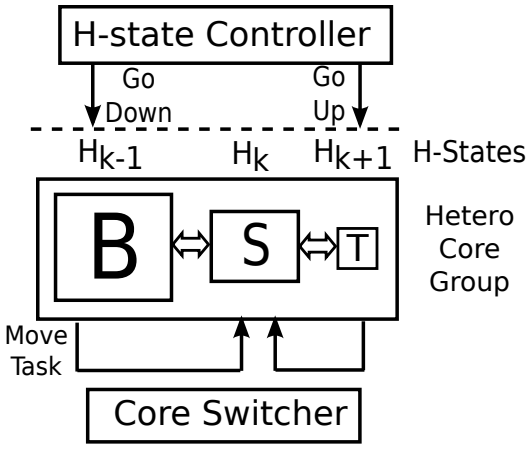


Figure 25: A core groups consisting of three heterogeneous cores: a big (B), a small (S), and a tiny (T) core exposed as three H-states.

of processors. Higher P-state numbers represent slower processor speeds. Thus, P0 is the highest-performance state, with P1 to Pn being successively lower-performance states. Similarly, an H-state is assigned to each type of core in the core group. A high-performance big core corresponds to a lower numbered H-state, while a low-power small core corresponds to a higher-numbered H-state. Thus, a core group exposes a set of H-states ($H_0 \ldots H_n$) which are controlled by an *H-state controller* module. Depending on the state transition logic and the resultant H-state, a *core switcher* transparently migrates the execution to the appropriate core. In this manner, applications perceive only homogeneous cores with larger dynamic power range than any of the individual cores.

The design of HeteroMates offers multiple advantages. First, H-state interface decouples heterogeneity from scheduling such that the scheduler need not deal with performance/power differences among cores. Instead, a separate H-state driver handles this transparently to the scheduler. Second, H-states can be implemented either in hardware, firmware, operating system, or even hypervisors, allowing a broader applicability. As an architectural solution, it provides a way to completely hide heterogeneity from the operating system, which is critical to support legacy software and

applications. Further, core groups provide a unified mechanism to easily accommodate a variable number of heterogeneous cores by adding an H-state for each type of core. Finally, core groups can also be useful when TDP (thermal-design-point) limits may constrain the number of cores that can be active simultaneously. As transistor density on modern processors keep increasing, such TDP limits are proving to be a critical design constraint in the form of *dark silicon* [22].

4.4.2 H-state Controller

H-states on a core group are controlled by the H-state controller, in a manner similar to frequency scaling operations performed by a CPU governor. A CPU governor is a kernel module that changes core P-states based on a policy. In a similar manner, the H-state controller performs H-state scaling operations. However, instead of changing voltage and frequency as in the case of P-states, a change in H-state causes the execution to move to a different core. The functions of the H-state controller and of the traditional P-state governor complement each other. For example, Figure 26 shows the combined P-state and H-state transition diagram for a two-core heterogeneous core group. Here, H_k corresponds to the small core, and H_{k-1} corresponds to the big core. P-state changes within a core are performed by the P-state governor, while cross-core migrations are governed by the H-state controller.

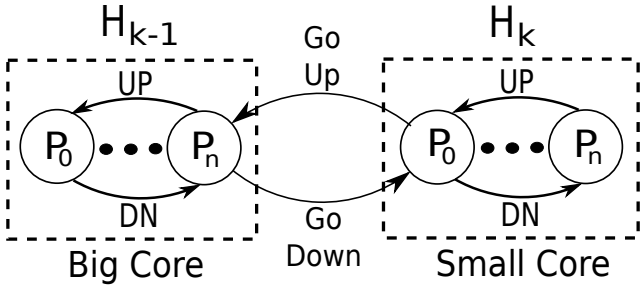


Figure 26: H-state and P-state transition state machines. H-state determine the core for execution, while P-states determine the frequency on that core.

CPU governors available in current operating systems (e.g., the ondemand governor in Linux [82]) dynamically change CPU frequency in response to CPU load (utilization). However, CPU load alone is not sufficient to drive H-state scaling operations, which also require determining whether a bigger or smaller core is more suitable for execution. Previous work on heterogeneous processor scheduling [54, 55, 92] has identified application IPC (instructions-per-cycle) as a key metric to select the right core for execution. Therefore, HeteroMates uses a combination of CPU load and application IPC to form the H-state transition logic shown in Figure 27.

The intuition behind the scaling algorithm can be explained as follows. An application with high CPU load but low IPC is likely to perform equally well on both big and small cores due to its low IPC requirements, which can easily be met on a small core. Applications with high IPC but small CPU load under-utilize the big core. Moving such applications to a smaller core results in higher utilization of the small core, but without a significant penalty in application performance. When both of these conditions are violated, the application is likely to gain performance by executing on a bigger core.

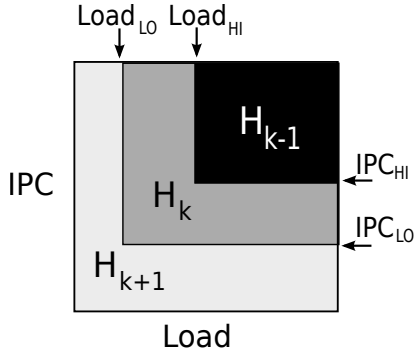


Figure 27: H-state scaling operations in response to application IPC and CPU load.

The H-state controller monitors application IPC and CPU load at periodic intervals and compares them with pre-defined thresholds to determine the resultant state. If both the IPC and load are above thresholds IPC_{HI} and $Load_{HI}$ respectively, the

core group is scaled up, i.e, moved to a higher-performance or lower numbered state. If either IPC or load are lower than thresholds IPC_{LO} and $Load_{LO}$, the H-state is scaled down to a lower-performance state. For values in between these thresholds, no H-state change is performed. These thresholds are defined for each type of core in the system. By setting different values for these thresholds, different policies can be enforced. For example, low values of thresholds force the execution to big cores, and thus prefer performance over power. Similarly, a policy having thresholds with high values picks smaller cores more often.

An H-state change operation causes the execution to switch to a different core. This switching overhead could be substantial due to migration latency and loss of private cached data if such changes are very frequent. In response, we use *history counters* to dampen core switching frequency. A switch is performed only after a certain number of consecutive identical H-state change requests are received. The history counter is a simple integer counter associated with each core group, which is incremented whenever consecutive intervals generate the same requests and reset otherwise.

4.4.3 Uncore-aware Operation

The energy efficiency of a platform is not only determined by the type of core used for execution, but also by the power consumption of the shared uncore subsystem. Workloads for which execution on a bigger core provides both higher performance and better energy efficiency due to improved performance scaling, should always be run on big cores as small core degrades both performance and efficiency. HeteroMates addresses this issue by adding the *energy override* condition in Equation 10 to the heuristic described earlier. If the energy consumption of the current H-state (H_{cur}) is greater than the energy consumption of the next higher state (H_{cur-1}), a scale up operation is performed to move the execution to the bigger core.

$$\textbf{if } \frac{Energy(H_{cur-1})}{Energy(H_{cur})} < 1 \textbf{ then } H_{next} = H_{cur-1} \tag{10}$$

For energy-aware operation, Equation 10 requires the energy consumption of the application to be estimated on a different core (H-state). This task can be divided into two components: processor power prediction and application behavior (e.g., execution time, IPC) prediction. CPU power visibility to the operating system is becoming increasingly important, with multiple CPU vendors providing hardware counters to measure the power of different components on the platform. Further, previous work has developed light-weight models to accurately predict per-core power using existing performance events [27]. Using a similar approach, this work also uses power models, described in Section 3.2.4, to obtain core and uncore power consumption.

In order to understand the impact of a core transition on application behavior, hardware assistance can be provided. For example, HeteroScouts [96] proposes hardware performance counters to predict workload behavior on a remote core (after-transition) from the parameters available on the local core (before-transition). Due to unavailability of such counters in current processors, simple prediction models are developed using experimental data. The following section provides details of the modeling methodology.

4.4.4 Remote Behavior Prediction

To model the relationship between application IPC on a big and a small core in our experimental platform, the client workloads in Table 7 and SPEC CINT2006 benchmarks are executed on both types of cores. Figure 28 plots the obtained $IPC_{scaling}$ data, defined as the ratio of the big core IPC and the small core IPC, as a function of the IPC on the big core. As evident from the figure, a linear curve fits the data

well, with the resultant model given by the equations below.

$$IPC_{scaling} = 0.6 * IPC_{big} + 1.01 \quad (11)$$

$$IPC_{scaling} = 1.31 * IPC_{small} + 0.94 \quad (12)$$

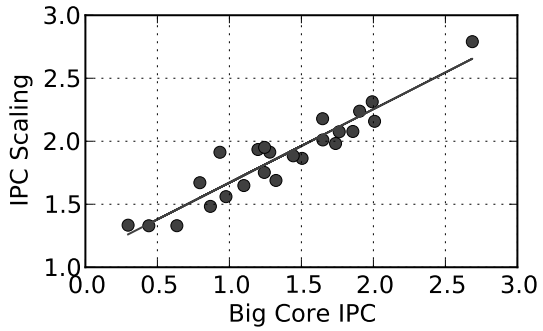


Figure 28: Modeling IPC scaling as a function of IPC

The impact of IPC scaling on the execution time of an application is workload dependent. CPU-bound workloads show a proportional relationship between IPC scaling and execution-time scaling. However, this does not hold true for many client workloads with significant idle phases, e.g., media and graphics workloads. For such workloads, execution time is not affected by the core performance. Instead, a change in core performance translates into change in core idle state residency. These conditions are modeled by applying the scaling function to the product of core active state (R_{active}) residency and execution time (t), as shown in Equation 13. The equation was experimentally verified using the client workloads in Table 7 as majority of the workloads closely follow the modeled relationship. In the online model, sampling interval is substituted for the execution time.

$$(R^{small}_{active} * t_{small}) = IPC_{scaling} * (R^{big}_{active} * t_{big}) \quad (13)$$

Further, the change in core idle residency (R_{idle}) impacts package idle state (U_{idle}) residency in an application dependent manner. Applications for which the package

becomes idle, as soon as the core becomes idle, show a strong correlation between core and package idle states. On the other hand, for multi-threaded applications and graphics-intensive applications, a core's idle state does not necessarily translate to the package idle state since the package can still be busy due to activity in another core or the graphics processor. Such applications show a weak or negligible correlation between core and package idle states. These two scenarios are modeled in Equation 14 where a difference of 20% between U_{idle} and R_{idle} is assumed as an indicator of weak correlation. For such cases, U_{idle} is assumed to be the same irrespective of the type of core used for execution.

$$U_{idle}^{small} = \begin{cases} U_{idle}^{big} & \text{if } U_{idle}^{big} \ll R_{idle}^{big}, \\ R_{idle}^{small} & \text{otherwise} \end{cases} \tag{14}$$

Using the models presented above and the power models described in Section 3.2.4, an application's relative energy consumption on two different H-states can be obtained. These values are used to perform energy override operations as defined earlier by Equation 10.

4.5 Implementation

HeteroMates is implemented for the Linux kernel. The current implementation considers systems involving pairs of heterogeneous cores. H-states are implemented by customizing the P-state tables on each core to expose two P-states corresponding to each core in a pair. H-state changes work in lock-step on both of these cores to avoid conflicting operations. An H-state change causes execution to switch cores instead of performing DVFS. Our current implementation does not consider traditional voltage and frequency scaling. This is because there is substantial previous work on DVFS [72, 87, 94, 104], which can be used to perform P-state scaling in addition to H-state transitions.

The H-state controller is implemented as a kernel module which runs on each

active core as a kernel thread. It periodically (40ms) reads various hardware performance monitoring counters (PMCs), applies models, and performs any H-state changes depending on the policy and thresholds chosen. The overhead of running models is measured to be small (approximately 2% increase in core active and 5% increase in package active residency). The core switcher is implemented in the OS kernel by changing the runqueue pointer for the tasks in the source runqueue to point to the destination runqueue. The overhead of this operation is minimal when runqueue length is not large, which we have observed as being the case for the typical client workloads used in our experiments. We note that similar functionality can be provided by hardware, to further reduce overheads. Also, only active cores are made available for scheduling to the Linux CFS scheduler. Inactive cores are put into an offline mode using a lightweight mechanism. A value of three is used for history counters.

4.6 Experimental Evaluation

4.6.1 Experimental Platform

Evaluations are carried out on the quad-core Intel i7-2600 client processor using proprietary Intel tools to defeature cores and emulate heterogeneity. A block diagram of the emulated platform is shown in Figure 29. Details of this platform along with the power models are described in Chapter 3.

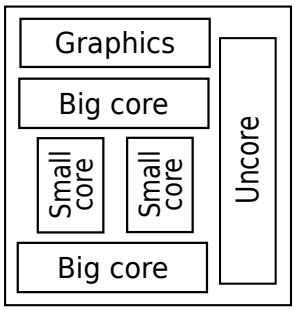


Figure 29: Experimental heterogeneous platform

4.6.2 Workloads

Table 7 provides a summary of client applications used in our analysis which include browsing, gaming, media, etc. and relevant performance metrics.

