Power, Performance, and Upheaval: An Opportunity for Managed Languages

Ting Cao

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

July 2014

© Ting Cao 2014

Except where otherwise indicated, this thesis is my own original work.

Ting Cao
28 July 2014

to the people who love me

Acknowledgments

There are many people and organizations I wish to acknowledge for their support and encouragement during the course of my PhD.

First I would like to thank my supervisors Prof. Steve Blackburn and Prof. Kathryn McKinley for their guidance and support. They are both very intelligent, inspirational, conscientious, generous, and truly considerate of students. This work would not have been possible without their help.

I would also like to express my gratitude to the Chinese Government and the Australian National University for the financial support. Many thanks to my undergraduate and master supervisor, Prof. Zhiying Wang, for his encouragement and support during my studies in China and my study overseas.

I would like to thank the members of the Computer System Group in the Research School of Computer Science, in particular Dr. Peter Strazdins, Prof. Alistair Rendell and Dr. Eric McCreath for their informative discussions and feedback.

I would also like to express my gratitude to my main student collaborators, Dr. Hadi Esmaeilzadeh, Xi Yang, Tiejun Gao, and Ivan Jibaja. I really enjoyed working with them and I have learnt a lot from them.

I am grateful to the support and help from everyone in our lab, Dr. John Zigman, Dr. Daniel Frampton, Xi Yang, Vivek Kumar, Rifat Shahriyer, Tiejun Gao, Yi Lin, Kunshan Wang, Brian Lee and James Bornholt. I feel very lucky to be able to work with these excellent and friendly fellows. Special thanks to Dr. John Zigman for his long term support of my work and for helping proofread my thesis.

Many thanks to my best friends in Australia, Dr. John Zigman, Tan Vo and Wensheng Liang. They care about me and are always there to provide assistance when I need it. Also, thanks to my friends Leilei Cao, Xinyue Han, Xu Xu, Jing Liu, Jie Liang, and Shi Wang. My life in Australia is much more colourful with their company.

Finally I would like to thank my family, especially my parents and grandparents for their positive role in my education and development. Thanks to my boyfriend Jiaxi Shi, who surprisingly is still my boyfriend after twelve years.

Abstract

Two significant revolutions are underway in computing. (1) On the hardware side, exponentially growing transistor counts in the same area, limited power budget and the breakdown of MOSFET voltage scaling are forcing power to be the first order constraint of computer architecture design. Data center electricity costs are billions of dollars each year in the U.S. alone. To address power constraints and energy cost, industry and academia propose Asymmetric Multicore Processors (AMP) that integrate general-purpose big (fast, high power) cores and small (slow, low power) cores. They promise to improve both single-thread performance and multi-threaded throughput with lower power and energy consumption. (2) On the software side, managed languages, such as Java and C#, and an entirely new software landscape of web applications have emerged. This has revolutionized how software is deployed, is sold, and interacts with hardware, from mobile devices to large servers. Managed languages abstract over hardware using Virtual Machine (VM) services (garbage collection, interpretation, and/or just-in-time compilation) that together impose substantial power and performance overheads. Thus, hardware support for managed software *and* managed software utilization of available hardware are critical and pervasive problems.

However, hardware and software researchers often examine the changes arising from these on going revolutions in isolation. Architects mostly grapple with microarchitecture design through the narrow software context of native sequential SPEC CPU benchmarks, while language researchers mostly consider microarchitecture in terms of performance alone. This dissertation explores the confluence of the two trends.

My thesis is that there exists a synergy between managed software and AMP architectures that can be automatically exploited to reduce VM overheads and deliver the efficiency promise of AMP architectures while abstracting over hardware complexity.

This thesis identifies a synergy between AMP and managed software, and addresses the challenge of exploiting it through the following three steps. (1) It first systematically measures and analyzes the power, performance, and energy characteristics of managed software compared to native software on current mainstream symmetric hardware, which motivates the next part of the thesis. (2) It next demonstrates that VM services fulfil the AMP workload requirements and tailored small cores for VM services deliver improvements in performance and energy. (3) Finally, it introduces a dynamic scheduling algorithm in the VM that manages parallelism, load balance and core sensitivity for efficiency.

This thesis is the first to quantitatively study measured power and performance

at the chip level across hardware generations using managed and native workloads, revealing previously unobserved hardware and software trends. This thesis proposes solutions to the 40% overhead of VM services by using tailored small cores in AMP architectures. This thesis introduces a new dynamic VM scheduler that offers transparency from AMP heterogeneity and substantial performance and energy improvements.

Those contributions show that the opportunities and challenges of AMP architectures and managed software are complementary. The conjunction provides a win-win opportunity for hardware and software communities now confronted with performance and power challenges in an increasingly complex computing landscape.

Contents

Acknowledgments **vii**

Abstract **ix**

1 Introduction **1**
1.1 Problem Statement . . . 1
1.2 Scope and Contributions . . . 2
1.3 Thesis Outline . . . 3

2 Background and Related Work **5**
2.1 Single-ISA Asymmetric Multicore Processors . . . 5
2.1.1 AMP Utilization . . . 6
2.2 Managed Language Virtual Machines . . . 7
2.2.1 Garbage Collector . . . 8
2.2.2 Just-In-Time Compiler . . . 9
2.2.3 Interpreter . . . 10
2.2.4 VM Overhead . . . 10
2.2.4.1 VM Performance Overhead Studies . . . 10
2.2.4.2 VM Energy Overhead Studies . . . 11
2.2.4.3 Hardware Support for GC . . . 12
2.3 Summary . . . 13

3 Power and Performance Characteristics for Language and Hardware **15**
3.1 Introduction . . . 15
3.2 Methodology . . . 16
3.2.1 Benchmarks . . . 16
3.2.1.1 Native Non-scalable Benchmarks . . . 18
3.2.1.2 Native Scalable Benchmarks . . . 19
3.2.1.3 Java Non-scalable Benchmarks . . . 19
3.2.1.4 Java Scalable Benchmarks . . . 20
3.2.2 Java Virtual Machines and Measurement Methodology . . . 20
3.2.3 Operating System . . . 21
3.2.4 Hardware Platforms . . . 21
3.2.5 Power Measurement . . . 23
3.2.6 Reference Execution Time, Reference Energy, and Aggregation . 23
3.2.7 Processor Configuration Methodology . . . 24
3.3 Perspective . . . 26

3.3.1 Power is Application Dependent 26
3.3.2 Historical Overview 28
3.3.3 Pareto Analysis at 45 nm 31
3.4 Feature Analysis 33
3.4.1 Chip Multiprocessors 34
3.4.2 Simultaneous Multithreading 36
3.4.3 Clock Scaling 38
3.4.4 Die Shrink 41
3.4.5 Gross Microarchitecture Change 43
3.4.6 Turbo Boost Technology 45
3.5 Summary 45

4 Asymmetric Multicore Processors and Managed Software 49
4.1 Introduction 49
4.2 Methodology 52
4.2.1 Hardware 53
4.2.2 Power and Energy Measurement 53
4.2.3 Hardware Configuration Methodology 55
4.2.3.1 Small Core Evaluation 56
4.2.3.2 Microarchitectural Characterization 56
4.2.4 Workload 56
4.2.5 Virtual Machine Configuration 57
4.2.5.1 GC 57
4.2.5.2 JIT 57
4.2.5.3 Interpreter 58
4.3 Motivation: Power and Energy Footprint of VM Services 58
4.4 Amenability of VM Services to a Dedicated Core 60
4.5 Amenability of VM services to Hardware Specialization 62
4.5.1 Small Core 62
4.5.2 Microarchitectural Characterization 64
4.5.2.1 Hardware Parallelism 64
4.5.2.2 Clock Speed 65
4.5.2.3 Memory Bandwidth 65
4.5.2.4 Last-level Cache Size 68
4.5.2.5 Gross Microarchitecture 68
4.5.3 Discussion 68
4.6 Modeling Future AMP Processors 69
4.7 Further Opportunity for the JIT 70
4.8 Summary 71

5 A VM Scheduler for AMP 73
5.1 Introduction 73
5.2 Workload Analysis & Characterization 75
5.3 Dynamically Identifying Workload Class 78

5.4 Speedup and Progress Prediction Model 80
5.5 The WASH Scheduling Algorithm 83
5.5.1 Overview 83
5.5.2 Single-Threaded and Low Parallelism WASH 84
5.5.3 Scalable Multithreaded WASH 85
5.5.4 Non-scalable Multithreaded WASH 86
5.6 Methodology 87
5.6.1 Hardware 87
5.6.2 Operating System 88
5.6.3 Workload 88
5.6.4 Virtual Machine Configuration 88
5.6.5 Measurement Methodology 88
5.7 Results 89
5.7.1 Single-Threaded Benchmarks 91
5.7.2 Scalable Multithreaded Benchmarks 91
5.7.3 Non-scalable Multithreaded Benchmarks 95
5.8 Summary 96

6 Conclusion **97**
6.1 Future Work 98
6.1.1 WASH-assisted OS Scheduling 98
6.1.2 Heterogeneous-ISA AMP and Managed Software 99

List of Figures

2.1 Basic Virtual Machine and structure. 8

3.1 Scalability of Java multithreaded benchmarks on i7 (45), comparing four cores with two SMT threads per core (4C2T) to one core with one SMT thread (1C1T). 20

3.2 The choice of JVM affects energy and performance. Benchmark time and energy on three different Java VMs on i7 (45), normalized to each benchmark's best result of the three VMs. A result of 1.0 reflect the best result on each axis. 21

3.3 Measured power for each processor running 61 benchmarks. Each point represents measured power for one benchmark. The '✗'s are the reported TDP for each processor. Power is application-dependent and does not strongly correlate with TDP. 28

3.4 Power / performance distribution on the i7 (45). Each point represents one of the 61 benchmarks. Power consumption is highly variable among the benchmarks, spanning from 23 W to 89 W. The wide spectrum of power responses from different applications points to power saving opportunities in software. 29

3.5 Power / performance tradeoff by processor. Each point is an average of the four workloads. Power per million transistor is consistent across different microarchitectures regardless of the technology node. On average, Intel processors burn around 1 Watt for every 20 million transistors. 30

3.6 Energy / performance Pareto frontiers (45 nm). The energy / performance optimal designs are application-dependent and significantly deviate from the average case. 32

3.7 CMP: Comparing two cores to one core. Doubling the cores is not consistently energy efficient among processors or workloads. 35

3.8 Scalability of single threaded Java benchmarks. Some single threaded Java benchmarks scale well. The underlying JVM exploits parallelism for compilation, profiling and GC. 36

3.9 SMT: one core with and without SMT. Enabling SMT delivers significant energy savings on the recent i5 (32) and the in-order Atom (45). . . . 37

3.10 Clock: doubling clock in stock configurations. Doubling clock does not increase energy consumption on the recent i5 (32). 39

3.11 Clock: scaling across the range of clock rates in stock configurations. Points are clock speeds. 40

3.12 Die shrink: microarchitectures compared across technology nodes. 'Core' shows Core 2D (65) / Core 2D (45) while 'Nehalem' shows i7 (45) / i5 (32) when two cores are enabled. Both die shrinks deliver substantial energy reductions. 42

3.13 Gross microarchitecture: a comparison of Nehalem with four other microarchitectures. In each comparison the Nehalem is configured to match the other processor as closely as possible. The most recent microarchitecture, Nehalem, is more energy efficient than the others, including the low-power Bonnell (Atom). 44

3.14 Turbo Boost: enabling Turbo Boost on i7 (45) and i5 (32). Turbo Boost is not energy efficient on the i7 (45). 46

4.1 Motivation: (a) VM services consume significant resources; (b) The naive addition of small cores *slows* down applications. 50

4.2 Hall effect sensor and PCI card on the Atom. 54

4.3 GC, JIT, and application power and energy on i7 4C2T at 3.4 GHz using Jikes RVM. The power demands of the GC and JIT are relatively uniform across benchmarks. Together they contribute about 20% to total energy. 59

4.4 Utility of adding a core dedicated to VM services on total energy, power, time, and PPE using Jikes RVM. Overall effect of binding GC, JIT, and both GC & JIT to the second core running at 2.8 GHz (dark), 2.2 GHz (middle), and 0.8 GHz (light) on the AMD Phenom II corrected for static power. The baseline uses one 2.8 GHz core. 61

4.5 Amenability of services and the application to an in-order-processor. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3. These results compare execution on an in-order Atom to an out-of-order i3. 63

4.6 Cycles executed as a function of clock speed, normalized to cycles at 1.6 GHz on the i7. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3. The workload is fixed, so extra cycles are due to stalls. 63

4.7 Microarchitectural characterization of VM services and application. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3. 66

4.8 Microarchitectural characterization of VM services and application. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3. 67

4.9 Modeling total energy, power, time, and PPE of an AMP system. The light bars show a model of an AMP with one 2.8 GHz Phenom II core and two Atom cores dedicated to VM services. The dark bars reproduce Phenom II results from Figure 4.4 that use a 2.8 GHz dedicated core for VM services. 69
4.10 Adjusting the JIT's cost-benefit model to account for lower JIT execution costs yields improvements in total application performance. 70

5.1 Linux OS scheduler (Oblivious) on homogeneous configurations of a Phenom II normalized to one big core. We classify benchmarks as single threaded, non-scalable multithreaded (MT), and scalable MT. Higher is better. 76
5.2 Execution time of Oblivious and bindVM on various AMP configurations, normalized to one 1B5S with Oblivious. Lower is better. 77
5.3 Fraction of time spent waiting on locks as a ratio of all cycles per thread in multithreaded benchmarks. The left benchmarks (purple) are scalable, the right five (pink) are not. Low ratios are highly predictive of scalability. 79
5.4 Accurate prediction of thread core sensitivity. Y-axis is predicted. X-axis is actual speedup. 82
5.5 All benchmarks: geomean time, power and energy with Oblivious, bindVM, and WASH on all three hardware configurations. Lower is better. 89
5.6 Normalized geomean time, power and energy for different benchmark groups. 90
5.7 Individual benchmark results on 1B5S. Lower is better. 92
5.8 Individual benchmark results on 2B4S. Lower is better. 93
5.9 Individual benchmark results on 3B3S. Lower is better. 94

List of Tables

3.1 Benchmark Groups; Source: SI: SPEC CINT2006, SF: SPEC CFP2006, PA: PARSEC, SJ: SPECjvm, D6: DaCapo 06-10-MR2, D9: DaCapo 9.12, and JB: pjbb2005. 17
3.2 Experimental error. Aggregate 95% confidence intervals for measured execution time and power, showing average and maximum error across all processor configurations, and all benchmarks. 18
3.3 The eight experimental processors and key specifications. 22
3.4 Average performance characteristics. The rank for each measure is indicated in small font. The machine list is ordered by release date. . . 25
3.5 Average power characteristics. The rank for each measure is indicated in small font. The machine list is ordered by release date. 25
3.6 Findings. We organize our discussion around these eleven findings from an analysis of measured chip power, performance, and energy on sixty-one workloads and eight processors. 27
3.7 Pareto-efficient processor configurations for each workload. Stock configurations are bold. Each '✔' indicates that the configuration is on the energy / performance Pareto-optimal curve. Native non-scalable has almost no overlap with any other workload. 32

4.1 Experimental processors. 53

5.1 Performance counters identified by PCA that most accurately predict thread core sensitivity. Intel Sandy Bridge simultaneously provides three fixed (F) and up to four other performance counters (V). AMD Phenom provides up to four performance counters. 81
5.2 Experimental processors. We demonstrate generality of core sensitivity analysis on both Intel and AMD processors. Intel supports various clock speeds, but all cores must run at the same speed. All scheduling results for performance, power, and energy use the Phenom II since it supports separate clocking of cores, mimicking an AMP. 87
5.3 Characterization of workload parallelism. Each of the benchmarks is classified as either single threaded, non-scalable or scalable. 88

Chapter 1

Introduction

This thesis addresses the problem of executing managed software on heterogeneous hardware efficiently and transparently.

1.1 Problem Statement

In the past decade, computer architecture design changed from being limited by transistors to being limited by power. While transistor numbers are still scaling up under Moore's law, the CMOS threshold voltage set by leakage current is not scaling down accordingly. As a consequence, under a limited power budget, the fraction of a chip that can be powered at full speed at one time is decreasing [Esmaeilzadeh et al., 2011a]. An obvious example is the use of Intel Turbo Boost technique [Intel, 2008]. The technique will power portions of the transistors on a chip at higher frequency while the other transistors are powered off. Another indication is that even as the transistor size has shrunk in each recent generation, the CPU frequency has not increased accordingly. Apart from power constraints, energy cost is also a serious issue in PC, server, and portable device markets. Globally, data centers are estimated currently to consume about US$30B worth of electricity per year [Piszczalski, 2012]. Power and energy problems in all market sectors are redefining the road for architecture development.

Heterogeneous hardware is recognised by academia and industry as a promising approach to exploit the abundant transistors of a chip for performance improvement under a limited power budget. It can accelerate the serial phases of applications on a few big cores and the parallel phases on many small cores. Some cores can be tailored for specific purposes. Kumar et al. model a single-ISA asymmetric multicore architecture [Kumar et al., 2003]. By running applications on the most appropriate cores, they reduce the total energy by 40%. Morad et al. show theoretically that single-ISA asymmetric multiprocessors can reduce power consumption by more than two thirds with similar performance compared to symmetric multiprocessors [Morad et al., 2006]. Vendors are building Asymmetric Multicore Processors (AMPs) already, such as ARM's big.LITTLE [Greenhalgh, 2011], Intel QuickIA [Chitlur et al., 2012], and NVIDIA Tegra 4 [NVIDIA, 2013]. However, heterogeneous processors expose hardware complexity to software developers. Without software support, the hetero-

geneous hardware will not be able to deliver on its promise.

The software community is facing orthogonal challenges of a similar magnitude, with major changes in the way software is deployed, sold, and interacts with hardware. Developers are increasingly choosing managed languages, sacrificing performance for programmer productivity, time-to-market, reliability, security, and portability. Smart phone and tablet applications are predominantly written in managed languages. Modern web services combine managed languages, such as PHP on the server side and JavaScript on the client side. In markets as diverse as financial software and cell phone applications, Java and .NET are the dominant choices. Until recently the performance overheads associated with managed languages were made tolerable by an exponential growth in sequential hardware performance. Unfortunately, this source of mitigation is drying up just as managed languages are becoming ubiquitous.

The hardware and software communities are thus both facing significant change and major challenges.

1.2 Scope and Contributions

The aim of my research is to mitigate the two major challenges happening in the software and hardware worlds—to explore the potential efficiency of heterogeneous processors while insulating software from this complexity.

There are generally two categories of heterogeneous architectures: those that integrate main cores with accelerators for specific domains (graphics, imaging, security, speech, etc.) and those that integrate general purpose cores which are normally based on the same ISA family but asymmetric in performance and power characteristics. Single-ISA AMPs allow threads to be scheduled between different core types, and do not necessarily force the redesign of existing software. The single-ISA AMP is the focus of my thesis. I use Java workloads as the representative of managed software, since Java has mature virtual machines and sophisticated benchmarks. While specific quantitative results may vary, the methodologies should be applicable to other managed languages.

Power measurement and analysis. To exploit the power, energy, and performance response of different hardware features running with various workloads on real processors, this thesis is the first to quantitatively study measured power and performance at the chip level across hardware generations, comparing managed against native workloads according to different hardware characteristics. The quantitative data reveals thirteen hardware or software findings. Two themes emerge: (1) energy efficient architecture design (i.e. less energy for the same task since a pure "race to finish" measure is not sufficient as it can use disproportionate amounts of power) is very sensitive to workload type (native or managed), and (2) each hardware feature elicits a different power and performance response. The variations in responses and the opportunity to mix them

motivate the exploration of managed software optimization combined with the use of AMPs.

AMPs and VM services. Researchers often examine the challenges of hardware and software in isolation. This thesis uses novel hardware/software co-design methodologies to solve the challenges together. Heterogeneous hardware will only be practical if it is transparent to application software. If every new generation of hardware requires application developers to change their code, developers are very unlikely to have the time or expertise to use it. This thesis shows that Virtual Machine (VM) services, such as garbage collection and just-in-time compilation, can exploit the potential efficiency of AMPs transparently to applications by running on the small cores. VM services understand and abstract the hardware. The tailored small cores of AMPs can hide the VM services overheads and improve total efficiency without any changes to applications.

Managed software scheduler for AMPs. To fully explore the efficiency of AMPs and abstract over the complexity, we consider the problem of scheduling managed application threads on AMPs. Managed applications consist of a *mess* of heterogeneous threads with different functions and amounts of work compared to classic scalable parallel workloads written in native languages. This thesis presents a dynamic scheduling algorithm called WASH (Workload Aware Scheduler for Heterogeneity). WASH can be integrated in managed language VMs to manage parallelism, load balance, and core sensitivity for both managed application and VM service threads. We show that WASH achieves substantial performance and energy improvements.

In summary, this thesis addresses the interaction of modern software with emerging hardware. It shows the potential of exploiting the VM abstraction layer to hide hardware complexity from applications, and at the same time exploiting the differentiated power and performance characteristics to substantially improve performance and energy efficiency.

1.3 Thesis Outline

The body of the thesis is structured around the three key contributions outlined above and starts with related work.

Chapter 2 discusses emerging AMPs and their potential for solving the power crisis. It presents the related work that proposes to deliver the potential efficiency of AMPs by scheduling threads to appropriate core types. It introduces a main source of overhead in managed software—VM services, as well as the relevant literature that attacks those overheads.

Chapters 3, 4, and 5 comprise the main body of the thesis, covering the three key contributions. Chapter 3 systematically analyzes the power, performance, and energy responses of managed software compared with native software, and a variety of hardware features across five technology generations. For native software, we

use SPEC CPU2006 and PARSEC as workloads. For managed software, we use SPECjvm, DaCapo and pjbb2005 as workloads. Chapter 4 identifies the complimentary relationship of AMP and VM services, which can hide each other's disadvantages—complexity and overheads, and exploit each other's advantages—efficiency and abstraction. Chapter 5 describes a new dynamic VM scheduler which schedules both managed application and VM threads to different core types of AMPs to improve efficiency. In both Chapter 4 and 5, we use DaCapo and pjbb2005 as managed workloads.

Finally, Chapter 6 concludes the thesis, summarizing how my contributions have identified, quantified, and addressed the challenges of managed software and AMP architectures. It further identifies key future research directions for emerging hardware and software.

Chapter 2

Background and Related Work

This thesis explores the interplay of Asymmetric Multicore Processors (AMPs) and managed software. This chapter provides the background information about the increasing necessity of AMP architectures and how to utilise them for efficiency. Section 2.1 introduces AMP architectures and the techniques to expose their potential energy and performance efficiency. It also explains how Virtual Machines (VMs) support managed software abstractions and ways to reduce the VM overheads. Section 2.2 provides an overview of VMs, with an emphasis on the three major services—the Garbage Collector (GC), the Just-In-Time compiler (JIT), and the interpreter.

2.1 Single-ISA Asymmetric Multicore Processors

This thesis focuses on single-ISA AMP architectures. This design consists of cores with the same ISA (instruction set architecture), but different microarchitectural features, speed, and power consumption. The single-ISA AMP architecture was proposed by Kumar et al. [2003] to reduce power consumption. The motivation is that different applications have different resource requirements during their execution. By choosing the most appropriate core from available cores from different points in the power and performance design space, specific performance and power requirements can be met. Morad et al. show that in theory single-ISA AMP architectures can reduce power consumption by more than two thirds with similar performance compared to symmetric multiprocessors [Morad et al., 2006]. Single-ISA AMP architectures can be constructed using a set of previously-designed processors with appropriately modified interfaces, thus reducing the design effort required. For example, Intel QuickIA integrates Xeon 5450 and Atom N330 as the big and small cores via the FSB (front side bus) [Chitlur et al., 2012]. ARM's big.LITTLE system connect the Cortex-A15 and Cortex-A7 as the big and small cores via the CCI-400 coherent interconnect [Greenhalgh, 2011]. Also, code can migrate among different cores without recompilation. However, AMP architectures cannot deliver on the promise of efficiency without software assistance to schedule threads to appropriate cores. The following subsection will discuss current proposals to utilize AMP architectures.

2.1.1 AMP Utilization

Several AMP schedulers are proposed in previous work. Most of them make scheduling decisions according to the thread's microarchitecture characteristics [Saez et al., 2011, 2012; Craeynest et al., 2012; Kumar et al., 2004; Becchi and Crowley, 2006], while others do not [Li et al., 2007; Mogul et al., 2008; Saez et al., 2010]. We will discuss them separately in the following text.

For those algorithms considering the thread's microarchitecture characteristics, the fundamental property for making scheduling decisions is speedup factor, that is, how much quicker a thread retires instructions on a fast core relative to a slow core [Saez et al., 2012]. Systems either measure or model the speedup factor. To directly determine speedup, the system runs threads on each core type, getting their IPC directly [Kumar et al., 2004; Becchi and Crowley, 2006]. To model the speedup, the system uses thread characteristics such as cache-miss rate, pipeline-stall rate and the CPU's memory-access latency. This information can be either gathered online by performance counters [Saez et al., 2011, 2012; Craeynest et al., 2012], or offline by using a reuse-distance profile [Shelepov et al., 2009]. By contrast to direct measurements, modelling does not need to migrate threads among different core types before making scheduling decisions.

Saez et al. gather memory intensity (cache miss rate) information online to model the speedup while they use an AMP platform with cores only differing in clock speed [Saez et al., 2011]. More recently, they consider an AMP composed of cores with different microarchitectures [Saez et al., 2012]. They change the retirement width of several out-of-order cores in a system to model the behaviour of in-order cores, while the other cores use their default settings. They use an additive regression technique from the WEKA machine learning package to find out the performance counters that contribute more significantly to the resulting speedup factor, which are IPC, LLC miss rate, L2 miss rate, execution stalls and retirement stalls. The modeled speedup will be the linear combination of those performance counter values with their additive-regression factors. Craeynest et al. consider cores with different microarchitecture too and use MLP and ILP information to estimate CPI for different core types [Craeynest et al., 2012]. They calculate MLP from LLC misses rate, big core reorder buffer size, and average data dependancy distance. They calculate ILP from instruction issue width and the probability of executing a certain number of instructions in a given cycle on the small cores.

There are several algorithms oblivious to the thread's microarchitecture characteristics. Li et al. propose a scheduling algorithm to ensure the load on each core type is proportional to its computing power and to ensure fast-core-first, which means threads run on fast cores whenever they are under-utilised [Li et al., 2007]. This algorithm does not consider which threads can benefit the most from the big cores' resources. Mogul et al. schedule frequently used OS functions to small cores [Mogul et al., 2008]. This algorithm can only benefit OS-intensive workloads. Saez et al. dynamically detect the parallelism of applications [Saez et al., 2010]. If the number of runnable threads of an application is higher than a threshold, the threads of this ap-

plication will primarily run on the slow cores. Otherwise those threads will primarily run on fast cores.

Whether or not the scheduling decisions can be changed while the threads run determines whether the AMP algorithm is *static* or *dynamic*. *Static* means threads will not be migrated again once decisions are made [Kumar et al., 2004; Shelepov et al., 2009; Saez et al., 2011], while *dynamic* adapts when thread behaviours change [Becchi and Crowley, 2006; Saez et al., 2012] or there is CPU load imbalance [Li et al., 2007]. For dynamic algorithms, the interval for sampling and migrating affects the performance, since migration can increase cache misses, especially for NUMA systems. Li et al. analyze the migration overhead and use a resident-set-based algorithm to predict the overhead before migration [Li et al., 2007].

Recent work also proposes to schedule critical sections to big cores in AMP architectures [Du Bois et al., 2013; Joao et al., 2012, 2013]. Du Bois et al. use a criticality time array with each entry for each thread to work out which one is the most critical thread [Du Bois et al., 2013]. After each time interval, the hardware will divide the time by active thread numbers in that period and add the value to the criticality time array. The thread with the highest value in the array will be the critical thread, and will be scheduled to the big cores. Joao et al. measure the cycles of threads having to wait for each bottleneck, and accelerate the most critical section on the big cores of the AMP architecture [Joao et al., 2012]. Their approach requires the programmer, compiler or library code to insert specific instructions to the source code, assisting hardware to keep track of the bottlenecks. Subsequently, they extend their work to detect threads that execute longer than other threads and put them on the big cores too [Joao et al., 2013]. They evaluate the results by using both a single multithreaded application and multiple multithreaded applications running concurrently. However, they always set the total number of threads equal to the number of cores.

The AMP scheduler proposed in this thesis schedules threads based on not only the number of threads, but also the scalable or non-scalable parallelism exhibited by those threads. It dynamically accelerates the critical thread and shares resources as appropriate. The prior algorithms do not consider scheduling appropriate threads on slower cores for energy efficiency, while our scheduler exploits to gain energy efficiency, especially for threads that are not on the critical path. Furthermore, none of the prior work considers application and VM service threads together.

2.2 Managed Language Virtual Machines

This section provides some background on modern virtual machines for managed languages. Figure 2.1 shows the basic VM structure. While executing managed software, the VM uses dynamic interpretation and JIT compilation to translate standardised portable bytecode to the binary code of the target physical machine. The JIT uses dynamic profiling to optimize frequently executed code for performance. The GC automatically reclaims memory not in use anymore. There are some other VM services too, such as a scheduler and finalizer. Most of those VM services are

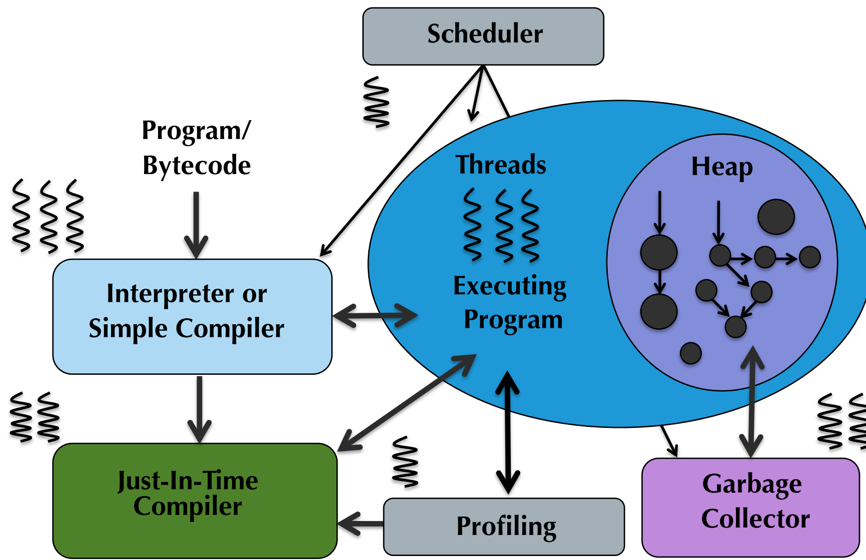


Figure 2.1: Basic Virtual Machine and structure.

multithreaded, asynchronous, non-critical and have their own hardware behaviour characteristics. This thesis will focus on the three main services: GC, JIT, and interpreter.

There are several mature Java VMs, such as Oracle's Hotspot, Oracle's JRockit, IBM's J9, and Jikes RVM [Alpern et al., 2005; The Jikes RVM Research Group, 2011]. This thesis mainly uses Oracle's Hotspot, the most widely used production JVM, and Jikes RVM, the most widely used research JVM.

2.2.1 Garbage Collector

Managed languages use GC to provide memory safety to applications. Programmers allocate *heap* memory and the GC automatically reclaims it when it becomes unreachable. GC algorithms are graph traversal problems, amenable to parallelization, and fundamentally memory-bound. There are several canonical GC algorithms, such as reference counting, mark-sweep, semi-space and mark-region. The following paragraphs will introduce the GCs used in this thesis.

Mark-region collectors divide the heap into fixed sized regions. The collectors allocate objects into free regions and reclaim regions containing no live objects. The best performing example of a mark-region algorithm is Immix [Blackburn and McKinley, 2008]. It uses a hierarchy of blocks and line regions, in which each block consists of some number of lines. Objects may cross lines but not blocks. Immix recycles partly used blocks by reclaiming at a fine-grained line granularity. It uses lightweight opportunistic evacuation to address fragmentation and achieves space efficiency, fast collection, and continuous allocation for application performance.