Table 7: Modern client workloads

Workload	Description	Metric
7zip	a parallelized version of 7zip is used to compress a text file	Time
applaunch	launches and executes a series of graphics-intensive applications	Load time
browse	loads a set of web-pages at 3s interval to emulate user's think time	Load time
canvas	HTML5 benchmark performs browser canvasing tests	FPS
eclipse	Java based benchmark runs performance tests for the Eclipse IDE	Time
gmagick	Image editing application is used to resize a set of images	Time
javascript	Javascript benchmark performs standard browser operations	Load time
lightsmark	renders scenes from a 3D game and measures graphics performance	FPS
mplayer	a H/W accelerated version of mplayer plays an HD movie clip (60s)	FPS
mytube	plays an H.264 streaming video inside the browser for 60s	FPS
openarena	plays a benchmarking demo from a 3D first-person-shooter game	FPS
palbum	photo album application flips through photographs at 0.5s interval	Load time
strike	replays a demo session of a web-based 2D game (60s)	FPS
x264	x264 media encoder is used to encode a media file	Time

4.6.3 Methodology

Two different policies are used, one performance-driven, the other power-driven. This is done by choosing different threshold values, obtained after experimenting with several combinations of thresholds. Table 8 summarizes the various thresholds used to cater to these policies. For a paired-core system, small cores can only perform scale up operations and not scale down, therefore, only HI thresholds are relevant for small cores. Similarly, only LO thresholds are relevant for the big cores. The first performance-driven policy favors performance over power by using big cores for execution in an aggressive manner. This is achieved by choosing smaller thresholds in the table. The power-driven policy, on the other hand, focuses on power by choosing bigger thresholds and forcing the execution to small cores more often. The evaluation is carried out by comparing the performance and energy consumption of the performance-driven policy with only big core execution and of the power-driven policy

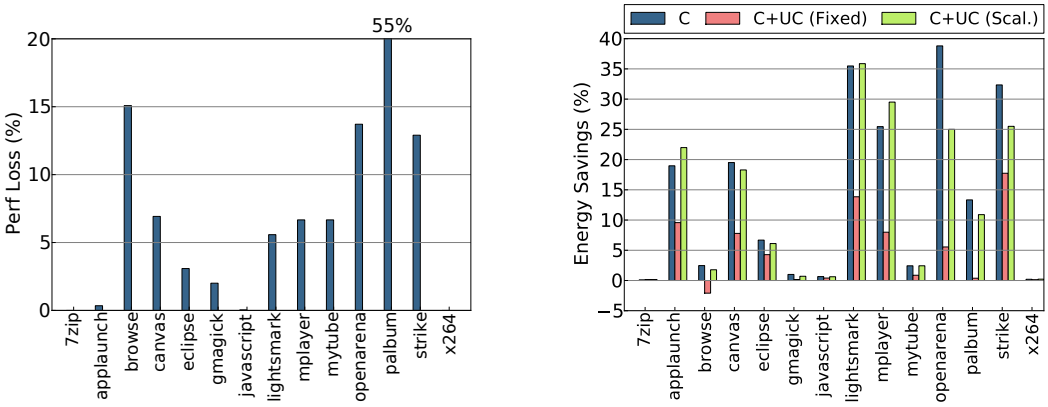
with just small core execution. These two comparison points provide us a perspective of the advantage of using heterogeneous cores over homogeneous configurations.

Table 8: Thresholds for performance- and power-driven policies

	Small Core		Big Core	
	IPC_{HI}	$Load_{HI}$	IPC_{LO}	$Load_{LO}$
Performance-driven	0.5	70%	0.8	40%
Power-driven	0.7	80%	1.25	50%

4.7 Experimental Results

4.7.1 Performance-driven Policy



(a) Performance loss wrt. big cores

(b) Energy savings wrt. big cores

Figure 30: Comparison of performance-driven policy with big core execution

Figure 30 provides results comparing the performance and energy consumption of the performance-driven policy with execution on big cores. Specifically, Figure 30(a) shows performance loss (%) with respect to the maximum performance achievable by using big cores for the entire execution, and Figure 30(b) shows corresponding energy savings by using small cores for partial execution when big core is not energy-efficient. Performance is measured based upon the metrics in Table 7, with inverse of latency as the metric for latency-oriented workloads. As evident from the figures, this

policy is able to achieve performance within 15% of the big core performance for all the workloads except browse and palbum. This high performance loss for these two workloads is due to their bursty nature, i.e., these applications exhibit sudden bursts of high activity during page-rendering. HeteroMates uses history counters to dampen core switching frequency, which requires multiple consecutive state change requests to be received before actually making the change. Due to this reason, these bursty applications observe a short delay before they are moved to the big core which incurs a higher performance degradation. However, the absolute increase in the latency for these applications may not be user-perceivable.

Figure 30(b) shows corresponding energy savings results for three scenarios: core-only savings (C), SoC-wide savings (C+UC) with a fixed uncore, and SoC-wide savings with a scalable uncore. As seen from the figure, it is able to save significant energy for several applications with a small performance degradation. Workload openarena achieves highest gains with 39% core energy savings. However, these savings are strongly affected when the power consumption of the uncore is taken into account. On the other hand, when a scalable uncore is used, these savings increase and become comparable (25%) to core-only energy savings.

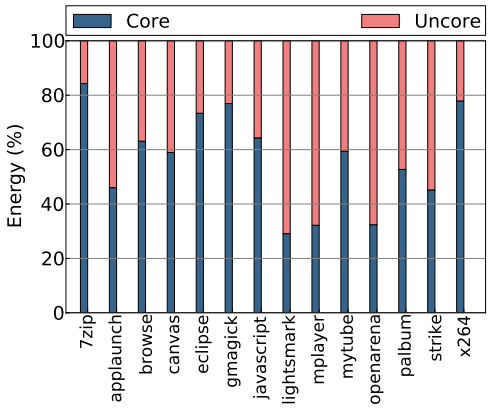
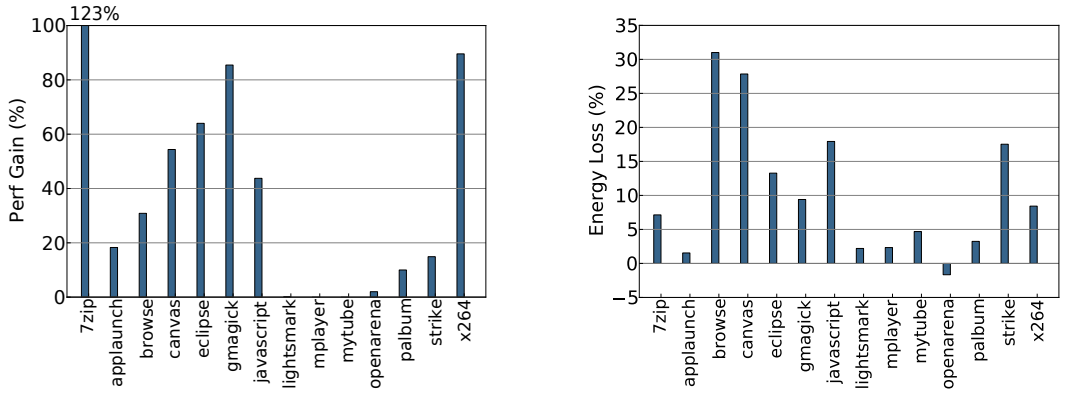


Figure 31: Core and uncore energy distribution

To elaborate on the importance of uncore power in total SoC power, Figure 31

shows the distribution of core and uncore energy consumption for various applications. Core energy component dominates for CPU-intensive applications like 7zip, eclipse, gmagick, and x264, while uncore component is significant for other applications including lightsmark, mplayer, and openarena. These results highlight the growing importance of uncore power in the processor power consumption and motivate the need for a scalable uncore design when seeking to obtain large gains from heterogeneous multicores.

4.7.2 Power-driven Policy



(a) Performance gain wrt. small cores

(b) Energy loss wrt. small cores

Figure 32: Comparison of power-driven policy with small core execution

Results for the power-driven policy are presented in Figure 32, where Figures 32(a) and 32(b) respectively, show performance gain and energy loss (SoC-wide) in comparison to small-core-only execution. As results show, this policy is able to achieve significant performance gains for many applications by selectively using big cores. Further, it is able to do so with only a small to moderate increase in energy consumption. For example, the browse and canvas workloads observe the highest increases in energy consumption of 31% and 28% respectively, while most of the other applications show a smaller increase. However, these two applications also show a 31% and 54% performance gain for the increased energy consumption due to their usage of

big cores. We note that some applications like lightsmark, mplayer, and openarena exhibit negligible performance improvement due to poor scalability.

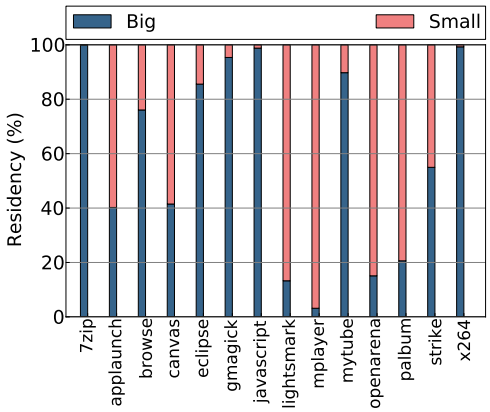
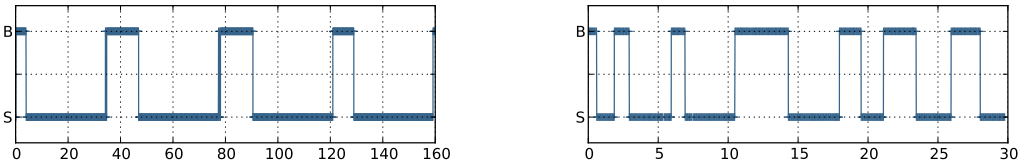


Figure 33: Residency on big and small cores

Results in Figure 33 show the percentage residency on big and small cores for all of the applications. Different applications exhibit different degrees of big and small core usage. For example, applications like 7zip, eclipse, and x264 with good performance scalability spend the majority of their execution on big cores. On the other hand, applications like lightsmark, mplayer, and palbum remain on small cores for a significant portion of their execution time. Other applications like applaunch, canvas, and strike make use of both types of cores during their execution. To illustrate this further, the big and small core usage profiles of the applaunch and strike workloads are shown in Figure 34. The applaunch workload launches and executes a series of graphics-intensive applications. The launch operation is CPU-intensive and performs better on a big-core, while the execution phase is accelerated using the on-die graphics processor and a small core provides comparable performance to the big core at a lower power. Therefore, this workload transits between big and small cores during launch and execution phases (see Figure 34(a)). Similarly, Figure 34(b) shows the execution profile for the strike gaming workload. This workload exhibit several phases with high activity (e.g., bots shooting) when big cores are used and phases with low

activity (e.g., bots aiming and moving) when small cores may suffice. In this manner, the appropriate core is used depending on the activity.



(a) applaunch (b) strike

Figure 34: Big (B) and small (S) core usage profile (x-axis: time(s))

4.8 Related Work

Heterogenous chip multiprocessors (CMPs) have been proposed to achieve higher energy-efficiency than symmetric multicore processors. Using a mix of big and small cores, different phases within an application can be mapped to the core which can run them most efficiently [55, 56]. Similarly, heterogeneous cores can be used to improve the performance of parallel applications by speeding up sequential phases within the application [42, 97]. Researchers have also developed appropriate scheduling algorithms to efficiently run applications on heterogeneous cores [33, 99, 54, 58, 63, 64, 92]. In addition, previous work has proposed runtime mechanisms to leverage different cores in mobile devices [44, 66] and virtualize heterogeneous multicore systems [52, 57].

There is also substantial previous work on dynamic voltage and frequency scaling (DVFS). Several techniques have been developed to dynamically select appropriate voltage and frequency for maximum efficiency [72, 87, 94, 104]. However, others have questioned the effectiveness of DVFS on modern processors [6].

In comparison, our work targets client devices where energy is a premium resource, with diverse application behavior and performance metrics. In that context, we extend the existing DVFS mechanisms to go beyond homogeneous cores and support core heterogeneity to enable a wide dynamic power range on these client devices. In

addition, we highlight the significance of uncore power in total SoC power and motivate the need for a scalable uncore for exploiting maximum gains from heterogeneous CMPs.

4.9 Summary

This chapter presents the *HeteroMates* solution, which utilizes heterogeneous multicores in order to provide a wide dynamic power range on client devices. It proposes *core groups*, an abstraction that groups together a small number of heterogeneous cores to form a single execution unit. Cores within a core group are exposed as multiple heterogeneity (H) states. H-state transitions are governed by an H-state controller, while a core switcher transparently migrates the task to the appropriate core depending on the resultant H-state. Using a diverse mix of client applications and an experimental heterogeneous platform, we show that heterogeneous CMPs can be used to provide a superior solution for client devices. We also highlight the growing importance of uncore power in total SoC power consumption and the need for a scalable uncore design to completely realize the intended gains.

CHAPTER V

HETEROVISOR: ELASTIC RESOURCE SCALING ON HETEROGENEOUS CLOUD PLATFORMS

5.1 Introduction

Elasticity is a key feature of cloud infrastructures, enabling 'on-demand' scaling of resources used by an application to match its requirements and user preferences [24]. Resource scaling techniques used by modern cloud platforms like Amazon's Elastic Compute Cloud (EC2) [5] and Google's Compute Engine [28], however, are coarse-grained, both in space and in time. These mechanisms involving use of different types of virtual machines (VMs) have substantial consequent monetary implications for customers due to the costs they incur for such fixed instances and the frequencies at which such heavy-weight scaling operations can be performed. Customers could implement their VM-internal solutions to this problem, but a *truly elastic* execution environment should provide 'fine-grained' scaling capabilities to *frequently* adjust the resource allocation of applications in an *incremental* manner. Given the competition among cloud providers for cheaper/better services, fine-grained resource management may prove to be a compelling feature of future cloud platforms [1, 24].

A clearly emerging trend shaping future cloud computing environments is the presence of heterogeneity in multiple subsystems of server platforms, including processors, memories, and storage. Processors can be heterogeneous in the levels of performance offered [23, 30, 55], like the big/little cores commonly found in today's client systems [18, 29, 78]. Similarly, memory heterogeneity could arise from the joint use of high speed 3D die-stacked memory, slower off-chip DRAM, and non-volatile memory [21, 69, 88]. Such heterogeneity presents known challenges to server system

management, but in this work, we view it as an opportunity to improve future systems' scaling capabilities, by making it possible for execution context to move among heterogeneous components via dynamic 'spill up' and 'spill down' operations, in a fine-grained manner and driven by application needs. A spill-up operation results in improved performance for the application by making use of the high-performance components, while a spill-down reduces resource usage by using slower resources.

HeteroVisor virtual machine monitor presented in this work hides the underlying complexity associated with platform heterogeneity from cloud applications yet provides them with a highly elastic execution environment. It offers a simple abstraction of a homogeneous scalable virtual resource to applications, which is then mapped appropriately to underlying heterogeneous platform components. More specifically, it exports the abstraction of *Elasticity (E) states* which provide guest VMs with a channel for dynamically expressing their resource requirements. The E-state interface is inspired by the already existing P-state interface [82] used by modern operating systems to scale the frequency and voltage of processors. In addition to processors, HeteroVisor incorporates multiple heterogeneous resources under a unified abstraction of E-states. By triggering transitions on these E-states, system- or application-level modules called *Elasticity drivers* (like the Linux CPU governor in the case of P-states) provide hints to the hypervisor on managing the resources assigned to each VM. HeteroVisor incorporates these hints to dynamically manage underlying heterogeneous resources, thus, making it possible to vary heterogeneous resource allocations, on a per-VM basis.

While the E-state abstraction is generic to be applicable to various resources, we specifically explore heterogeneous processors in this work. With heterogeneous CPUs, E-states are used to provide the abstraction of a scalable virtual CPU (vCPU) to applications which operate at a requested elastic speed, different than those of the cores underneath. This is achieved by appropriate mapping of the vCPUs to

heterogeneous cores and imposing *usage caps* on vCPUs, thus, limiting their usage of the physical processor. Similarly with heterogeneous memories, E-states can provide an abstraction of performance-scalable memory, where multiple performance levels are obtained by adjusting the allocation of fast vs. slower memory resources.

HeteroVisor is implemented in the Xen hypervisor, along with a simple E-state driver for the guest virtual machines. In order to evaluate its utility and overheads with actual applications and workloads, not relying on architectural simulators, CPU throttling is used to emulate processor heterogeneity. With workloads that use traces from Google cluster data [40], experimental evaluations show that by exploiting heterogeneity in an unobtrusive way, HeteroVisor makes it possible to achieve on-demand performance boosts and cost savings for cloud applications with diverse resource requirements. Specifically, the scaling mechanisms provide upto 2.3x improved quality-of-service (QoS), while also reducing average resource usage and thus cost for these workloads. Further, two usage policies are compared, showing that different trade-offs between QoS and cost can be achieved using the proposed mechanisms.

5.2 Elasticity using Heterogeneity

5.2.1 Elasticity in Clouds

Elasticity, i.e., ability to scale resources on-demand to minimize cost is one of the most attractive feature of the cloud computing environments. The resources can be scaled either in 'scale out' or 'scale up' manner [24]. A scale-out operation implies dynamically varying the number of VM instances used by an application. Commercial cloud services like Amazon EC2 AutoScale [3] rely on such server-level scaling where application resources can be increased in the form of additional VMs of fixed instance types. Further, these instances can only be rented in the order of several minutes to a full hour and are charged for the whole instance even if partially used [4]. Thus, scaling out is a rather heavy-weight and coarse-grained operation having high cost

implications for the end-user (see Table 9). Server scaling also does not provide a way to improve the performance of existing resources owned by an application, e.g., moving from a VM instance to another type of instance with different resources requires a VM restart in Amazon EC2. Previous work has highlighted the need for long running instances in datacenters for predictable performance [13].

Table 9: Mechanisms for elastic resource scaling in clouds

	Scale out	**Scale up**
Scaling Method	VM Instances	Resource Shares
Resource Granularity	Coarse	Fine
Time Granularity	Slow	Fast
Software Changes	High	Minimal

Thus, light-weight, fine-grained resource scaling methods can be vital for cloud systems which can be enabled using 'scale up' operations by adjusting the share of platform resources owned by a VM (refer Table 9). Fine-grained elasticity enables a user to start a VM with a basic configuration and dynamically build its platform configuration as needed. Such scaling techniques may be sufficient and in fact, better suited for many users than VM-level scaling methods, e.g., when a VM goes through sudden short bursts requiring higher allocation of resources. Thus, a user can simply request the resources it needs and rent durations can also be much shorter (on the order of seconds) to reduce costs. Another advantage to this approach is that it can be transparent to the VM and applications, not requiring sophisticated software changes to deal with varying resources. Several techniques have been proposed in literature to enable fine-grained resource management in clouds [13, 83, 81, 93]. However, each of these focus on a specific approach, without a unifying mechanism to enable their adoption.

5.2.2 Exploiting Heterogeneity

This work exploits platform heterogeneity to enhance the elastic resource scaling capabilities for cloud systems using 'spill' operations, i.e., changing the allocation of VM resources in heterogeneous components. Consider a resource such as memory with components of three different performance characteristics, i.e., die-stacked DRAM as the fast resource, off-chip DRAM as the medium-performance resource, and non-volatile memory as the slow resource. Each of these components support a different performance range. However, the performance of memory subsystem can be adjusted by varying the allocation of memory to an application in these three levels. As a higher share of the VM resources are allocated in the faster component (e.g., moving application data to on-chip memory from off-chip DRAM), its performance increases. This is denoted as a 'spill up' operation as shown in Figure 35. Similarly, by spilling down the application resources (e.g., ballooning out VM pages to persistent memory), its performance can be dynamically adjusted. In this manner, it provides the abstraction of a scalable memory using spill operations over heterogeneous components.

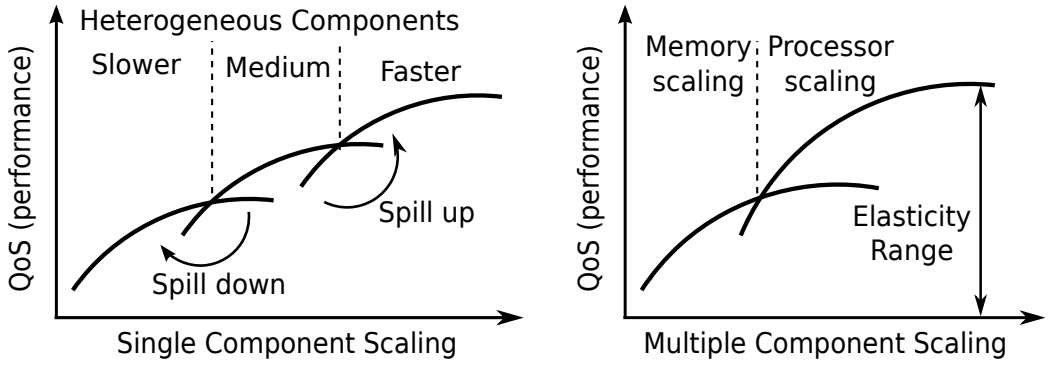


Figure 35: Using heterogeneity to enable resource scaling

Further, these scaling mechanisms can be applied to multiple platform resources such as processor, memory, or storage components to provide an overall extended elasticity range to the applications as shown in Figure 35. In this manner, their use for the joint management of heterogeneous processors and memories, make it possible, for

instance, to use a slow processor with rapidly accessible memory for a data-intensive code, while a CPU-intensive code with good cache behavior may be well-served with slower memory components. The spill operations, however, may govern diversity in these components. The processor scaling is achieved by appropriate scheduling of vCPUs to heterogeneous cores and capping their usage of these cores to achieve a target speed. Memory spill operations, on the other hand, are not as easily managed as those for processors since a VM's performance is sensitive to its memory access patterns which are not directly visible to the hypervisor. The next section describes various mechanisms that are incorporated into HeteroVisor to implement these scaling mechanisms.

5.3 Design

Using heterogeneous platform resources, HeteroVisor provides fine-grained elasticity for cloud platforms. To incorporate heterogeneity into the scaling methods, there are several principles that we follow in our design.

- Adhering to the philosophy that cloud platforms should sell resources and not performance, VMs should explicitly request resources from the cloud provider. This design requiring application VMs to specify their resource requirements is common to IaaS platforms where users select different types of VM instances.

- Typically special software support is required for managing heterogeneity. Diversity across vendors and rapidly changing hardware make it difficult for operating systems to incorporate explicit mechanisms for managing these components. Thus, the complexity of managing heterogeneous components should be hidden from the users.

- The resource scaling interface should be generic and extensible to allow its use on various platforms with different heterogeneous configurations. It should allow

scaling of resources in incremental ways and should be light-weight in nature for frequent reconfiguration. It should also work with multiple resources.

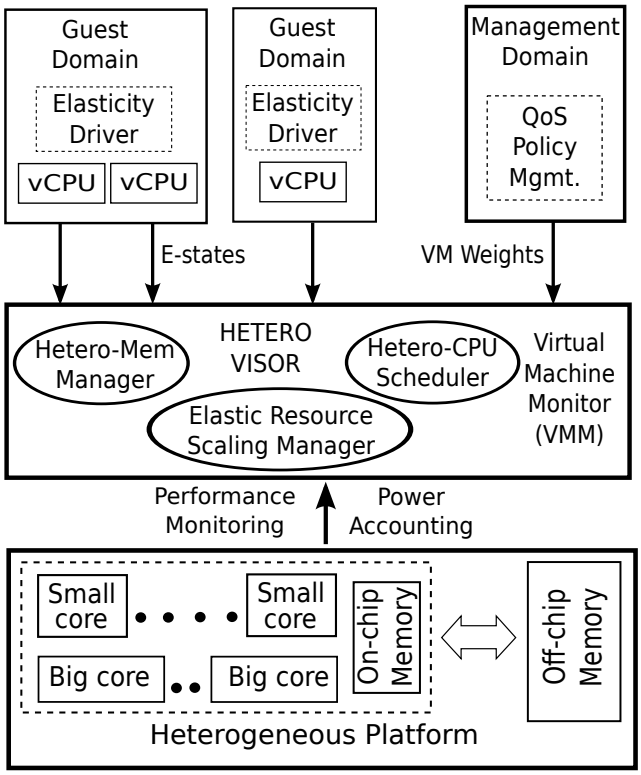


Figure 36: System architecture for HeteroVisor

Figure 36 shows the overall architecture of the system, including various components and their interactions. The underlying platform consists of heterogeneous resources such as CPU, memory and provides capabilities for performance and power monitoring. The platform is shared by multiple guest virtual machines where each VM communicates with the hypervisor about its resource requirements through the *elasticity (E) state* interface (detailed in Section 5.3.1). These E-states are controlled by an *E-state driver* module, allowing the guest VM to communicate its changing resource usage as state transitions. The hypervisor contains various heterogeneity-aware resource managers such as CPU scheduler, memory manager, and a scaling manager. The scaling manager is the higher-level resource allocator which takes into

account various E-state inputs from the VMs and policy constraints from the management domain to partition various resources across all the VMs, while CPU and memory manager own the responsibility of enforcing these partitions as dictated by the scaling manager and also managing them efficiently for each VM. Each of these components are described in detail in the following sections.

5.3.1 Elasticity States

To enable fine-grained resource management, a VM should be allowed to express its resource requirements. Inspired by the P-state (performance-state) interface [82] defined by the ACPI standard and used by operating systems to request hardware to control CPU voltage and frequency (DVFS), we propose E-state (elasticity-state) abstraction shown in Figure 37. E-states are hints provides by a VM to the hypervisor to guide its resource allocation which are controlled by a VM-specific E-state driver (like CPU governor in the case of P-states).

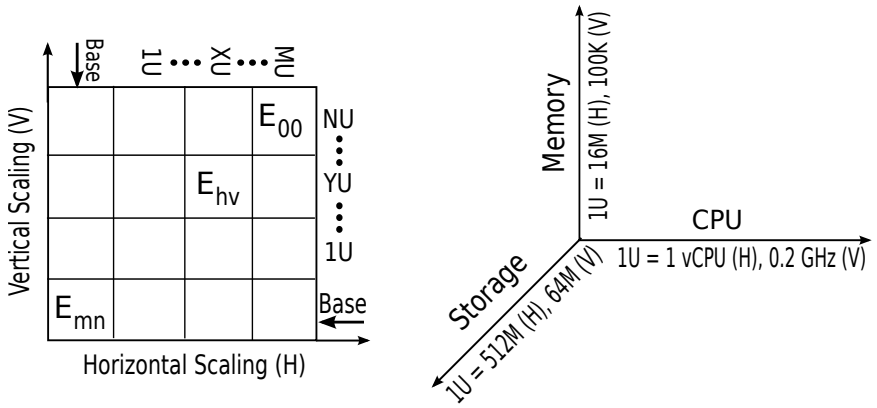


Figure 37: Elasticity state abstraction for resource scaling

E-state interface defines multiple states where each state (e.g., E_{hv}) corresponds to a different resource configuration. E-states are arranged along two dimensions, corresponding to horizontal and vertical scaling through a single E-state interface. Horizontal scaling allows platform scaling, i.e., adding virtual resources to the application using hot-plug based mechanisms and vertical scaling implies boosting the

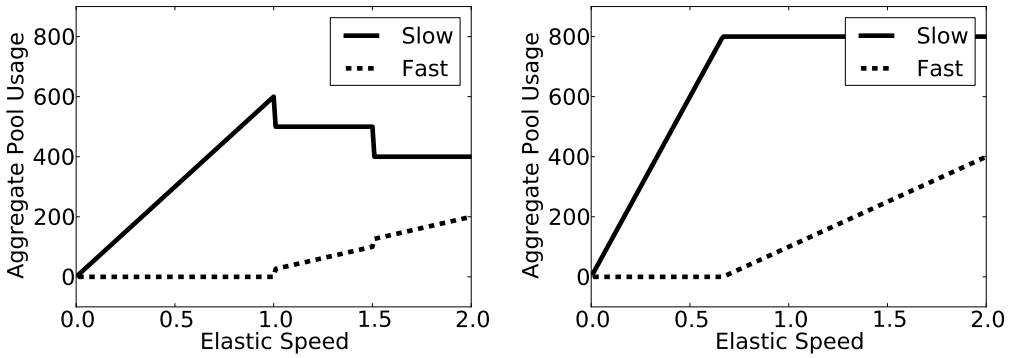
performance of existing platform resources. It should be noted that both horizontal and vertical scaling are scale up methods, separate from the scale out methods which vary the number of VM instances. As in the case of P-states, a higher numbered E-state (E_{mn}) represents lower resource configuration, while a lower numbered E-state (E_{00}) implies high-performance mode. Further, these states are specific for each scalable component such as processor, memory, and storage subsystems. A change in the E-state implies a request to change the allocation of resources to that VM by a certain number of resource units (U). For the CPU component, a horizontal E-state operation changes the number of vCPUs, while the vertical scaling adjusts its speed in units of CPU frequency. Similarly, for the memory subsystem, horizontal scaling is achieved by changing its overall memory assignment while vertical scaling adjusts its allocation in fast/slow memory (at page granularity). This work focuses on vertical scaling dimension in the presence of heterogeneous resources.