Production GCs normally divide the heap into spaces for objects of different ages, which are called generational GCs [Lieberman and Hewitt, 1983; Ungar, 1984; Moon,

1984]. The generational hypothesis states that most objects have very short lifetimes, therefore generational GCs can attain greater collection efficiency by focusing collection effort on the most recently allocated objects. The youngest generation is normally known as the *nursery* and the space containing the oldest objects is known as the *mature space*. Different collection policies can be applied to each generation. For example, Jikes RVM uses a generational collector with an evacuating nursery and an Immix mature space for its production configuration. The default collector for Oracle Hotspot uses an evacuating nursery, a pair of semi-spaces as the second generation, and a mark-sweep-compact mature space. Generational GCs are very effective. The majority of collectors in practical systems are generational.

There are *stop-the-world* and *concurrent* GCs, depending on whether the application execution halts or not during a collection. As the application halts and guarantees not to change the object graph while a collection, stop-the-world GC is simpler to implement and faster than concurrent GC. However, it is not suited to real-time or interactive programs where application pauses can be unacceptable. Concurrent GC is designed to reduce this disruption and improve application responsiveness. For concurrent GC, both collector and application threads are running simultaneously and synchronize only occasionally. Many concurrent collectors have been proposed, see for example [Ossia et al., 2002; Printezis and Detlefs, 2000; O'Toole and Nettles, 1984]. The concurrent mark-sweep collector in Jikes RVM uses a classic snapshot-at-the-beginning algorithm [Yuasa, 1990].

2.2.2 Just-In-Time Compiler

High performance VMs use a JIT to dynamically optimize code for frequently executed methods and/or traces. Because the code will have already executed at the time the optimizing JIT compiles it, the runtime has the opportunity to dynamically profile the code and tailor optimizations accordingly. The JIT will compile code asynchronously with the application and may compile more than one method or trace at once.

Compilation strategies may be incrementally more aggressive according to the heat of the target code. Thus, a typical JIT will have several optimization levels. The first level may apply register allocation and common sub-expression elimination and the second level applies optimizations that require more analysis, e.g., loop invariant code motion and loop pipelining. The compiler also performs feedback-directed optimizations. For example, it chooses which methods to inline at polymorphic call sites based on the most frequently executed target thus far. Similarly, it may perform type specialization based on the most common type, lay out code based on branch profiles, and propagate run-time constant values. These common-case optimizations either include code that handles the less frequent cases or that falls back to the interpreter or recompiles when the assumptions fail.

Each system uses a cost model that determines the expected cost to compile the code and the expected benefit. For example, the cost model used in Jikes RVM uses the past to predict the future [Arnold et al., 2000]. Offline, the VM first computes

compiler DNA: (1) the average compilation cost as a function of code features such as loops, lines of code, and branches, and (2) average improvements to code execution time due to the optimizations based on measurements on a large variety of code. The compiler assumes that if a method has executed for 10% of the time thus far, it will execute for 10% of the time in the future. At run time, the JIT recompiles a method at a higher level of optimization if the predicted cost to recompile it at that level and the reduction in the method execution time will reduce total time.

2.2.3 Interpreter

Many managed languages support dynamic loading and do not perform ahead-of-time compilation. The language runtime must consequently execute code immediately, as it is loaded. Modern VMs use interpretation, template compilation to machine code, or simple compilation without optimizations (all of which we refer to as interpretation for convenience). An interpreter is thus highly responsive but offers poor code quality. Advanced VMs will typically identify frequently executed code and dynamically optimize it using an optimizing compiler. In steady state, performance-critical code is optimized and the remaining code executes via the interpreter. One exception is the .NET framework for C#, which compiles all code with many optimizations immediately, only once, at load time. The interpreter itself is not parallel, but it will reflect any parallelism in the application it executes.

2.2.4 VM Overhead

This section will introduce the related work on studying overheads of VM services and how to reduce those overheads while improving managed software performance. Most of the works focus on improving the software algorithm or building specific hardware for that purpose. There are few works focusing on fulfilling the task through utilizing the available hardware efficiently.

2.2.4.1 VM Performance Overhead Studies

There have been many studies analyzing and optimizing the performance overheads of VM services since the earlier days of language VMs such as Lisp interpreter [McCarthy, 1978], SmallTalk VM [Deutsch and Schiffman, 1984] and others. Here we summarize some of the more recent relevant papers.

Arnold et al. describe the adaptive optimization system of Jikes RVM [Arnold et al., 2000]. The system uses low-overhead sampling technique to drive adaptive and online feedback-directed multilevel optimization. The overhead of this optimization system is 8.6% in start-up period and 6.0% for long-running period (accumulative timings of five runs for the same benchmark) using SPECjvm98.

Ha et al. introduce a concurrent trace-based JIT that use novel lock-free synchronization to trace, compile, install, and stitch traces on a separate core to improve responsiveness and throughput [Ha et al., 2009]. It also opens up the possibility of

increasing the code quality with compiler optimizations without sacrificing the application pause time. The paper shows the interpreter and JIT time in Tamarin-Tracing VM using the SunSpider JavaScript benchmark suite. When using sequential JIT, the compilation time ranges from 0.2% to 24.6% and the range for interpreter is 0.4% to 58.0%. By using concurrent JIT, the throughput can increase 5% on average and up to 36%.

Blackburn et al. give detailed performance studies of three whole heap GCs and generational counterparts: semi-space, mark-sweep and reference counting [Blackburn et al., 2004]. They measure the GC and application execution time for different benchmarks as a function of heap size for different GC algorithms. They also evaluate the impact of GC on application's cache locality. The conclusions from experiments include that (1) GC and total execution time are sensitive to heap size; (2) generational GC performs much better than their whole heap variants; and (3) the contiguous allocation of collectors attains significant locality benefits over free-list allocators.

2.2.4.2 VM Energy Overhead Studies

In contrast to the number of studies conducted into performance, there are fewer studies that evaluate power and energy overhead of VM services.

Chen et al. study mark-sweep GC using an energy simulator and the Shade SPARC simulator, which is configured to be a cacheless SoC [Chen et al., 2002]. They use embedded system benchmarks, ranging from utility programs used in hand-held devices to wireless web browser to game programs. Their results show that the GC costs 4% in total energy. They also develop a mechanism to improve leakage energy by using a GC-controlled optimization to shut off memory banks without live data.

Velasco et al. study the energy consumption of state-of-the-art GCs (e.g., mark-sweep, semi-space, and generational GC) and their impact on total energy cost for designing embedded systems [Velasco et al., 2005]. They use Jikes RVM, Dynamic SimpleScalar (DSS) [Huang et al., 2003], and combine DSS with a CACTI energy/delay/area model to calculate energy. Their energy simulation results follow the performance measurements from prior work [Blackburn et al., 2004]. They use SPECjvm98 benchmarks and divide the benchmarks into three scenarios: limited memory use, C-like memory use, and medium to high amounts of memory use. For the second scenario, the copying collector with mark-sweep gets the best energy results and the generational GC achieves the best for the third scenario. However, their results show GC costs 25% to 50% in the second and third scenarios, which is a big contrast with prior results.

Hu and John evaluate the performance and energy overhead of GC and JIT compilation, and their impact on application energy consumption on a hardware adaption framework [Hu et al., 2005] implemented with DSS and Jikes RVM [Hu and John, 2006]. In their evaluation, for SPECjvm benchmarks, JIT costs around 10% in total on average. GC costs depend on the heap size, ranging from 5% to 18% on average. By using the adaptive framework, they study the preferences of configurable units

(issue queue, reorder buffer, L1 and L2 caches) on the JIT and GC. Their results show that GCs prefer simple cores for energy efficiency. GCs can use smaller cache, issue queue, and reorder buffer with minimal performance impact. A JIT requires larger data caches than normal applications, but smaller issue queues and reorder buffers.

While the work described above uses simulators, our work measures the power and energy overhead of GC and JIT on real machines. The interpreter's power and energy overhead was not measured due to the sampling frequency limitations of the measurement hardware.

2.2.4.3 Hardware Support for GC

There exist many proposals for hardware supported GC [Moon, 1985; Ungar, 1987; Wolczko and Williams, 1992; Nilsen and Schmidt, 1994; Wise et al., 1997; Srisa-an et al., 2003; Meyer, 2004, 2005, 2006; Stanchina and Meyer, 2007b,a; Horvath and Meyer, 2010; Click et al., 2005; Click, 2009; Cher and Gschwind, 2008; Maas et al., 2012; Bacon et al., 2012]. The goals for this work are mainly to eliminate GC pauses and improve safety, reliability, and predictability. Some works aim to have a better use of the available hardware by offloading GC to it.

The first use of hardware to support GC was in Lisp machines [Moon, 1985]. In those machines, special microcode accompanies the implementation of each memory fetch or store operation. The worst-case latencies are improved, but the runtime overhead and the throughput are not improved. The following projects, Smalltalk on a RISC (SOAR) [Ungar, 1987] and Mushroom [Wolczko and Williams, 1992], target improving the throughput, but not the worst-case latencies of GC. At that time, since the target audience was small for such special-purpose architectures, major software developers did not consider it economical to port their products to specialized architectures.

Starting in the 1990s, to avoid the pitfalls of special purpose machines, researchers proposed active memory modules with hardware support for GC, including Nilsen and Schmidt's garbage collected memory module [Nilsen and Schmidt, 1994], Wise et al.'s reference count memory [Wise et al., 1997] and Srisa-an et al.'s active memory processor [Srisa-an et al., 2003]. This technology investment may be shared between users of many different processor architectures. However, the hardware cost for the memory module is relatively high and depends on the size of the garbage collected memory.

Meyer's group published a series of works exploring hardware support for GC [Meyer, 2004, 2005, 2006; Stanchina and Meyer, 2007b,a; Horvath and Meyer, 2010]. They initially developed a novel processor architecture with an object-based RISC main core and a small, microcoded on-chip GC coprocessor. Because of the tight coupling of processor and collector, all synchronization mechanisms for real-time GC are efficiently realized in hardware. Both read barrier checking and read barrier fault handling are entirely realized in hardware [Meyer, 2004, 2005, 2006]. Based on this initial system, they then introduced a hardware write barrier for generational GC to detect inter-generational pointers and execute all related book-keeping operations

entirely in hardware. The runtime overhead of generational GC is reduced to near zero [Stanchina and Meyer, 2007a]. In recent work, they developed a low-cost multi-core GC coprocessor to solve the synchronization problem for parallel GCs [Horvath and Meyer, 2010].

Azul Systems built a custom chip to run Java business applications [Click et al., 2005; Click, 2009]. They redesigned about 50% of the CPU and built their own OS and VM. The chips have special support for read and write barriers, fast user-mode trap handlers, cooperative pre-emption, and special TLB support, which enable a highly concurrent, parallel and compacting GC algorithm capable of handling very large heaps and large numbers of processors.

Several proposals use specialized processing units designed for something else to run GC. Cher and Gschwind offload the mark phase to the SPE co-processor on a Cell system [Cher and Gschwind, 2008]. Maas et al. offload GC to a GPU, motivated by the observation that consumer workloads often underutilize GPU and create an opportunity to offload some system tasks to GPU [Maas et al., 2012]. They show a new algorithm, and variations thereof, for performing the mark phase of a mark-sweep GC on the GPU by using a highly parallel queue-based breadth-first search. They use an AMD APU integrating CPU and GPU into a single device as a test platform. The performance result of this GPU-based GC turns out to be 40 to 80% slower than CPU-based collector, partly because of the large data transfer overhead between the CPU and GPU, and the limited memory space for the GPU.

In very recent work, Bacon et al. implement the first complete GC in hardware (as opposed to hardware-assist or microcode) [Bacon et al., 2012]. By using a concurrent snapshot algorithm and synthesising it into hardware, the collector provides single-cycle access to the heap and never stalls the mutator for a single cycle. Compared to the work before, the CPU does not need modification, and also the heap size does not need to be larger than the maximum live data. However, they trade flexibility in memory layout for large gains in collector performance. The shape of the objects (the size of the data fields and the location of pointers) is fixed. Results show this complete hardware GC achieves higher throughput and lower latency, memory usage, and energy consumption than stop-the-world collection.

Our work focuses more broadly on all VM services and in the context of AMP hardware, which is general purpose.

2.3 Summary

This chapter discussed how the prior work addresses the AMP architectures hardware complexity, as well as the overheads for managed software. In the following chapters, we will explain how to bring the two tracks together to hide each other's disadvantages and exploit their advantages. To more deeply understand the interaction of hardware and software in a modern setting, we start with measurements. We perform a detailed analysis of the power, energy and performance characteristics of managed as well as native software on a range of hardware features.

Chapter 3

Power and Performance Characteristics for Language and Hardware

To improve the efficiency of hardware supporting managed software and managed software utilising available hardware, we need to comprehensively understand the power, performance and energy characteristics of current hardware features and software. This chapter reports and analyzes measured chip power and performance on five process technology generations covering six hardware features (chip multiprocessors, simultaneous multithreading, clock scaling, die shrink, microarchitecture and turbo boost) executing a diverse set of benchmarks.

This chapter is structured as follows. Section 3.2 describes the hardware, workload, measurements, and software configuration. Section 3.3 examines the energy tradeoffs made by each processor over time and conducts a Pareto energy efficiency analysis to find out which hardware settings are more efficient for managed and native benchmarks. Section 3.4 explores the energy impact of hardware features.

This chapter describes work published in the paper "Looking Back on the Language and Hardware Revolutions: Measured Power, Performance, and Scaling" [Esmaeilzadeh, Cao, Yang, Blackburn, and McKinley, 2011b], in the paper "What is Happening to Power, Performance, and Software?" [Esmaeilzadeh, Cao, Yang, Blackburn, and McKinley, 2012a], and in the paper"Looking Back and Looking Forward: Power, Performance, and Upheaval" [Esmaeilzadeh, Cao, Yang, Blackburn, and McKinley, 2012b]. All the data are published in the ACM Digital Library as a companion to this paper [Esmaeilzadeh, Cao, Yang, Blackburn, and McKinley, 2011c]. In this collaboration, I was primarily responsible for managed software and hardware feature analysis.

3.1 Introduction

Quantitative performance analysis is the foundation for computer system design and innovation. In their classic paper, Emer and Clark noted that "A lack of detailed timing information impairs efforts to improve performance" [Emer and Clark, 1984].

They pioneered the quantitative approach by characterizing instruction mix and cycles per instruction on timesharing workloads. They surprised expert reviewers by demonstrating a gap between the theoretical 1 MIPS peak of the VAX-11/780 and the 0.5 MIPS it delivered on real workloads [Emer and Clark, 1998]. Industry and academic researchers in software and hardware all use and extend this principled performance analysis methodology. Our research applies this quantitative approach to measured power. This work is timely because the past decade heralded the era of power and energy constrained hardware design. A lack of detailed energy measurements is impairing efforts to reduce energy consumption on modern workloads.

Using controlled hardware configurations, we explore the energy impact of hardware features and workloads. We perform historical and Pareto analyses that identify the most power and performance efficient designs in our architecture configuration space. Our data quantifies a large number of workload and hardware trends with precision and depth, some known and many previously unreported. Our diverse findings include the following: (a) native sequential workloads do not approximate managed workloads or even native parallel workloads; (b) diverse application power profiles suggest that future applications and system software will need to participate in power optimization and management; and (c) software and hardware researchers need access to real measurements to optimize for power and energy.

3.2 Methodology

This section describes our benchmarks, compilers, Java Virtual Machines, operating system, hardware, and performance measurement methodologies.

3.2.1 Benchmarks

The following methodological choices in part prescribe our choice of benchmarks. (1) *Individual benchmark performance and average power:* We measure execution time and average power of individual benchmarks in isolation and aggregate them by workload type. While multi-programmed workload measurements, such as SPECrate can be valuable, the methodological and analysis challenges they raise are beyond the scope of this thesis. (2) *Language and parallelism:* We systematically explore native / managed, and scalable / non-scalable workloads. We create four benchmark groups in the cross product and weight each group equally.

Native Non-scalable: C, C++ and Fortran single threaded benchmarks from SPEC CPU2006.

Native Scalable: Multithreaded C and C++ benchmarks from PARSEC.

Java Non-scalable: Single and multithreaded benchmarks that do not scale well from SPECjvm, DaCapo 06-10-MR2, DaCapo 9.12, and pjbb2005 [Blackburn et al., 2006].

Java Scalable: Multithreaded benchmarks from DaCapo 9.12, selected because their performance scales similarly to Native Scalable on the i7 (45).

Grp	Src	Name	Time (s)	Description
Native Non-scalable	SI	perlbench	1037	Perl programming language
		bzip2	1563	bzip2 Compression
		gcc	851	C optimizing compiler
		mcf	894	Combinatorial opt/singledepot vehicle scheduling
		gobmk	1113	AI: Go game
		hmmer	1024	Search a gene sequence database
		sjeng	1315	AI: tree search & pattern recognition
		libquantum	629	Physics / Quantum Computing
		h264ref	1533	H.264/AVC video compression
		omnetpp	905	Ethernet network simulation based on OMNeT++
		astar	1154	Portable 2D path-finding library
		xalancbmk	787	XSLT processor for transforming XML
	SF	gamess	3505	Quantum chemical computations
		milc	640	Physics/quantum chromodynamics (QCD)
		zeusmp	1541	Physics/Magnetohydrodynamics based on ZEUS-MP
		gromacs	983	Molecular dynamics simulation
		cactusADM	1994	Cactus and BenchADM physics/relativity kernels
		leslie3d	1512	Linear-Eddy Model in 3D computational fluid dynamics
		namd	1225	Parallel simulation of large biomolecular systems
		dealII	832	PDEs with adaptive finite element method
		soplex	1024	Simplex linear program solver
		povray	636	Ray-tracer
		calculix	1130	Finite element code for linear and nonlinear 3D structural applications
		GemsFDTD	1648	Solves the Maxwell equations in 3D in the time domain
		tonto	1439	Quantum crystallography
		lbm	1298	Lattice Boltzmann Method for incompressible fluids
		sphinx3	2007	Speech recognition
Native Scalable	PA	blackscholes	482	Prices options with Black-Scholes PDE
		bodytrack	471	Tracks a markerless human body
		canneal	301	Minimizes the routing cost of a chip design with cache-aware simulated annealing
		facesim	1230	Simulates human face motions
		ferret	738	Image search
		fluidanimate	812	Fluid motion physics for realtime animation with SPH algorithm
		raytrace	1970	Uses physical simulation for visualization
		streamcluster	629	Computes an approximation for the optimal clustering of a stream of data points
		swaptions	612	Prices a portfolio of swaptions with the Heath-Jarrow-Morton framework
		vips	297	Applies transformations to an image
		x264	265	MPEG-4 AVC / H.264 video encoder
Java Non-scalable	SJ	compress	5.3	Lempel-Ziv compression
		jess	1.4	Java expert system shell
		db	6.8	Small data management program
		javac	3.0	The JDK 1.0.2 Java compiler
		mpegaudio	3.1	MPEG-3 audio stream decoder
		mtrt	0.8	Dual-threaded raytracer
		jack	2.4	Parser generator with lexical analysis
	D6	antlr	2.9	Parser and translator generator
		bloat	7.6	Java bytecode optimization and analysis tool
	D9	avrora	11.3	Simulates the AVR microcontroller
		batik	4.0	Scalable Vector Graphics (SVG) toolkit
		fop	1.8	Output-independent print formatter
		h2	14.4	An SQL relational database engine in Java
		jython	8.5	Python interpreter in Java
		pmd	6.9	Source code analyzer for Java
		tradebeans	18.4	Tradebeans Daytrader benchmark
		luindex	2.4	A text indexing tool
	JB	pjbb2005	10.6	Transaction processing, based on SPECjbb2005
Java Scalable	D9	eclipse	50.5	Integrated development environment
		lusearch	7.9	Text search tool
		sunflow	19.4	Photo-realistic rendering system
		tomcat	8.6	Tomcat servlet container
		xalan	6.9	XSLT processor for XML documents

Table 3.1: Benchmark Groups; Source: SI: SPEC CINT2006, SF: SPEC CFP2006, PA: PARSEC, SJ: SPECjvm, D6: DaCapo 06-10-MR2, D9: DaCapo 9.12, and JB: pjbb2005.

	Execution Time		Power	
	average	max	average	max
Average	1.2%	2.2%	1.5%	7.1%
Native Non-scalable	0.9%	2.6%	2.1%	13.9%
Native Scalable	0.7%	4.0%	0.6%	2.5%
Java Non-scalable	1.6%	2.8%	1.5%	7.7%
Java Scalable	1.8%	3.7%	1.7%	7.9%

Table 3.2: Experimental error. Aggregate 95% confidence intervals for measured execution time and power, showing average and maximum error across all processor configurations, and all benchmarks.

Native and managed applications embody different tradeoffs between performance, reliability and portability. In this setting, it is impossible to meaningfully separate language from workload. We therefore offer no commentary on the *virtue* of a language choice, but rather, reflect the measured reality of two workload classes that are ubiquitous in today's software landscape.

We draw 61 benchmarks from six suites to populate these groups. We weight each group equally in our aggregate measurements; see Section 3.2.6 for more details on aggregation. We use Java to represent the broader class of managed languages because of its mature VM technology and benchmarks. Table 3.1 shows the benchmarks, their groupings, the suite of origin, the *reference running time* (see Section 3.2.6) to which we normalize our results, and a short description. In the case of native benchmarks, all single threaded benchmarks are non-scalable and all parallel multithreaded native benchmarks are scalable up to eight hardware contexts, the maximum we explore. By scalable, we mean that adding hardware contexts improves performance. Bienia et al. also find that the PARSEC benchmarks scale up to 8 hardware contexts [Bienia et al., 2008]. To create a comparable group of scalable Java programs, we put multithreaded Java programs that do not scale well in the non-scalable category.

3.2.1.1 Native Non-scalable Benchmarks

We use 27 C, C++, and Fortran codes from the SPEC CPU2006 suite [Standard Performance Evaluation Corporation, 2006] for Native Non-scalable and all are *single threaded*. The 12 SPEC CINT benchmarks represent compute-intensive integer applications that contain sophisticated control flow logic, and the 15 SPEC CFP benchmarks represent compute-intensive floating-point applications. These native benchmarks are compiled ahead-of-time. We chose Intel's *icc* compiler because we found that it consistently generated better performing code than *gcc*. We compiled all of the Native Non-scalable benchmarks with version 11.1 of the 32-bit Intel compiler suite using the `-o3` optimization flag, which performs aggressive scalar optimizations. We used the 32-bit compiler suite because the 2003 Pentium 4 (130) does not support 64-

bit. This flag does not include any automatic parallelization. We compiled each benchmark once, using the default Intel compiler configuration, without setting any microarchitecture-specific optimizations, and used the *same* binary on all platforms. We exclude 410.bwaves and 481.wrf because they failed to execute when compiled with the Intel compiler. Three executions are prescribed by SPEC. We report the mean of these three successive executions. Table 3.2 shows that aggregate 95% confidence intervals are low for execution time and power: 1.2% and 1.5% respectively.

3.2.1.2 Native Scalable Benchmarks

The Native Scalable benchmarks consists of 11 C and C++ benchmarks from the PARSEC suite [Bienia et al., 2008]. The benchmarks are intended to be diverse and forward looking parallel algorithms. All but one uses POSIX threads and one contains some assembly code. We exclude freqmine because it is not amenable to our scaling experiments, in part, because it does not use POSIX threads. We exclude dedup from our study because it has a large working set that exceeds the amount of memory available on the 2003 Pentium 4 (130). The multithreaded PARSEC benchmarks include *gcc* compiler configurations, which worked correctly. The icc compiler failed to produce correct code for many of the PARSEC benchmarks with similar configurations. Consequently we used the PARSEC default gcc build scripts with gcc version 4.4.1. The gcc scripts use `-O3` optimization. We report the mean of five successive executions of each benchmark. We use five executions, which as Table 3.2 shows, gives low aggregate 95% confidence intervals for execution time and power: 0.9% and 2.1% on average.

3.2.1.3 Java Non-scalable Benchmarks

The Java Non-scalable group includes benchmarks from SPECjvm, both releases of DaCapo, and pjbb2005 that do not scale well. It includes both single threaded and multithreaded benchmarks. SPECjvm is intended to be representative of client-side Java programs. Although the SPECjvm benchmarks are over ten years old and Blackburn et al. have shown that they are simple and have a very small instruction cache and data footprint [Blackburn et al., 2006], many researchers still use them. The DaCapo Java benchmarks are intended to be diverse, forward-looking, and non-trivial [Blackburn et al., 2006; The DaCapo Research Group, 2006]. The benchmarks come from major open source projects under active development. Researchers have not reported extensively on the 2009 release, but it was designed to expose richer behavior and concurrency on large working sets. We exclude tradesoap because its heavy use of sockets suffered from timeouts on the slowest machines. We use pjbb2005, which is a fixed-workload variant of SPECjbb2005 [Standard Performance Evaluation Corporation, 2010] that holds the workload, instead of time, constant. We configure pjbb2005 with 8 warehouses and 10,000 transactions per warehouse. We include the following multithreaded benchmarks in Java Non-scalable: pjbb2005, avrora, batik, fop, h2, jython, pmd, and tradebeans. As we show below, these applications do not scale well.

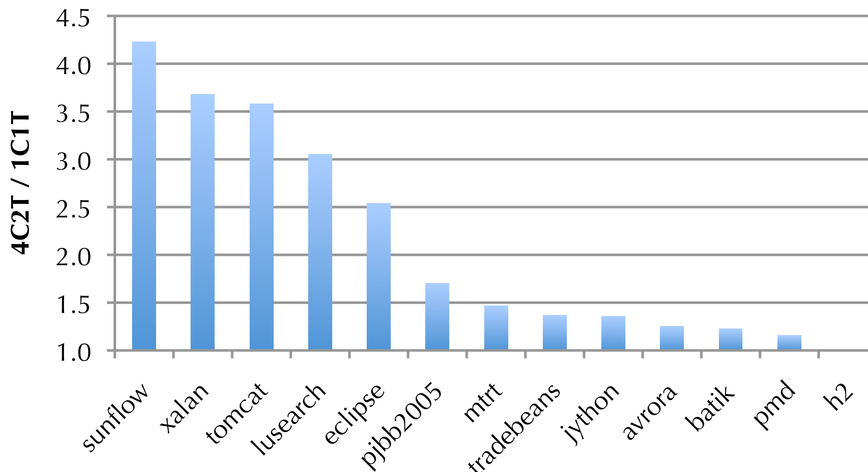


Figure 3.1: Scalability of Java multithreaded benchmarks on i7 (45), comparing four cores with two SMT threads per core (4C2T) to one core with one SMT thread (1C1T).

Section 3.2.2 discusses the measurement methodology for Java. Table 3.2 indicates low aggregate 95% confidence intervals for execution time and power: 1.6% and 1.5%.

3.2.1.4 Java Scalable Benchmarks

The Java Scalable group includes the multithreaded Java benchmarks that scale similarly to the Native Scalable benchmarks. Figure 3.1 shows the scalability of the multithreaded Java benchmarks. The five most scalable are: sunflow, xalan, tomcat, lusearch and eclipse, all from DaCapo 9.12. Together, they speed up on average by a factor of 3.4 given eight hardware contexts compared to one context on the i7 (45). Our Native Scalable benchmarks scale better on this hardware, improving by a factor of 3.8. Although eliminating lusearch and eclipse would improve average scalability, it would reduce the number of benchmarks to three, which we believe is insufficient. Table 3.2 shows low aggregate 95% confidence intervals for execution time and power: 1.8% and 1.7%.

3.2.2 Java Virtual Machines and Measurement Methodology

We report results using an Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 VM. We did some additional experiments with Oracle JRockit build R28.0.0-679-130297 and IBM J9 build pxi3260sr8. Figure 3.2 shows the difference of running time and energy for Java benchmarks on three different VMs on an i7 (45) (each point is one benchmark). Exploring the influence of the choice of native compiler and JVM on power and energy is an interesting avenue for future research.

To measure both Java Non-scalable and Java Scalable, we follow the recommended methodologies for measuring Java [Georges et al., 2007; Blackburn et al., 2008]. We use the `-server` flag and fix the heap size at a generous 3× the minimum required

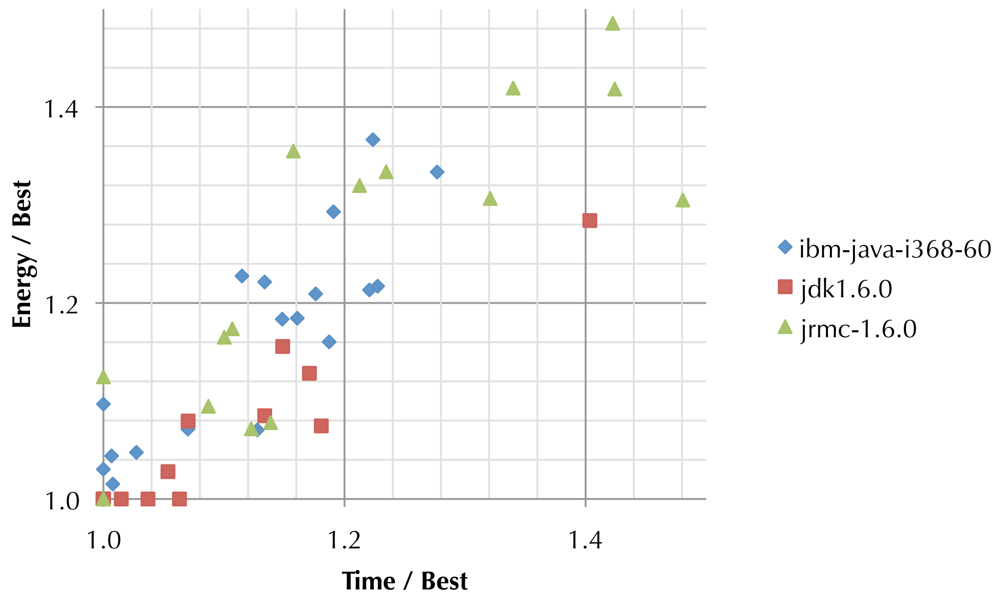


Figure 3.2: The choice of JVM affects energy and performance. Benchmark time and energy on three different Java VMs on i7 (45), normalized to each benchmark's best result of the three VMs. A result of 1.0 reflect the best result on each axis.

for each benchmark. We did not set any other JVM flags. We report the fifth iteration of each benchmark within a single invocation of the JVM to capture steady state behavior. This methodology avoids class loading and heavy compilation activity that often dominates the early phases of execution. The fifth iteration may still have a small amount of compiler activity, but has sufficient time to optimize frequently executed code. We perform this process twenty times and report the mean. Table 3.2 reports the measured error. We require twenty invocations to generate a statistically stable result because the adaptive JIT and GC induce non-determinism. In contrast to the compiled ahead-of-time native configurations, Java compilers may dynamically produce microarchitecture-specific code.

3.2.3 Operating System

We perform all the experiments using 32-bit Ubuntu 9.10 Karmic with the 2.6.31 Linux kernel. We use a 32-bit OS and compiler builds because the 2003 Pentium 4 (130) does not support 64-bit. Exploring the impact of word size is also interesting future work.