5.3.2 Elasticity Manager

Heterogeneous resources consisting of components with different speeds can be used to provide a virtual scalable resource. In this section, we describe in detail how this can be achieved for heterogeneous cores, however, this notion can be extended to other resources as well. Further, the formulation is presented for two different types of cores which can be generalized to multiple levels as well.

Given a platform configuration with heterogeneous cores, the objective of the elasticity manager is to provide homogeneous virtual cores with a desired speed, different from the speed of the cores underneath. This can be achieved by appropriate scheduling of the vCPUs on these heterogeneous cores and assigning a *usage cap* to each vCPU, limiting its usage of the physical resources. For such scaling, all the vCPUs are scheduled on slow cores initially with fast cores kept idle. As vCPUs are scaled up, the slow core cap of the vCPUs is increased to meet the desired scaling

speed. When slow cores cycles are saturated, further scaling results in vCPUs being scheduled to fast cores, providing higher scaled speeds than that are possible with slow cores only.



(a) 6 vCPU VM (b) 12 vCPU VM

Figure 38: Models for vCPU scaling using heterogeneity

The expressions for the corresponding usage caps of various cores for achieving a given effective processing speed can be obtained by formulating it as a linear optimization problem, solvable using standard solvers. Since the allocation problem needs to be computed in kernel-space, instead of relying on external solvers, we obtain a closed-form solution for a special case of two types of cores, slow (s) and fast (f), where slow cores have lower ownership-cost than fast cores, i.e., prioritizing allocation to slow cores before using fast cores. The formulation and derivation of the expressions are presented in Appendix B. Figure 38 plots the resultant equations for a configuration with 8 slow cores with 1x speed and 4 fast cores with 4x speed. The figure shows the aggregate slow and fast pool usage for a VM (total percentage utilization cap assigned collectively to all the vCPUs) as we vary the elastic core speed. Two different VM configurations are plotted by varying the number of vCPUs in the VM (v_n) to 6 and 12.

In both the cases, slow pool usage first increases linearly as we increase the elastic core speed (solid lines). Once slow cores are saturated at usage value 600 for 6

vCPUs and at 800 for 12 vCPUs (constrained by 8 physical slow cores), fast pool usage gradually increases (dotted lines) to obtain the requested elastic scaling. For example, a VM with 12 vCPUs at speed 1U exhibits 800% slow pool utilization (8 slow cores fully utilized) and 100% fast pool usage (1 fast core with speed 4x). We also see jumps in the CPU usage with v_n equal to 6 at speed 1 and 1.5 which happens due to the shift of a slow pool vCPU to the fast pool.

In order to perform elastic scaling, vCPUs are partitioned into two pools, one corresponding to the each type of core, i.e., slow and fast pool. Each vCPU executes within a particular pool and load-balanced among other vCPUs belonging to that pool as shown in Figure 39. Due to this partitioning of vCPUs in pools, there may arise performance imbalance among vCPUs. To deal with this, a rotation is performed periodically among pools to exchange a vCPU, thus, giving every vCPU a chance to run on the fast cores, resulting into balanced performance. Such migrations have very little cost if done infrequently and particularly if cores share a last-level cache.

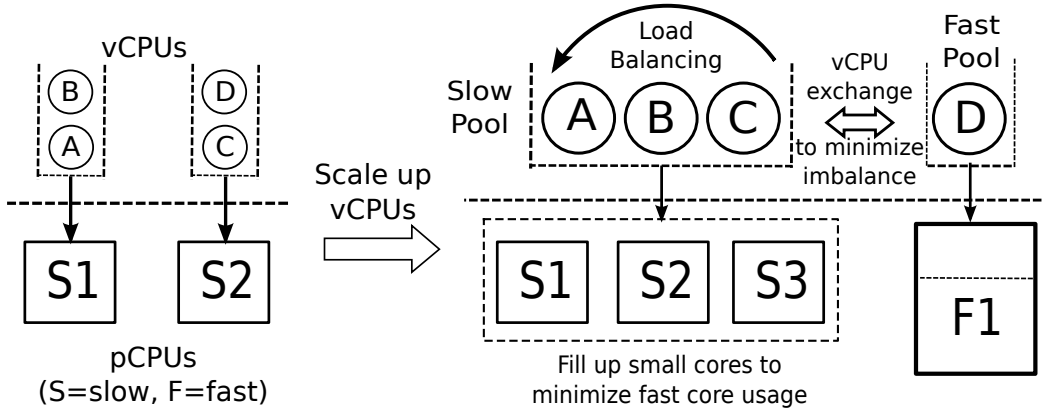


Figure 39: Virtual core scaling using heterogeneous cores

Extending the notion of elasticity to memory subsystem, the spill operations can be performed by migrating pages between different memories. Thus, it provides the interface of a performance-scalable memory over heterogeneous components by

changing a VM's allocation in different memories, i.e., use of fast memory for high-performance E-states and slow memory for slower E-states. Chapter 3 describes management for heterogeneous memories involving fast die-stacked memory and slow off-chip DRAMs. Since the die-stacked memory is small in capacity in comparison to the off-chip DRAM, a subset of pages from application's memory need to be chosen to be placed in the stacked-DRAM. For this purpose, it is important to detect and manage application's 'hot' pages that are critical to its performance. Hot pages can be detected using page-table access-bit history based mechanisms [61] and then actively managed by moving them in and out of fast/slow memories. Such migrations require remapping guest page tables which are hidden from the hypervisor. In order to do this in a guest-transparent way, guest's page tables can be *mirrored* in the hypervisor. A detailed description of the mechanisms involved in these operations is beyond the scope of this thesis.

5.3.3 Elasticity Driver

Elasticity drivers are the guest-specific component of the HeteroVisor stack, allowing guest VM to guide resource allocation by triggering E-state transitions, in a manner similar to the CPU governor which makes P-state (performance-state) changes in the context of DVFS (dynamic voltage and frequency scaling) [82]. Various resource management policies can be implemented by using different implementations of the driver. Thus, an application can choose a specific driver catered to its requirements. Various solutions (e.g., RightScale [91]) are already available which implement resource scaling controllers for various applications. The E-state driver is a step forward in this direction, allowing fine-grained resource management. Traditional VMs with static configurations are supported as well, though with a cost/performance penalty of over/under-provisioning of resources. We have currently implemented a simple reactive heuristic in the E-state driver, however, more sophisticated controllers can be

designed such as prediction based mechanisms [93] that model application behavior to determine the resultant E-state changes.

Our driver implementation uses a combination of utility factor (*util*) and application performance (*qos*) to form the E-state transition logic. The utility factor is analogous to CPU/memory utilization, i.e., the percentage of resources (CPU usage cap or fast memory pages) consumed by a VM against its assigned usage. Similarly, QoS metrics such as response time or response rate can be obtained from the application. Using these two metrics, E-state scaling operations are executed as shown in Algorithm 3. E-state driver defines four thresholds: qos_{hi}, qos_{lo}, $util_{hi}$, and $util_{lo}$. If *qos* is lower than minimum required performance qos_{lo} or utility factor is higher than $util_{hi}$ mark, E-state scale up operation is requested. Scale down logic requires *qos* to be higher than qos_{hi} and *util* to be lower than $util_{lo}$ thresholds.

Algorithm 3: Elasticity-driver scaling heuristic

```
E_last ← E_cur
if util > util_hi OR qos < qos_lo then
    E_next ← E_cur−1 ;                                  // Scale up
else if util < util_lo AND qos > qos_hi then
    E_next ← E_cur+1 ;                                  // Scale down
else
    E_next ← E_cur ;                                    // No change
```

The intuition behind the scaling algorithm can be explained as follows. If the application performance is lower than its SLA, a scale up operation is issued to improve performance. Similarly, if the utility factor is too high which may cause SLA violations, more processing capacity is requested again. On the other hand, if application performance is higher than its desired SLA, a scale down operation can be issued to reduce its resource usage. However, it additionally requires the utility factor to be low so that the scale down operation does not lead to violations after resources are scaled. In order to avoid oscillations due to transitory application behavior, history counters are used to dampen switching frequency. A switch is

requested only after a fixed number of consecutive identical E-state change requests are received. The history counter is a simple integer counter, which is incremented whenever consecutive intervals generate the same requests and reset otherwise.

5.3.4 Discussion

There are few issues that should be mentioned regarding the limitations of our current implementation.

The evaluation in this work considers only single-VM scenarios. The allocation problem among competing VMs can be more challenging [108], requiring priority management and handling over-subscription requests which may need mechanisms for relinquishing resources owned by a VM. The problem becomes more complex when considering multi-resource allocation scenarios [26] where different resources affect the QoS for various applications in an application dependent manner. Similarly, current E-state driver performs elastic scaling along only one resource axis at a time. However, a real system can perform scaling across multiple resources simultaneously, coordinating their usage to optimize application performance/cost. Earlier work can be leveraged for coordinated scaling in such multi-resource grids [20].

Further, we currently consider systems having two levels of CPU heterogeneity only but it can be generalized to more heterogeneous systems. Regarding memory heterogeneity, it may also need to account for NUMA systems along with heterogeneity, i.e., moving pages from slow memory to local vs. remote fast memory. Mechanisms to intelligently handle such cases are not described in this work.

5.4 Implementation

HeteroVisor is implemented by augmenting the Xen CPU scheduler [9] which uses a credit-based scheduling mechanism to accomplish fair sharing of physical cores among virtual CPUs. We extend the credit scheduler by adding two different types of credits: slow and fast. Credits represent the resource right of a VM to execute on the

respective types of cores and are distributed periodically (30ms) to each running VM. A vCPU owns one type of credits during one accounting period. As the VM executes, its credits are decremented periodically (30ms) based upon the type of cores it uses. A vCPU can execute as far as it has positive credits available. Once it has consumed all the credits, it is made offline by putting it into a separate 'parking queue' until the next allocation period. At this point, the credits are redistributed to each VM and its vCPUs are made available for scheduling again. Further, a circular queue of vCPUs is maintained to perform a vCPU rotation between slow and fast cores periodically (10 scheduler ticks, i.e., at a frequency of 300ms). This granularity is found to be sufficient for long-running server workloads, however, using a faster period is trivial as well.

The E-state driver is implemented as a Linux kernel module which periodically changes E-states by issuing a hypercall to the Xen. The E-state driver uses a QoS interface in the form of a proc file to which the application periodically writes its QoS metric. In addition, it reads the CPU utility factor from the hypervisor through a hypercall interface. In our current implementation, a single E-state is assigned to each individual VM which are shared by all the vCPUs belonging to that VM. However, the implementation can easily be extended to per-vCPU E-states as well. The E-state driver runs once every second, with a value of three for the history counter.

5.5 Evaluation

5.5.1 Experimental Setup

Our experimental platform consists of a dual-socket 12 core Intel Westmere server with 12GB DDR3 memory. In order to experiment with real platform and workloads, we emulate heterogeneity on this platform. Processor heterogeneity is emulated using CPU throttling by writing to CPU MSRs which allows changing the duty cycle of each core independently. For the purpose of experiments, a platform configuration

consisting of eight slow cores and four fast cores is considered where slow and fast cores are distributed uniformly on each socket to minimize migration overheads. The performance ratio between fast and slow cores is kept at 4x. Experiments are conducted using a VM with 12 vCPUs, providing an elasticity range up to 2U. Having an E-state step of 0.2U gives us 10 CPU E-states from E0 (2U) to E9 (0.2U) which are exported by the E-state interface.

Table 10: Thresholds for QoS- and resource-driven scaling policies

	qos_{hi}	$util_{lo}$	qos_{lo}	$util_{hi}$
ES-Q	1/5	40	1/10	90
ES-R	1/5	50	1/15	95

In our experiments, response time is chosen as the QoS metric (lower is better), implying an inverse value of latency is used in the QoS thresholds for the driver. A latency value of 10ms is chosen as the SLA corresponding to which two policies are evaluated by using different thresholds for the scaling algorithm. The thresholds for these policies are shown in Table 10 which are obtained after experimenting with several different values. The first QoS-driven policy (ES-Q) is performance-sensitive while the second resource-driven policy (ES-R) has higher affinity for lower speeds, and thus, is driven by higher resource savings.

5.5.2 Workloads

Evaluation is carried out using the Apache web-server based application which services a stream of incoming requests by executing a CPU-intensive computation kernel in response to each request. A change in the input request rate causes a change in the resources used by the server. In addition, several other benchmarks including SPEC CPU2006 and SPECjbb are also included in the analysis.

In order to simulate variable resource usage behavior, workload profiles based on data from Google cluster traces are used [40]. Specifically, Google cluster data

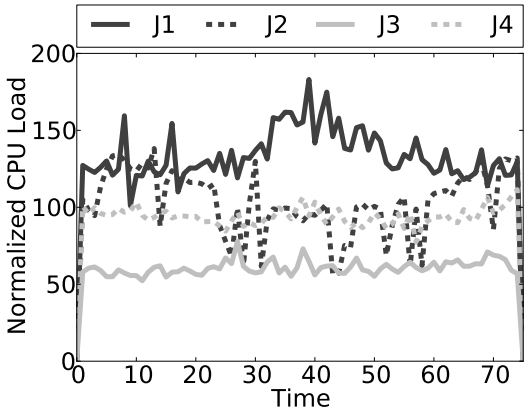


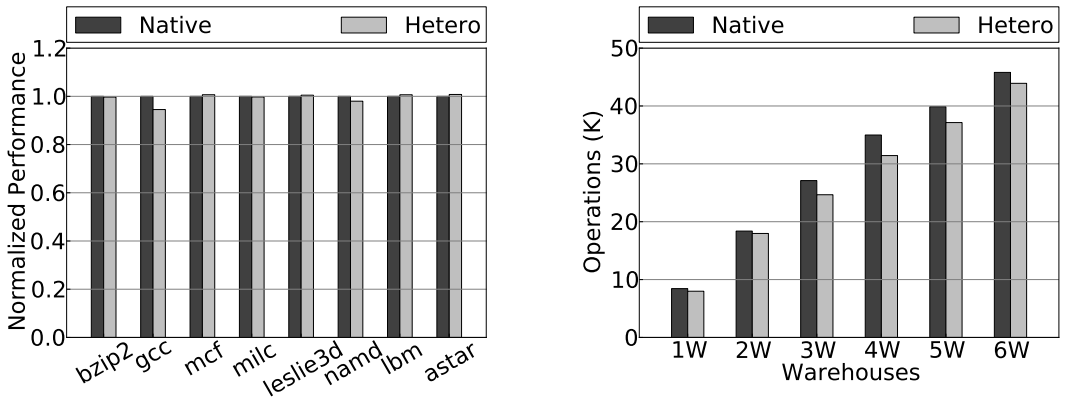
Figure 40: Workload traces based on Google cluster data [40]

provides normalized CPU utilization of a set of jobs over several hours from one of their production clusters. The dataset consists of four types of jobs from which we obtain the average CPU load of each type of job, with resultant data shown in Figure 40. As seen from the figure, workload J1 has constant high CPU usage while J2 has varying behavior, with phases of high and low usage. In comparison, workload J3 and J4 have uniform CPU usage, with J3 having significant idle component. These traces are replayed by varying the input request rate in proportion to the CPU load, with each data point maintained for 20 seconds. It is to be noted that the data presented in the graphs is averaged across the entire cluster instead of retrieved from a single server instance since the dataset does not provide the machine mapping, but we believe that these jobs provide us a good mix to test different dynamic workload scenarios present in server systems.

5.6 Results

The key objective of the evaluation is to analyze the performance and resource saving gains attainable using fine-grained elastic scaling over static allocation schemes.

First, we evaluate the overhead associated with scaling operations in Figure 41(a) that compares the performance of several SPEC CPU2006 benchmarks with composed

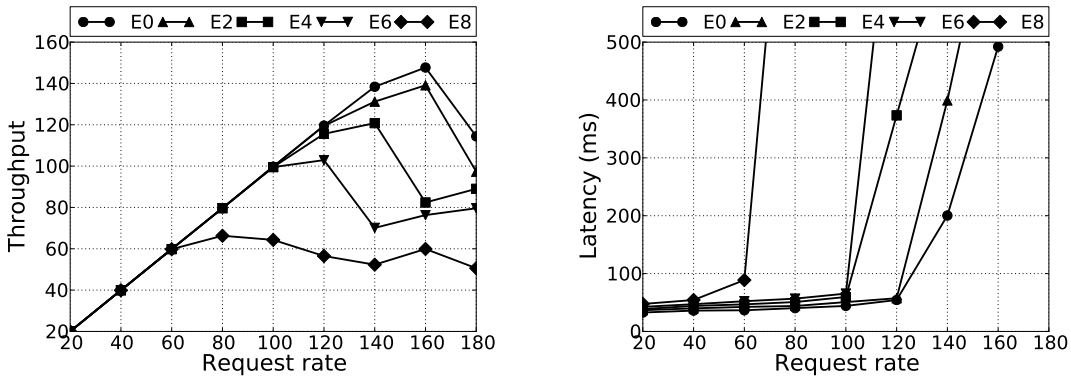


(a) SPECCPU (b) SPECJBB

Figure 41: Performance comparison of heterogeneous configurations with the native platform

virtual platforms using heterogeneous cores (8S+4F) against standard homogeneous configurations (12S). Both the configurations operate at an elastic core speed of 1U. The data suggests that HeteroVisor provides comparable performance for all of the benchmarks to that exhibited by the standard homogeneous platform, implying minimal overhead associated with scaling operations. In order to evaluate multi-threaded execution scenarios, Figure 41(b) shows the performance score for SPECjbb2005, a Java multi-tier warehouse benchmark, at different configurations by increasing the number of warehouses. As it can be seen from the results, performance results for the both the cases closely follow each other with increasing threads, showing its applicability to multi-threaded applications as well.

We next conduct benchmarking experiments by observing response time and throughput variation of the web-server in response to increasing request rate at different elastic speeds. The corresponding results are shown in Figure 42 where different curves correspond to elastic core speeds varying from E8 (0.4U) to E0 (2U) in increments of 0.4U. As evident from the figure, different core speeds behave similarly at low load points. Further, each curve has a tipping point where throughput starts to degrade and latency starts to rise quickly. This point corresponds to the maximum



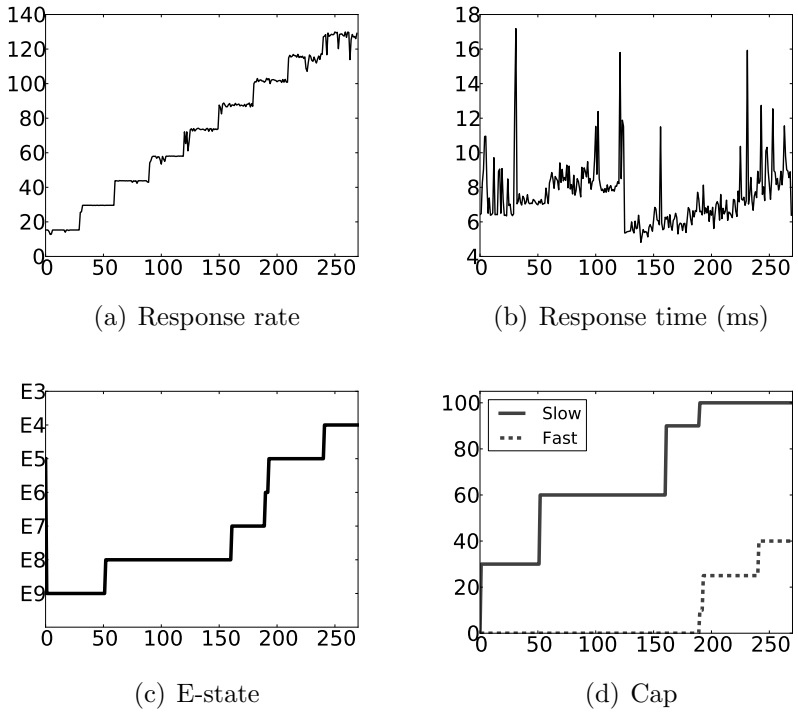
(a) Throughput (b) Latency

Figure 42: QoS variation with different E-states

service rate of the server while providing acceptable performance. Beyond this point, a higher core speed provides higher throughput and lower latency. Thus, low speed cores can be used at low request rates for saving resources, while high-speed cores are needed to maintain performance when server load is high.

Next, E-state scaling mechanisms are evaluated by executing the web-server application with increasing load and observing dynamic scaling of E-states (see Figure 43). Specifically, Figure 43(a) and 43(b) show the response rate and response time for this workload. As shown in the figure, the throughput rises gradually as the load is increased. The corresponding latency curve is relatively flat as the E-state driver scales E-states to maintain latency within SLA (10ms). We also notice few spikes in the latency graph which happen due to increase in the input load. In response, E-state is scaled up to bring it back down. The resultant E-state graph is shown in Figure 43(c) where E-states are scaled from from E9 to E4 in multiple steps. Also shown is the corresponding slow core and fast core usage in Figure 43(d). Initially, slow core load is increased which get saturated at time 185s, at which point fast cores are used, gradually increasing their usage.

Our next results evaluate the four workloads based on Google cluster traces shown



(a) Response rate

(b) Response time (ms)

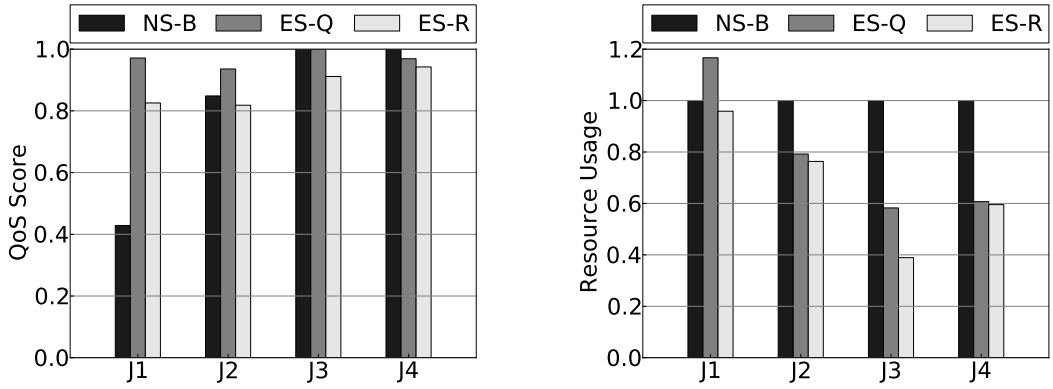
(c) E-state

(d) Cap

Figure 43: Elastic scaling experiment using the webserver workload (x-axis = time (s))

in Figure 40. The results in Figure 44 compare the QoS and resource usage for the base configuration without any elastic scaling (NS-B) with HeteroVisor based elastic scaling with the two policies ES-Q and ES-R given in Table 10. The base platform configuration consists of 12 slow cores, each with an elastic speed of 1U. The QoS score graph shows percentage of queries for which the service latency falls within SLA (10ms). Similarly, resource usage graphs compare the relative usage of various configurations where a linear relationship is assumed between E-states and their resource usage.

As the results demonstrate, both the policies provide much higher QoS than the base system for workload J1. Specifically, QoS-sensitive policy ES-Q results in 97% QoS score, with 17% resource usage penalty, while the resource-driven policy ES-R provides lower QoS (83%), with lower usage (0.96x). In comparison, the base platform



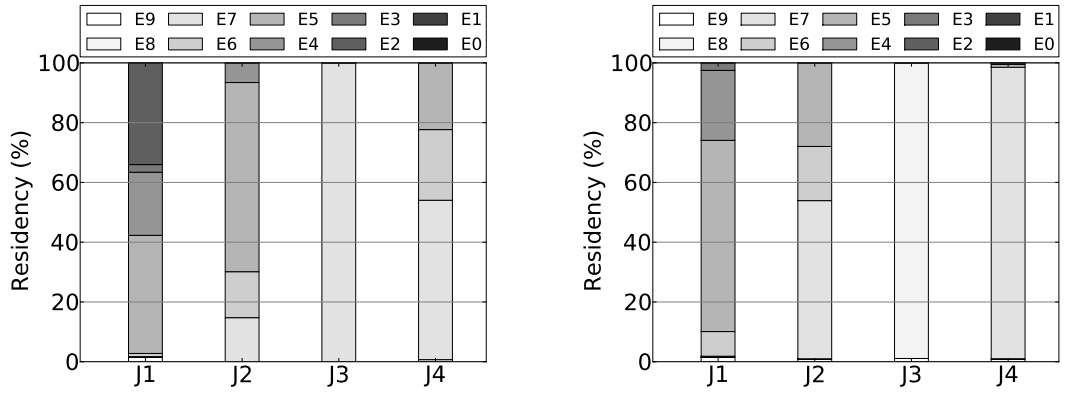
(a) QoS Score

(b) Resource Usage

Figure 44: Experimental results for CPU E-state scaling

can only sustain 43% QoS. Thus, HeteroVisor can scale up platform resources to provide better performance when system load is high. For workload J2, ES-Q exhibits 9% higher and ES-R results in 3% lower QoS, while also reducing the resource usage by 21% and 24% respectively. Thus, resources are scaled up and down to meet the desired performance requirement. For J3 with low input load, HeteroVisor yields resource savings while also maintaining QoS, i.e., it generates 100% and 91% QoS score with 42% and 61% lower resource usage for the two policies. In this manner, scaling down resources during low load periods produces savings for these jobs. Finally, uniformly behaving workload J4 also shows comparable performance with significant resource savings across these configurations (~40%). Thus, E-states enable dynamic scaling of resources providing high-performance when required (as for J1) and resulting in cost savings for low activity workloads like J3 and J4.

To illustrate the elastic scaling of resources further, Figure 45 shows the residency distribution (%) in each E-state for each of the four jobs for CPU scaling experiments. The states are color coded by their gray-scale intensity, meaning a high-performance E-state is depicted by a darker color in comparison to a low-performing E-state. The graphs in Figure 45(a) and 45(b) correspond to the ES-Q and ES-R policies.



(a) QoS-driven (ES-Q) (b) Resource-driven (ES-R)

Figure 45: E-state residencies for different scaling policies

As seen in the figure, different E-states dominate different workloads. J1 has large shares of E-states E2, E4, and E5 due to its high activity profile. For low-CPU J3 workload, slower state E7 is dominant under ES-Q policy while ES-R policy has highest residency in the state E8. Similarly, J4 spends majority of its execution time in state E7 and E5, while J2 has mixed usage of E8, E7, E6, and E5 states. Thus, different workloads make use of different elasticity states. The corresponding E-state switch profiles for the ES-Q policy are shown in Figure 46. For each workload, a similarity can be observed between the load profile in Figure 40 and these E-state changes. Both J1 and J4 stay in lower E-states initially and scale up when the demand increases. J3 stays in a single E-state, while J2 has several E-state transitions due to its variable load. Thus, HeteroVisor dynamically scales resources to match the input load requirement.

In this manner, HeteroVisor exploits platform heterogeneity and enables dynamic scaling of resources to meet desired application performance/cost trade-offs. As shown by the experimental data, it not only better services load peaks in comparison to homogeneous platforms (upto 2.3x) but also provides savings (average 21%) scaling down resources during idle periods. Also, E-state driver can be customized to meet

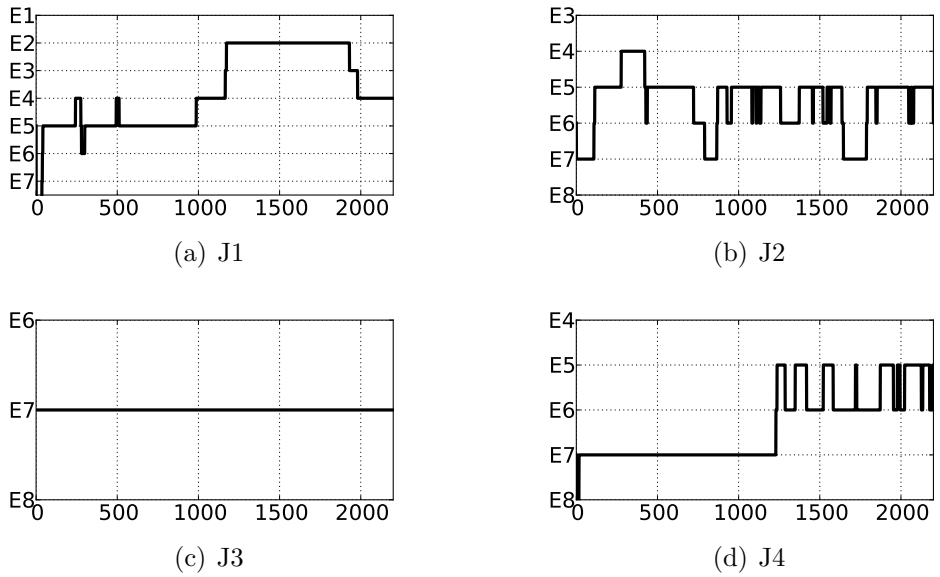


Figure 46: E-state switch profiles showing usage of various states (x-axis = time (s), y-axis = E-states)

different user requirements, either meeting high QoS requirement using an aggressive policy or reducing resource usage while maintaining performance by using a conservative policy.