3.2.4 Hardware Platforms

We use eight IA32 processors, manufactured by Intel using four process technologies (130 nm, 65 nm, 45 nm, and 32 nm), representing four microarchitectures (NetBurst, Core, Bonnell, and Nehalem). Table 3.3 lists processor characteristics: uniquely iden-

	Pentinum 4 HT 2.4C	Core 2 Duo E6600	Core 2 Quad Q6600	Atom 230	Core i7 920	Core 2 Duo E7600	Atom D510	Core i5 670
μArch	NetBurst	Core	Core	Bonnell	Nehalem	Core	Bonnell	Nehalem
Processor	Northwood	Conroe	Kentsfield	Diamondville	Bloomfield	Wolfdale	Pineview	Clarkdale
sSpec	SL6WF	SL9S8	SL9UM	SLB6Z	SLBCH	SLGTD	SLBLA	SLBLT
Release Date	May '03	Jul '06	Jan '07	Jun '08	Nov '08	May '09	Dec '09	Jan '10
Price (USD)	—	$316	$851	$29	$284	$133	$63	$284
CMP/SMT	1C2T	2C1T	4C1T	1C2T	4C2T	2C1T	2C2T	2C2T
LLC (B)	512 K	4 M	8 M	512 K	8 M	3 M	1 M	4 M
Clock (GHz)	2.4	2.4	2.4	1.7	2.7	3.1	1.7	3.4
Tech (nm)	130	65	65	45	45	45	45	32
Trans (M)	55	291	582	47	731	228	176	382
Die (mm^2)	131	143	286	26	263	82	87	81
VID (V)	—	0.85 - 1.50	0.85 - 1.50	0.90 - 1.16	0.80 - 1.38	0.85 - 1.36	0.80 - 1.17	0.65 - 1.40
TDP (W)	66	65	105	4	130	65	13	73
FSB (MHz)	800	1066	1066	533	—	1066	665	—
B/W (GB/s)	—	—	—	—	25.6	—	—	21.0
DRAM	DDR-400	DDR2-800	DDR2-800	DDR2-800	DDR3-1066	DDR2-800	DDR2-800	DDR3-1333

Table 3.3: The eight experimental processors and key specifications.

tifying sSpec number, release date / price; CMP and SMT (*n*C*m*T means *n* cores, *m* SMT threads per core), die characteristics; and memory configuration. Intel sells a large range of processors for each microarchitecture—the processors we use are just samples within that space. Most of our processors are mid-range desktop processors. The release date and release price in Table 3.3 provides the context regarding Intel's placement of each processor in the market. The two Atoms and the Core 2Q (65) *Kentsfield* are extreme points at the bottom and top of the market respectively.

3.2.5 Power Measurement

In contrast to whole system power studies [Isci and Martonosi, 2003; Bircher and John, 2004; Le Sueur and Heiser, 2010], we measure on-chip power. Whole system studies measure AC current to an entire computer, typically with a clamp ammeter. To measure on-chip power, we must isolate and measure DC current to the processor on the motherboard, which cannot be done with a clamp ammeter. We use *Pololu's ACS714* current sensor board, following prior methodology [Pallipadi and Starikovskiy, 2006]. The board is a carrier for *Allegro's* $\pm 5\,A$ *ACS714* Hall effect-based linear current sensor. The sensor accepts a bidirectional current input with a magnitude up to $5\,A$. The output is an analog voltage ($185\,mV/A$) centered at $2.5\,V$ with a typical error of less than 1.5%. The sensor on i7 (45), which has the highest power consumption, accepts currents with magnitudes up to $30\,A$.

Each of our experimental machines has an isolated power supply for the processor on the motherboard, which we verified by examining the motherboard specification and confirmed empirically. This requirement precludes measuring, for example the Pentium M, which would have given us a 90 nm processor. We place the sensors on the $12\,V$ power line that supplies only the processor. We experimentally measured voltage and found it was very stable, varying less than 1%. We send the measured values from the current sensor to the measured machine's USB port using *Sparkfun's Atmel AVR Stick*, which is a simple data-logging device. We use a data-sampling rate of 50 Hz. We execute each benchmark, log its measured power values, and then compute the average power consumption over the duration of the benchmark.

To calibrate the meters, we use a current source to provide 28 reference currents between 300 mA and 3 A, and for each meter record the output value (an integer in the range 400-503). We compute linear fits for each of the sensors. Each sensor has an R^2 value of 0.999 or better, which indicates an excellent fit. The measurement error for any given sample is about 1%, which reflects the fidelity of the quantization (103 points).

3.2.6 Reference Execution Time, Reference Energy, and Aggregation

As is standard, we weight each benchmark equally within each workload group, since the execution time of the benchmark is not necessarily an indicator of benchmark importance. For example, the benchmark running time for SPEC CPU2006 and PARSEC can be two orders of magnitude longer than DaCapo. Furthermore, we want to

represent each of our benchmark groups equally. These goals require (1) a reference execution time and a reference energy value for each benchmark for normalization, and (2) an average of the benchmarks in each workload group. Since the average power of a benchmark is not directly biased by execution time, we use it directly in the power analysis. We also normalize energy to a reference, since energy is the integral of power over time and practically we use the average of the sampled power multiplied by time to calculate energy.

Table 3.1 shows the reference running time we use to normalize the execution time and energy results. As we need to fairly evaluate each machine, not their behaviour relative to a specific machine, to avoid biasing performance measurements to the strengths or weaknesses of one architecture, we normalize individual benchmark execution times to its average execution time executing on four architectures. We choose the Pentium 4 (130), Core 2D (65), Atom (45), and i5 (32) to capture all four microarchitectures and all four technology generations in this study. The reference energy is the average power on these four processors times the average runtime instead of the average energy of those four processors to be consistent throughout. Given a power and time measurement, we compute energy and then normalize it to the reference energy.

Table 3.1 shows that the native workloads tend to execute for much longer than the managed workloads. Measuring their code bases is complicated because of the heavy use of libraries by the managed languages and by PARSEC. However, some native benchmarks are tiny and many PARSEC codes are fewer than 3000 lines of non-comment code. These estimates show that the size of the native and managed application code bases alone (excluding libraries) does not explain the longer execution times. There is no evidence that native execution times are due to more sophisticated applications; instead these longer execution times are often due to more repetition.

The averages equally weight each of the four benchmark groups. We report results for each group by taking the arithmetic mean of the benchmarks within the group. We use the mean of the four groups for the overall average. This aggregation avoids bias due to the varying number of benchmarks within each group (from 5 to 27). Table 3.4 and Table 3.5 show the normalized performance and the measured power for each of the processors and each of the benchmark groups. The table indicates the weighted average (Avg^w), which is the average of the four groups and we use throughout the chapter, and for comparison, the simple average of all of the benchmarks (Avg^b). The table also records the highest and lowest performance and power measures seen on each of the processors.

3.2.7 Processor Configuration Methodology

We evaluate the eight stock processors and configure them for a total of 45 processor configurations. We produce power and performance data for each benchmark for each configuration, which is available for downloading at the ACM Digital Library [Esmaeilzadeh, Cao, Yang, Blackburn, and McKinley, 2011c]. To explore the influence of architectural features, we control for clock speed and hardware contexts.

Processor	Speedup Over Reference							
	NN	NS	JN	JS	Avg^w	Avg^b	Min	Max
Pentium 4	0.91 $_{6}$	0.79 $_{7}$	0.80 $_{6}$	0.75 $_{7}$	0.82 $_{6}$	0.85 $_{6}$	0.51 $_{6}$	1.25 $_{6}$
Core 2 Duo E6600	2.02 $_{5}$	2.10 $_{5}$	1.99 $_{5}$	2.04 $_{5}$	2.04 $_{5}$	2.03 $_{5}$	1.40 $_{4}$	2.85 $_{5}$
Core 2 Quad Q6600	2.04 $_{4}$	3.62 $_{3}$	2.04 $_{4}$	3.09 $_{3}$	2.70 $_{3}$	2.41 $_{4}$	1.39 $_{5}$	4.67 $_{3}$
Atom 230	0.49 $_{8}$	0.52 $_{8}$	0.53 $_{8}$	0.52 $_{8}$	0.52 $_{8}$	0.51 $_{8}$	0.39 $_{8}$	0.75 $_{8}$
Core i7 920	3.11 $_{2}$	6.25 $_{1}$	3.00 $_{2}$	5.49 $_{1}$	4.46 $_{1}$	3.84 $_{1}$	2.16 $_{2}$	7.60 $_{1}$
Core 2 Duo E7600	2.48 $_{3}$	2.76 $_{4}$	2.49 $_{3}$	2.44 $_{4}$	2.54 $_{4}$	2.53 $_{3}$	1.45 $_{3}$	3.71 $_{4}$
Atom D510	0.53 $_{7}$	0.96 $_{6}$	0.61 $_{7}$	0.86 $_{6}$	0.74 $_{7}$	0.66 $_{7}$	0.41 $_{7}$	1.17 $_{7}$
Core i5 670	3.31 $_{1}$	4.46 $_{2}$	3.18 $_{1}$	4.26 $_{2}$	3.80 $_{2}$	3.56 $_{2}$	2.39 $_{1}$	5.42 $_{2}$

Table 3.4: Average performance characteristics. The rank for each measure is indicated in small font. The machine list is ordered by release date.

Processor	Power (W)							
	NN	NS	JN	JS	Avg^w	Avg^b	Min	Max
Pentium 4	42.1 $_{7}$	43.5 $_{6}$	45.1 $_{7}$	45.7 $_{6}$	44.1 $_{6}$	43.5 $_{7}$	34.5 $_{7}$	50.0 $_{6}$
Core 2 Duo E6600	24.3 $_{5}$	26.6 $_{4}$	26.2 $_{5}$	28.5 $_{4}$	26.4 $_{5}$	25.6 $_{5}$	21.4 $_{5}$	32.3 $_{4}$
Core 2 Quad Q6600	50.7 $_{8}$	61.7 $_{8}$	55.3 $_{8}$	64.6 $_{8}$	58.1 $_{8}$	55.2 $_{8}$	45.6 $_{8}$	77.3 $_{7}$
Atom 230	2.3 $_{1}$	2.5 $_{1}$	2.3 $_{1}$	2.4 $_{1}$	2.4 $_{1}$	2.3 $_{1}$	1.9 $_{1}$	2.7 $_{1}$
Core i7 920	27.2 $_{6}$	60.4 $_{7}$	37.5 $_{6}$	62.8 $_{7}$	47.0 $_{7}$	39.1 $_{6}$	23.4 $_{6}$	89.2 $_{8}$
Core 2 Duo E7600	19.1 $_{3}$	21.1 $_{3}$	20.5 $_{3}$	22.6 $_{3}$	20.8 $_{3}$	20.2 $_{3}$	15.8 $_{3}$	26.8 $_{3}$
Atom D510	3.7 $_{2}$	5.3 $_{2}$	4.5 $_{2}$	5.1 $_{2}$	4.7 $_{2}$	4.3 $_{2}$	3.4 $_{2}$	5.9 $_{2}$
Core i5 670	19.6 $_{4}$	29.2 $_{5}$	24.7 $_{4}$	29.5 $_{5}$	25.7 $_{4}$	23.6 $_{4}$	16.5 $_{4}$	38.2 $_{5}$

Table 3.5: Average power characteristics. The rank for each measure is indicated in small font. The machine list is ordered by release date.

We selectively down-clock the processors, disable cores, disable simultaneous multi-threading (SMT), and disable Turbo Boost [Intel, 2008]. Intel markets SMT as Hyper-Threading [Intel Corporation, 2011]. The stock configurations of Pentium 4 (130), Atom (45), Atom D (45), i7 (45), and i5 (32) include SMT (Table 3.3). The stock configurations of the i7 (45) and i5 (32) include Turbo Boost, which automatically increases frequency beyond the base operating frequency when the core is operating below power, current, and temperature thresholds [Intel, 2008]. We control each variable via the BIOS. We experimented with operating system configuration, which is far more convenient, but it was not sufficiently reliable. For example, operating system scaling of hardware contexts often caused power consumption to increase as hardware resources were decreased! Extensive investigation revealed a bug in the Linux kernel [Li, 2011]. We use all the means at our disposal to isolate the effect of various architectural features using stock hardware, but often the precise semantics are undocumented. Notwithstanding such limitations, these processor configurations help quantitatively explore how a number of features influence power and performance in real processors.

3.3 Perspective

We organize our analysis into eleven findings, which we list in Table 3.6. We begin with broad trends. We first show that applications exhibit a large range of power and performance characteristics that are not well summarized by a single number. This section then conducts a Pareto energy efficiency analysis for all of the 45 nm processor configurations. Even with this modest exploration of architectural features, the results indicate that each workload prefers a different hardware configuration for energy efficiency.

3.3.1 Power is Application Dependent

The nominal thermal design power (TDP) for a processor is the amount of power the chip may dissipate without exceeding the maximum transistor junction temperature. Table 3.3 lists TDP for each processor. Because measuring real processor power is difficult and TDP is readily available, TDP is often substituted for real measured power [Chakraborty, 2008; Hempstead et al., 2009; Horowitz et al., 2005]. Figure 3.3 shows that this substitution is problematic. It plots on a logarithmic scale, measured power for each benchmark on each stock processor as a function of TDP and indicates TDP with an ✗. TDP is strictly higher than actual power. The gap between peak measured power and TDP varies from processor to processor and TDP is up to a factor of four higher than measured power. The variation among benchmarks is highest on the i7 (45) and i5 (32), likely reflecting their advanced power management. For example on the i7 (45), measured power varies between 23 W for 471.omnetpp and 89W for fluidanimate! The smallest variation between maximum and minimum is on the Atom (45) at 30%. This trend is not new. All the processors exhibit a range of benchmark-specific power variation. TDP loosely correlates

Findings

Power consumption is highly application dependent and is poorly correlated to TDP.

Power per transistor is relatively consistent within microarchitecture family, independent of process technology.

Energy-efficient architecture design is very sensitive to workload. Configurations in the Native Non-scalable *Pareto Frontier substantially differ from all the other workloads.*

Comparing one core to two, enabling a core is not consistently energy efficient.

The JVM induces parallelism into the execution of single threaded Java benchmarks.

Simultaneous multithreading (SMT) delivers substantial energy savings for recent hardware and for in-order processors.

The most recent processor in our study does not consistently increase energy consumption as its clock increases.

The power / performance response to clock scaling of Native Non-scalable *differs from the other workloads.*

Two recent die shrinks deliver similar and surprising reductions in energy, even when controlling for clock frequency.

Controlling for technology, hardware parallelism, and clock speed, the out-of-order architectures have similar energy efficiency as the in-order ones.

Turbo Boost is not energy efficient on the i7 (45).

Table 3.6: Findings. We organize our discussion around these eleven findings from an analysis of measured chip power, performance, and energy on sixty-one workloads and eight processors.

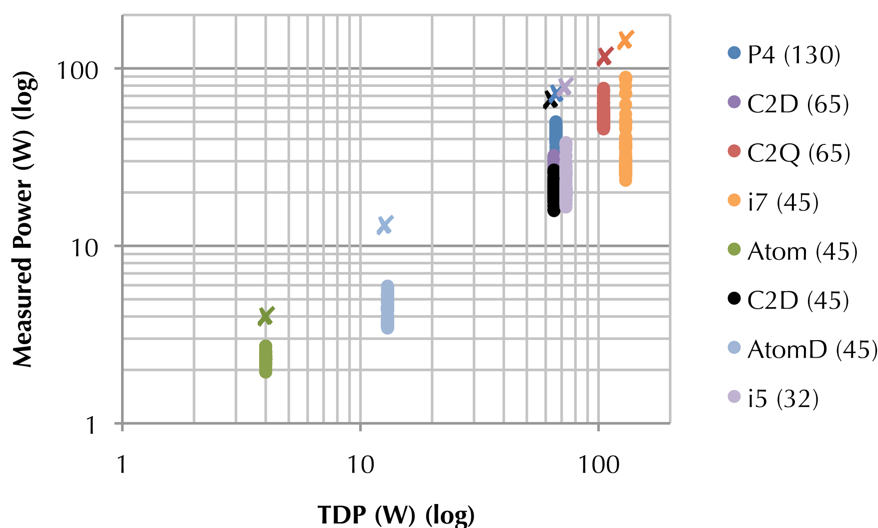


Figure 3.3: Measured power for each processor running 61 benchmarks. Each point represents measured power for one benchmark. The '✗'s are the reported TDP for each processor. Power is application-dependent and does not strongly correlate with TDP.

with power consumption, but it *does not* provide a good estimate for (1) maximum power consumption of individual processors, (2) comparing among processors, or (3) approximating benchmark-specific power consumption.

> **Finding:** *Power consumption is highly application dependent and is poorly correlated to TDP.*

Figure 3.4 plots power versus relative performance for each benchmark on the i7 (45) with eight hardware contexts. Native (red) and managed (green) are differentiated by color, whereas scalable (triangle) and non-scalable (circle) are differentiated by shape. Unsurprisingly, the scalable benchmarks (triangles) tend to perform the best and consume the most power. More unexpected is the range of power and performance characteristics of the non-scalable benchmarks. Power is not strongly correlated with performance across workload or benchmarks. The points would form a straight line if the correlation were strong. For example, the point on the bottom right of the figure achieves almost the best relative performance and lowest power. The correlation coefficient of relative speedup and power is 0.752 for all benchmarks. Java scalable has the strongest correlation among the four groups with the coefficient as 0.946, while native non-scalable has the weakest with the coefficient as -0.32. The coefficients for Java non-scalable and native scalable are 0.418 and 0.374 respectively.

3.3.2 Historical Overview

Figure 3.5(a) plots the average power and performance for each processor in their stock configuration relative to the reference performance, using a log / log scale. For

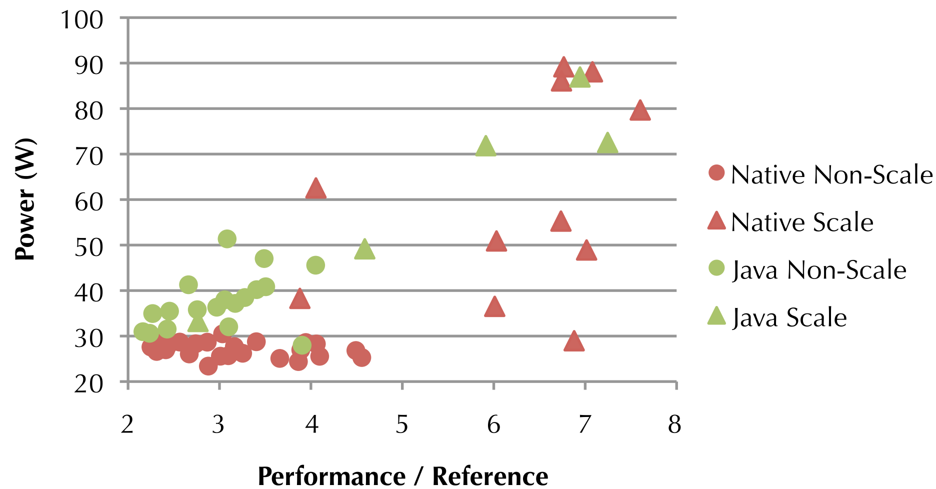
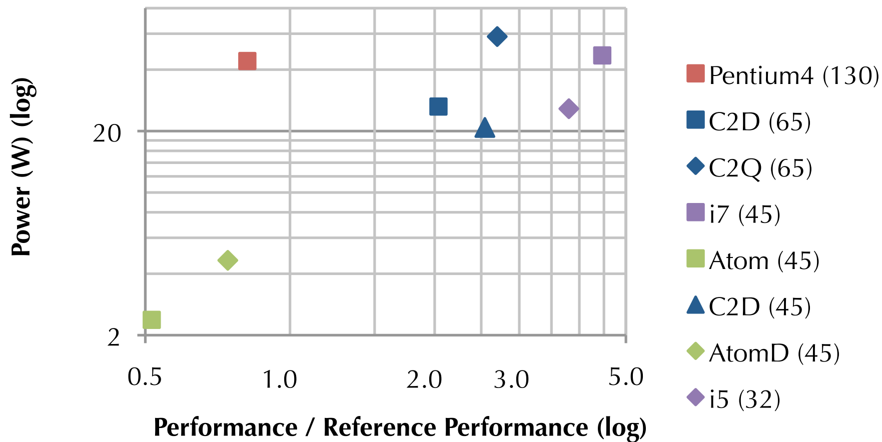


Figure 3.4: Power / performance distribution on the i7 (45). Each point represents one of the 61 benchmarks. Power consumption is highly variable among the benchmarks, spanning from 23 W to 89 W. The wide spectrum of power responses from different applications points to power saving opportunities in software.

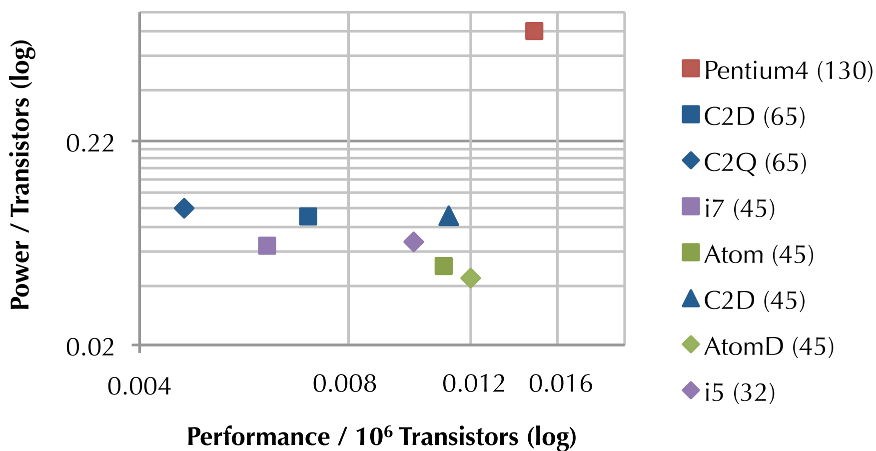
example, the i7 (45) points are the average of the workloads derived from the points in Figure 3.4. Both graphs use the same color for all of the experimental processors in the same family. The shapes encode release age: a square is the oldest; the diamond is next; and the triangle is the youngest, smallest technology in the family.

While historically, mobile devices have been extensively optimized for power, general-purpose processor design until recently has not. Several results stand out illustrating that power is now a first-order design goal and trumps performance in some cases. (1) The Atom (45) and Atom D (45) are designed as low power processors for a different market, however they successfully execute all these benchmarks and are the most power-efficient processors. Compared to the Pentium 4 (130), they degrade performance modestly and reduce power enormously, consuming as little as one twentieth the power. Device scaling from 130 nm to 45 nm contributes significantly to the power reduction from Pentium to Atom. (2) Comparing between 65 nm and 45 nm generations using the Core 2D (65) and Core 2D (45) shows only a 25% increase in performance, but a 35% drop in power. (3) Comparing the two most recent 45 nm and 32 nm generations using the i7 (45) and i5 (32) shows that the i5 (32) delivers about 15% less performance, while consuming about 40% less power. This result has three root causes: (1) the i7 (45) has four cores instead of two on the i5 (32); (2) since half the benchmarks are scalable multithreaded benchmarks, the software parallelism benefits more from the additional two cores, increasing the advantage to the i7 (45); and (3) the i7 (45) has significantly better memory performance. Comparing the Core 2D (45) to the i5 (32) where the number of processors are matched, the i5 (32) delivers 50% better performance, while consuming around 25% more power than the Core 2D (45).

Contemporaneous comparisons also reveal the tension between power and per-



(a) Power / performance tradeoffs have changed from Pentium 4 (130) to i5 (32).



(b) Power and performance per million transistors.

Figure 3.5: Power / performance tradeoff by processor. Each point is an average of the four workloads. Power per million transistor is consistent across different microarchitectures regardless of the technology node. On average, Intel processors burn around 1 Watt for every 20 million transistors.

formance. For example, the contrast between the Core 2D (45) and i7 (45) shows that the i7 (45) delivers 75% more performance than the Core 2D (45), but this performance is very costly in power, with an increase of nearly 100%. These processors thus span a wide range of energy tradeoffs within and across the generations. Overall, these results indicate that optimizing for both power and performance is proving a lot more challenging than optimizing for performance alone.

Figure 3.5(b) explores the effect of transistors on power and performance by dividing them by the number of transistors in the *package* for each processor. We include all transistors because our power measurements occur at the level of the package, not the die. This measure is rough and will downplay results for the i5 (32) and Atom D (45), each of which have a GPU in their package. Even though the benchmarks do not exercise the GPUs, we cannot discount them because the GPU transistor counts on the Atom D (45) are undocumented. Note the similarity between the Atom (45), Atom D (45), Core 2D (45), and i5 (32), which at the bottom right of the graph, are the most efficient processors by the transistor metric. Even though the i5 (32) and Core 2D (45) have five to eight times more transistors than the Atom (45), they all eke out very similar performance and power per transistor. There are likely bigger differences to be found in power efficiency per transistor between chips from different manufactures.

Finding: *Power per transistor is relatively consistent within microarchitecture family, independent of process technology.*

The left-most processors in the graph yield the smallest amount of performance per transistor. Among these processors, the Core 2D (65) and i7 (45) yield the least performance per transistor and use the largest caches among our set. The large 8 MB caches are not effective. The Pentium 4 (130) is perhaps most remarkable—it yields the most performance per transistor and consumes the most power per transistor by a considerable margin. In summary, performance per transistor is inconsistent across microarchitectures, but power per transistor correlates well with microarchitecture, regardless of technology generation. There are two likely factors which govern this, which are: (a) the traditional voltage scaling for transistors has been slowed down or stopped under 90 nm [Venkatesh et al., 2010], so the power cost of a chip mostly depend on the power saving techniques used by its microarchitecture; and (b) the power cost of on-chip interconnects varies with microarchitecture and scales less favourably than that of transistors across technology generations [Kahng et al., 2009].

3.3.3 Pareto Analysis at 45 nm

The Pareto optimal frontier defines a set of choices that are most efficient in a tradeoff space. Prior research uses the Pareto frontier to explore power versus performance with *models* that derive potential architectural designs on the frontier [Azizi et al., 2010]. We present a Pareto frontier derived from *measured performance and power*. We hold the process technology constant by using the four 45 nm processors: Atom (45),

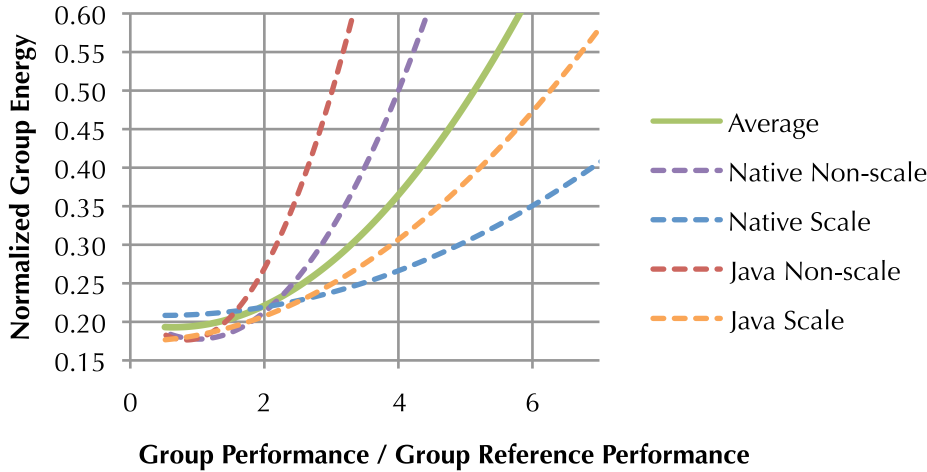


Figure 3.6: Energy / performance Pareto frontiers (45 nm). The energy / performance optimal designs are application-dependent and significantly deviate from the average case.

	Atom (45) 1C2T @ 1.7GHz	Core 2D (45) 2C1T @ 1.6GHz	**Core 2D (45) 2C1T @ 3.1GHz**	i7 (45) 1C1T @ 2.7GHz No TB	i7 (45) 1C1T @ 2.7GHz	i7 (45) 1C2T @ 1.6GHz	i7 (45) 1C2T @ 2.4GHz	i7 (45) 2C1T @ 1.6GHz	i7 (45) 2C2T @ 1.6GHz	i7 (45) 4C1T @ 2.7GHz No TB	i7 (45) 4C1T @ 2.7GHz	i7 (45) 4C2T @ 1.6GHz	i7 (45) 4C2T @ 2.1GHz	i7 (45) 4C2T @ 2.7GHz No TB	**i7 (45) 4C2T @ 2.7GHz**
Native Non-scalable				✔	✔	✔	✔								
Java Non-scalable	✔		✔				✔	✔		✔	✔				✔
Native Scalable	✔	✔										✔	✔	✔	✔
Java Scalable	✔								✔			✔	✔	✔	✔
Average	✔								✔			✔	✔	✔	✔

Table 3.7: Pareto-efficient processor configurations for each workload. Stock configurations are bold. Each '✔' indicates that the configuration is on the energy / performance Pareto-optimal curve. Native non-scalable has almost no overlap with any other workload.

Atom D (45), Core 2D (45), and i7 (45). We expand the number of processor configurations from 4 to 29 by configuring the number of hardware contexts (SMT and CMP), by clock scaling, and disabling / enabling Turbo Boost. The 25 non-stock configurations represent alternative design points. For each configuration, we compute the averages for each workload and their average to produce an energy / performance scatter plot (not shown here). We next pick off the frontier — the points that are not

dominated in performance or energy efficiency by any other point — and fit them with a polynomial curve. Figure 3.6 plots these polynomial curves for each workload and the average. The rightmost curve delivers the best performance for the least energy.

Each row of Table 3.7 corresponds to one of the five curves in Figure 3.6. The check marks identify the Pareto-efficient configurations that define the bounding curve and include 15 of 29 configurations. Somewhat surprising is that none of the Atom D (45) configurations are Pareto efficient. Notice the following. (1) Native non-scalable shares only one choice with any other workload. (2) Java Scalable and the average share all the same choices. (3) Only two of eleven choices for Java Non-scalable and Java Scalable are common to both. (4) Native non-scalable does not include the Atom (45) in its frontier. This last finding contradicts prior simulation work, which concluded that dual-issue in-order cores and dual-issue out-of-order cores are Pareto optimal for Native Non-scalable [Azizi et al., 2010]. Instead we find that all of the Pareto-efficient points for Native Non-scalable in this design space are quad-issue out-of-order i7 (45) configurations.

Figure 3.6 starkly shows that each workload deviates substantially from the average. Even when the workloads share points, the points fall in different places on the curves because each workload exhibits a different energy / performance trade-off. Compare the scalable and non-scalable benchmarks at 0.40 normalized energy on the y-axis. It is impressive how well these architectures effectively exploit software parallelism, pushing the curves to the right and increasing performance from about 3 to 7 while holding energy constant. This measured behavior confirms prior model-based observations about the role of software parallelism in extending the energy / performance curve to the right [Azizi et al., 2010].

Finding: *Energy-efficient architecture design is very sensitive to workload. Configurations in the Native Non-scalable Pareto frontier differ substantially from all other workloads.*

One might try to conclude that managed language workloads are less energy-efficient compared to native workloads from Figure 3.6 since the curves are steeper, but the workloads are quite different. SPEC CPU is CPU-intensive workload with scientific codes, whereas DaCapo is client-side Java with language, query, and event processing programs. For many years hardware development has focused on efficiency gains based on SPEC CPU and similar native benchmarks, rather than managed workloads. It is to be expected that native benchmark energy efficiency scales more reasonably than managed benchmarks. In summary, architects should use a variety of workloads, and in particular, should avoid only using Native Non-scalable workloads.

3.4 Feature Analysis

This section explores the energy impact of hardware features through controlled experiments. We present two pairs of graphs for feature analysis experiments as shown

in Figure 3.7, for example. The top graph compares relative power, performance, and energy as an average of the four workload groups. The bottom graph breaks down energy by workload group. In these graphs, higher is better for performance. Lower is better for power and energy.

3.4.1 Chip Multiprocessors

Figure 3.7 shows the average power, performance, and energy effects of chip multiprocessors (CMPs) by comparing one core to two cores for the two most recent processors in our study. We disable Turbo Boost in these analyses because it adjusts power dynamically based on the number of idle cores. We disable Simultaneous Multithreading (SMT) to maximally expose thread-level parallelism to the CMP hardware feature. Figure 3.7(a) compares relative power, performance, and energy as a weighted average of the workloads. Figure 3.7(b) breaks down the energy as a function of workload. While average energy is reduced by 9% when adding a core to the i5 (32), it is increased by 12% when adding a core to the i7 (45). Figure 3.7(a) shows that the source of this difference is that the i7 (45) experiences twice the power overhead for enabling a core as the i5 (32), while producing roughly the same performance improvement.