5.7 Related Work

5.7.1 Resource Management in Clouds

There has been substantial prior work on elastic resource scaling for server systems. In comparison to cluster-level scaling solutions [3, 25, 43], HeteroVisor focuses on platform-level scaling methods for fine-grained resource scaling. RaaS (resource-as-a-service) computing paradigm argues in favor of fine-grained resource management for future cloud platforms [1]. Similarly, Kaleidoscope also makes a case for fine-grained elasticity in clouds and presents techniques based on VM cloning for achieving the same [13]. The ideas presented by these studies motivate the need for fine-grained resource management in clouds as explored in this work. Q-clouds described Q-state

abstraction to mitigate performance interference effects in shared virtualized platforms [75]. Further, market based allocation methods for datacenter applications have also been analyzed [31, 103]. These techniques can be incorporated into the design of elasticity drivers to generate interference-aware or revenue-aware resource bids. AutoPilot, CloudScale, and VirtualPower proposed hypervisor-level mechanisms for elastic scaling of cloud resources [76, 81, 93]. All of these techniques motivate the need for adaptive resource usage for maximizing efficiency. However, none of these address the presence of platform heterogeneity.

5.7.2 Heterogeneous Processor Scheduling

Heterogeneous multicore processors have been proposed to achieve higher energy-efficiency than symmetric multicore processors [30, 55]. Earlier work has demonstrated the need for compute heterogeneity in datacenters to efficiently support a wide variety of applications [6, 10, 48, 59, 105] and shown its presence in commercial cloud platforms like EC2 [79]. Several implementations of heterogeneous processor architecture have been released by CPU vendors [18, 29, 78, 45]. In order to manage these platforms, appropriate techniques have been developed to efficiently run applications on heterogeneous cores [32, 54, 64, 92]. Further, mechanisms to effectively virtualize heterogeneous multicore platforms have also been investigated [52]. HeteroVisor adopts an alternative approach by hiding heterogeneity from the OS scheduler, exposing a homogeneous scalable resource interface. Finally, several cloud schedulers have also been proposed to incorporate heterogeneity [19, 74]. These cluster-level techniques are complementary to HeteroVisor that works at the platform level.

5.8 Summary

In summary, this work presents HeteroVisor system for managing heterogeneous resources in elastic cloud platforms, providing fine-grained scaling capabilities for applications. To manage heterogeneity, it provides the abstraction of elasticity (E) states

to the guest machine which an E-state driver could use to elastically request resources on-demand. The proposed abstractions are applicable to multiple resources and levels of heterogeneity. Demonstrating its application to the processor subsystem, various techniques are presented to manage these heterogeneous resources in an elastic manner. The HeteroVisor solution is implemented in the Xen hypervisor along with a simple E-state driver for two scaling policies, QoS-driven and resource-driven. Evaluation is carried out using real-world traces on an emulated heterogeneous platform, showing that HeteroVisor can provide VMs with the capabilities to quickly obtain resource for handling load spikes or minimize cost during low load periods.

CHAPTER VI

CONCLUSIONS & FUTURE WORK

6.1 Conclusions

This dissertation investigates the use of heterogeneous platform resources to provide fine-grain resource scaling capabilities on future systems. Using the abstraction of 'resource states', it decouples heterogeneity from the resource management operations to provide both high performance and resource efficiency in a seamless manner. The work described considers both mobile systems and cloud platforms, focusing on CPU and memory subsystems. Specific contributions from the thesis include:

- A performance and energy analysis of modern client and server workloads on a heterogeneous multicore platform.
- Impact of uncore subsystem on the energy-efficiency of heterogeneous cores.
- Mechanisms for software-controlled management of heterogeneous memory platforms consisting of fast die-stacked memory and slow off-chip memory.
- HeteroMates solution for client devices to extend their dynamic range using 'core groups' abstraction
- HeteroVisor system to enhance the elastic resource scaling capabilities of cloud platforms using heterogeneous components.

The proposed solutions and evaluations from this work lead to several conclusions. First, heterogeneity is a viable approach to provide fine-grain and elastic scaling for both client devices and cloud platforms. Using such mechanisms can lead to higher performance-levels and resource-efficiency gains than that can be obtained

using homogeneous configurations. As demonstrated by HeteroMates and HeteroVisor solutions, new system abstractions are required for smoother adoption of such heterogeneous platforms into mainstream. Moreover, for such methods to be effective, scalability across various components needs to be explored and employed. A non-scalable component like uncore can significantly affect the gains achievable from other scalable components like core.

It should also be mentioned that this thesis evolved from the HeteroMates work on mobile devices to the HeteroVisor solution for cloud systems. HeteroMates using the H-state abstraction allows only grouping of CPU cores where a fraction of the cores are always idle. On the other hand, HeteroVisor uses a more generalized abstraction of E-states where all of the cores can be used simultaneously. HeteroMates approach may be feasible for the client devices with small number of cores, but it is not practical for server systems with large number of cores due to excessive cost implications. This observation resulted into us exploring the HeteroVisor solution for server systems, breaking the strict grouping of cores. However, the resultant solution need not be limited to servers only as the approach can also be used for mobile devices, thus, improving die utilization over the core groups abstraction used by HeteroMates. We have not performed this evaluation as part of this thesis.

From the experiences during the course of this study, a fundamental question regarding heterogeneity arises: *whether it is worth the effort*. In the author's opinion, the need for improved performance/efficiency with various technological limits approaching [12, 22] and competition among various vendors to offer attractive new features is likely to push in favor of heterogeneity. It is also likely that adoption of heterogeneous platforms would be dependent on the application domain. For example, authors have questioned the effectiveness of certain use cases of heterogeneous processors for datacenter applications [36, 37]. This dissertation particularly focused on mobile devices and datacenter environments. However, another key area

related to high-performance computing (HPC) also remains interesting. Desire for ever-increasing performance, along with strong emphasis on energy-efficiency makes heterogeneity an attractive option for these systems.

The challenge with heterogeneity lies in managing these platforms which can be done at various levels. At one extreme, it could be completely hidden from the software as in the case of multiple levels of CPU caches. This approach may be suitable for easier adoption of such platforms, but the need for performance/efficiency requires software involvement. Another alternative would be to explore scenarios where heterogeneity is exposed and explicitly controlled by the user VMs/applications [33]. This approach provides users more control over the desired allocations, however, requires sophisticated software support. Thus, hiding heterogeneity within hardware can be too restrictive while exposing them to applications can be too disruptive. Rather, a balanced approach is required which gives enough control to the applications, but does not overload them with complexity.

Orthogonal to the approach taken in this dissertation where underlying platform has heterogeneous resources, architectural techniques exploring mechanisms for a single scalable component which can morph into different types of resources such as a highly parallel processor vs. a high speed serial execution unit also look quite promising [46, 53, 71]. Software mechanisms proposed in this thesis become highly relevant for such platforms where exposing heterogeneity is not an option. Another aspect of heterogeneity that was not explored as part of this thesis is concerning functional heterogeneity where various cores differ in their instruction sets as well, having either shared ISA [90] or disjoint ISA as in the case of accelerators [17, 34, 101]. Though more challenging in terms of management due to their incompatible functionality, the level of performance/efficiency provided by these specialized systems makes them the way to go forward.

6.2 Future Work

There are several directions that become open from the work in this dissertation.

First, the elasticity-state concept presented in this dissertation is applicable to multiple resources, namely processor, memory, and storage. This thesis presented detailed implementation and evaluation for the processor subsystem. Thus, investigating challenges associated with other components would be an area of research which has many open questions. For example, managing heterogeneous memory resources involves efficient page migration mechanisms which needs further investigation [61]. Another related issue is scaling in the presence of multi-level heterogeneity involving a hierarchy of heterogeneous resources. For instance, managing stacked memory, off-chip DRAM, and NVRAM together using both hot page migrations and ballooning. Further, it is unclear how these mechanisms can be extended to incorporate functionally heterogeneous components. A possibility would be to assign a state to each accelerator and manage them implicitly using fault-and-migrate technique [90]. Such inclusion would make these abstractions even more generic. In addition, efficient horizontal scaling of platform resources requires more research [98]. First, new OS mechanisms are required to make such mechanisms efficient as existing mechanisms are not suitable for frequent reconfiguration [83]. Further, such scaling mechanisms may not be transparent to the user and thus require additional runtime support. For example, adding a virtual CPU to a VM may currently require restarting the application with a different number of threads to match the underlying platform configuration. Solutions like Elastin [77] are a step in this direction, but these frameworks target specific application domains. More work is required for such mechanisms to become mainline.

With the introduction of fine-grain resource scaling mechanisms on cloud platforms, a whole new area becomes open concerning the design of policies for requesting

and allocating resources in the presence of multiple competing users. For the applications, controllers should be designed to bid for resources taking into account desired QoS and budget constraints. The controllers described in this thesis are reactive which may lead to performance violations or can be slow to reach a stable state. So further work exploring predictive controllers such as AutoScale, CloudScale, etc. [80, 81, 93] can improve such methods. Similarly, the host system should perform QoS-aware distribution of constrained platform resources across VMs, trying to maximize its gain. Market-based allocation mechanisms based on game theory become relevant in this context [31]. The problem becomes more complex when considering multi-resource allocation scenarios, along with heterogeneity, where different resources affect QoS for various applications in an application dependent manner [26]. Further, cluster-level scheduler mechanisms need to be integrated into platform-level scaling mechanisms for QoS-aware placement and migration of virtual machines [15, 43].

Finally, detailed analysis is required to determine what constitutes an ideal heterogeneous platform, i.e., the characteristics of various components and the size of each of them to be included on the platform. The analysis should include the cost of each component, the performance/power properties of them, the nature of applications for the target domain, the willingness of customers to buy such systems and corresponding gains. In addition, heterogeneity can be incorporated at different levels such as socket-, platform-, or cluster-level in the case of processors. Each level provides different level of flexibility and complexity, thus, determining the appropriate level of heterogeneity needs further investigation. For the architecture researchers, it would also be an interesting venue of research to look into the design of scalable uncore, analyzing which components can be scaled and what the associated performance overheads would be. Also, novel uncore-aware scheduling algorithms need to be devised such as delayed execution to coordinate core idle states, thus, maximizing

uncore sleep time and energy savings. Another related issue is the challenge of managing heterogeneous resources while taking user-perceived performance into account. A solution to deal with this problem would be to maintain a history of previous allocations and corresponding performance metrics, and thus, scaling resources for different applications to meet the desired performance levels [100].

Overall, heterogeneity is still an evolving space, with several innovations likely to appear in future platforms. A hardware/software co-designed approach is key in making them a favorable element in the computing eco-system.

APPENDIX A

CLIENT WORKLOAD SUITE

A.1 Browser

Web-browsing is the most common usage of mobile devices. Users perform various tasks using their browsers. We pick multiple applications under this category to evaluate different use cases.

- browse: This workload fetches a set of popular web pages from a web server and renders them in the browser. Performance metric is the average load time of a page which it measures by inserting JavaScript commands into the pages. The onload event is used to know when the browser finishes loading the page.
- javascript: This workload is based on the sunspider benchmark which performs various standard javascript tests including math, string, crypto operations etc.
- palbum: A photo-album application that flips through a set of photographs in the browser using Javascript. Performance metric is defined as the average load time for a photograph.

Web-based workloads are executed with a client-server setup, i.e., browser and a web-server running on two different systems. The client machine is the machine-under-test with a heterogeneous CPU configuration where we focus on the browser performance. A second machine acts as the web-server serving the requests from the client machine.

A.2 Gaming

Gaming is another popular usage of mobile devices. Two games are included under this component which are representative of the games played in low-power devices.

- chess: GNU Chess game is used to evaluate 2D gaming scenario. Various moves from a previously played game session are stored in a file and loaded into the game at start up. The benchmarking part replays these moves automatically by sending key strokes using *xautomation* utility.
- strike: A benchmarking demo of a shooting game is played for 30s to evaluate this usage scenario. The demo simulates a scrolling shooter similar to the Raiden arcade game and displays the achieved frame-rate at the end of demo.

A.3 Multimedia

Multimedia is an integral component of end-user devices where users perform a variety of operations ranging from media consumption, creation, and editing. Several use-case scenarios involving images and videos are included in this component.

- animate: Picture animation is used to animate a pre-defined sequence of images. Picture animation is a command-line executable, which is part of the ImageMagick software package.
- convert: Convert command-line utility from the ImageMagick program is used to resize a set of 100 images in batch mode.
- mencoder: A media file is encoded from H.264 format to AVI format with MPEG4 codec using mencoder utility in command-line mode.
- mplayer: To evaluate video-playback performance, mplayer plays an HD clip of the popular Elephant's Dream movie for a total of 1000 frames in noframedrop mode and measures the achieved frame rate.

A.4 Productivity

Productivity applications are also increasingly used on various smart mobile devices. This benchmark is adopted from TMAPP suite [47] and assesses various functions of OpenOffice applications using Office *macro* scripts for automation.

- calc: The Calc module consists of starting the application, opening a Calc document with tables and graphs, calculating various data points in multiple tables and automatically creating graphs in the document, saving it, and closing the document and the application.
- impress: The Impress module consists of starting the application, opening a presentation document with animation, modifying it and saving it, showing a slideshow from first to last page and closing the document and application.
- writer: The Writer module consists of launching the application, opening a document, saving after modifying it, scrolling through from top to bottom, and closing the document and the application.

A.5 Utility

Several utilities are also included in the analysis to evaluate the performance of various operations performed on client devices.

- 7zip: A parallelized version of popular 7zip application is used to compress a text file (20MB chunk from Wikipedia text) using LZMA compression.
- diskscan: To evaluate the behavior of I/O intensive applications, this workload simulates disk I/O operations common in reading/compiling kernel trees.
- gtkperf: This benchmark evaluates the performance of various GUI elements such as text box, progress bar, buttons, etc. and measures average latency.
- pguard: GNU Privay Guard app is used to encrypt a large file (512MB) using a given passphrase
- sqlite: This lightweight database application is used in mobile applications for storing useful information (e.g., cookies in browser). This workload performs a series of mysql operations.
- wget: To evaluate download performance, wget utility is used to download a large media file from the web-server.

APPENDIX B

VIRTUAL CORE SCALING MODELS

The expressions for the corresponding usage of various cores for achieving a given effective processing speed can be obtained as follows. Consider a heterogeneous CPU configuration consisting of H types of cores with n_i cores of type i and processing speed p_i. To compose v_n virtual cores with speed p_v, various types of cores can be utilized partially. It is assumed that sufficient cores are available to meet the processing requirement. Let us denote the number of virtual cores that belong to the pool of core type i as v_i and the fraction of that core assigned to the virtual core as u_i. Thus, the total processing capacity of the virtual cores should match the processing speed of the physical cores as shown in Equation 15, subjected to the constraints in Equation 16. If each core type has a corresponding cost (infrastructure and operational) given by the function $tco(i)$, an objective function for minimizing the TCO can be expressed as in Equation 17. This formulation is a linear optimization problem which can be solved using standard solvers.

$$\sum_{i=1}^{H} v_i * u_i * p_i = v_n * p_v \tag{15}$$

$$\sum_{i} v_i = v_n \qquad 0 \leq u_i \leq 1 \qquad 0 \leq v_i * u_i \leq n_i \quad (i : 1 \rightarrow H) \tag{16}$$

$$minimize\left(\sum_{i=1}^{H} v_i * u_i * tco(i)\right) \tag{17}$$

Since the allocation problem needs to be computed in the kernel-space, and thus, should be fast, instead of relying on external solvers, we obtain a closed-form solution for a special case of two types of cores, slow (s) and fast (f), where slow cores have lower TCO than faster cores. Since abundant slow cores are utilized first, using scarce

fast resources only when necessary, if the total required processing capacity ($v_n * p_v$) is smaller than that of all the slow cores ($n_s * p_s$), only slow cores are used, i.e., both v_f and u_f become zero. Thus, nthe umber of slow cores v_s becomes equal to v_n, with u_s calculated as shown below.

$$u_s = \frac{v_n * p_v}{v_s * p_s} \tag{18}$$

If, however, both types of cores should be utilized for higher core speeds, a fraction of the virtual cores are run on the slow pool, while the remaining are run on the fast pool. Using Equation 15, fast core utilization u_f can be expressed in the form of Equation 19. Since core usage can not be greater than 1, solving this equation for v_f gives us Equation 20. v_f is thus chosen to be the smallest integer satisfying this condition to minimize fast core usage.

$$u_f = \frac{v_n * (p_v - p_s * u_s) + v_f * p_s * u_s}{v_f * p_f} \tag{19}$$

$$v_f \geq \frac{v_n(p_v - p_s * u_s)}{p_f - p_s * u_s} \tag{20}$$

It further leads into two cases: first when the number of virtual cores is smaller than the number of slow cores ($v_n < n_s$) and second when they are larger in number. Since slow cores are used fully to minimize fast core usage, when there are enough physical cores available for all the virtual cores, u_s becomes 1 for the first scenario. On the other hand, slow core resources are shared by all the virtual cores belonging to the slow core pool in the second case, implying u_s is set equal to the ratio of n_s and v_s.

Substituting and solving these equations bring us to the complete results in Equation 21 which shows the slow and fast core usage for various scenarios. Using this equation along with Equations 15 and 16, all the required variables (v_s, u_s, v_f, u_f) can be obtained.

$$\{u_s, u_f\} = \begin{cases} \frac{p_v}{p_s}, 0 & \text{if } p_v \leq \min(1, \frac{n_s * p_s}{v_n}), \\ 1, \frac{v_n * p_v - (v_n - v_f) * p_s}{v_f * p_f} \;\middle|\; v_f = \lceil \frac{v_n(p_v - p_s)}{p_f - p_s} \rceil & \text{else if } \frac{n_s}{v_n} \geq 1, \\ \frac{n_s}{v_n - v_f}, \frac{v_n * p_v - n_s * p_s}{v_f * p_f} \;\middle|\; v_f = \lceil \frac{v_n * p_v - n_s * p_s}{p_f} \rceil & \text{otherwise} \end{cases} \tag{21}$$

The maximum core speed is determined by maximizing the fast core usage and using slow cores for the remaining capacity as given below.

$$v_p^{max} = \frac{\min(n_f, v_n) * p_f + (v_n - \min(n_f, v_n)) * p_s}{v_n} \tag{22}$$

Using this maximum speed, formulation can be extended to multi-level heterogeneous cores by using two levels of cores upto this maximum speed and moving a level higher in the heterogeneity for larger speeds.

REFERENCES

[1] Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., and Tsafrir, D., "The resource-as-a-service (RaaS) cloud," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, (Berkeley, CA, USA), pp. 12–12, USENIX Association, 2012.

[2] Ahmad, F., Chakradhar, S. T., Raghunathan, A., and Vijaykumar, T. N., "Tarazu: optimizing MapReduce on heterogeneous clusters," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 61–74, ACM, 2012.

[3] Amazon.com, "Amazon EC2 Auto Scaling." `http://aws.amazon.com/autoscaling/`. [Online].

[4] Amazon.com, "Amazon EC2 Pricing." `http://aws.amazon.com/ec2/pricing/`. [Online].

[5] Amazon.com, "Amazon Elastic Compute Cloud (EC2)." `http://aws.amazon.com/ec2/`. [Online].

[6] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V., "FAWN: a fast array of wimpy nodes," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, (New York, NY, USA), pp. 1–14, ACM, 2009.

[7] Apparao, P., Iyer, R., and Newell, D., "Implications of cache asymmetry on server consolidation performance," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 24–32, 2008.

[8] Balakrishnan, S., Rajwar, R., Upton, M., and Lai, K., "The impact of performance asymmetry in emerging multicore architectures," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, (Washington, DC, USA), pp. 506–517, IEEE Computer Society, 2005.

[9] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[10] Barroso, L. A., "Brawny cores still beat wimpy cores, most of the time," *Micro, IEEE*, vol. 30, pp. 20 –24, july-aug. 2010.

[11] Bienia, C., Kumar, S., Singh, J. P., and Li, K., "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[12] Borkar, S. and Chien, A. A., "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[13] Bryant, R., Tumanov, A., Irzak, O., Scannell, A., Joshi, K., Hiltunen, M., Lagar-Cavilla, A., and de Lara, E., "Kaleidoscope: cloud micro-elasticity via VM state coloring," in *Proceedings of the sixth conference on Computer systems*, EuroSys '11, (New York, NY, USA), pp. 273–286, ACM, 2011.

[14] Cao, T., Blackburn, S. M., Gao, T., and McKinley, K. S., "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2012.

[15] Chen, G., He, W., Liu, J., Nath, S., Rigas, L., Xiao, L., and Zhao, F., "Energy-aware server provisioning and load dispatching for connection-intensive internet services," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, (Berkeley, CA, USA), pp. 337–350, USENIX Association, 2008.

[16] Chitlur, N., Srinivasa, G., Hahn, S., Gupta, P. K., Reddy, D., Koufaty, D., Brett, P., Prabhakaran, A., Zhao, L., Ijih, N., Subhaschandra, S., Grover, S., Jiang, X., and Iyer, R., "QuickIA: Exploring heterogeneous architectures on real prototypes," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2012.

[17] Chung, E. S., Milder, P. A., Hoe, J. C., and Mai, K., "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2010.

[18] Cole, B., "Samsung reveals big-little 8-core ARM for mobiles." `http://www.embedded.com/electronics-news/4404964/Samsung-reveals-big-little-8-core-ARM-for-mobiles`, 2013. [Online].

[19] Delimitrou, C. and Kozyrakis, C., "Paragon: QoS-aware scheduling for heterogeneous datacenters," in *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, (New York, NY, USA), pp. 77–88, ACM, 2013.

[20] Deng, Q., Meisner, D., Bhattacharjee, A., Wenisch, T. F., and Bianchini, R., "CoScale: Coordinating CPU and memory system DVFS in server systems," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, (Washington, DC, USA), pp. 143–154, IEEE Computer Society, 2012.

[21] Dong, X., Xie, Y., Muralimanohar, N., and Jouppi, N. P., "Simple but effective heterogeneous main memory with on-chip memory controller support," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2010.

[22] Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D., "Dark silicon and the end of multicore scaling," in *Proceeding of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[23] Fedorova, A., Saez, J. C., Shelepov, D., and Prieto, M., "Maximizing power efficiency with asymmetric multicore systems," *Commun. ACM*, vol. 52, pp. 48–57, Dec. 2009.

[24] Galante, G. and Bona, L. C. E. d., "A survey on cloud computing elasticity," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, (Washington, DC, USA), pp. 263–270, IEEE Computer Society, 2012.

[25] Gandhi, A., Zhu, T., Harchol-Balter, M., and Kozuch, M. A., "SOFTScale: stealing opportunistically for transient scaling," in *Proceedings of the 13th International Middleware Conference*, Middleware '12, (New York, NY, USA), pp. 142–163, Springer-Verlag New York, Inc., 2012.

[26] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I., "Dominant resource fairness: fair allocation of multiple resource types," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 24–24, USENIX Association, 2011.

[27] Goel, B., McKee, S. A., Gioiosa, R., Singh, K., Bhadauria, M., and Cesati, M., "Portable, scalable, per-core power estimation for intelligent resource management," in *Proceedings of the International Conference on Green Computing*, GREENCOMP '10, (Washington, DC, USA), pp. 135–146, IEEE Computer Society, 2010.

[28] Google, "Google Compute Engine." `https://cloud.google.com/products/compute-engine/`. [Online].

[29] Greenhalgh, P., "Big.LITTLE Processing with ARM CortexTM-A15 & Cortex-A7." White paper, ARM, Sept 2011.

[30] Grochowski, E., Ronen, R., Shen, J., and Wang, H., "Best of both latency and throughput," in *Proceedings of the IEEE International Conference on Computer Design*, ICCD '04, (Washington, DC, USA), pp. 236–243, IEEE Computer Society, 2004.

[31] Guevara, M., Lubin, B., and Lee, B. C., "Navigating heterogeneous processors with market mechanisms," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pp. 95–106, 2013.

[32] Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "HeteroMates: Providing high dynamic power range on client devices using heterogeneous core groups," in *Green Computing Conference (IGCC), 2012 International*, pp. 1 –10, june 2012.

[33] Gupta, V., Knauerhase, R., Brett, P., and Schwan, K., "Kinship: efficient resource management for performance and functionally asymmetric platforms," in *Proceedings of the ACM International Conference on Computing Frontiers*, CF '13, (New York, NY, USA), pp. 16:1–16:10, ACM, 2013.

[34] Gupta, V., Schwan, K., Tolia, N., Talwar, V., and Ranganathan, P., "Pegasus: coordinated scheduling for virtualized accelerator-based systems," in *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2011.

[35] Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "The forgotten 'uncore': on the energy-efficiency of heterogeneous cores," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 34–34, USENIX Association, 2012.

[36] Gupta, V. and Nathuji, R., "Analyzing performance asymmetric multicore processors for latency sensitive datacenter applications," in *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[37] Gupta, V., Nathuji, R., and Schwan, K., "An analysis of power reduction in datacenters using heterogeneous chip multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 3, pp. 87–91, 2011.

[38] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[39] Gutierrez, A., Dreslinski, R. G., Wenisch, T. F., Mudge, T., Saidi, A., Emmons, C., and Paver, N., "Full-system analysis and characterization

of interactive smartphone applications," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, (Washington, DC, USA), pp. 81–90, IEEE Computer Society, 2011.

[40] Hellerstein, J. L., "Google cluster data." Google research blog, Jan. 2010. Posted at `http://googleresearch.blogspot.com/2010/01/google-cluster-data.html`.

[41] Henning, J. L., "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, Sept. 2006.

[42] Hill, M. D. and Marty, M. R., "Amdahl's law in the multicore era," *Computer*, vol. 41, pp. 33–38, July 2008.

[43] Hong, Y.-J., Xue, J., and Thottethodi, M., "Dynamic server provisioning to minimize cost in an IaaS cloud," in *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '11, (New York, NY, USA), pp. 147–148, ACM, 2011.

[44] Hruby, T., Bos, H., and Tanenbaum, A. S., "When slower is faster: On heterogeneous multicores for reliable systems," in *Proceedings of the 2013 USENIX conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), USENIX Association, 2013.

[45] Intel, "Intel Xeon Phi Coprocessor - the Architecture." `http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner`, 2013. [Online].

[46] Ipek, E., Kirman, M., Kirman, N., and Martinez, J. F., "Core fusion: accommodating software diversity in chip multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, (New York, NY, USA), pp. 186–197, ACM, 2007.

[47] Issa, J. and Figueira, S., "Performance and power-consumption analysis of mobile internet devices," in *Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, PCCC '11, (Washington, DC, USA), pp. 1–6, IEEE Computer Society, 2011.

[48] Janapa Reddi, V., Lee, B. C., Chilimbi, T., and Vaid, K., "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 314–325, ACM, 2010.

[49] Jiang, X., Madan, N., Zhao, L., Upton, M., Iyer, R., Makineni, S., Newell, D., Solihin, D., and Balasubramonian, R., "CHOP: Adaptive filter-based DRAM caching for CMP server platforms," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, 2010.

[50] Jiang, X., Mishra, A., Zhao, L., Iyer, R., Fang, Z., Srinivasan, S., Makineni, S., Brett, P., and Das, C. R., "ACCESS: Smart scheduling for asymmetric cache CMPs," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, (Washington, DC, USA), pp. 527–538, IEEE Computer Society, 2011.

[51] Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Geiger: monitoring the buffer cache in a virtual machine environment," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, (New York, NY, USA), pp. 14–24, ACM, 2006.