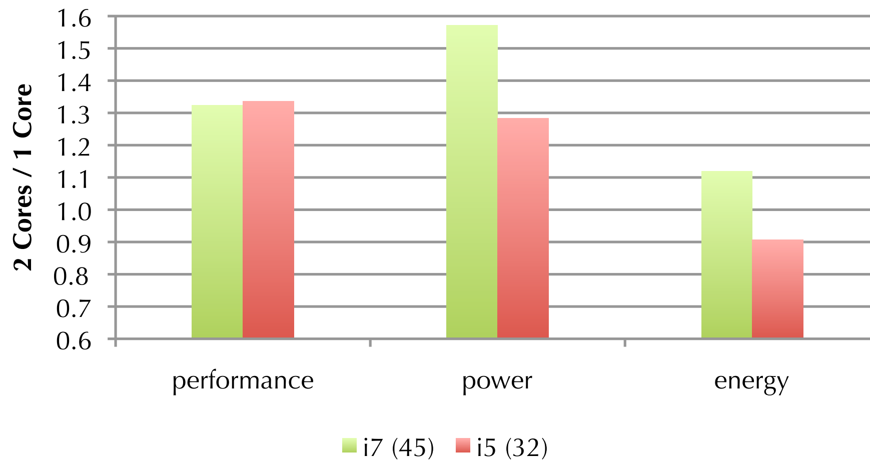
Finding: *Comparing one core to two, enabling a core is not consistently energy efficient.*

Figure 3.7(b) shows that Native Non-scalable and Java Non-scalable suffer the most energy overhead with the addition of another core on the i7 (45). As expected, performance for Native Non-scalable is unaffected. However, turning on an additional core for Native Non-scalable leads to a power increase of 4% and 14% respectively for the i5 (32) and i7 (45), translating to energy overheads.

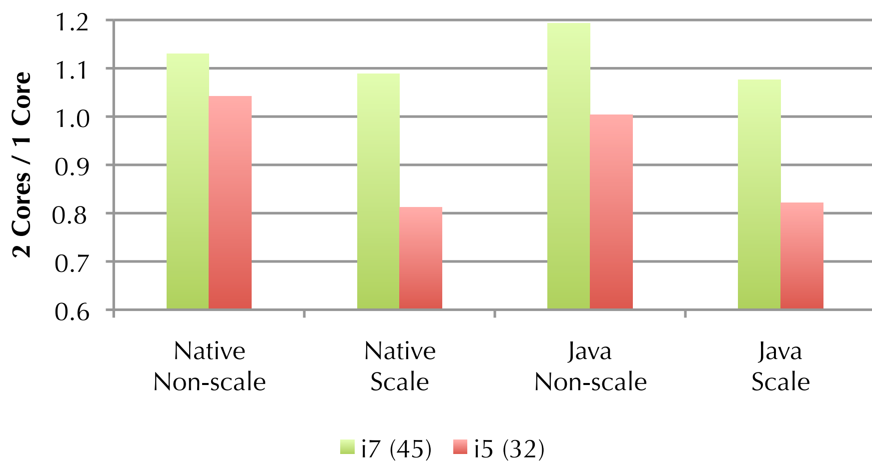
More interesting is that Java Non-scalable does not incur an energy overhead when enabling another core on the i5 (32). In fact, we were surprised to find that the reason for this is that the *single threaded* Java Non-scalable workload runs faster with two processors! Figure 3.8 shows the scalability of the single threaded subset of Java Non-scalable on the i7 (45), with SMT disabled, comparing one and two cores. Although these Java benchmarks are single threaded, the JVMs on which they execute are not.

Finding: *The JVM induces parallelism into the execution of single threaded Java benchmarks.*

Since VM services for managed languages, such as JIT, GC, and profiling, are often concurrent and parallel, they provide substantial scope for parallelization, even within ostensibly sequential applications. We instrumented the HotSpot JVM and confirmed that its JIT and GC are parallel. Detailed performance counter measurements revealed that the GC induced memory system improvements with more cores by reducing the collector's cache displacement effect on the application thread.



(a) Impact of doubling the number of cores on performance, power, and energy, averaged over all four workloads.



(b) Energy impact of doubling the number of cores for each workload.

Figure 3.7: CMP: Comparing two cores to one core. Doubling the cores is not consistently energy efficient among processors or workloads.

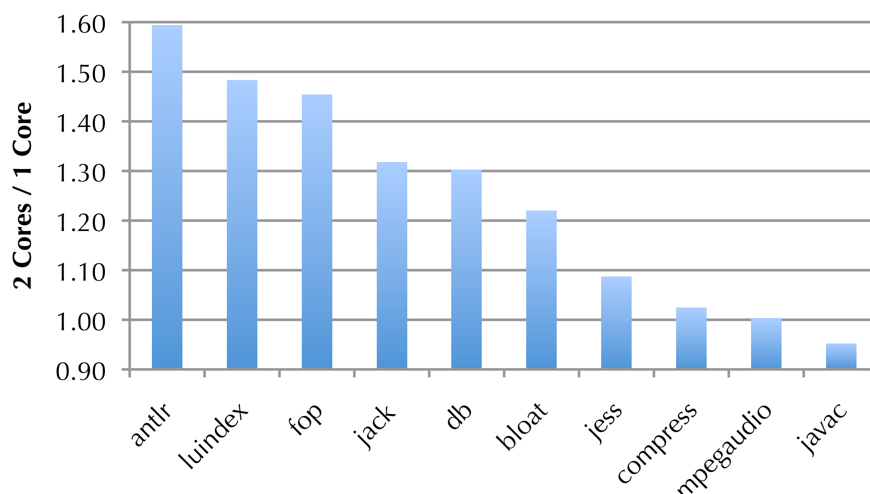


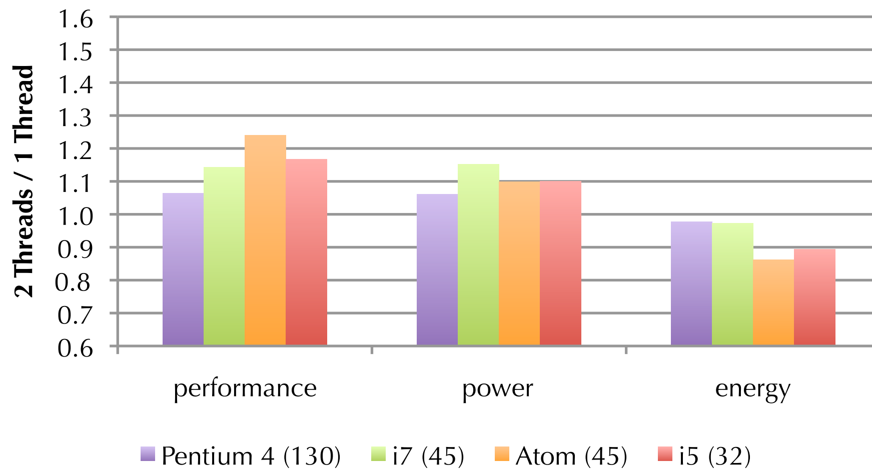
Figure 3.8: Scalability of single threaded Java benchmarks. Some single threaded Java benchmarks scale well. The underlying JVM exploits parallelism for compilation, profiling and GC.

3.4.2 Simultaneous Multithreading

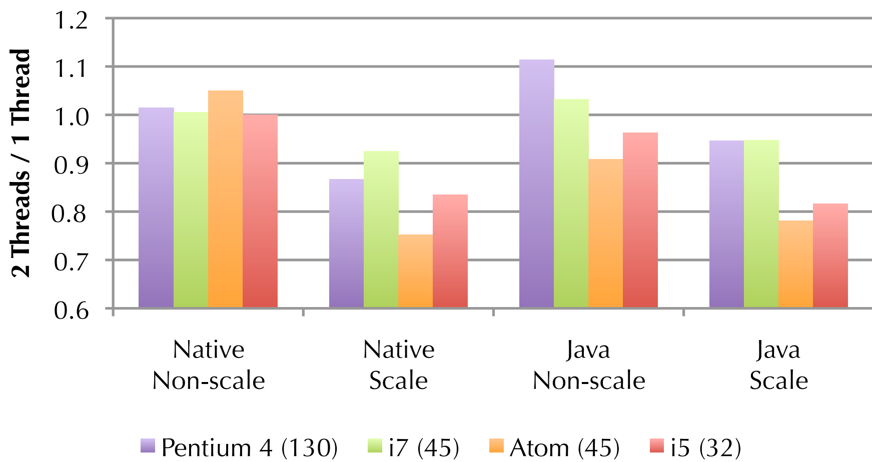
Figure 3.9 shows the effect of disabling simultaneous multithreading (SMT) [Tullsen et al., 1995] on the Pentium 4 (130), Atom (45), i5 (32), and i7 (45). Each processor supports two-way SMT. SMT provides fine-grain parallelism to distinct threads in the processors' issue logic and in modern implementations, threads share all processor components (e.g., execution units, caches). Singhal states that the small amount of logic exclusive to SMT consumes very little power [Singhal, 2011]. Nonetheless, this logic is integrated, so SMT contributes a small amount to total power even when disabled. Our results therefore slightly underestimate the power cost of SMT. We use only one core, ensuring SMT is the sole opportunity for thread-level parallelism. Figure 3.9(a) shows that the performance advantage of SMT is significant. Notably, on the i5 (32) and Atom (45), SMT improves average performance significantly without much cost in power, leading to net energy savings.

Finding: *SMT delivers substantial energy savings for recent hardware and for in-order processors.*

Given that SMT was and continues to be motivated by the challenge of filling issue slots and hiding latency in wide issue superscalars, it may appear counter intuitive that performance on the dual-issue in-order Atom (45) should benefit so much more from SMT than the quad-issue i7 (45) and i5 (32) benefit. One explanation is that the in-order pipelined Atom (45) is more restricted in its capacity to fill issue slots. Compared to other processors in this study, the Atom (45) has much smaller caches. These features accentuate the need to hide latency, and therefore the value of SMT. The performance improvements on the Pentium 4 (130) due to SMT are half to one third that of more recent processors, and consequently there is no net energy advantage.



(a) Impact of enabling two-way SMT on a single-core with respect to performance, power, and energy, averaged over all four workloads.



(b) Energy impact of enabling two-way SMT on a single-core for each workload.

Figure 3.9: SMT: one core with and without SMT. Enabling SMT delivers significant energy savings on the recent i5 (32) and the in-order Atom (45).

This result is not so surprising given that the Pentium 4 (130) is the first commercial implementation of SMT.

Figure 3.9(b) shows that, as expected, the Native Non-scalable workload experiences very little energy overhead due to enabling SMT, whereas Figure 3.7(b) shows that enabling a core incurs a significant power and thus energy penalty. The scalable workloads unsurprisingly benefit most from SMT.

The excellent energy efficiency of SMT is impressive on recent processors as compared to CMP, particularly given its very low die footprint. Compare Figure 3.7 and 3.9. SMT provides less performance improvement than CMP—SMT adds about half as much performance as CMP on average, but incurs much less power cost. The results on the modern processors show SMT in a much more favorable light than in Sasanka et al.'s model-based comparative study of the energy efficiency of SMT and CMP [Sasanka et al., 2004].

3.4.3 Clock Scaling

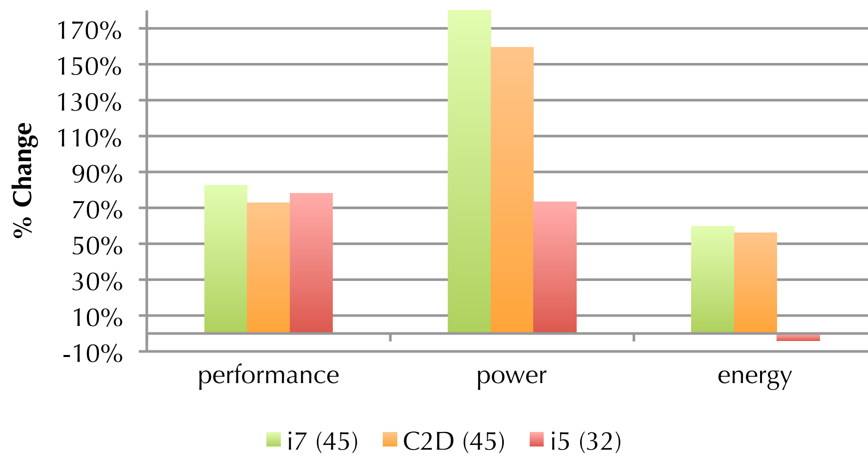
We vary the processor clock on the i7 (45), Core 2D (45), and i5 (32) between their minimum and maximum settings. The range of clock speeds are: 1.6 to 2.7 GHz for i7 (45); 1.6 to 3.1 GHz for Core 2D (45); and 1.2 to 3.5 GHz for i5 (32). We uniformly disable Turbo Boost to produce a consistent clock rate for comparison; Turbo Boost may vary the clock rate, but only when the clock is set at its highest value. Each processor is otherwise in its stock configuration. Figures 3.10(a) and 3.10(b) express changes in power, performance, and energy with respect to doubling in clock frequency over the range of clock speeds to normalize and compare across architectures.

The three processors experience broadly similar increases in performance of around 80%, but *power differences vary substantially, from 70% to 180%*. On the i7 (45) and Core 2D (45), the performance increases require disproportional power increases—consequently energy consumption increases by about 60% as the clock is doubled. The i5 (32) is starkly different—doubling its clock leads to a slight energy reduction.

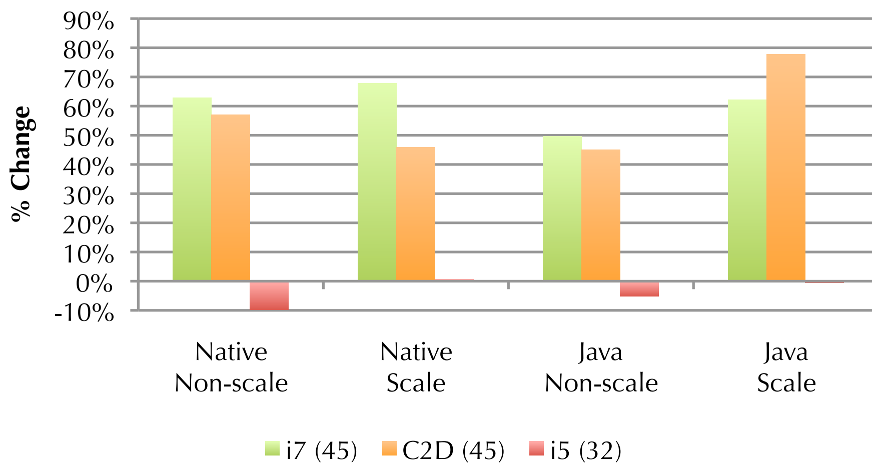
Finding: *The most recent processor in our study does* not *consistently increase energy consumption as its clock increases.*

Figure 3.11(a) shows that this result is consistent across the range of i5 (32) clock rates. A number of factors may explain why the i5 (32) performs relatively so much better at its highest clock rate: (a) the i5 (32) is a 32 nm process, while the others are 45 nm; (b) the power-performance curve is non-linear and these experiments may observe only the upper (steeper) portion of the curves for i7 (45) and Core 2D (45); (c) although the i5 (32) and i7 (45) share the same microarchitecture, the second generation i5 (32) likely incorporates energy improvements; (d) the i7 (45) is substantially larger than the other processors, with four cores and a larger cache.

Finding: *The power / performance response to clock scaling of* Native Non-scalable *differs from the other workloads.*

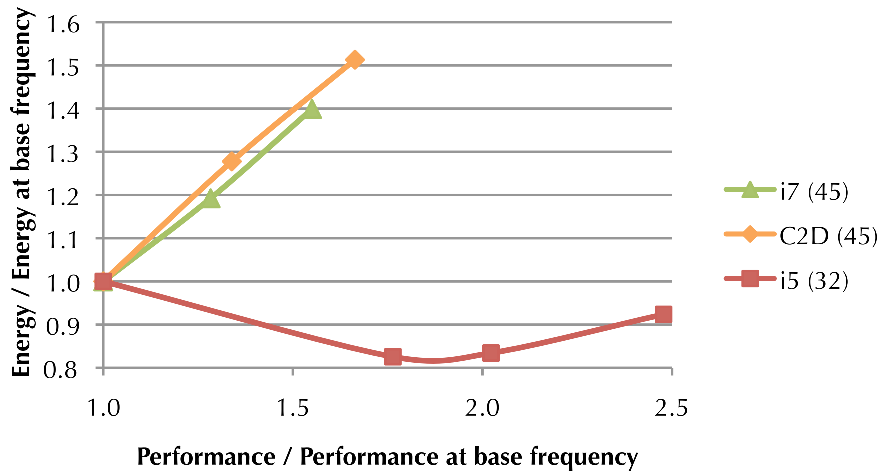


(a) Impact of doubling clock with respect to performance, power, and energy, averaged over all four workloads.

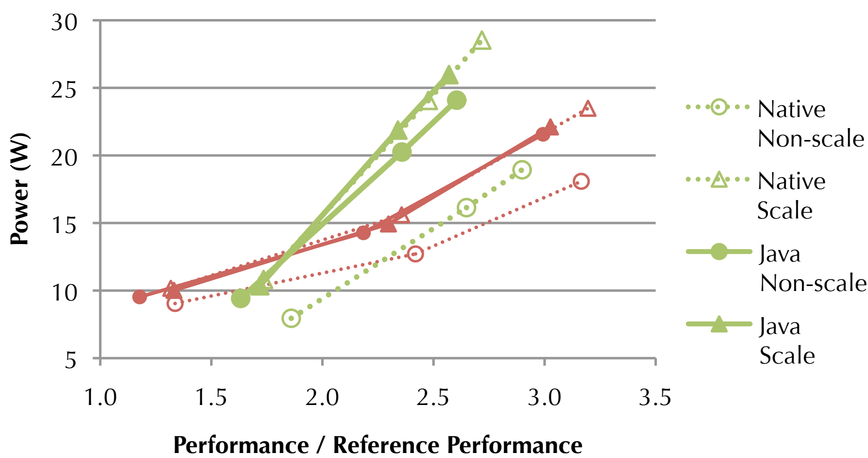


(b) Energy impact of doubling clock in stock configurations for each workload.

Figure 3.10: Clock: doubling clock in stock configurations. Doubling clock does not increase energy consumption on the recent i5 (32).



(a) Impact of scaling up the clock with respect to performance and energy over all four workloads.



(b) Impact of scaling up the clock with respect to performance and power on the i7 (45) (green) and i5 (32) (red) by workload.

Figure 3.11: Clock: scaling across the range of clock rates in stock configurations. Points are clock speeds.

Figure 3.10(b) shows that doubling the clock on the i5 (32) roughly maintains or improves energy consumption of all workloads, with Native Non-scalable improving the most. For the i7 (45) and Core 2D (45), doubling the clock raises energy consumption. Figure 3.11(b) shows that Native Non-scalable has a different power / performance behavior compared to the other workloads and that this difference is largely independent of clock rate. The Native Non-scalable workload draws less power overall, and power increases less steeply as a function of performance increases. Native non-scalable (SPEC CPU2006) is the most widely studied workload in the architecture literature, but it is the outlier. These results reinforce the importance of including scalable and managed workloads in energy evaluations.

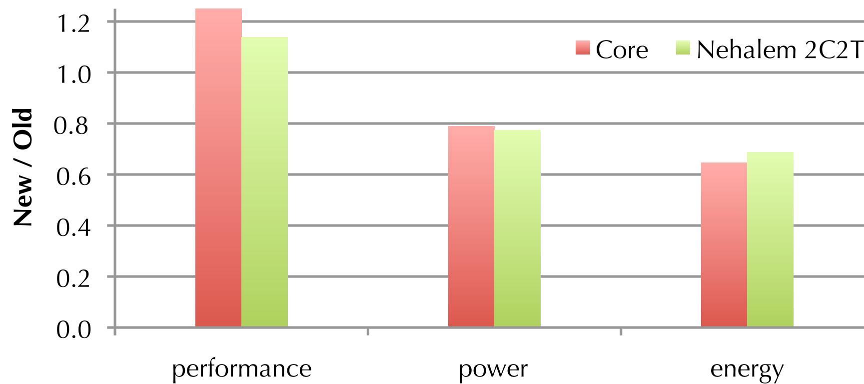
3.4.4 Die Shrink

We use processor pairs from the Core (Core 2D (65) / Core 2D (45)) and Nehalem (i7 (45) / i5 (32)) microarchitectures to explore die shrink effects. These hardware comparisons are imperfect because they are not straightforward die shrinks. To limit the differences, we control for hardware parallelism by limiting the i7 (45) to two cores. The tools and processors at our disposal do not let us control the cache size nor for other microarchitecture changes that accompany a die shrink. We compare at stock clock speeds and control for clock speed by running both Cores at 2.4 GHz and both Nehalems at 2.66 GHz. We do not directly control for core voltage, which differs across technology nodes for the same frequency. Although imperfect, these are the first published comparisons of measured energy efficiency across technology nodes.

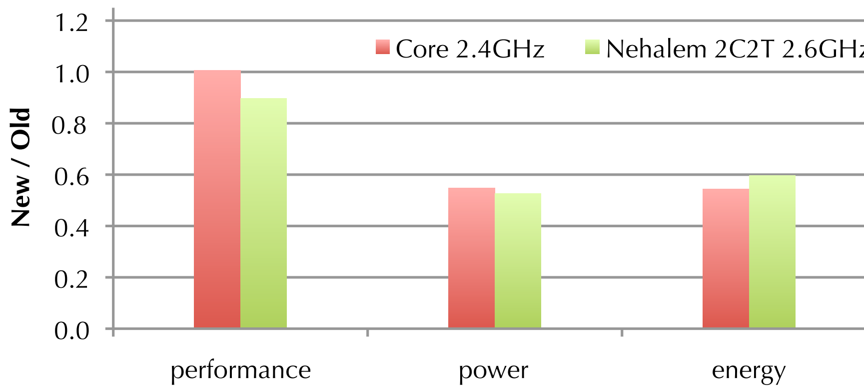
Finding: *Two recent die shrinks deliver similar and surprising reductions in energy, even when controlling for clock frequency.*

Figure 3.12(a) shows the power and performance effects of the die shrinks with the stock clock speeds for all the processors. The newer processors are significantly faster at their higher stock clock speeds and significantly more power efficient. Figure 3.12(b) shows the same experiment, but down-clocking the newer processors to match the frequency of their older peers. Down-clocking the new processors improves their relative power and energy advantage even further. Note that as expected, the die shrunk processors offer no performance advantage once the clocks are matched, indeed the i5 (32) performs 10% slower than the i7 (45). However, power consumption is reduced by 47%. This result is consistent with expectations, given the lower voltage and reduced capacitance at the smaller feature size.

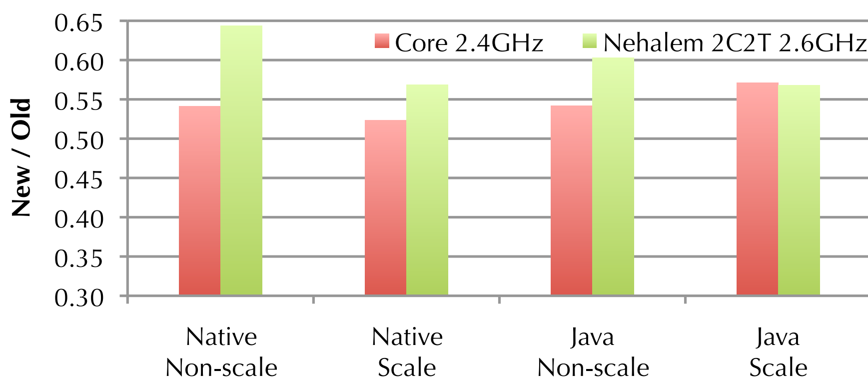
Figures 3.12(a) and 3.12(b) reveal a striking similarity in power and energy savings between the Core (65 nm / 45 nm) and Nehalem (45 nm / 32 nm) die shrinks. This data suggests that Intel maintained the same rate of energy reduction across the two most recent generations. As a point of comparison, the models used by the International Technology Roadmap for Semiconductors (ITRS) predicted a 9% increase in frequency and a 34% reduction in power from 45 nm to 32 nm [ITRS Working Group, 2011]. Figure 3.12(a) is both more and less encouraging. Clock speed increased by



(a) Impact of die shrinking with respect to performance, power, and energy over all four workloads. Each processor uses its native clock speed.



(b) Impact of die shrinking with respect to performance, power, and energy over all four workloads. Clock speeds are matched in each comparison.



(c) Energy impact of die shrinking for each workload. Clock speeds are matched in each comparison.

Figure 3.12: Die shrink: microarchitectures compared across technology nodes. 'Core' shows Core 2D (65) / Core 2D (45) while 'Nehalem' shows i7 (45) / i5 (32) when two cores are enabled. Both die shrinks deliver substantial energy reductions.

26% in the stock configurations of the i7 (45) to the i5 (32) with an accompanying 14% increase in performance, but power reduced by 23%, less than the 34% predicted. To more deeply understand die shrink efficiency on modern processors requires measuring more processors in each technology node.

3.4.5 Gross Microarchitecture Change

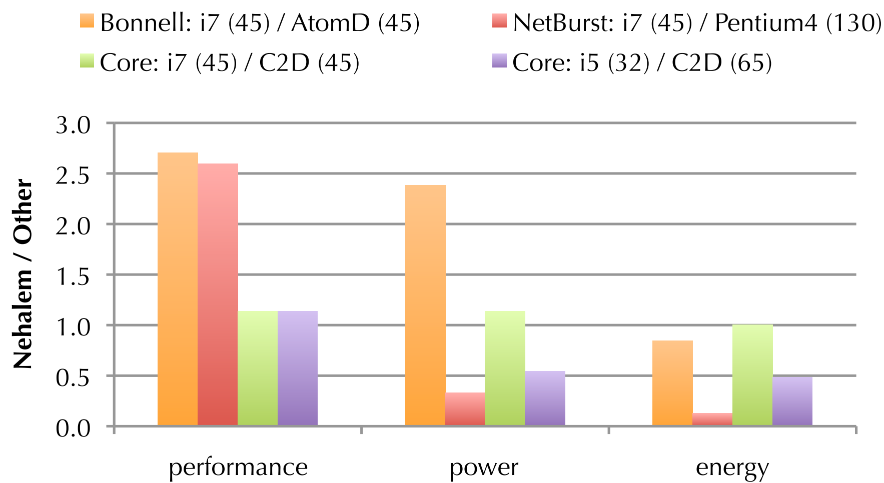
This section explores the power and performance effect of gross microarchitectural change by comparing microarchitectures while matching features such as processor clock, degree of hardware parallelism, process technology, and cache size.

Figure 3.13 compares the Nehalem i7 (45) with the NetBurst Pentium 4 (130), Bonnell Atom D (45), and Core 2D (45) microarchitectures, and it compares the Nehalem i5 (32) with the Core 2D (65). Each comparison configures the Nehalems to match the clock speed, number of cores, and hardware threads of the other architecture. Both the i7 (45) and i5 (32) comparisons to the Core show that the move from Core to Nehalem yields a small 14% performance improvement. This finding is not inconsistent with Nehalem's stated primary design goals, i.e., delivering scalability and memory performance.

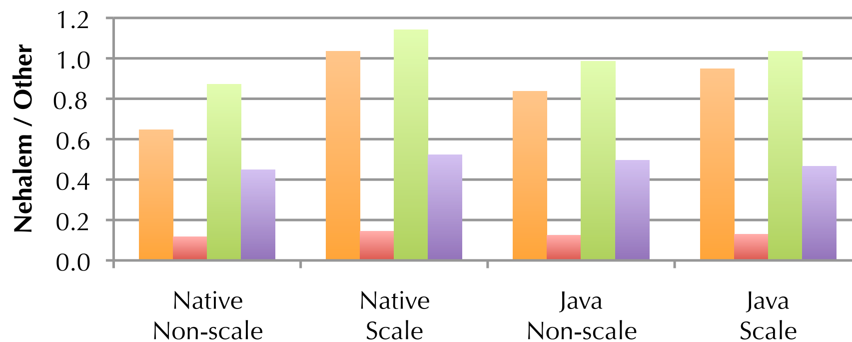
Finding: *Controlling for technology, hardware parallelism, and clock speed, the out-of-order architectures have similar energy efficiency as the in-order ones.*

The comparisons between the i7 (45) and Atom D (45) and Core 2D (45) hold process technology constant at 45 nm. All three processors are remarkably similar in energy consumption. This outcome is all the more interesting because the i7 (45) is disadvantaged since it uses fewer hardware contexts here than in its stock configuration. Furthermore, the i7 (45) integrates more services on-die, such as the memory controller, that are off-die on the other processors, and thus outside the scope of the power meters. The i7 (45) improves upon the Core 2D (45) and Atom D (45) with a more scalable, much higher bandwidth on-chip interconnect, that is not exercised heavily by our workloads. It is impressive that, despite all of these factors, the i7 (45) delivers similar energy efficiency to its two 45 nm peers, particularly when compared to the low-power in-order Atom D (45).

It is unsurprising that the i7 (45) performs 2.6$\times$ faster than the Pentium 4 (130), while consuming one third the power, when controlling for clock speed and hardware parallelism (but not for factors such as memory speed). Much of the 50% power improvement is attributable to process technology advances. This speedup of 2.6 over seven years is however *substantially* less than the historical factor of eight improvement experienced in every prior seven year time interval between 1970 through the early 2000s. This difference in improvements marks the beginning of the power-constrained architecture design era.



(a) Impact of microarchitecture change with respect to performance, power, and energy, averaged over all four workloads.



(b) Energy impact of microarchitecture for each workload.

Figure 3.13: Gross microarchitecture: a comparison of Nehalem with four other microarchitectures. In each comparison the Nehalem is configured to match the other processor as closely as possible. The most recent microarchitecture, Nehalem, is more energy efficient than the others, including the low-power Bonnell (Atom).

3.4.6 Turbo Boost Technology

Intel Turbo Boost Technology on Nehalem processors over-clocks cores under the following conditions [Intel, 2008]. With Turbo Boost enabled, *all* cores can run one "step" (133 MHz) faster if temperature, power, and current conditions allow. When only one core is active, Turbo Boost may clock it an additional step faster. Turbo Boost is only enabled when the processor executes at its default highest clock setting. This feature requires on-chip power sensors. We verified empirically on the i7 (45) and i5 (32) that all cores ran 133 MHz faster with Turbo Boost. When only one core was active, the core ran 266 MHz faster. Since the i7 (45) runs at a lower clock (2.67 GHz) than the i5 (32) (3.46 GHz), it experiences a relatively larger boost.

Finding: *Turbo Boost is not energy efficient on the i7 (45).*

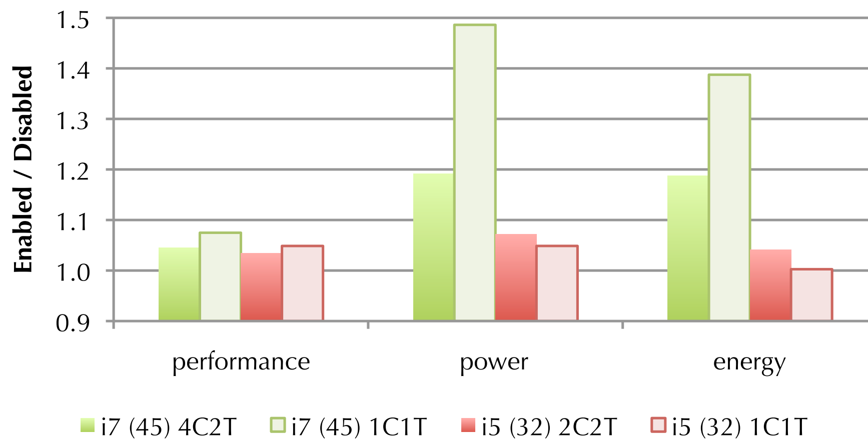
Figure 3.14(a) shows the effect of disabling Turbo Boost at the BIOS on the i7 (45) and i5 (32) in their stock configurations (dark) and when we limit each machine to a single hardware context (light). With the single hardware context, Turbo Boost will increment the clock by two steps if thermal conditions permit. The actual performance changes are well predicted by the clock rate increases. The i7 (45) clock step increases are 5 and 10%, and the actual performance increases are 5 and 7%. The i5 (32) clock step increases are 4 and 8%, and the actual performance increases are 3 and 5%. However, the i7 (45) responds with a substantially higher power increase and consequent energy overhead, while the i5 (32) is essentially energy-neutral. That difference can be accounted for as a result of: (a) the static power for i5 (32) taking a larger portion of total power compared to that of the i7 (45); (b) the i7 (45) experiencing a relatively greater boost to that of the i5 (32), since the i7 (45) runs at a lower clock; and (c) the improvements in the turbo boost technique of the i5 (32) over that of the i7 (45).