[52] Kazempour, V., Kamali, A., and Fedorova, A., "AASH: an asymmetry-aware scheduler for hypervisors," in *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '10, (New York, NY, USA), pp. 85–96, ACM, 2010.

[53] Khubaib, Suleman, M. A., Hashemi, M., Wilkerson, C., and Patt, Y. N., "MorphCore: An energy-efficient microarchitecture for high performance ILP and high throughput TLP," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, (Washington, DC, USA), pp. 305–316, IEEE Computer Society, 2012.

[54] Koufaty, D., Reddy, D., and Hahn, S., "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 125–138, ACM, 2010.

[55] Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M., "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, (Washington, DC, USA), pp. 81—-, IEEE Computer Society, 2003.

[56] Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I., "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, (Washington, DC, USA), pp. 64—-, IEEE Computer Society, 2004.

[57] Kwon, Y., Kim, C., Maeng, S., and Huh, J., "Virtualizing performance asymmetric multi-core systems," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 45–56, ACM, 2011.

[58] Lakshminarayana, N. B., Lee, J., and Kim, H., "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, (New York, NY, USA), pp. 25:1—-25:12, ACM, 2009.

[59] Lang, W., Patel, J. M., and Shankar, S., "Wimpy node clusters: what about non-wimpy workloads?," in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, (New York, NY, USA), pp. 47–55, ACM, 2010.

[60] Lee, C., Kim, E., and Kim, H., "The AM-Bench: An Android multimedia benchmark suite," Tech. Rep. GIT-CERCS-12-04, Georgia Institute of Technology, 2012.

[61] Lee, M., Gupta, V., and Schwan, K., "Software-controlled transparent management of heterogeneous memory resources in virtualized systems," in *Proceedings of the 2013 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, (New York, NY, USA), ACM, 2013.

[62] Lee, M. and Schwan, K., "Region scheduling: efficiently using the cache architectures via page-level affinity," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 451–462, ACM, 2012.

[63] Li, T., Baumberger, D., Koufaty, D. A., and Hahn, S., "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, (New York, NY, USA), pp. 53:1—-53:11, ACM, 2007.

[64] Li, T., Brett, P., Knauerhase, R., Koufaty, D., Reddy, D., and Hahn, S., "Operating system support for overlapping-ISA heterogeneous multi-core architectures," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, 2010.

[65] Lim, K., Turner, Y., Santos, J. R., AuYoung, A., Chang, J., Ranganathan, P., and Wenisch, T. F., "System-level implications of disaggregated memory," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[66] Lin, F. X., Wang, Z., LiKamWa, R., and Zhong, L., "Reflex: using low-power processors in smartphones without knowing them," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 13–24, ACM, 2012.

[67] Loh, G. H., "3D-stacked memory architectures for multi-core processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, (Washington, DC, USA), pp. 453–464, IEEE Computer Society, 2008.

[68] Loh, G. H., "The cost of uncore in throughput-oriented many-core processors," in *In Proc. of Workshop on Architectures and Languages for Throughput Applications (ALTA)*, 2008.

[69] Loh, G. H., Jayasena, N., McGrath, K., O'Connor, M., Reinhardt, S., and Chung, J., "Challenges in heterogeneous die-stacked and off-chip memory systems," in *In Proc. of 3rd Workshop on SoCs, Heterogeneity, and Workloads (SHAW)*, (New Orleans, LA, USA), Feb 2012.

[70] Lu, P. and Shen, K., "Virtual machine memory access tracing with hypervisor exclusive cache," in *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, (Berkeley, CA, USA), pp. 3:1–3:15, USENIX Association, 2007.

[71] Lukefahr, A., Padmanabha, S., Das, R., Sleiman, F. M., Dreslinski, R., Wenisch, T. F., and Mahlke, S., "Composite cores: Pushing heterogeneity into a core," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, (Washington, DC, USA), pp. 317–328, IEEE Computer Society, 2012.

[72] Miyoshi, A., Lefurgy, C., Van Hensbergen, E., Rajamony, R., and Rajkumar, R., "Critical power slope: understanding the runtime effects of frequency scaling," in *Proceedings of the 16th international conference on Supercomputing*, ICS '02, (New York, NY, USA), pp. 35–44, ACM, 2002.

[73] Mogul, J. C., Mudigonda, J., Binkert, N., Ranganathan, P., and Talwar, V., "Using asymmetric single-ISA CMPs to save energy on operating systems," *IEEE Micro*, vol. 28, pp. 26–41, May 2008.

[74] Nathuji, R., Isci, C., and Gorbatov, E., "Exploiting platform heterogeneity for power efficient data centers," in *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, (Washington, DC, USA), pp. 5–, IEEE Computer Society, 2007.

[75] Nathuji, R., Kansal, A., and Ghaffarkhah, A., "Q-clouds: managing performance interference effects for QoS-aware clouds," in *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, (New York, NY, USA), pp. 237–250, ACM, 2010.

[76] Nathuji, R. and Schwan, K., "VirtualPower: coordinated power management in virtualized enterprise systems," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, (New York, NY, USA), pp. 265–278, ACM, 2007.

[77] Neamtiu, I., "Elastic executions from inelastic programs," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, (New York, NY, USA), pp. 178–183, ACM, 2011.

[78] nvidia Corporation, "Variable SMP: A multi-core CPU architecture for low power and high performance." White paper, 2011.

[79] Ou, Z., Zhuang, H., Nurminen, J. K., Ylä-Jääski, A., and Hui, P., "Exploiting hardware heterogeneity within the same instance type of Amazon EC2," in *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, (Berkeley, CA, USA), pp. 4–4, USENIX Association, 2012.

[80] Padala, P., Hou, K.-Y., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., and Merchant, A., "Automated control of multiple virtualized resources," in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, (New York, NY, USA), pp. 13–26, ACM, 2009.

[81] Padala, P., Shin, K. G., Zhu, X., Uysal, M., Wang, Z., Singhal, S., Merchant, A., and Salem, K., "Adaptive control of virtualized resources in utility computing environments," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, (New York, NY, USA), pp. 289–302, ACM, 2007.

[82] Pallipadi, V. and Starikovskiy, A., "The ondemand governor: Past, present and future," *Linux Symposium*, vol. 2, pp. 223–238, 2006.

[83] Panneerselvam, S. and Swift, M. M., "Chameleon: operating system support for dynamic processors," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 99–110, ACM, 2012.

[84] Polfliet, S., Ryckbosch, F., and Eeckhout, L., "Optimizing the datacenter for data-centric workloads," in *Proceedings of the international conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 182–191, ACM, 2011.

[85] Poovey, J. A., Conte, T. M., Levy, M., and Gal-On, S., "A benchmark characterization of the EEMBC benchmark suite," *IEEE Micro*, vol. 29, pp. 18–29, Sept. 2009.

[86] Qureshi, M. and Loh, G., "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pp. 235–246, 2012.

[87] Rajamani, K., Hanson, H., Rubio, J., Ghiasi, S., and Rawson, F., "Application-aware power management," in *2006 IEEE International Symposium on Workload Characterization*, pp. 39–48, IEEE, Oct. 2006.

[88] Ramos, L. E., Gorbatov, E., and Bianchini, R., "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 85–95, ACM, 2011.

[89] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., "Evaluating MapReduce for multi-core and multiprocessor systems," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pp. 13–24, 2007.

[90] Reddy, D., Koufaty, D., Brett, P., and Hahn, S., "Bridging functional heterogeneity in multicore architectures," *SIGOPS Oper. Syst. Rev.*, vol. 45, pp. 21–33, Feb. 2011.

[91] RightScale, "RightScale Cloud Management." `http://www.rightscale.com`. [Online].

[92] Saez, J. C., Prieto, M., Fedorova, A., and Blagodurov, S., "A comprehensive scheduler for asymmetric multicore systems," in *5th EuroSys*, (New York, NY, USA), pp. 139–152, 2010.

[93] Shen, Z., Subbiah, S., Gu, X., and Wilkes, J., "CloudScale: elastic resource scaling for multi-tenant cloud systems," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, (New York, NY, USA), pp. 5:1–5:14, ACM, 2011.

[94] Snowdon, D., Le Sueur, E., Petters, S., and Heiser, G., "Koala: A platform for OS-level power management," in *Proceedings of the 4th ACM European conference on Computer systems*, pp. 289–302, ACM, 2009.

[95] Spiliopoulos, V., Kaxiras, S., and Keramidas, G., "Green governors: A framework for continuously adaptive DVFS," in *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2011.

[96] Srinivasan, S., Iyer, R., Zhao, L., and Illikkal, R., "HeteroScouts: hardware assist for OS scheduling in heterogeneous CMPs," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, pp. 341–342, June 2011.

[97] Suleman, M. A., Mutlu, O., Qureshi, M. K., and Patt, Y. N., "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, (New York, NY, USA), pp. 253–264, ACM, 2009.

[98] Suleman, M. A., Qureshi, M. K., and Patt, Y. N., "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 277–286, ACM, 2008.

[99] Van Craeynest, K., Jaleel, A., Eeckhout, L., Narvaez, P., and Emer, J., "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, (Washington, DC, USA), pp. 213–224, IEEE Computer Society, 2012.

[100] Vasić, N., Novaković, D., Miučin, S., Kostić, D., and Bianchini, R., "DejaVu: accelerating resource allocation in virtualized environments," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (New York, NY, USA), pp. 423–436, ACM, 2012.

[101] Venkatesh, G., Sampson, J., Goulding, N., Garcia, S., Bryksin, V., Lugo-Martinez, J., Swanson, S., and Taylor, M. B., "Conservation cores: reducing the energy of mature computations," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, (New York, NY, USA), pp. 205–218, ACM, 2010.

[102] Waldspurger, C. A., "Memory resource management in VMware ESX server," in *Proceedings of the 5th USENIX conference on Operating systems design and implementation*, OSDI'02, (Berkeley, CA, USA), USENIX Association, 2002.

[103] Wang, W., Liang, B., and Li, B., "Revenue maximization with dynamic auctions in IaaS cloud markets," in *Quality of Service (IWQoS), 2013 IEEE/ACM 21st International Symposium on*, pp. 1–6, 2013.

[104] Weissel, A. and Bellosa, F., "Process cruise control: event-driven clock scaling for dynamic power management," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '02, (New York, NY, USA), pp. 238–246, ACM, 2002.

[105] Wong, D. and Annavaram, M., "KnightShift: Scaling the energy proportionality wall through server-level heterogeneity," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, (Washington, DC, USA), pp. 119–130, IEEE Computer Society, 2012.

[106] Woo, D. H., Seong, N. H., Lewis, D., and Lee, H.-H., "An optimized 3D-stacked memory architecture by exploiting excessive, high-density TSV bandwidth," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pp. 1–12, 2010.

[107] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I., "Improving MapReduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[108] Zhao, W. and Wang, Z., "Dynamic memory balancing for virtual machines," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, (New York, NY, USA), pp. 21–30, ACM, 2009.

VITA

Vishal Gupta was born and grew up in Sri Ganganagar, Rajasthan, India. He earned his Bachelor of Technology degree in Computer Science & Engineering in 2006 from the Indian Institute of Technology Madras, India. Subsequently, he received his M.S. in Computer Science from the University of North Carolina at Chapel Hill in year 2008 before moving to the Georgia Institute of Technology, Atlanta for his Ph.D. During his doctoral studies at Georgia Tech, he was a member of the CERCS systems research group, working as a research assistant with his advisor Dr. Karsten Schwan. His research interests lie within systems software, particularly focused on virtualized and distributed systems. He currently works as a software developer with VMware, Inc.

PUBLICATIONS

A list of Vishal Gupta's publications from his doctoral studies:

1. Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "Core groups: System abstractions for extending the dynamic range of client devices using heterogeneous cores," *Sustainable Computing: Informatics and Systems*, vol. 3, no. 3, pp. 194 – 206, 2013.

2. Lee, M., Gupta, V., and Schwan, K., "Software-controlled transparent management of heterogeneous memory resources in virtualized systems," in *Proceedings of the 2013 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, (Seattle, WA, USA), ACM, June 2013.

3. Gupta, V. and Schwan, K., "Brawny vs. Wimpy: Evaluation and analysis of modern workloads on heterogeneous processors," in *Proceedings of the IEEE 22nd International Heterogeneity in Computing Workshop*, HCW'13, (Boston, MA, USA), IEEE Computer Society, May 2013.

4. Gupta, V. and Schwan, K., "PowerTune: Differentiated power allocation in over-provisioned multicore systems," in *Proceedings of the IEEE The Ninth Workshop on High-Performance, Power-Aware Computing*, HPPAC '13, (Boston, MA, USA), IEEE Computer Society, May 2013.

5. Gupta, V., Kim, H., and Schwan, K., "A power-performance analysis of memory-intensive parallel applications on a manycore platform," in *Proceedings of the 2012 19th International Conference on High Performance Computing: Student Research Symposium*, HIPC:SRS '12, (Pune, India), Dec 2012.

6. Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "HeteroMates: Providing high dynamic power range on client devices using heterogeneous core groups," in *Proceedings of the IEEE International Green Computing Conference (IGCC), 2012*, (San Jose, CA, USA) pp. 1–10, IEEE Computer Society, June 2012.

7. Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "The forgotten 'uncore': On the energy-efficiency of heterogeneous cores," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, (Boston, MA, USA), pp. 34–34, USENIX Association, June 2012.

8. Gupta, V., Brett, P., Koufaty, D., Reddy, D., Hahn, S., Schwan, K., and Srinivasa, G., "Extending the dynamic power range of client devices using heterogeneous multicore processors," in *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads*, SHAW-3, (New Orleans, LA, USA), Feb 2012.

9. Gupta, V., Nathuji, R., and Schwan, K., "An analysis of power reduction in datacenters using heterogeneous chip multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, no. 3, pp. 87–91, 2011. [Also accepted in ACM SIGMETRICS 2011 GreenMetrics Workshop (San Jose, CA, USA), June 2011].

10. Gupta, V. and Nathuji, R., "Analyzing performance asymmetric multicore processors for latency sensitive datacenter applications," in *Workshop on Power aware computing and systems*, HotPower'10, (Vancouver, BC, Canada), USENIX Association, Oct 2010.