Figure 3.14(b) shows that when all hardware contexts are available (dark), the non-scalable benchmarks consume relatively more energy than scalable benchmarks on the i7 (45) in its stock configuration. Because the non-scalable native and sometimes Java utilize only a single core, Turbo Boost will likely increase the clock by an additional step. Figure 3.14(a) shows that this technique is power-hungry on the i7 (45).

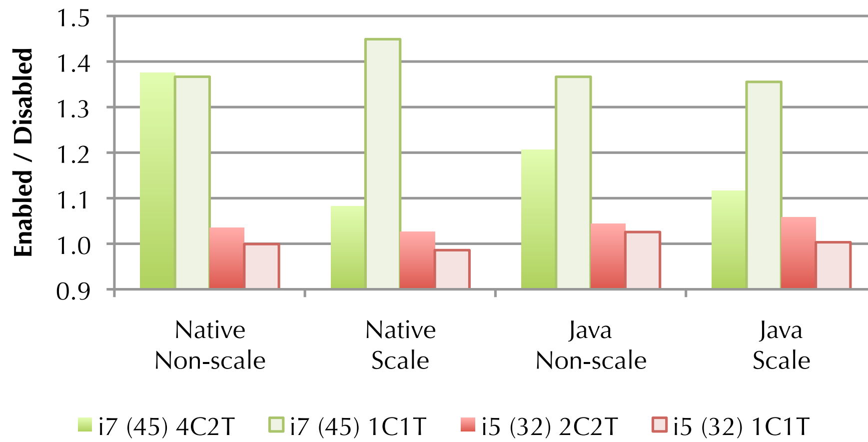
3.5 Summary

This chapter explored the performance, power, and energy impact of a variety of hardware features on both managed and native software. The quantitative data in this chapter revealed the extent of some known and previously unobserved hardware and software trends. We highlighted eleven findings from the analysis of the data we gathered, from which two themes emerge.

(1) *Workload:* The power, performance, and energy trends of native workloads do not approximate managed workloads. For example, native single threaded workloads never experience performance or energy improvements from CMPs or SMT,



(a) Impact of enabling Turbo Boost with respect to performance, power, and energy, averaged over all four workloads.



(b) Energy impact of enabling Turbo Boost for each workload.

Figure 3.14: Turbo Boost: enabling Turbo Boost on i7 (45) and i5 (32). Turbo Boost is not energy efficient on the i7 (45).

while single threaded Java workloads run on average about 10% faster and up to 60% faster on two cores when compared to one core due to the parallelism exploited by the VM. The results recommend architects always include native and managed workloads when designing and evaluating energy-efficient hardware. The diverse application power profiles suggest that software needs to participate in power optimization.

(2) *Architecture:* Clock scaling, microarchitecture, SMT, CMP and turbo boost each elicit a huge variety of power, performance, and energy responses. For example, halving the clock rate of the i5 (32) increases its energy consumption around 4%, whereas it decreases the energy consumption of the i7 (45) and Core 2D (45) by around 60%, i.e., running the i5 (32) at its peak clock rate is as energy efficient as running it at its lowest, whereas running the i7 (45) and Core 2D (45) at their lowest clock rate is substantially more energy efficient than their peak. This variety and the difficulty of obtaining power measurements recommends exposing on-chip power meters and when possible, structure-specific power meters for cores, caches, and other structures. Just as hardware event counters provide a quantitative grounding for performance innovations, power meters are necessary for energy optimizations.

The conclusion of this chapter motivates the following chapters in this thesis. The real machine power measurement framework and hardware/software configuration methodology explored in this chapter will be used throughout the thesis. The hardware characteristics identified in this chapter form a basis to model and tailor future AMP architectures. These hardware insights combined with the behaviour of the workloads evaluated leads us to investigate the significant characteristics of VMs, such as parallelism, in the next chapter in order to improve managed software efficiency on AMP architectures.

Chapter 4

Asymmetric Multicore Processors and Managed Software

The previous chapter explores the energy and performance characteristics of managed software with respect to a variety of hardware features. We further develop the power measurement methodology of the previous chapter and utilize that methodology to explore a greater range of hardware and software configurations. Using those results and modelling, we show how to exploit the differentiated characteristics of Virtual Machine (VM) services to improve total power, performance, and energy on Asymmetric Multicore Processors (AMPs), which require differentiated software to be effective.

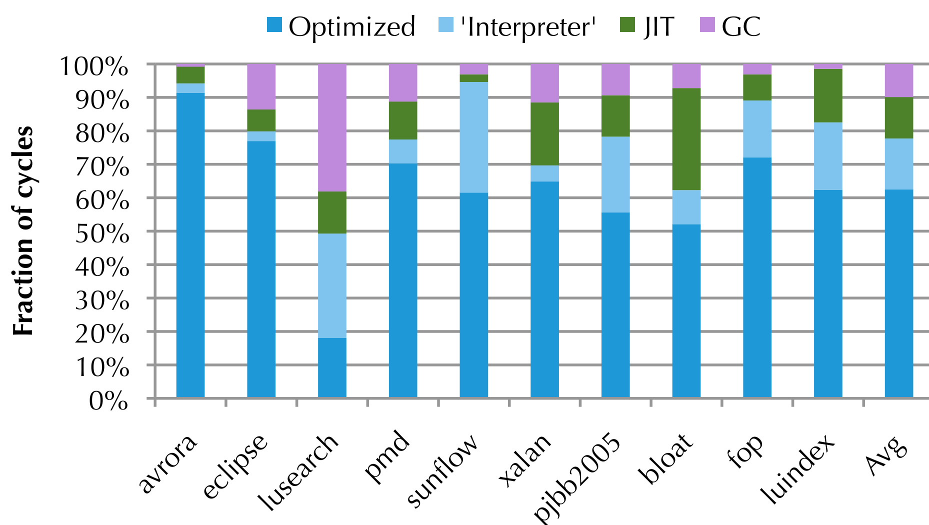
Section 4.2 describes the hardware, measurements, workload, and software configuration used in this chapter. Section 4.3 starts with a motivating analysis of the power and energy footprint of VM services on orthodox hardware. Section 4.4 explores whether the services will benefit from dedicated parallel hardware and be effective even if the hardware is slow. Section 4.5 explores the amenability of VM services to alternative core designs, since AMP has the opportunity to include a variety of general purpose cores tailored to various distinct workload types. Section 4.6 models VM services running on a hypothetical AMP scenario. Section 4.7 discusses a new opportunity offered by executing the Just-in-Time compiler (JIT) on small cores.

This chapter describes work published in the paper "The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software" [Cao, Blackburn, Gao, and McKinley, 2012].

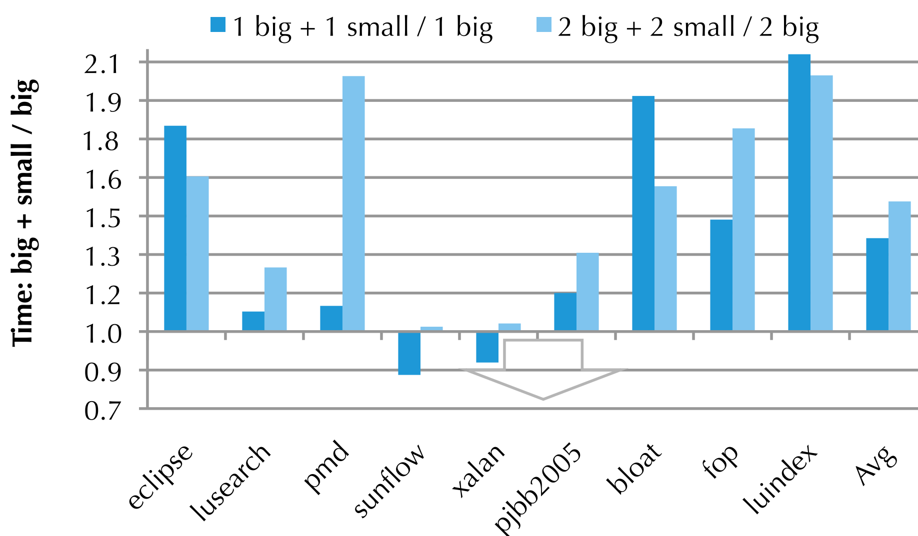
4.1 Introduction

This chapter uses a hardware-software cooperative approach to address the major challenges happening in hardware and software communities.

On the hardware side, we explore single-ISA AMP architectures. On the software side, we explore VM services, such as the interpreter, JIT, profiler, and Garbage Collector (GC), which provide much of the abstraction of managed languages, and also much of their overheads. Since VM services execute together with every managed



(a) Fraction of cycles spent in VM services on an i7 1C1T @ 3.4 GHz. See Section 4.2 for methodology.



(b) Execution time increase due to *adding* small cores on an AMD Phenom II[1].

Figure 4.1: Motivation: (a) VM services consume significant resources; (b) The naive addition of small cores *slows* down applications.

application, improvements to VM services will transparently improve all managed applications.

This chapter identifies and leverages unique combinations of four software attributes for exploiting AMP: (1) parallelism, (2) asynchrony, (3) non-criticality, and (4) hardware sensitivity. We go beyond improving performance on AMP architecture by also seeking to improve power, energy, and performance per energy (PPE); the reciprocal of the EDP (Energy-Delay Product) metric [Laros III et al., 2013].

We show that VM services are *a lucrative target* because they consume almost 40% of total time and energy and they exhibit the requisite software characteristics. Figure 4.1(a) shows the fraction of cycles Java applications spend in VM services. GC consumes 10% and JIT consumes 12% of all cycles on average. An additional 15% of total time is spent executing unoptimized code (e.g., via the interpreter). Total time in VM services ranges from 9% to 82%. Prior GC performance results on industry VMs confirm this trend: IBM's J9, JRockit, and Oracle's HotSpot JDK actually show an even higher average fraction of time spent on GC [Ha et al., 2008]. VM services in less mature VMs, such as JavaScript and PHP VMs, likely consume an even higher fraction of total cycles. Reducing the power and PPE of VM services is thus a promising target.

Figure 4.1(b) shows that naively executing managed applications on AMP platforms without any VM or operating system support is a very bad idea. In this figure, we measure the effect of adding small (slow, low power) cores to the performance of Java applications in Jikes RVM. The downward pointing grey arrow indicates that lower is better on this graph. The big (fast, high power) cores are out-of-order x86 cores running at the default 2.8 GHz and the small ones are simply down-clocked to 0.8 GHz on an AMD Phenom II. We evaluate both the addition of one slow core to one fast core, and two slow cores to two fast cores. The results are similar. Even though these configurations are set by the OS and provide strictly more hardware resources, they slow down applications by 35% and 50% on average!

Using hardware and software configuration, power measurements, and modeling, this chapter shows how to exploit the characteristics of GC, interpreter, and JIT workloads on AMP hardware to improve total power, performance, energy, and PPE. Each VM component has a unique combination of (1) parallelism, (2) asynchrony, (3) non-criticality and (4) hardware sensitivity.

GC. Because GC may be performed asynchronously and a concurrent collector is normally not on the critical path, it is amenable to executing on a separate core. The resources applied to GC need to be tailored according to the work rate generated by the application to ensure that the collector keeps up and remains off the application's critical path. The computation the collector performs, a graph traversal, is parallel and thus benefits from more than one separate core. Furthermore, GC is memory bound and many high-performance, power-hungry hardware features that improve application PPE are inefficient for GC. GC does not benefit from a high clock rate or instruction level parallelism (ILP), but it does benefit from higher memory band-

[1] The avrora benchmark is excluded from Figure 4.1(b) due to its erratic behavior in this heterogeneous environment.

width. Consequently, adding in-order small cores with high memory bandwidth for GC does not slow GC performance much, or at all, given the right design. The result is improvement to system PPE.

JIT. The JIT is also asynchronous and exhibits some parallelism. Because its role is optimization, the JIT is generally non-critical, which also makes it amenable to executing on separate cores. The JIT workload itself is very similar to Java application workloads and PPE therefore benefits from big core power-hungry features, such as out-of-order execution, bandwidth, and large caches. We show that because the JIT is not on the critical path, executing it at the highest performance is not important. Furthermore, putting the JIT on the small core takes it off the application's critical path. Slowing the JIT down on a small core does not matter because the JIT can deliver optimized code fast enough at much less power. On a small core, we can make the JIT more aggressive, such that it elects to optimize code earlier and delivers more optimized application code, with little power cost. The resulting system design improves total performance, energy, and PPE.

Interpreter. The interpreter is, however, on the critical path and is not asynchronous. The interpreter parallelism reflects the applications' parallelism and can thus often benefit from multiple cores. However, we find that the interpreter has very low ILP, a small cache footprint, and does not use much memory bandwidth. Therefore, executing interpreter threads on a high performance, high power core is inefficient, and a better choice for PPE and energy is to execute the interpreter on small cores.

Other VM Services. Other VM services may present good opportunities for heterogeneous multicore architectures, but are beyond the scope of this thesis. These services include zero-initialization [Yang et al., 2011], finalization, and profiling. For example, feedback-directed-optimization (FDO) depends on profiling of the running application. Such profiling is typically implemented as a producer-consumer relationship, with the instrumented application as the producer and one or more profiling threads as the consumers [Ha et al., 2011]. The profiler is parallel and exhibits an atypical memory-bound execution profile, making it a likely candidate for heterogeneous multicore architectures. However, we do not explore profiling here.

4.2 Methodology

We base our study on real power and performance measures of existing hardware, leveraging our existing well-developed tools and methodology related in Section 3.2. Evaluating the power and performance of future AMP systems is complex, just as evaluating managed languages can be challenging [Blackburn et al., 2006]. This section describes the hardware, measurements, workload, and software configuration that we use to explore hardware and software energy efficiency. We have made all of our data publicly available online.[2] This data includes quantitative measures of

[2] http://cecs.anu.edu.au/~steveb/downloads/results/yinyang-isca-2012.zip

Processor	**i7 (32)** Core i7-2600	**i3 (32)** Core i3-2120	**AtomD (45)** AtomD510	**Phenom II (45)** X6 1055T
Architecture	Sandy Bridge	Sandy Bridge	Bonnell	Thuban
Technology	32 nm	32 nm	45 nm	45 nm
CMP & SMT	4C2T	2C2T	2C2T	6C1T
LLC	8 MB	3 MB	1 MB	6 MB
Frequency	3.4 GHz	3.3 GHz	1.66 GHz	2.8 GHz
Transistor No	995 M	504 M	176 M	904 M
TDP	95 W	65 W	13 W	125 W
DRAM Model	DDR3-1333	DDR3-1333	DDR2-800	DDR3-1333

Table 4.1: Experimental processors.

experimental error and evaluations that we could not report here due to space constraints.

4.2.1 Hardware

Table 4.1 lists characteristics of the four experimental machines we use in this study. Hardware parallelism is indicated in the CMP & SMT row, the same as in Section 3.2.4. The notation nCmT means the machine has n cores (CMP) and m simultaneous hardware threads (SMT) on each core. The Atom and Sandy Bridge families are at the two ends of Intel's product line. Atom has an in-order pipeline, small caches, a low clock frequency, and is low power. Sandy Bridge is Intel's newest generation high performance architecture. It has an out-of-order pipeline, sophisticated branch prediction, prefetching, Turbo Boost power management, and large caches. We use two Sandy Bridge machines to explore hardware variability, such as cache size, within a family. We choose Sandy Bridge 06_2AH processors because they provide an on-chip RAPL energy performance counter [David et al., 2010]. We use the AMD Phenom II since it exposes independent clocking of cores to software, whereas the Intel hardware does not.

Together we use this hardware to mimic, understand, measure, and model AMP designs that combine *big* (fast, high power) cores with *small* (slow, low power) cores in order to meet power constraints and provide energy efficiency architectures.

4.2.2 Power and Energy Measurement

We use on-chip energy counters provided on Intel's Sandy Bridge processors [David et al., 2010], and an improvement to the Hall effect sensor methodology that we previously introduced in Section 3.2.5. Intel recently introduced user-accessible energy counters as part of a new hardware feature called RAPL (Runtime Average Power Limit). The system has three components: power measurement logic, a power limiting algorithm, and memory power limiting control. The power measurement logic uses activity counters and predefined weights to record accumulated energy in MSRs

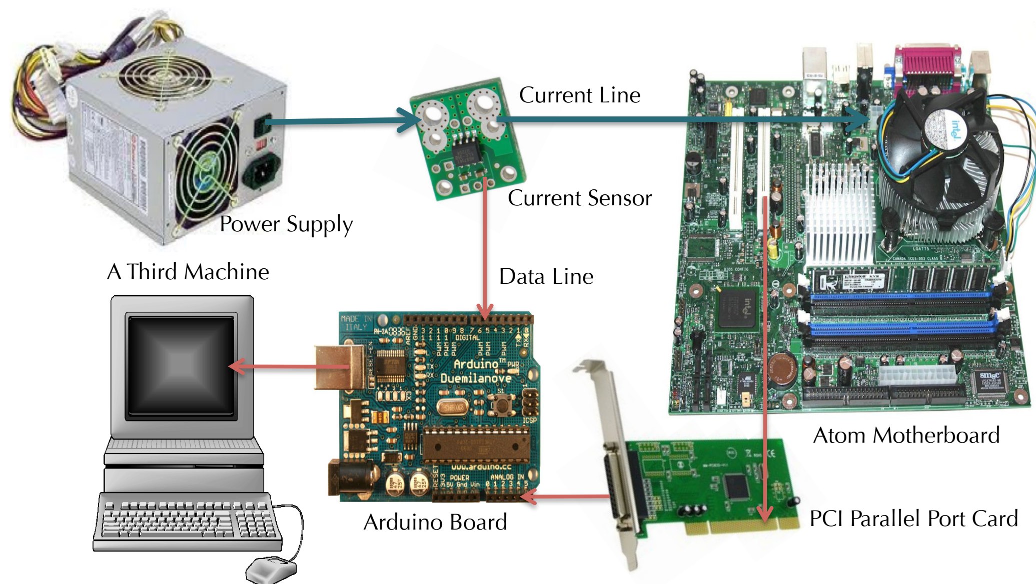


Figure 4.2: Hall effect sensor and PCI card on the Atom.

(Machine State Registers). The values in the registers are updated every 1 msec, and overflow about every 60 seconds [Intel, 2011]. Reading the MSR, we obtain package, core, and uncore energy. Key limitations of RAPL are: (1) it is only available on processors with the Sandy Bridge microarchitecture, and (2) it has a temporal resolution of just 1 msec, so it cannot resolve short-lived events. Unfortunately, VM services such as the GC, JIT and interpreter often occur in phases of less than 1 msec.

We extend the prior methodology for measuring power with the Hall effect sensor by raising the sample rate from 50 Hz to 5 KHz and using a PCI card to identify execution phases. Figure 4.2 shows a Pololu ACS714 Hall effect linear current sensor positioned between the power supply and the voltage regulator supplying the chip. We read the output using an Arduino board with an AVR microcontroller. We connect a PCI card to the measured system and the digital input of a Arduino board and use the PCI bus to send signals from the measured system to the microcontroller to mark the start and end of each execution phase of a VM service. Using the PCI card allows us to demarcate execution phases at a resolution of 200 μsec or better so we can attribute each power sample to the application or to the VM service being measured.

One limitation of this method is that the Hall effect sensor measures the voltage regulator's power consumption. We compared the Hall effect sensor to RAPL measurements. As expected, power is higher for the Hall effect sensor: 4.8% on average, ranging form 3% to 7%. We adjust for the voltage regulator by subtracting a nominal 5% voltage regulator overhead from Hall effect sensor measurements. We were unsuccessful in using this higher sample rate methodology on the AMD Phenom II, so we are limited to the lower sample rate methodology for it.

When measuring the i3 and i7, we use RAPL to measure energy and divide the

measurement by time to present average power. Conversely, when measuring the Atom and AMD, we use the Hall effect sensor to take power samples and integrate them with respect to time to present energy. We calculate performance as 1/time for a given workload. We compute relative performance per energy (PPE) values based on the experiment. For example, when we compute PPE of hardware feature X compared to feature Y running the same workload, we calculate:

$$\frac{PPE_X}{PPE_Y} = \frac{\frac{workload/run\ time_X}{energy_X}}{\frac{workload/run\ time_Y}{energy_Y}} = \frac{run\ time_Y \cdot energy_Y}{run\ time_X \cdot energy_X}$$

Because the interpreter normally executes in phases that are shorter than any of our methodologies can resolve, we only directly measure the power and energy of the interpreter by running the JVM in interpreter-only mode.

Static Power Estimates We take care to account for the static power consumption of cores when we model AMP systems using real hardware. Unfortunately, per-core static power data for production processors is not generally available, nor easy to measure [Le Sueur, 2011]. By measuring power with cores in different idle states, we estimate the per-core static power on the Phenom II at 3.1 W per core. For the Atom, we make a conservative estimate of 0.5 W per core. Note that static power consumption is temperature sensitive. The approach taken here, which assumes static power is constant and thus can be approximated by idle power, is therefore a first-order approximation. In the absence of better quality data, we were conservative and were able to demonstrate that our results are quite robust with respect to these estimates. The evaluation in Sections 4.4 and 4.6 uses the six-core Phenom II to model a single core system and a two core AMP system. Because the Phenom II powers the unused cores, we subtract the static power contribution of unused cores. When we model two Atom cores and one Phenom II core as part of the AMP system in Section 4.6, we subtract the static power for five AMD cores and add the static power for two Atom cores.

4.2.3 Hardware Configuration Methodology

We use hardware configuration to explore the amenability of future AMP hardware to managed languages. We are unaware of any publicly available simulators that provide the fine-grained power and performance measurements necessary for optimizing application software together with hardware. Compared to simulation, hardware configuration has the disadvantage that we can explore fewer hardware parameters and designs. Hardware configuration has an enormous execution time advantage; it is orders of magnitude faster than simulation. In practice, time is limited and consequently, we explore more software configurations using actual hardware. Measuring real hardware greatly reduces, but does not completely eliminate, the effect of inaccuracies due to modelling.

4.2.3.1 Small Core Evaluation

To understand the amenability of the various services to small cores in an AMP design, we use the i3 and Atom processors. The processors differ in two ways that are inconsistent with a single-die setting: a) they have different process technologies (32 nm vs 45 nm), and b) they have different memory speeds (1.33 GHz vs 800 MHz). Our comparisons adjust for both by down-clocking the i3's memory speed to match the Atom, and by approximating the effect of the technology shrink. Section 3.4.4 has found that a die shrink from 45 nm to 32 nm reduces processor power by 45% on two Intel architectures. We use the same factor, but do not adjust clock speed, on the grounds that a simple low power core may well run at a lower frequency.

To evaluate the overall power and PPE effects of deploying the GC and JIT on a low power core, we use the Phenom II and the Hall effect sensor without the PCI card. This methodology's inability to measure fine grain events is inconsequential because we are measuring overall system power and performance in the experiments where the Phenom II is used. As mentioned above, the Phenom II's separately clocked cores make it a good base case.

4.2.3.2 Microarchitectural Characterization

As we did in Section 3.2.7, we evaluate microarchitectural features using BIOS configuration. We explore the effect of *frequency scaling* on the i7, varying the clock from 1.6 GHz to 3.4 GHz. We normalize to 1.6 GHz. We explore the effect of *hardware parallelism* on the Phenom II by varying the number of participating CMP cores and on the i7 by varying CMP and SMT. We explore the effect of *last level cache* sizes by comparing the i7 and i3, each configured to use two cores at 3.4 GHz, but with 8 MB and 3 MB of LLC respectively. To explore sensitivity to *memory bandwidth*, we use the i3 with 800 MHz single channel memory relative to the default 1.33 GHz dual channel memory. To understand the effect of *gross microarchitectural change*, we compare the i3 and Atom running at the same clock speed, and make adjustments for variation in process technology, reducing the power of the Atom by 45% to simulate fabrication at 32 nm (see Section 3.4.4).

4.2.4 Workload

We use ten widely used Java benchmarks taken from the DaCapo suites and SPECjbb in this chapter, which can run successfully on Jikes RVM with its replay compilation: bloat, eclipse, and fop (DaCapo-2006); avrora, luindex, lusearch, pmd, sunflow, and xalan (DaCapo-9.12); and pjbb2005. All are multithreaded except for fop, luindex, and bloat. These benchmarks are non-trivial real-world open source Java programs [Blackburn et al., 2006].

4.2.5 Virtual Machine Configuration

All our measurements follow Blackburn et al.'s best practices for Java performance analysis [Blackburn et al., 2006] with the following adaptations to deal with the limitations of the power and energy measurement tools at our disposal.

4.2.5.1 GC

We focus our evaluation on concurrent GC because it executes concurrently with respect to the application and is thus particularly amenable to AMP. We also evaluate Jikes RVM's default production GC, generational Immix [Blackburn and McKinley, 2008]. We evaluated, but do not report, four other GCs to better understand GC workloads in the context of AMP. The measurements that we do not explicitly report here are available in our online data. All of our evaluations are performed within Jikes RVM's memory management framework, MMTk [Blackburn et al., 2004].

We report time for the production collector in Figure 4.1(a) because it is the best performing collector and thus yields the lowest time. In all other cases, we use the concurrent collector. Because GC is a time-space tradeoff, the available heap space determines the amount of work the GC does, so it must be controlled. We use a heap $1.5\times$ the minimum in which the collectors executes and is typical. For the concurrent collector, the time-space tradeoff is significantly more complex because of the concurrency of the collector's work, so we explicitly controlled the GC workload by forcing regular concurrent collections every 8 MB of allocation for avrora, fop, and luindex, which have a low rate of allocation, and 128 MB for the remaining benchmarks.

Measurement of concurrent GC faces two major challenges: (a) Except using RAPL on Sandy Bridge, we have no way of measuring the power and/or energy of a particular thread, and thus we cannot directly measure power or energy of concurrent collection, and (b) unlike full heap stop-the-world (STW) collectors, which suspend all application threads when collecting, concurrent collectors require the application to perform modest housekeeping work which can not be directly measured because it is finely entangled within the application. We use a STW variation on our concurrent collector to solve this methodological challenge. Using the STW-concurrent collector, a modified concurrent collector which behaves in all respects as a concurrent collector except that the application is stopped during GC phase, we can isolate and measure the application energy consumption and running time. Using concurrent GC, we measure the total energy and time of application and GC. We can then deduce the net overhead of concurrent GC by subtracting application energy and time from total energy and time measured when using concurrent GC.

The GC implementations expose software parallelism and exploit all available hardware contexts.

4.2.5.2 JIT

Because the unit of work for the JIT when executing normally is too fine grained for either RAPL or Hall effect measurements, we perform and measure all JIT work at

once from a *replay* profile on Jikes RVM [Blackburn et al., 2006]. *Replay compilation* removes the nondeterminism of the adaptive optimization system. The methodology uses a compilation profile produced on a previous execution that records what the adaptive compiler chose to do. It first executes one iteration of the benchmark without any compilation, forcing all classes to be loaded. Then the compilation profile is applied en mass, invoking the JIT once for each method identified in the profile. We measure the JIT as it compiles all of the methods in the profile. We then disable any further JIT compilation and execute and measure the application on the second iteration. The application thus contains the same mix of optimized and unoptimized code as it would have eventually had with regular execution, but now we can measure both the compiler and application independently and thus the experiments are repeatable and measurable with small variation. To decrease or eliminate GC when measuring the JIT, we use Jikes RVM's default generational Immix GC, since it performs the best, and set the heap size to be four times the minimum size required.

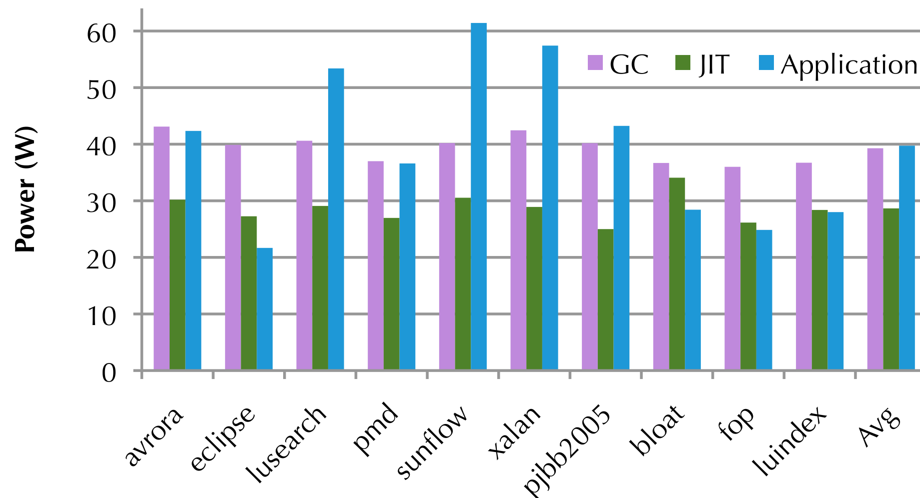
Normally the JIT executes asynchronously to the application and GC. Although the JIT compiler could readily exploit parallelism by compiling different code in multiple threads, Jikes RVM's JIT is not parallel. We thus evaluate the JIT on a single hardware context. When we evaluate the JIT and GC together we use multiple hardware contexts.

4.2.5.3 Interpreter

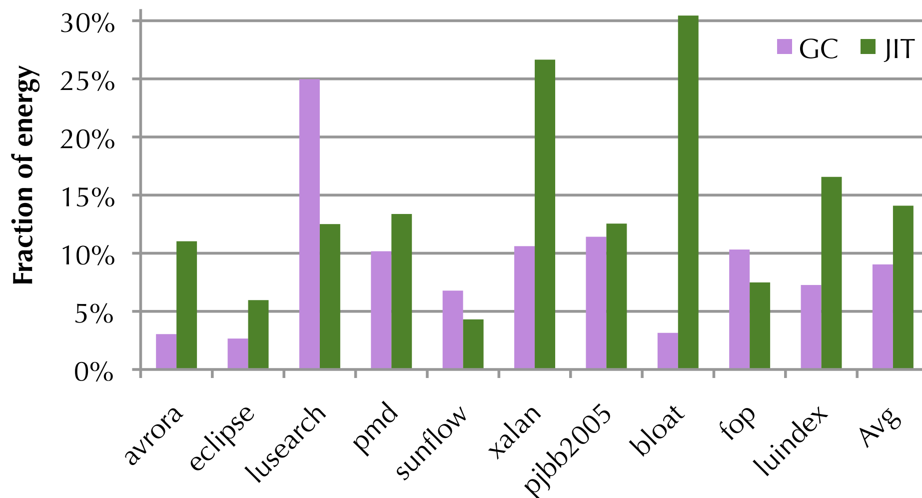
Evaluating the interpreter is challenging because interpretation is finely interwoven with optimized code execution. We evaluate the interpreter two ways, using two JVMs, the Oracle HotSpot JDK 1.6.0 and Jikes RVM. HotSpot interprets bytecodes and Jikes RVM template compiles them. Both adaptively (re)compile hot methods to machine code. We first use Jikes RVM and a timer-based sampler to estimate the fraction of time spent in interpreted (template-compiled) code. We use HotSpot to evaluate the interpreter's microarchitectural sensitivity. We execute HotSpot with the compiler turned off, so that all application code is interpreted. Because the interpreter is tightly coupled with the application, it exhibits no independent parallelism and cannot execute asynchronously with respect to the application. However, it reflects all software parallelism inherent in the application.

4.3 Motivation: Power and Energy Footprint of VM Services

Figure 4.3 shows the overall contribution of the GC and JIT of Jikes RVM to system power and energy on existing hardware. As we mentioned in the previous section, the interpreter's fine-grained entanglement with the application makes it too difficult to isolate and measure its power or energy with current tools. Figure 4.1(a) shows that the fraction of cycles due to the interpreter is significant. Although we do not evaluate the interpreter further here, it remains a significant target for power and energy optimization on future AMP systems. Figure 4.3 reports isolated GC and JIT power and energy on a stock i7 (4C2T at 3.4 GHz). Our methodology generates



(a) Average power for GC, JIT, and application.



(b) Fraction of energy due to GC and JIT.

Figure 4.3: GC, JIT, and application power and energy on i7 4C2T at 3.4 GHz using Jikes RVM. The power demands of the GC and JIT are relatively uniform across benchmarks. Together they contribute about 20% to total energy.

typical JIT and GC workloads. We use the JIT workload during warm-up (the first iteration of each benchmark), in which it performs most of its work. We use the GC and application workload in steady state (5th iteration), where the GC sees a typical load from the application and the application spends most its time in optimized code. Since the GC is concurrent and parallel, it utilizes all available hardware. The JIT is concurrent, but single threaded, although JIT compilation is not intrinsically single threaded. The benchmarks themselves exhibit varying degrees of parallelism.

Figure 4.3(a) shows that power consumption is quite uniform for the GC and JIT regardless of benchmark on the i7, at around 40 W and 30 W respectively. The JIT has lower power because it is single threaded, whereas the parallel GC uses all four cores and SMT. This power uniformity shows that the GC and JIT workload are both relatively benchmark independent. By contrast, application power varies by nearly a factor of three, from 22 W to 62 W, reflecting the diverse requirements and degrees of parallelism among the benchmarks.

Figure 4.3(b) shows that the GC and JIT each contribute about 10% on average to the total energy consumption, totalling about 20%. This confirms our hypothesis that VM services may provide a significant opportunity for energy optimization. Recall from Figure 4.1(a) that GC and JIT totally use around 20% CPU cycles (although with different software and hardware configurations), which also provides a significant opportunity for performance optimization. The figure also shows significant variation in total GC and JIT energy consumption across benchmarks, despite their uniform power consumption. This variation, together with the data in Figure 4.1(a), reflects the different *extents* to which the benchmarks exercise the JIT and GC. Some benchmarks require more or less GC and JIT services, even though the behavior of each service is quite homogeneous.

4.4 Amenability of VM Services to a Dedicated Core

The above results demonstrate the potential for energy savings and we now explore the amenability of executing VM services on dedicated small cores. This experiment explores whether the services will a) benefit from dedicated parallel hardware, and b) be effective, even if the hardware is slow.

Because we are not yet considering the small core microarchitecture, we use existing stock hardware. We choose the AMD Phenom II because it provides independent frequency scaling of the cores. We model the small core by down-clocking a regular core from 2.8 GHz to 2.2 GHz and 0.8 GHz. We subtract our conservative estimate of static power of 3.1 W per core from our measurements to avoid overstating our results. (Section 4.2.2 explains the estimation.)

We bind the VM service(s) to separate cores and measure the entire system. We use regular adaptive JIT because it interleaves its work asynchronously and in parallel with the application (replay compilation, although easier to measure, does neither). We measure total time for the first iteration of each benchmark because the first iteration is a representative JIT workload that imposes significant JIT activity

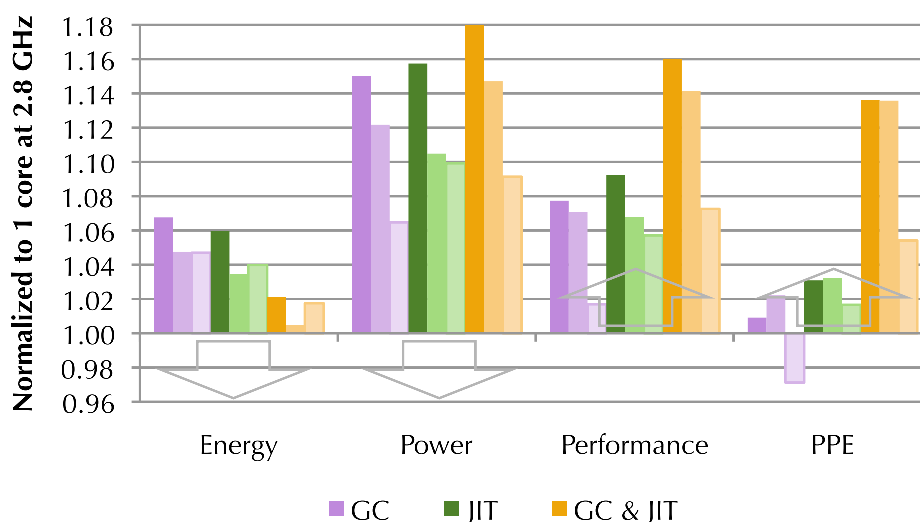


Figure 4.4: Utility of adding a core dedicated to VM services on total energy, power, time, and PPE using Jikes RVM. Overall effect of binding GC, JIT, and both GC & JIT to the second core running at 2.8 GHz (dark), 2.2 GHz (middle), and 0.8 GHz (light) on the AMD Phenom II corrected for static power. The baseline uses one 2.8 GHz core.

(see Figure 4.1(a)).

Figure 4.4 shows the effect of adding a dedicated core for GC and JIT services on energy, power, performance, and PPE. The light grey arrows show which direction is better (lower for energy and power, and higher for performance and PPE). The leftmost bar in each cluster shows the effect when the additional core runs at 2.8 GHz, the same speed as the main core. This result evaluates our first question: whether the system benefits from hardware parallelism dedicated to VM services. For both the GC and JIT, the introduction of dedicated hardware improves performance by 8-10% while increasing power by around 15%. They independently increase energy by around 6% and marginally improve PPE. When both GC and JIT are bound to the additional core the effect is amplified, leading to a net PPE improvement of 13% and a 2% increase in energy. This data shows that simple binding of the JIT and GC to an additional core is effective.

In this chapter however, we are exploring future systems in which adding more big cores is not feasible because of power or energy constraints. If it were, dedicating a big core to the VM services rather than sharing it with the application is probably a poor design choice.

The lighter bars in Figure 4.4 show the effect of slowing down the dedicated VM services core. The power overhead of the additional core is reduced by nearly half when the dedicated core is slowed down to 0.8 GHz. However performance also suffers, particularly for GC. Nonetheless, the PPE improves for the JIT and JIT & GC cases, which indicates that the GC and JIT could efficiently utilize a small core. A very promising result from Figure 4.4 is that executing JIT and GC on the dedi-

cated core at 2.2 GHz delivers a 13% higher PPE than one processor with virtually no energy cost. From this figure, we can also see that while we reduce the JIT core frequency from 2.8 GHz to 0.8 GHz (reduced 70%), the application performance only reduces around 3%. So even though we slow the JIT down, it still can deliver optimized code fast enough, which provides the opportunity to run JIT on a small core with much lower power.

We also measure the effect on power and energy of introducing a small core *without* binding the VM services. This configuration led to a significant slow down, which is illustrated in Figure 4.1(b). There was also a modest increase in energy, which together lead to a 30% degradation in PPE. This result emphasizes that without binding of tasks or some other guidance to the OS scheduler, the addition of a small core is counterproductive.

4.5 Amenability of VM services to Hardware Specialization

A single-ISA heterogeneous processor design has the opportunity to include a variety of general purpose cores tailored to various distinct, yet ubiquitous, workload types. Such a design offers an opportunity for very efficient execution of workloads that do not well utilize big out-of-order designs. We now explore the amenability of VM services to alternative core designs. We start with measuring a stock small processor, the in-order, low power, Atom. We then evaluate how the VM services respond to a range of microarchitectural variables such as clock speed, cache size, etc. To ease measurement, this analysis evaluates the GC and application with respect to a steady state workload and the replay JIT workload. We evaluate the interpreter using the HotSpot JDK with its JIT disabled and measure the second iteration of the benchmark, which reflects steady state for the interpreter.

4.5.1 Small Core

We first compare contemporary in-order cores to out-of-order cores. We use an i3 and Atom with the same degree of hardware parallelism (2C2T), and run with the same memory bandwidth to focus on the microarchitectural differences. We do not adjust for clock frequency in this experiment on the grounds that the small core may well run at a lower clock, so they execute at their 3.4 GHz and 1.66 GHz default frequencies respectively. We explore the effect of clock scaling separately, below. An important difference between the two machines is their process technology. To estimate the effect of shrinking the process, we project the Atom power consumption data to 32 nm by reducing measured power by 45%, as described in Section 4.2.3.1.

Figure 4.5 shows the effect on energy, power, performance, and PPE when moving from the i3 to the in-order Atom. The figure shows, unsurprisingly, that the Atom offers lower energy, power, and performance in all cases. Power is uniformly reduced by around 90%. Of these, the GC benefits the most because its performance decrease

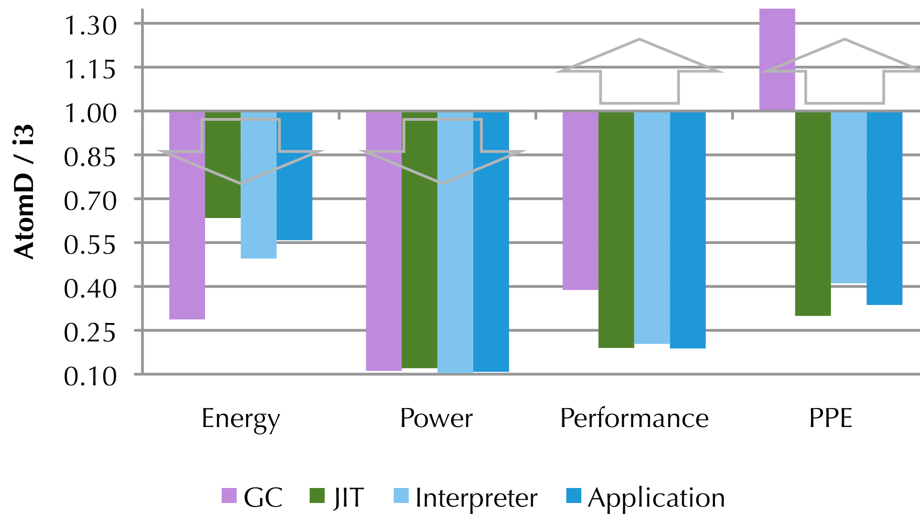


Figure 4.5: Amenability of services and the application to an in-order-processor. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3. These results compare execution on an in-order Atom to an out-of-order i3.

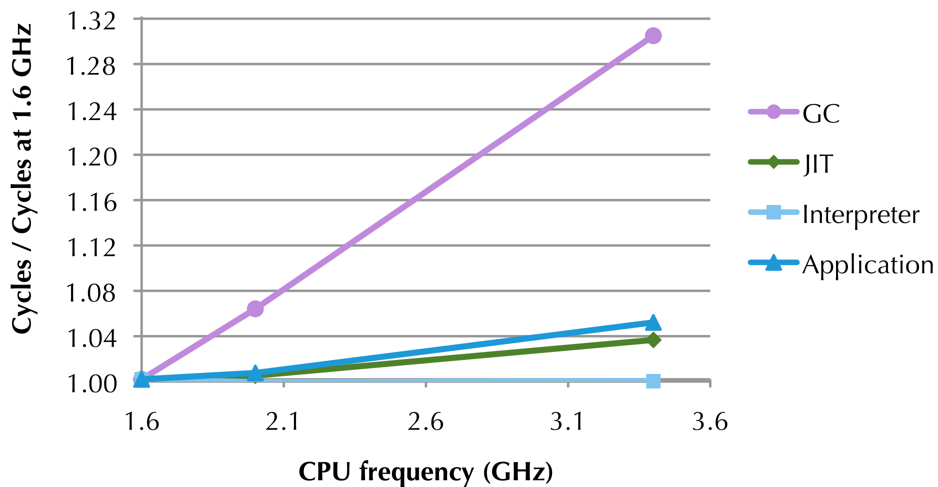


Figure 4.6: Cycles executed as a function of clock speed, normalized to cycles at 1.6 GHz on the i7. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3. The workload is fixed, so extra cycles are due to stalls.

is less than for the others. The consequence is greater energy reduction and a net improvement in PPE of 35%. The JIT, interpreter, and application all see degradations in PPE of around 60-70%.

This data makes it emphatically clear that the simple memory-bound graph traversal at the heart GC is much better suited to small low power in-order processors. In the case of the JIT, the 35% energy reductions may trump degraded PPE because the

JIT can be trivially parallelized, which improves performance without significantly increasing energy, leading to improved PPE. The small core will be a power- and energy-efficient execution context for the interpreter in contexts when the performance reduction is tolerable. Since performance-critical code is typically optimized and therefore not interpreted, it is plausible that interpreted code is less likely to be critical and therefore more resilient to execution on a slower core.

Figure 4.6 provides insight into why the GC does so well on the Atom. This graph shows the effect on the total cycles executed when scaling the clock on a stock i7. The application and JIT execute just 4-5% more cycles as the clock increases from 1.6 GHz to 3.4 GHz while the GC executes 30% more cycles. These extra cycles are due to stalls. The result is unsurprising given the memory-bound nature of GC. It is also interesting to note what this graph reveals about the interpreter. The higher clock speed induces no new stalls, so the interpreter's performance scales perfectly with the clock frequency. This result reflects the fact that an interpreter workload will invariably have very good instruction locality because the interpreter loop dominates execution. We confirmed this hypothesis by directly measuring the frequency of last level cache accesses by the interpreter and found that it was 70% lower than the application code. Both the interpreter and the GC exhibit atypical behavior that suitably tuned cores should be able to aggressively exploit for greater energy efficiency.

4.5.2 Microarchitectural Characterization

This section further explores the sensitivity of the GC, JIT, interpreter, and application workloads to microarchitectural characteristics. This characterization gives more insight into the amenability of the services to hardware specialization. Figure 4.7 and 4.8 show the result of varying: hardware parallelism, clock speed, memory bandwidth, cache size, and gross microarchitecture. For the SMT and CMP experiments, we drop the three single threaded benchmarks, and use the seven multithreaded ones to focus on sensitivity with software parallelism. Similarly, six of the applications fit in the smaller 3 MB last-level cache, so we use the ones that do not (bloat, eclipse, pmd and pjbb2005) to compare with VM services. To generate the GC workload, we drop avrora, fop, and luindex here because they have high variation and low GC time. Note the different y-axis scales on each graph.

4.5.2.1 Hardware Parallelism

We study the effects of hardware parallelism using the i7. To evaluate CMP performance, we compare 1C1T and 2C1T configurations, which disable hyperthreading and ensure that software parallelism is maximally exposed to the availability of another core. Conversely, to evaluate SMT performance, we compare 1C1T and 1C2T configurations, which use just one core to maximize exposure of software parallelism to the addition of SMT. In all cases, the processor executes at its 3.4 GHz stock frequency. Because the JIT of Jikes RVM is single threaded, we omit the JIT from this

part of the study. However, some JITs are parallel and would be amenable to hardware parallelism.

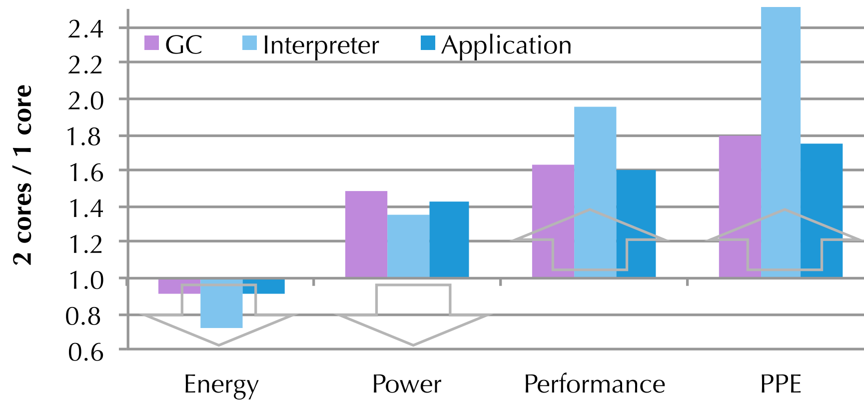
Figure 4.7(a) shows that increasing the number of cores improves the PPE of both GC and multithreaded applications similarly. The interpreter sees an even greater benefit, which is likely due to the lower rate of dependencies inherent in the slower interpreter loop relative to typical optimized application code. Figure 4.7(b) shows that SMT does not improve PPE as much as CMP, but effectively decreases energy. In Section 3.4.2, we have shown SMT requires very little additional power as compared to CMP and is a very effective choice given a limited power budget. The advantage of the interpreter over the other workloads is even more striking here. Six multithreaded benchmarks exhibit dramatic interpreter performance improvements on SMT and CMP respectively: sunflow (87%, 164%), xalan (54%, 119%), avrora (55%, 74%), pmd (45%, 113%), lusearch (53%, 111%), and pjbb2005 (36% and 67%). Eclipse is the only multithreaded benchmark that did not improve due to hardware parallelism when fully interpreted. These results show that VM services can very effectively utilize available hardware parallelism.

4.5.2.2 Clock Speed

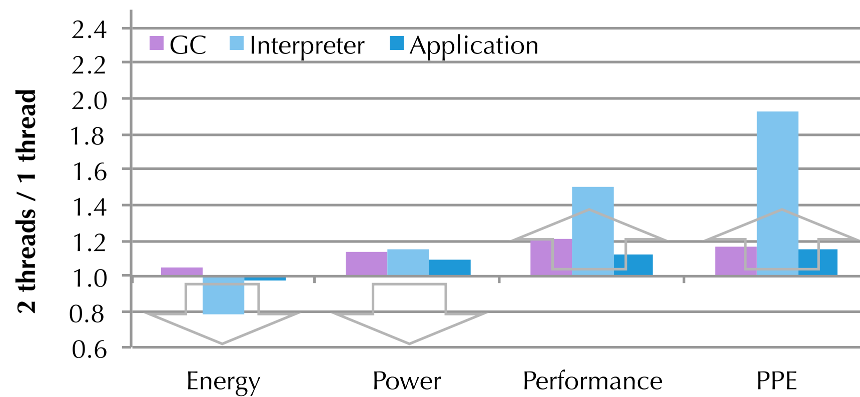
Figure 4.7(c) plots performance (x-axis) and energy (y-axis) as a function of clock frequency on the i7. The single threaded JIT uses a 1C1T configuration, while all other workloads use the stock 4C2T configuration. Values are normalized to those measured at the minimum clock frequency (1.6 GHz). The different responses of each workload is striking. The JIT, interpreter, and application all improve their performance two-fold as the clock increases. On the other hand, the GC performance improves very little beyond 2.0 GHz. Figure 4.6 and our measurements of cache miss rates suggest that memory stalls are the key factor in this result. The different workloads also have markedly different energy responses. The single threaded JIT energy only increases by 4% going from the lowest to the maximum clock speed, whereas the application and GC energy consumption increase by 25% and 23% respectively with clock speed increases.

4.5.2.3 Memory Bandwidth

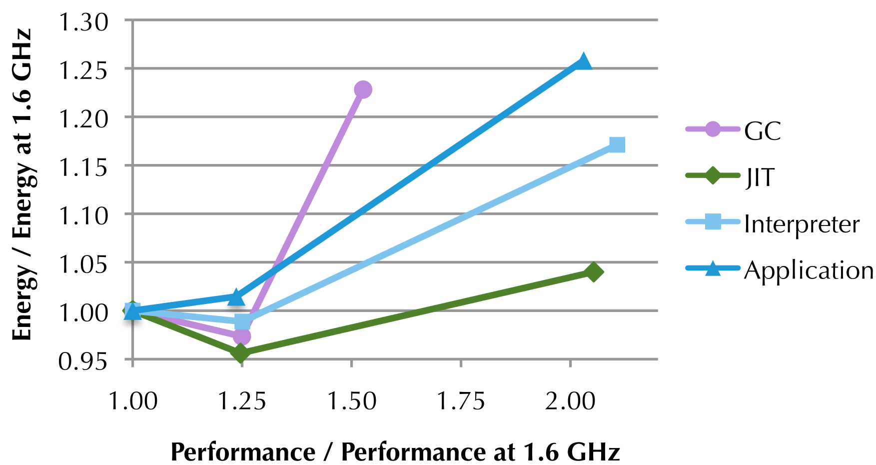
Figure 4.8(a) shows the effect of increasing memory bandwidth from a single channel at 0.8 GHz to two channels at 1.33 GHz on the i7. The increase in memory bandwidth reduces CPU energy for all workloads, but most strikingly for GC. The GC performance increases dramatically, leading to a 2.4× improvement in PPE. Of course the increased memory bandwidth will lead to increased energy consumption by the off-chip memory subsystem, and our measurements are limited to the processor chip and thus not captured by this data. The GC's response to memory bandwidth is unsurprising, given that the workload is dominated by a graph traversal. It is interesting to note that while the JIT and the application also see significant, if less dramatic, improvements, the interpreter does not. The results in Figure 4.8(a) are



(a) Effect of CMP on GC, interpreter, and multithreaded application on i7.

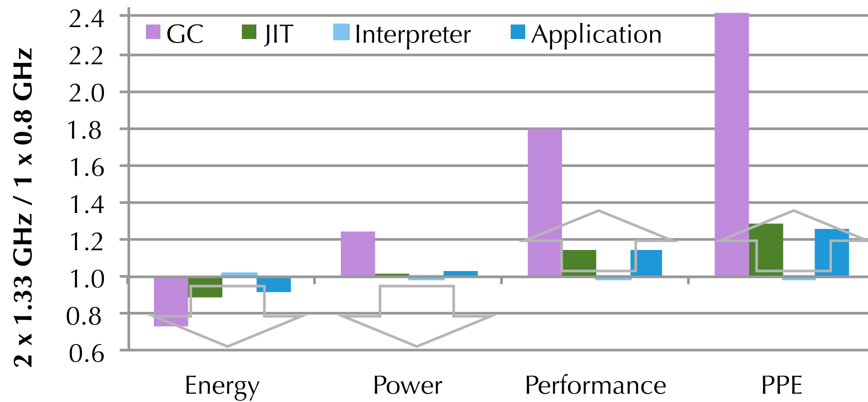


(b) Effect of SMT on GC, interpreter, and multithreaded application on i7.

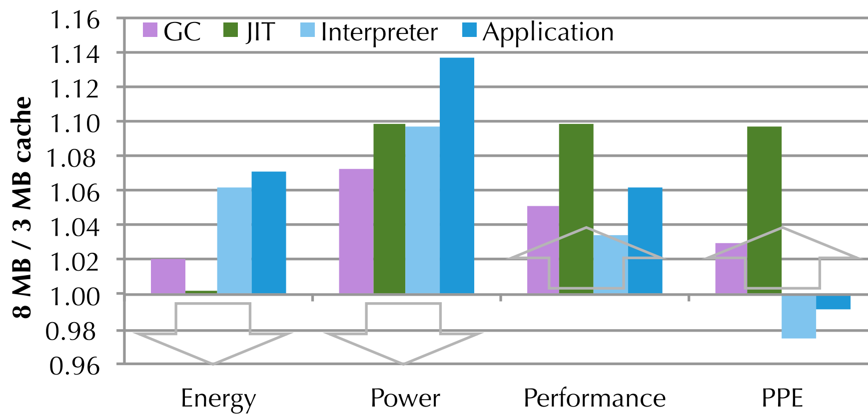


(c) Effect of clock frequency on performance and energy. Points are clock speeds 1.6, 2.0, and 3.4 GHz (from left to right) on i7.

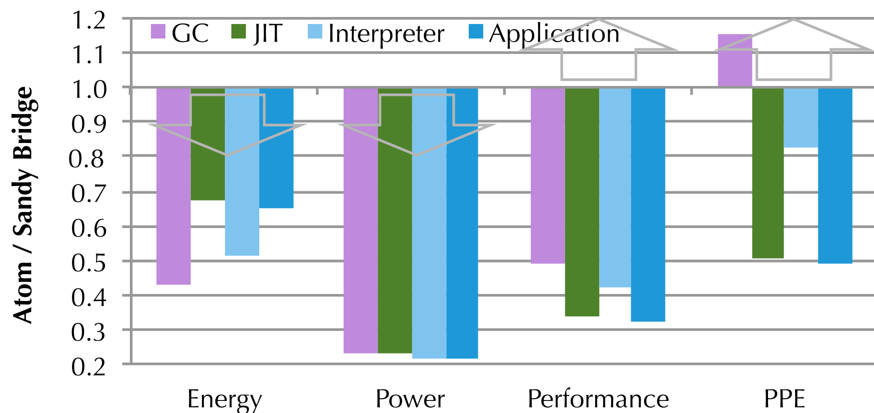
Figure 4.7: Microarchitectural characterization of VM services and application. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3.



(a) Effect of memory bandwidth (1.33 GHz x 2 channels vs 0.8 GHz x 1 channel) on i7.



(b) Effect of LL cache size (8 MB vs 3 MB) on i7 and i3.



(c) Effect of gross architecture on Sandy Bridge and Atom at same frequency, hardware threads, and process technology.

Figure 4.8: Microarchitectural characterization of VM services and application. GC, JIT, and interpreter use Jikes RVM. Interpreter uses HotSpot JDK, as stated in Section 4.2.5.3.

quite consistent with the results in Figure 4.6. The GC is memory-bound and sensitive to memory performance. The interpreter has excellent locality and is relatively insensitive to memory performance.

4.5.2.4 Last-level Cache Size

A popular use of abundant transistors is to increase cache size. In Figure 4.8(b), the experiment evaluates the approximate effect of increased cache size on the GC and application using the i7 and i3. We configure them with the same hardware parallelism (2C2T) and clock frequency (3.4 GHz). After controlling for hardware parallelism and clock speed, the most conspicuous difference between the systems is their last level cache: the i7's LL cache is 8 MB and i3's is 3 MB. Six of our ten benchmarks are insensitive to the large cache size. Cache is not a good use of transistors for them. The four cache-sensitive benchmarks (bloat, eclipse, pmd and pjbb2005) have large minimum heap sizes. For these benchmarks, the increase in LL cache size from 3 MB to 8 MB is only a small amount of the average maximum volume of live objects. Note the y-axis scale in Figure 4.8(b). Even these benchmarks only improve performance by 6% with a larger cache. The larger cache improves JIT performance and PPE by 10%, but these improvements are very modest in comparison with the other hardware features explored in Figure 4.7 and Figure 4.8. The interpreter, which has good locality, sees a reduction in PPE when the cache size is increased.

4.5.2.5 Gross Microarchitecture

Figure 4.8(c) compares the impact of gross microarchitecture change using the Atom and i3, controlling for clock frequency (1.66 GHz for Atom and 1.6 GHz for i3), hardware parallelism (2C2T), and memory bandwidth. We configure them both to use 800 MHz single channel memory. We adjust for process technology on the Atom, as we do above, by scaling by 45%. The results make it clear that the GC is better suited to the small in-order Atom than a big core, with the high performance architectural features of the i3. GC has a better PPE on the Atom than on the i3. The interpreter does not do as well as the GC, but the benefit of the i3 is muted compared to its performance benefit for the JIT and the application.

4.5.3 Discussion

These results show that each of the VM services exhibit a distinctly different response to microarchitectural features. The similarities between the JIT and application are not a surprise. The characteristics of compilers have been studied extensively in the literature and are included in many benchmark suites including the ones we use here. On the other hand, the GC and interpreter each exhibit striking deviations from the application. This heterogeneity presents designers with a significant opportunity for energy optimizations that tune small general purpose cores more closely to the microarchitectural needs of this ubiquitous workload.

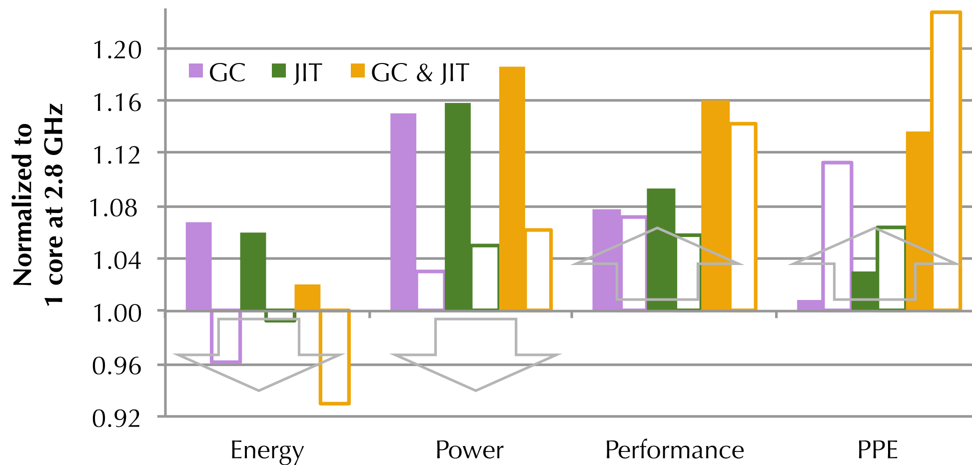


Figure 4.9: Modeling total energy, power, time, and PPE of an AMP system. The light bars show a model of an AMP with one 2.8 GHz Phenom II core and two Atom cores dedicated to VM services. The dark bars reproduce Phenom II results from Figure 4.4 that use a 2.8 GHz dedicated core for VM services.

4.6 Modeling Future AMP Processors

This section evaluates a hypothetical AMP scenario. The absence of a suitable AMP processor in silicon and our software constraints for executing on an x86 ISA motivate using a model. Our model combines measured application power and performance results on the big Phenom II core at 2.8 GHz with the power and performance results for GC and JIT on two small Atom cores at 1.66 GHz. We compare this model to the Phenom II AMP core results from Figure 4.4, which we corrected for static power. Because the Atom cores are projected into the Phenom II die, we assume the Phenom II's memory bandwidth and LL cache size, and accordingly raise the Atom's performance by 79% and 5% for the GC and 2% and 10% for the JIT, based on our measurements in Figures 4.8(a) and 4.8(b).

For the small cores, which run the VM services, we model GC throughput by straightforwardly scaling up our measures of GC throughput on the regular Atom according to the adjustments for the Phenom II cache size and memory bandwidth. We do the same for the JIT, however, we also scale the JIT from 1C1T to 2C2T, making the assumption that the JIT will scale as well as typical applications. This adjustment is conservative since the JIT is embarrassingly parallel.

To make our model more accurate, we capture the effect on the application of slowing down the GC and JIT. We leverage the results on the Phenom II that downclock cores in Figure 4.4. The modeled performance for the GC on the two Atom cores is slightly better than on the Phenom II at 2.2 GHz, and the JIT modeled on the two Atom cores is slightly better than on the Phenom II at 0.8 GHz. We thus use the application measurements when running with the 2.2 GHz and 0.8 GHz dedicated VM services cores respectively. This measurement conservatively captures the effect on the application of slowing the GC and JIT.

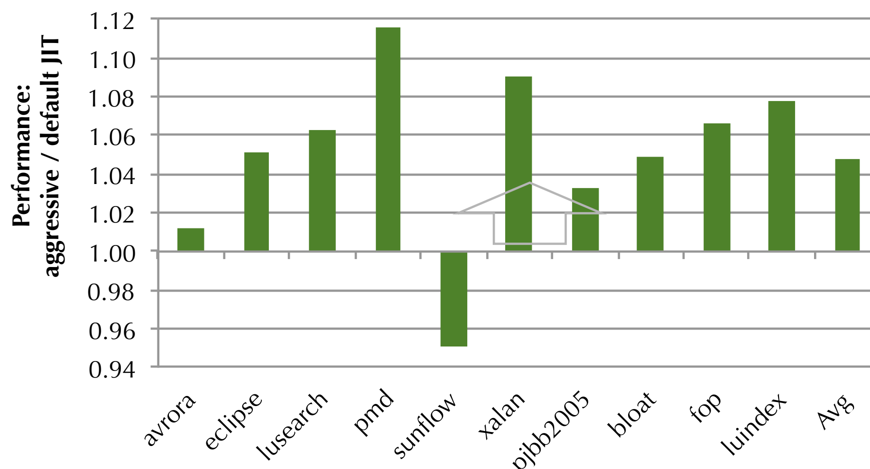


Figure 4.10: Adjusting the JIT's cost-benefit model to account for lower JIT execution costs yields improvements in total application performance.

Figure 4.9 shows that even our conservative model of the two Atom cores (light bars) is promising. For the GC alone, energy improves by 4% due to a performance improvement of 7% but at only a 3% power cost. The JIT alone has slightly worse total performance impact, but at a commensurate power cost. Individually, the PPE improvement is 11% and 6% from GC and JIT respectively. When we bind both the GC and JIT to the small cores, the result is more than additive results because of the better utilization of the small cores. Total performance is very similar to that of the 2.8 GHz dedicated core but power, energy, and PPE are all markedly improved. The performance improvement is 13% with only a 5% power cost, resulting in a 7% energy improvement and a 22% PPE improvement.

Discussion. To realize these gains requires more research and infrastructure. At least, we will need (1) big/small core hardware with new hardware features, e.g., voltage scaling on a thread basis to, for example, accelerate the interpreter; (2) on-chip power meters, (3) OS or VM scheduling support; (3) concurrent GC and parallel JIT tuned for the hardware; and (4) scheduling algorithms to coordinate application and VM threads.

4.7 Further Opportunity for the JIT

Although the JIT is broadly similar to the applications and therefore may not offer an opportunity for tuned hardware, executing the JIT on small cores does offer a new opportunity to improve application code quality. Minimizing JIT cost means that in theory, the JIT can optimize more code, more aggressively, improving application code quality. Figure 4.10 evaluates this hypothesis. It compares the total performance (GC, application, and interpreter, but not JIT) using the standard optimization decisions that Jikes RVM produces on the first iteration using its cost benefit analysis (see Section 2.2.2), to a more aggressive cost model that compiles code sooner and

compiles more code. We configure the compiler cost model by reducing the cost of compilation by a factor of 10. Figure 4.10 shows that making the JIT more aggressive improves total performance by around 5%. This result suggests that software/hardware co-design has the potential to improve power, performance, and PPE further.

4.8 Summary

This chapter first showed the advantages and disadvantages of VMs and AMP architectures. VM services provide abstractions but consume significant resources. AMP architectures provide potential efficiency but behave inefficiently without software support. We identified that the opportunities and challenges of AMP processors and managed software are complementary. The VM's overhead can be hidden by utilizing the small cores on an AMP, and the AMP's hardware complexity can be abstracted by the managed languages. Using power and performance measurements of modern hardware and very conservative models, we showed that the addition of small cores for GC and JIT services alone should deliver, at least, improvements in performance of 13%, energy of 7%, and performance per energy of 22%.

This chapter explored efficiency improvements by running VM services on the small cores of AMP architectures. The next chapter extends the exploration of managed languages and AMP processors to application threads and VM services threads, and develops an algorithm that uses online metrics of parallelism, communication, and core sensitivity to schedule threads to the appropriate cores for improved system efficiency.

Chapter 5

A VM Scheduler for AMP

The previous chapter shows that by manually scheduling Virtual Machine (VM) services to the small cores on Asymmetric Multicore Processors (AMPs), the VM can improve performance and energy. This chapter presents a dynamic VM scheduler that shields both the VM threads and application threads from the hardware complexity of AMP architectures, resulting in substantial performance and energy improvements for managed applications.

Section 5.2 evaluates how well the existing schedulers perform for different benchmark groups. Section 5.3 presents how we dynamically identify benchmark parallelism online and Section 5.4 presents our model to predict thread sensitivity to different core types. Section 5.5 introduces the WASH (Workload Aware Scheduler for Heterogeneity) algorithm. Section 5.6 describes the hardware, measurements, workload, and software configuration used in this chapter. Section 5.7 shows the performance and energy improvements by using WASH on three different big/small core configurations.

This chapter describes work in the paper under review "Dynamic Analysis and Scheduling Managed Software on Asymmetric Multicore Processors" [Jibaja, Cao, Blackburn, and McKinley, 2014]. In this collaboration, I was primarily responsible for the detection of thread contention, dynamic profiling, and the scheduling algorithm.

5.1 Introduction

This chapter presents a fully automated AMP-aware runtime system that transparently delivers on the performance and energy efficiency potential of AMP. Prior work focused on: core sensitivity of each thread [Saez et al., 2011, 2012; Craeynest et al., 2012; Kumar et al., 2004; Becchi and Crowley, 2006]; using the number of threads to determine a schedule [Saez et al., 2010]; prioritizing threads on the critical path in multithreaded applications [Du Bois et al., 2013; Suleman et al., 2010; Joao et al., 2012, 2013]; or statically binding special threads to cores as in Chapter 4;. Each approach addresses only a part of the problem and is thus insufficient. Core sensitivity identifies the different microarchitectural needs of threads; the number of threads and critical paths are essential to efficient scheduling; and exploiting a priori knowledge of special purpose threads is profitable. Unfortunately, this prior work either

requires applications that: are parallel and scalable; have one thread or N where N is the number of hardware contexts; or have critical locks annotated by the developer so that the runtime can identify bottlenecks [Du Bois et al., 2013; Joao et al., 2012, 2013]. In contrast, this paper presents an AMP runtime system that does not restrict the number of threads; automatically identifies application parallelism (none, scalable, non-scalable); assesses and manages thread criticality, progress, and core sensitivity; and requires no changes to applications.

We explore managed language workloads and their specific challenges and opportunities. By comparison to native parallel workloads in prior work, managed workloads are complex and *messy*. Because managed runtime VMs contain helper threads, such as the garbage collector (GC), just-in-time compiler (JIT), profiler, and scheduler that run together with the application, they offer a mess of heterogeneous threads with different functions and amounts of work. Adding to this mess, applications that run on managed VMs use concurrency in diverse ways, and as a result are often not scalable. On the other hand, VMs offer an opportunity because they already profile, optimize, and schedule applications, and have a priori knowledge about their helper threads.

This chapter presents the design and implementation of an AMP-aware runtime that consists of: (1) a model of core sensitivity that predicts and assesses thread performance on frequency-scaled cores and expected thread progress using performance counter data; (2) a dynamic parallelism analysis that piggybacks on the underlying locking mechanism to determine scalability, cores, and contention; and (3) a new Workload Aware Scheduler for Heterogeneous systems (WASH) that optimizes based on thread type (application or runtime helper), each thread's core sensitivity, progress, and parallelism.

A key contribution of this work is the VM's dynamic parallelism analysis. We show that the dynamic parallelism analysis is necessary to produce good schedules. To maximize performance and energy efficiency: (1) the scheduler must identify critical thread(s) in non-scalable workloads and preferentially schedule them on big cores; (2) the scheduler must place the single thread in a sequential application on a big core and all runtime helper threads on small cores; and (3) it must fairly schedule scalable applications threads on *both* big and small cores based on thread progress. We demonstrate that our dynamic parallelism analysis accurately classifies lock contention as a function of thread count, work, and waiting time on a range of applications and it differentiates variants of a single application configured either as scalable or non-scalable.

We implement our AMP runtime in Jikes RVM. We evaluate 14 benchmarks from DaCapo and SPECjbb (five single threaded, five non-scalable, and four scalable) executing on the AMD Phenom II with core-independent frequency scaling configured as an AMP. We compare to the Linux OS AMP-oblivious round-robin scheduler, which works well for scalable workloads but underutilizes big cores in other workloads; and the fixed static assignment of helper threads to small cores from Chapter 4, which works well on single threaded benchmarks but underutilizes the small cores in other workloads. We show that neither of these approaches work consistently well

on AMP with non-scalable workloads. WASH consistently improves performance and energy in all hardware configurations by utilizing both small and big cores more appropriately. WASH delivers substantial improvements over the prior work: 20% average performance and 9% energy on a range of AMP hardware configurations. More efficient AMP designs that perform voltage scaling (not only frequency scaling) and/or combine different microarchitectures will likely perform better.

Our approach successfully exploits two key aspects of a managed runtime, but both could be generalized to a kernel-level scheduler. (1) We exploit a priori knowledge of managed runtime helper threads and their respective priorities. The VM could communicate this information via thread priorities (e.g., assigning low priority to JIT and GC threads), such that the OS will preferentially schedule them on small cores. (2) We exploit the biased locking optimization in modern managed runtimes [Bacon et al., 1998; Dice et al., 2010; Android, 2014], which cost-effectively identifies *true* contention. Prior work relied on programmer assistance to identify contention [Joao et al., 2012, 2013], in part since neither Linux or Windows implement biased locking. However, some pthread implementations implement lock optimizations [Android, 2014], and the utility of such optimizations in an AMP setting may motivate their wider adoption. Although our evaluation focusses on managed languages, we are optimistic that this approach applies beyond the confines of managed languages.

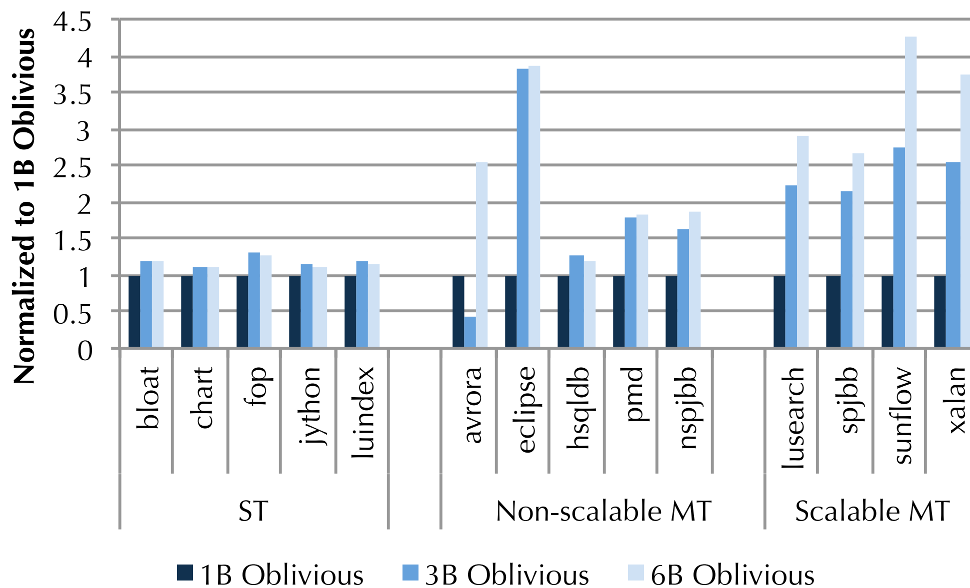
5.2 Workload Analysis & Characterization

The analysis in this section leads to the following insights. (1) Existing schedulers perform well for one of single-threaded or multithreaded workloads, but not both. (2) No scheduler works consistently well for non-scalable multithreaded workloads. (3) The OS currently does not have sufficient information to differentiate application and helper threads. Section 5.3 presents our online analysis that dynamically identifies parallelism and Section 5.4 presents an offline analysis that produces our core sensitivity model.

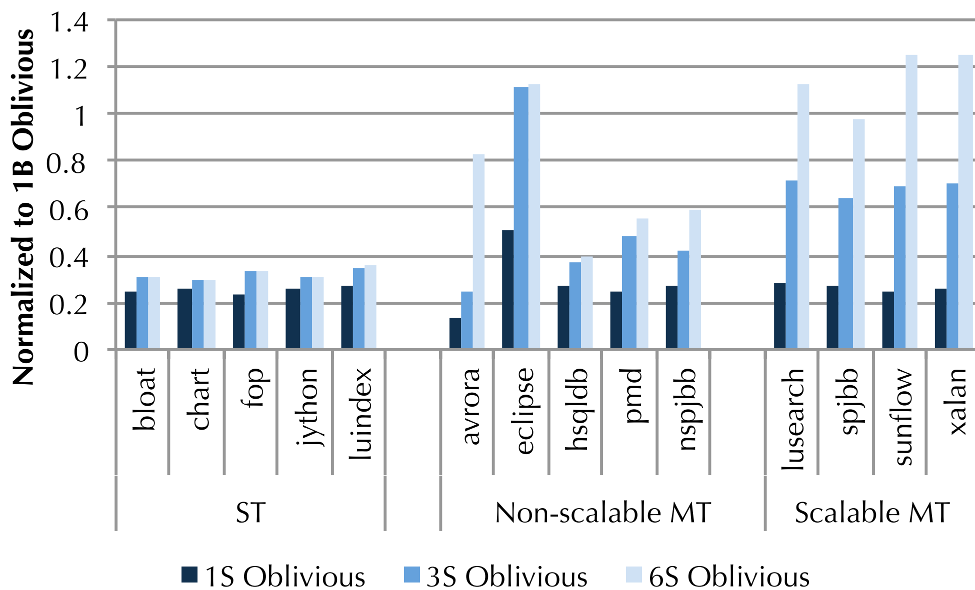
We first explore scalability on small numbers of homogeneous cores by configuring a six-core Phenom II to execute with one, three, and six cores at two processor speeds: big (B: 2.8 GHz) and small (S: 0.8 GHz) and execute the DaCapo Java benchmarks with Jikes RVM and a 2.6.32 Linux kernel. Section 5.6 describes our methodology in detail.

Figure 5.1(a) shows the speedup of each benchmark on big core configurations and Figure 5.1(b) shows the same experiment on small core configurations, both normalized to one big core. Higher is better. We use the default Linux scheduler (Oblivious) as the base case. This scheduler initially schedules threads round robin on each core, seeks to keep all cores busy, and avoids migrating threads between cores [Sarkar, 2010]. It is oblivious to different core capabilities; a shortcoming not exposed on the homogeneous hardware in this experiment.

We classify four of eight multithreaded benchmarks (lusearch, sunflow, spjbb and



(a) Speedup on one, three and six 2.8 GHz *big* cores.



(b) Speedup on one, three and six 0.8 GHz *small* cores.

Figure 5.1: Linux OS scheduler (Oblivious) on homogeneous configurations of a Phenom II normalized to one big core. We classify benchmarks as single threaded, non-scalable multithreaded (MT), and scalable MT. Higher is better.

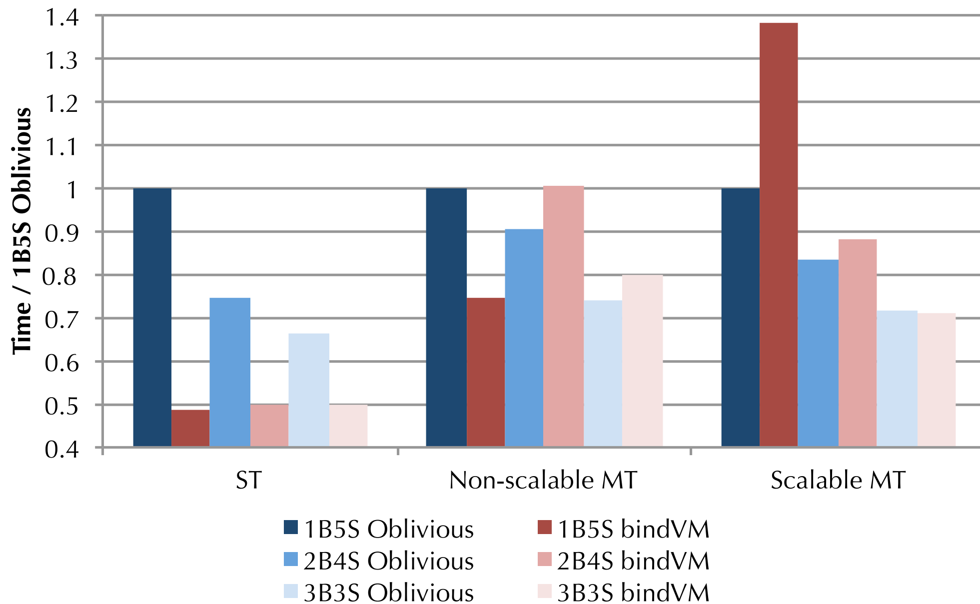


Figure 5.2: Execution time of Oblivious and bindVM on various AMP configurations, normalized to one 1B5S with Oblivious. Lower is better.

xalan) as scalable because they improve both from one to three cores, and from three to six cores. Even though five other multithreaded benchmarks respond well to additional cores, they do not improve consistently as a function of the number of cores. For instance, avrora performs worse on three big cores than on one, and eclipse performs the same on three and six cores on both big and small cores. The reason for avrora's unusual behaviour is that its rate of lock inflation (from cheap lock to heavy lock [Pizlo et al., 2011]) increases hugely from one core to multicore, more than two orders of magnitude. We found that pjbb2005 does not scale in its default configuration. We increased the size of its workload to produce a scalable variant, which we call spjbb. As comparison, the original pjbb2005 is called nspjbb in this chapter. The number of application threads and these results yield the benchmark classifications of single threaded (no parallelism), non-scalable multithreaded, and scalable multithreaded.

Because the VM itself is multithreaded, as noted in Section 3.4.1, all single threaded applications in Figure 5.1 improve slightly as a function of cores. Just observing speedup as a function of cores is insufficient to differentiate single threaded applications from multithread. For example, fop and hsqldb have similar responses to the number of cores, but fop is single threaded and hsqldb is multithreaded. This result motivates using the VM, since it knows a priori how many and which threads are application rather than VM service threads.

We now explore how well current schedulers perform on AMP hardware. We measure the default Linux scheduler (Oblivious) and the VM-aware scheduler proposed in Chapter 4 (bindVM) which schedules VM services on small cores and application threads on big cores. Chapter 4 shows that bindVM improves over Oblivious

on 1B5S and Figure 5.2 confirms this result. Regardless of the hardware configuration (1B5S, 2B4S, or 3B3S), bindVM performs best for single-threaded benchmarks. Improvements are highest for single-threaded benchmarks on average, because VM threads are non-critical and execute concurrently with the application thread. For performance, the application threads should always have priority over VM threads on the big cores.

The scalable applications tell a different story. Oblivious performs much better than bindVM on 1B5S because bindVM restricts the application to just the one big core, leaving small cores underutilized. In theory, on this Phenom II system, pairing one big core with five small cores adds 41.6% more instruction execution capacity compared to six small cores. Ideally, an embarrassingly parallel program will see this improvement on 1B5S. Our experiments show that for scalable benchmarks, exchanging one small core for a big core boosts performance by 33% on average, which is impressive but short of the ideal 41.6% (not illustrated).

For non-scalable benchmarks, neither scheduler is always best. bindVM performs best on 1B5S, but worse on 2B4S and 3B3S than Oblivious. Intuitively, with only one big core, binding the VM threads gives application threads more access to the one big core. However, with more big cores, round robin does a better job of equal access of the application threads to the big cores. In summary, neither scheduler consistently performs best across various AMP hardware configurations or workloads, motivating a better approach.

5.3 Dynamically Identifying Workload Class

This section describes our dynamic analysis that automatically classifies application parallelism. We exploit the VM thread scheduler, the Java language specification, and the VM biased lock implementation to classify applications as multithreaded scalable (MT) or multithreaded non-scalable. The VM identifies application threads separately from those it creates for GC, compilation, profiling, and other VM services. We simply count application threads to determine whether the application is single or multithreaded.

Our modified VM further classifies multithreaded applications as scalable and non-scalable. Compared to the approach used by Joao et al. [2013], which requires the programmer, compiler, or library to insert specific instructions in the source code to help the hardware identify bottlenecks, our approach is fully automated and requires no changes to applications. To differentiate between scalable and non-scalable multithreaded benchmarks, we calculate the ratio between the amount of time each thread contends (waits) for another thread to release a lock and the total execution time of the thread thus far. When this ratio is high and the thread is responsible for some threshold of execution time as a function of the total available hardware resources (e.g., 1% with two cores, 0.5% with four cores, and so on), we categorize the benchmark as non-scalable. We set this threshold based on the number of cores and threads.

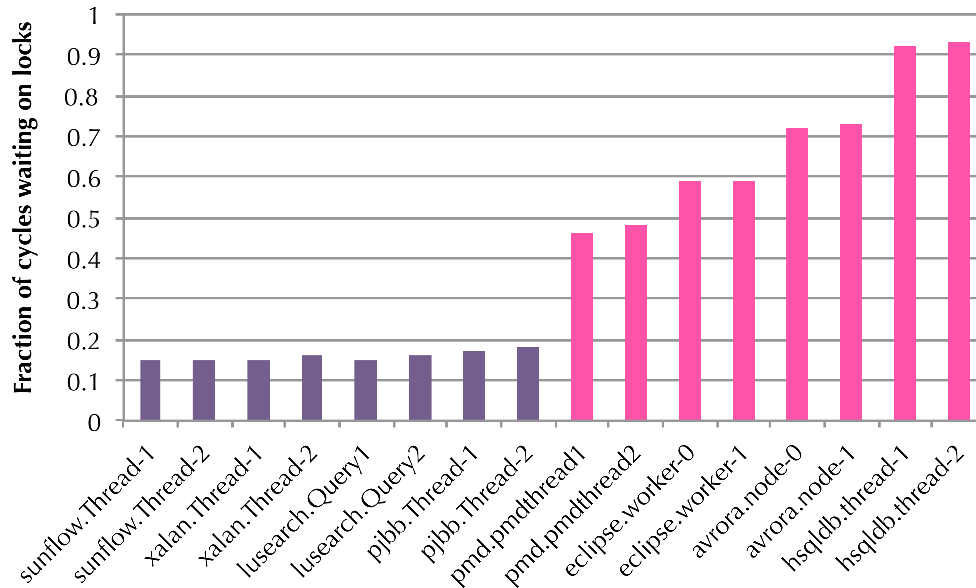


Figure 5.3: Fraction of time spent waiting on locks as a ratio of all cycles per thread in multithreaded benchmarks. The left benchmarks (purple) are scalable, the right five (pink) are not. Low ratios are highly predictive of scalability.

To compute the time waiting ratio, we piggyback on the runtime's locking implementation and thread scheduler. Only when a thread tries to acquire a lock and fails, does the VM scheduler put the thread on a wait queue, a heavyweight operation. We time how long the thread sits on the wait queue using the RDTSC instruction, which incurs an overhead of around 50 cycles each call. The VM implements biased locking, which lowers locking overheads by 'biasing' each lock to an owner thread, making the common case of taking an owned lock very cheap, at the expense of more overhead in the less common case where an unowned lock is taken [Bacon et al., 1998; Russell and Detlefs, 2006]. This optimization allows us to place our instrumentation on the rarely called code path, resulting in negligible overhead and narrowing our focus to cases of true contention.

Joao et al. [2013] achieved the same outcome of focus on true contention and very low overhead by requiring the application programmer to annotate — manually or semi-automatically — all critical locks. Our approach does not require the program to be annotated, and could be adopted in any system that uses biased locking or similar optimizations. Modern JVMs such as Jikes RVM and HotSpot already implement such optimizations. Although by default Windows OS and Linux do not implement biased locking, it is in principle possible and in practice Android does so in its Bionic implementation of the pthread library [Bacon et al., 1998; Dice et al., 2010; Android, 2014].

Figure 5.3 shows the results for two representative threads from the multithreaded benchmarks executing on the 1B5S configuration. Threads in the scalable benchmarks all have low locking ratios and those in the non-scalable benchmarks all have

high ratios. We observe that a low locking ratio is necessary but not sufficient. Scalable benchmarks typically employ homogeneous threads (or sets of threads) that perform about the same amount and type of work. When we examine the execution time of each thread in these benchmarks, their predicted sensitivity, and retired instructions, we observe that for spjbb, sunflow, xalan, and lusearch, threads are homogeneous. Our dynamic analysis inexpensively observes the progress of threads, scaled by their core assignment, and determines whether they are homogeneous or not.

5.4 Speedup and Progress Prediction Model

To effectively schedule threads on AMPs, the system must consider core features and the sensitivity of a thread to those features. When multiple threads compete for big cores, ideally the scheduler will execute the threads that benefit the most and are most critical on the big cores. For example, a thread that is memory bound will not benefit much from a higher instruction issue rate.

Offline, we create a predictive model that we use at run time. The model takes as input performance counters and predicts slow down and speedup, as appropriate. We use a linear regression model based on performance counters for two reasons. First, we find that for a given program, speedup has a roughly linear relationship that is well predicted by linear regression for the AMP hardware that we consider in this thesis. This result is consistent with Saez et al. [2012]. Second, the linear regression estimates are very inexpensive to compute from performance counters at runtime, a requirement for quick scheduling decisions. The resulting estimator is sufficiently accurate such that it does not affect the basic WASH algorithm, although a better predictor could improve our results.

We use Principal Component Analysis (PCA) to learn the most significant performance monitoring events and then use linear regression to model the relationship between those events and the speedup of a big core over a small core. Since each processor may use at most a limited number of performance counters, we use PCA analysis to select the most predictive ones. We consider two potential architectures to show the generality of our methodology: the frequency-scaled Phenom II and a hypothetical big/small design composed of an Intel Atom and i7. Our methodology is similar to Saez et al. [2012]. They use the additive regression model and machine learning to pick out the events that contribute most significantly to speedup. They also show a detailed analysis on the correlation between different performance events and the speedup. We use performance events on a faster processor to predict slow down on the small core when cores have very different architecture features.

It is not generally possible to predict speedup from small to big when the microarchitectures differ a lot. For example, given a small core with a single-issue CPU and a big core with multiway issue, if the single-issue core is always stalled, it is easy to predict that the thread won't benefit from more issue slots. However, if the single-issue core is operating at its peak issue rate, no performance counter on it will

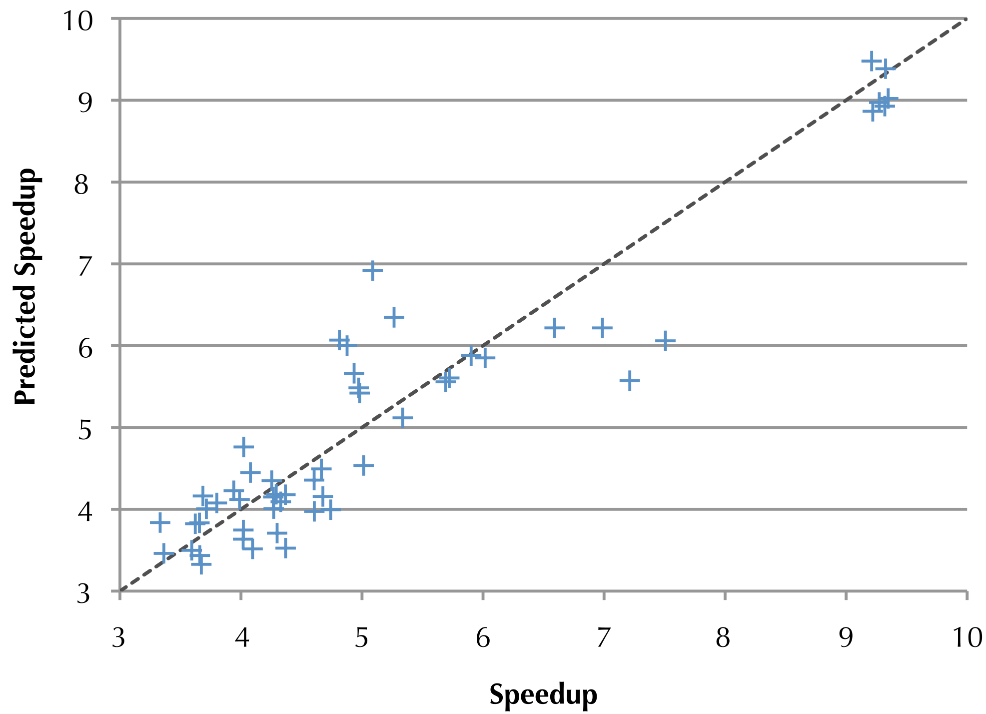
Performance counters	
Intel	**AMD**
F: INSTRUCTION_RETIRED	V: RETIRED_INSTRUCTIONS
F: UNHALTED_REFERENCE_CYCLES	V: REQUESTS_TO_L2:ALL
F: UNHALTED_CORE_CYCLES	V: L2_CACHE_MISS:ALL
V: UOPS_RETIRED:STALL_CYCLES	V: CPU_CLK_UNHALTED
V: L1D_ALL_REF:ANY	
V: L2_RQSTS:REFERENCES	
V: UOPS_RETIRED:ACTIVE_CYCLES	

Table 5.1: Performance counters identified by PCA that most accurately predict thread core sensitivity. Intel Sandy Bridge simultaneously provides three fixed (F) and up to four other performance counters (V). AMD Phenom provides up to four performance counters.

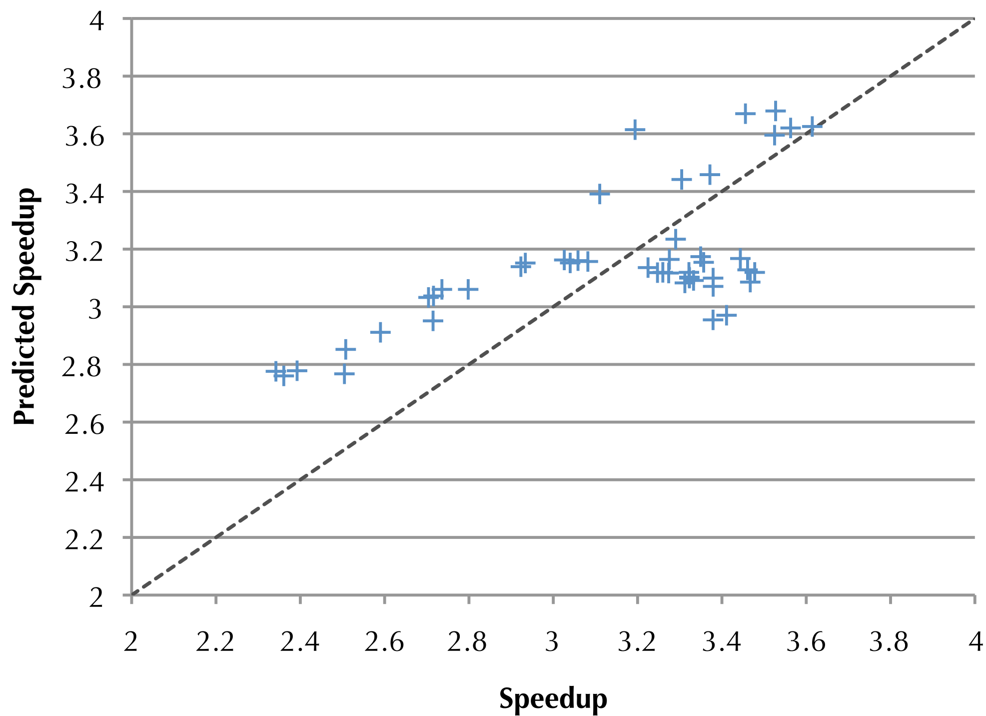
reveal how much potential speedup will come from multi-issue. Saez et al. [2012] observed this problem as well. With the frequency-scaled AMPs, our model can and does predict both speedups and slow downs because the microarchitecture does not change.

We execute and measure all of the threads with the comprehensive set of the performance monitoring events, including the energy performance counter on Sandy Bridge. We only train on threads that contribute more than 1% of total execution time, to produce a model on threads that perform significant amounts of application and VM work. In the PCA model, we first take the threads as PCA variables and then consider the performance event data for each thread as the observations for each variable. With this transformation, we determine the weights and scores for each principle component. We use the absolute value of the first principle component's weight to establish the importance of each performance event (greater absolute value means that they will have more effect on the final score). We then take performance events as variables, and their data for each thread as the observations. This time, we use the first principle component's scores to decide the similarity of the performance events (events with similar scores are similar). Based on the weights and scores, we incrementally eliminate performance events with the same scores (redundancy) and those with low weights (not predictive), to derive the N most predictive events, where N is the number of simultaneously available hardware performance counters. We set N to the maximum that the architecture will report at once. This analysis results in the performance events listed in Table 5.1. By using linear regression on the selected performance events and the speedup of a big core over a small core, we determine the linear model parameters for each event.

Figures 5.4(a) and 5.4(b) show the results of the learned models for predicting relative performance between the two Intel processors and the frequency-scaled AMD Phenom II. These results predict each application thread from a model trained on all the other applications threads, using leave-one-out validation. When executing benchmarks, we use the model trained on all the other benchmarks in *all* experi-



(a) Intel: i7 vs Atom



(b) Phenom II: 2.8 vs 0.8 GHz

Figure 5.4: Accurate prediction of thread core sensitivity. Y-axis is predicted. X-axis is actual speedup.

ments. These results show that the PCA methodology has good predictive power. The mean absolute percentage error for the predicated speedup relative to the measured value of the Intel and AMD machines are 9% and 8%, and the ranges are (0%, 36%), (0%, 23%) respectively. The residual variance for the Intel machine is 21.77 with 55 threads, and for the AMD machine it is 4.01 with 48 threads.

Our dynamic analysis monitors thread criticality and progress. It uses the retired instructions performance counter and scales it by the executing core capacity. Like some adaptive optimization systems [Arnold et al., 2000], we predict that a thread will execute for the same fraction of time in the future as it has in the past. To correct for different core speeds, we normalize retired instructions based on the speedup prediction we calculate from the performance events. This normalization gives threads executing on small cores an opportunity to out-rank threads that execute on the big cores. Our model predicts fast-to-slow well for the i7 and Atom. Our model predicts both slow-to-fast and fast-to-slow with frequency scaled cores. Thread criticality is decided based on the predicted gain for a thread that stays on or migrates to a big core.

5.5 The WASH Scheduling Algorithm

This section describes how we use core sensitivity, thread criticality, workload and core capacity to schedule threads. We consider the most general case: both the application *and* runtime contain multiple threads that exhibit varying degrees of heterogeneity and parallelism. We implement the WASH algorithm by setting thread affinities, which direct the OS to bind thread execution to one or more nominated cores. We do not require any change to the OS scheduler. Our runtime scheduler uses the existing POSIX interface to communicate thread affinities for big and small cores to the OS. The VM guides the scheduling decisions with the dynamic analysis described above. We believe that because our approach monitors the threads and adjusts the schedule accordingly, it would work even if the OS scheduler changes, but leave this evaluation to future work.

Our dynamic analysis leverages the VM's profiling infrastructure. High performance managed runtimes use feedback directed optimization (FDO) to target compiler JIT optimizations [Arnold et al., 2000]. In particular, we use Jikes RVM, which samples each thread. It sets a timer and samples the call stack to determine hot methods to compile at higher levels of optimization. We add to this sampling the gathering of per-thread performance counter data at low overhead. We use 40 ms as the quantum at which WASH reassesses each thread and adjusts its core affinity as necessary. Craeynest et al. [2012] show no discernible migration overhead on shared-LLC AMP processors when the quantum is 25 ms or greater.

5.5.1 Overview

The scheduler starts with a default *fallback* policy of using affinity masks to assign all application threads to big cores and all VM threads to small cores. The scheduler

uses the fallback policy each time a new thread is created. These defaults follow Chapter 4, which shows that GC and JIT threads are good candidates for small cores. The VM trivially identifies its service threads and explicitly binds them to the small cores. For long-lived application threads, the starting point is immaterial. For very short lived application threads that do not last a full time quantum, this fallback policy accelerates them. All subsequent scheduling decisions are made periodically based on dynamic information. Every time quantum, WASH reassesses thread sensitivity, criticality, and progress and then adjusts the affinity between threads and cores accordingly.

We add to the VM the dynamic parallelism classification and the core sensitivity models described in Sections 5.3 and 5.4. The core sensitivity model takes as input performance counter values for each thread and predicts how much the big core benefits the thread. The dynamic parallelism classifier uses a threshold for the waiting time. It examines the number of threads and their dynamic waiting time to classify applications as single threaded or multithreaded scalable or multithreaded non-scalable.

WASH uses this classification to choose a specialized scheduling strategy. For scalable applications, WASH keeps track of historical thread execution to give application threads equal access to big cores. For single-threaded applications and non-scalable applications with fewer application threads than big cores, WASH binds application threads to big cores and the default OS scheduler selects cores for VM threads. When the number of non-scalable application threads exceeds the number of big cores, WASH prioritizes threads on the big cores based on core capacity, thread efficiency, and thread criticality. WASH assesses scalability dynamically and adjusts among the policies accordingly.

Algorithm 1 shows the pseudo-code for the WASH scheduling algorithm. WASH makes three main decisions. (1) When the number of application threads is less than or equal to the number of big cores, WASH schedules them on the big cores. (2) For workloads that are scalable, WASH ensures that all threads have equal access to the big cores using the execution history. This strategy often follows the default round robin OS scheduler. (3) For other workloads, the algorithm will find the most critical alive threads whose instruction retirement rates on big cores match the rate at which the big cores can retire instructions. VM service threads are scheduled to the small cores, unless the number of application threads is less than the number of big cores.

The next three subsections discuss each workload in turn.

5.5.2 Single-Threaded and Low Parallelism WASH

When the application creates a thread, WASH's fallback policy sets the thread's affinity to the set of big cores. At each time quantum, WASH reassess the thread schedule. In the case when WASH dynamically detects that the application has one application thread or the number of application threads $|T_A|$ is less than or equal to the number of big cores $|C_B|$, then WASH sets the affinity for the application threads to the big cores. If there are fewer application threads than big cores, WASH sets the affinity

Algorithm 1 WASH

```
function WASH(T_A, T_V, C_B, C_S, t)
    T_A: Set of application threads
    T_V: Set of VM services threads, where T_A ∩ T_V = ∅
    C_B: Set of big cores, where C_B ∩ C_S = ∅
    C_S: Set of small cores, where C_B ∩ C_S = ∅
    t: Thread to schedule, where t ∈ T_A ∪ T_V
    if |T_A| ≤ |C_B| then
        if t ∈ T_A then
            Set Affinity of t to C_B
        else
            Set Affinity of t to C_B ∪ C_S
        end if
    else
        if t ∈ T_A then
            if ∀τ ∈ T_A(LockCycle%(τ) < Lock_Threshold) then
                Set Affinity of t to C_B ∪ C_S
            else
                T_Active ← {τ ∈ T_A : IsActive(τ)}
                Rank_t ← {τ ∈ T_Active : ExecRate(τ) > ExecRate(t)}
                if Σ_{τ∈Rank_t} ExecRate(τ) < ExecRate(C_B) then
                    Set Affinity of t to C_B
                else
                    Set Affinity of t to C_B ∪ C_S
                end if
            end if
        else
            Set Affinity of t to C_S
        end if
    end if
end function
```

for the VM threads such that they may execute on the remaining big cores or on the small cores. It does this by setting the affinity of the VM threads to all cores, which translates to the relative complement of $C_B \cup C_S$ with respect to $|T_A|$ big cores being used by T_A. If there are no available big cores, WASH sets the affinity for all VM threads to execute on the small cores, following Chapter 4. Single-threaded applications are the most common example of this scenario.

5.5.3 Scalable Multithreaded WASH

When the VM detects a scalable $|T_A| > |C_B|$ and homogenous workload, then the analysis in Section 5.2 shows that the default round-robin OS scheduler does an excellent job of scheduling application threads. So WASH only sets VM threads T_V to small cores C_S and does not set application threads T_A to any particular core. We use our efficient lock contention analysis to empirically establish a threshold $\text{Lock}_{\text{Threshold}}$ of 0.5 contention level (time spent contended / time executing) beyond which a thread is classified as contended (see Figure 5.3). The multithreaded work-

load is counted as scalable only when none of its threads are contended.

5.5.4 Non-scalable Multithreaded WASH

The most challenging case for AMP scheduling is how to prioritize non-scalable applications when the number of threads outstrips the number of cores and all threads compete for both big and small cores. WASH continuously determines for each thread whether performance will be improved by scheduling each thread on a big or small core. Our algorithm is based on two main considerations: (a) how important the thread is to the overall progress of the application, and (b) the relative capacity of big cores compared to small cores to retire instructions.

We rank the importance of each thread based on its relative capacity to retire instructions, seeking to accelerate threads that dominate in terms of productive work.

For each active thread T_{Active} (non-blocked for the last two scheduling quanta), we compute a running total of retired instructions that corrects for core capability. If a thread runs on a big core, we simply accumulate its retired instructions from the dynamic performance counter. When the thread runs on a small core, we increase its retired instructions total by the actual retired instructions multiplied by the predicted speedup from executing on the big core. We use the speedup model and the dynamic performance counter values to predict speedup. Thus we assess importance on a level playing field — judging each thread's progress as if it had access to big cores. Notice that threads that will benefit little from the big core will naturally have lower importance (regardless of which core they are running on in any particular time quantum), and that conversely threads that will benefit greatly from the big core will have their importance inflated accordingly. We call this rank computation *adjusted priority* and compute it for all active threads. We rank all active threads based on this adjusted priority.

We next select a set of the highest ranked threads to execute on the big cores. We do *not* size the set according to the fraction of cores that are big ($B/(B+S)$), but *instead* size the thread set according to the big cores' relative capacity to retire instructions, $\text{ExecRate}(C_B)$ ($B_{RI}/(B_{RI}+S_{RI})$). For example, in a system with one big core and five small cores, where the big core can retire instructions at five times the rate of each of the small cores, $B_{RI}/(B_{RI}+S_{RI}) = 0.5$. In that case, we will assign to the big cores the top N most important threads such that the adjusted retired instructions of those N threads is 0.5 of the total.

The effect of this algorithm is twofold. First, overall progress is maximized by placing on the big cores the threads that are both critical to the application and that will benefit from the speedup. Second, we avoid over or under subscribing the big cores by scheduling according to the capacity of those cores to retire instructions. Compared to prior work that considers scheduling parallel workloads on AMPs [Saez et al., 2012; Joao et al., 2013], this algorithm explicitly focuses on non-scalable parallelism. By detecting contention and modeling total thread progress (regardless of core assignment), our model corrects itself when threads compete for big cores yet cannot get them.

Processor	i7 Core i7-2600	Atom AtomD510	Phenom II X6 1055T
Architecture	Sandy Bridge	Bonnell	Thuban
Technology	32 nm	45 nm	45 nm
CMP & SMT	4C2T	2C2T	6C1T
LLC	8 MB	1 MB	6 MB
Frequency	3.4 GHz	1.66 GHz	B: 2.8 GHz; S: 0.8 GHz
Transistor No	995 M	176 M	904 M
TDP	95 W	13 W	125 W
DRAM Model	DDR3-1333	DDR2-800	DDR3-1333

Table 5.2: Experimental processors. We demonstrate generality of core sensitivity analysis on both Intel and AMD processors. Intel supports various clock speeds, but all cores must run at the same speed. All scheduling results for performance, power, and energy use the Phenom II since it supports separate clocking of cores, mimicking an AMP.

5.6 Methodology

This section describes the hardware, measurements, workload, and software configuration that we use to explore hardware and software energy efficiency in this chapter. We measure and report power and performance measures of existing hardware, leveraging prior tools and methodology described in Section 4.2.

5.6.1 Hardware

For the same reasons as in Section 4.2.3, we use configurations of stock hardware to explore AMP hardware for managed languages. We continue to use the Intel Sandy Bridge Core i7, the Atom, and the AMD Phenom II (with clock scaling) to explore and characterize sensitivity to AMP and methodologies for predicting core sensitivity and parallelism. Table 5.2 lists characteristics of the experimental machines used in this chapter. As in Section 4.2.2, we use Sandy Bridge 06_2AH processors because they provide an on-chip RAPL MSR energy performance counter which can measure each thread's energy consumption. The Hall effect sensor methodology described in Section 3.2.5 is used here to measure power and energy on the Phenom II. We show that our performance counter selection methodology and our model to predict thread speed generalizes across predicting the Atom from the Sandy Bridge and different speeds of the Phenom II. We accurately predict sensitivity to fast and slow cores for these two configurations.

We measure the performance and energy efficiency of our scheduler using the Phenom II, as it allows us to independently change the clock speed of each core, whereas the Intel hardware does not.

single threaded	bloat	chart	fop	jython	luindex
non-scalable	avrora	eclipse	hsqldb	nspjbb	pmd
scalable	lusearch	sunflow	xalan	spjbb	

Table 5.3: Characterization of workload parallelism. Each of the benchmarks is classified as either single threaded, non-scalable or scalable.

5.6.2 Operating System

We perform all of the experiments using Ubuntu 10.04 lucid with the 2.6.32 Linux kernel. We use the default Linux scheduler which is oblivious to different core capabilities, seeks to keep all cores busy and balanced based on the task numbers on each core, and tries not to migrate threads between cores [Sarkar, 2010].

5.6.3 Workload

We use thirteen widely used Java benchmarks taken from the DaCapo suites and SPECjbb in this chapter, which can run successfully on Jikes RVM: bloat, eclipse, fop, chart, jython, and hsqldb (DaCapo-2006); avrora, luindex, lusearch, pmd, sunflow, and xalan (DaCapo-9.12); and pjbb2005. Finding that pjbb2005 does not scale well, we increased the parallelism in the workload by modifying the JBB inputs to increase the number of transactions from 10,000 to 100,000, yielding spjbb, which scales on our six core machine. Table 5.3 shows the application parallelism (none, non-scalable, and scalable) in these benchmarks determined by the analysis in Section 5.2 of performance on *homogeneous* CMPs. We organize our analysis and results around this classification because we find that it determines a good choice of scheduling algorithm.

5.6.4 Virtual Machine Configuration

We use Jikes RVM configured with a concurrent Mark-Sweep GC. All our measurements follow Blackburn et al.'s best practices for Java performance analysis with the following modifications of note [Blackburn et al., 2006]. As in Section 4.2.5.1, regular concurrent collection is forced every 8 MB of allocation for avrora, fop and luindex, which have low allocation rate, and 128 MB for the other benchmarks. The number of collection threads is configured to be the same as the number of available cores. The default JIT settings are used in Jikes RVM, which intermixes compilation with execution.

5.6.5 Measurement Methodology

We execute each benchmark in each configuration 20 or more times and report first iteration results, which mix compilation and execution. For plotting convenience, we omit confidence intervals. For the WASH scheduling algorithm, the largest 95%

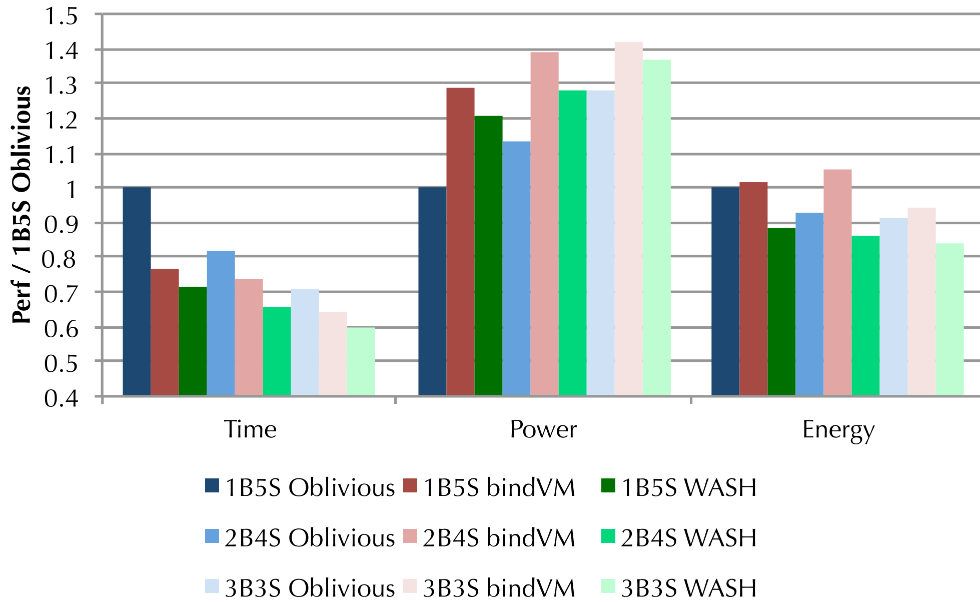


Figure 5.5: All benchmarks: geomean time, power and energy with Oblivious, bindVM, and WASH on all three hardware configurations. Lower is better.

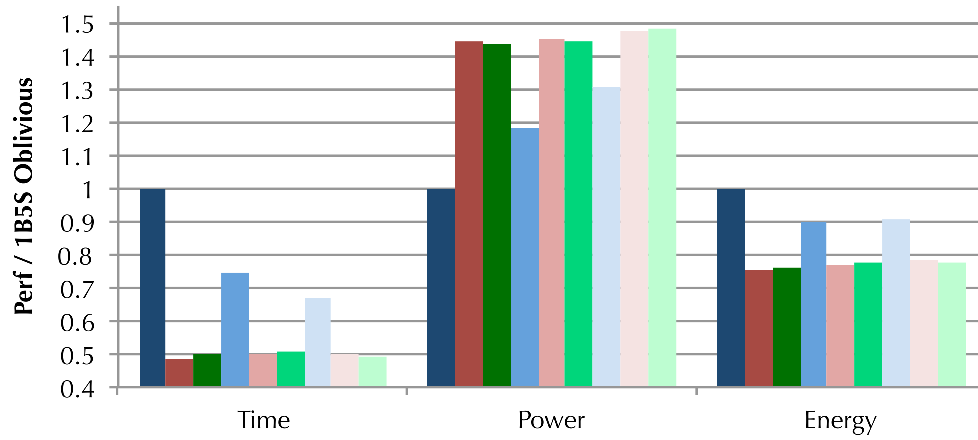
confidence interval for time measurements with 20 invocations is 5.2% and the average is 1.7%. For bindVM, the 95% confidence interval for time measurements with 20 invocations is 1.6% and the average is 0.7%. However, for Oblivious, the greatest confidence interval with 20 invocations is 15% and the average is 5.4%. Thus, we run the benchmarks with Oblivious for 60 invocations, which lowers the largest error to 9.6% and the average to 3.7%. The confidence intervals are a good indicator of performance predictability of each of the algorithms.

5.7 Results

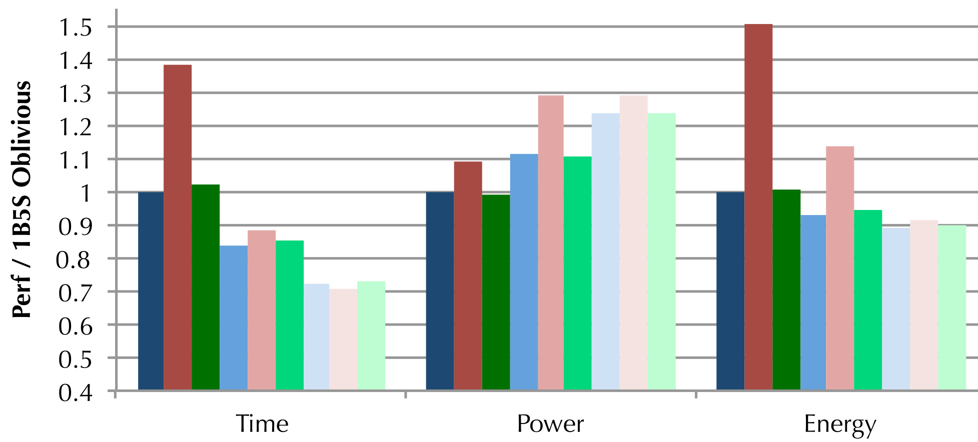
We compare WASH to the default OS scheduler (Oblivious) and VM threads to the small cores (bindVM). Figure 5.5 summarizes the overall performance, power, and energy results on three AMD hardware configurations: 1B5S (1 Big core and 5 Small cores), 2B4S and 3B3S. Figure 5.6 shows the results for each benchmark group. Figures 5.7, 5.8, and 5.9 show the individual benchmark results on the same hardware. We normalize to Oblivious and lower is better.

Figure 5.5 shows that WASH consistently improves performance and energy on average. Oblivious has the worst average time on all hardware configurations and even though it has the best power cost, 20% less power than WASH, it still consumes the most energy. Oblivious treats all the cores the same and evenly distributed threads, with the result that the big core may be underutilized and critical threads may execute unnecessarily on a small core.

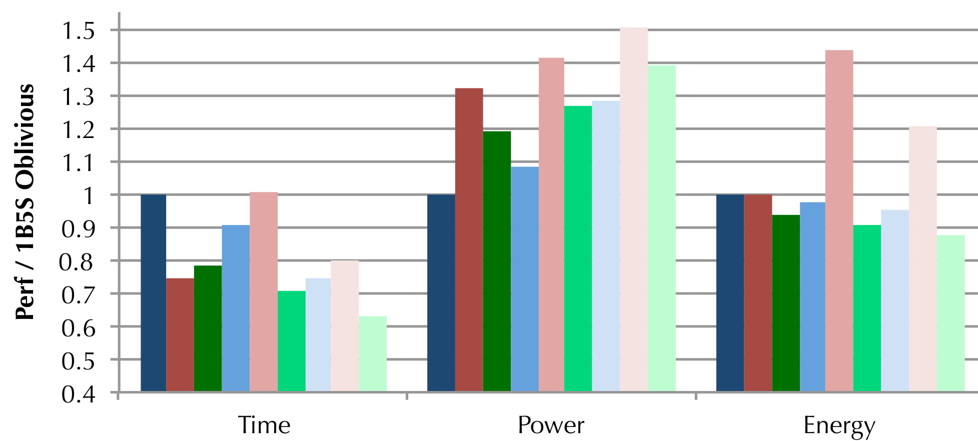
WASH attains its performance improvement by using more power than Oblivious, but at less additional power than bindVM. The bindVM scheduler has lower

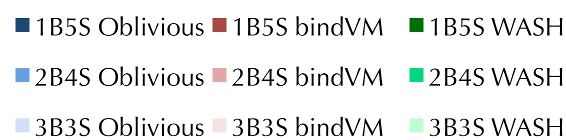


(a) Single-threaded benchmarks.



(b) Scalable multithreaded.





(c) Non-scalable multithreaded.

Figure 5.6: Normalized geomean time, power and energy for different benchmark groups.

average time compared to Oblivious, but it has the worst energy and power cost in all hardware settings, especially on 2B4S. It costs 20% more energy than WASH and 10% more than Oblivious. The bindVM scheduler overloads the big cores, with work that can be more energy efficiently performed by a small core leading to high power consumption and underutilizing the small cores.

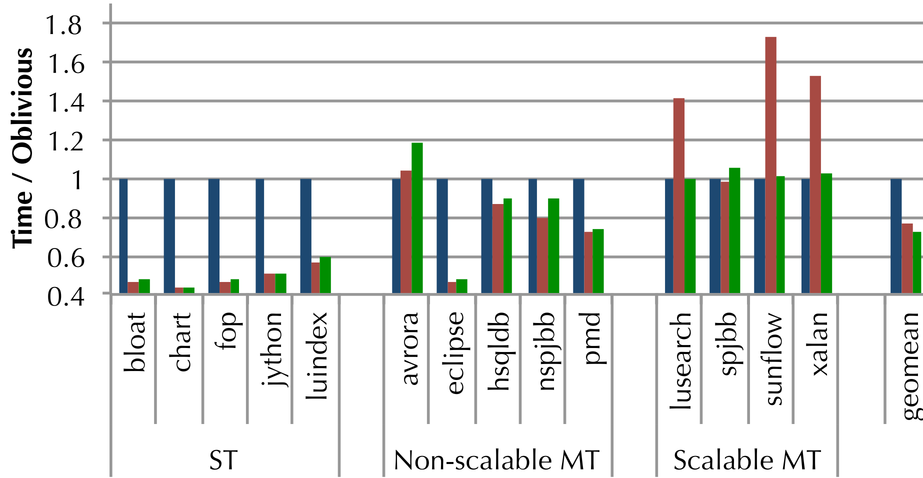
WASH and bindVM are especially effective in 1B5S compared to Oblivious, where the importance of correct scheduling decisions is accentuated most strongly. On 1B5S, WASH reduces the geomean time by 27% compared to Oblivious, and by about 7% comparing to bindVM. For energy, WASH saves more than 11% compared to bindVM and Oblivious. WASH consistently improves over bindVM at lower power costs. In the following subsections, we structure our detailed analysis based on workload categories.

5.7.1 Single-Threaded Benchmarks

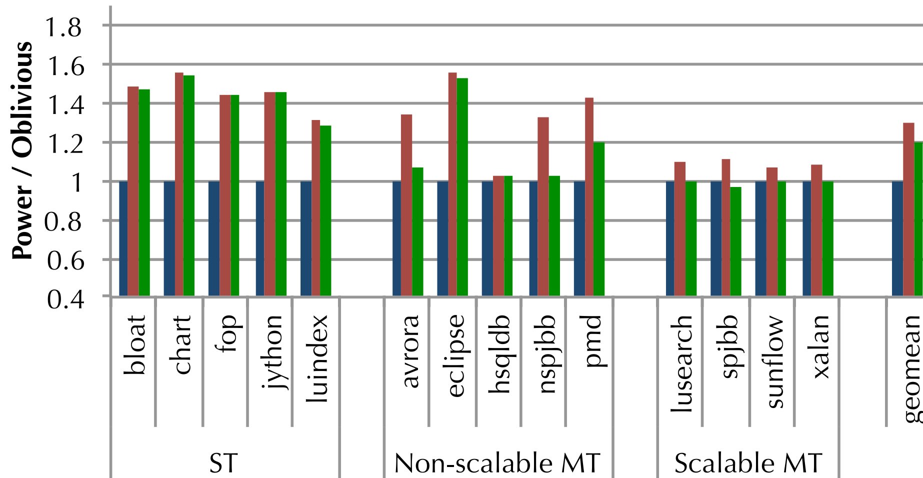
Figure 5.6(a) shows that for single-threaded benchmarks, both WASH and bindVM perform very well in terms of total time and energy on all hardware settings, while Oblivious performs poorly. Consider 1B5S in Figure 5.7. Compared to Oblivious scheduling, WASH and bindVM reduce benchmark time by 40% to 60%. Figure 5.7(c) shows that WASH reduces energy by 20% to 30% and increases average power by 30% to 55%. Oblivious performs poorly because it is unaware of the heterogeneity among the cores, so with high probability in the 1B5S case, it schedules the application thread onto a small core. In this scenario, both WASH and bindVM will schedule the application thread to the big core and all GC threads to the small cores. When the number of big cores increases, as in Figure 5.8(a) and Figure 5.9(a), then there is a smaller distinction between the two policies because the VM threads may be scheduled on big as well as small cores. In steady state the other VM threads do not contribute greatly to total time, as long as they do not interfere with the application thread. One thing to note is that the power consumption is higher for bindVM and WASH than for Oblivious. When the application thread migrates to a small core, it consumes less power by about 35% on average compared to bindVM and WASH, but the loss in performance outweighs the decrease in power. Thus total energy is reduced by 30 to 20% by bindVM and WASH. WASH and bindVM perform very similarly in the single-threaded scenario on all configurations.

5.7.2 Scalable Multithreaded Benchmarks

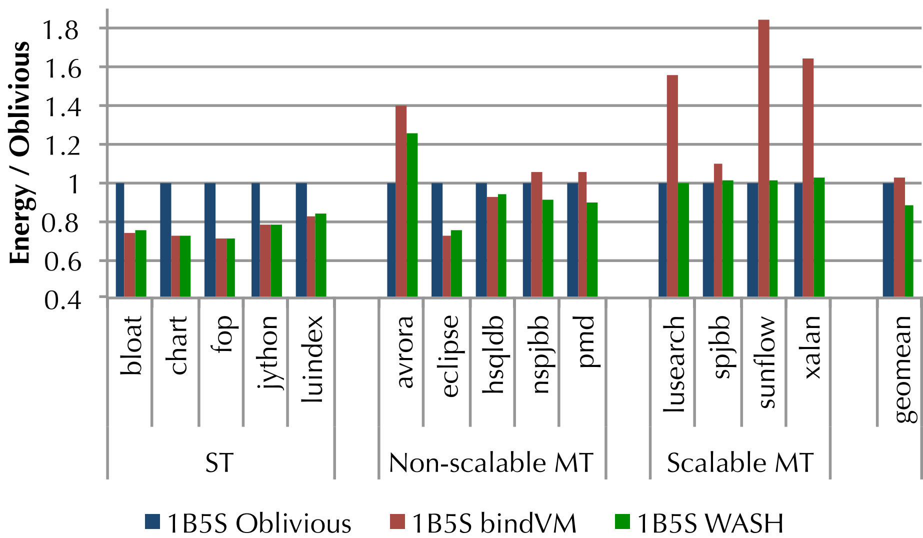
Figure 5.6(b) shows that for scalable multithreaded benchmarks, WASH and Oblivious perform very well in both execution time and energy on all hardware configurations, while bindVM performs poorly. Consider 1B5S in Figure 5.7. Compared to WASH and Oblivious scheduling, bindVM increases time by 40% to 70%, increases energy by 55% to 80%, and power by 5% to 10%. By using the contention information we gather online, WASH detects scalable benchmarks. Since WASH correctly identifies the workload, WASH and Oblivious generate similar results. The reason bindVM



(a) Time on 1B5S.

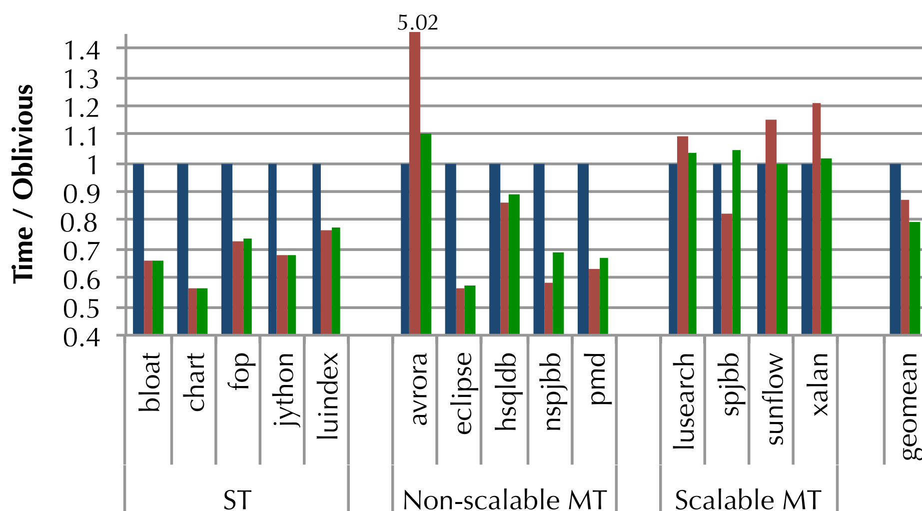


(b) Power on 1B5S.

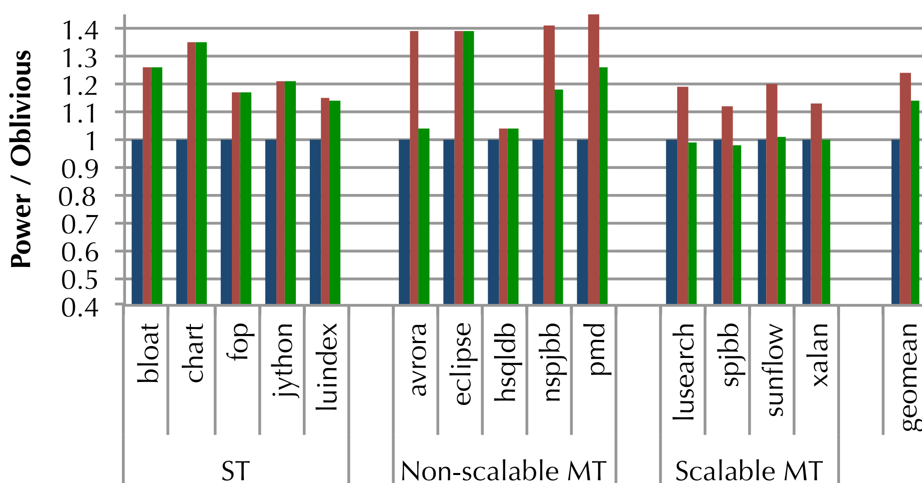


(c) Energy on 1B5S.

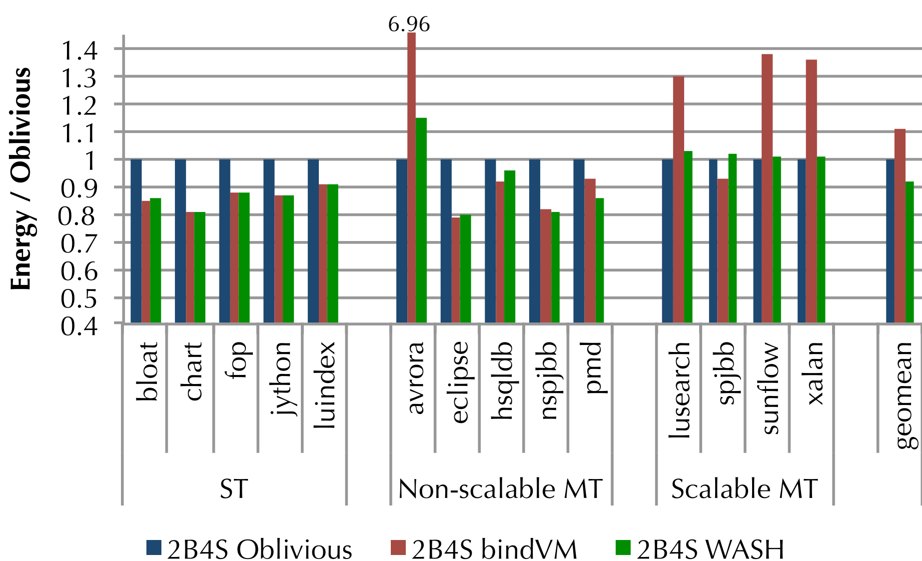
Figure 5.7: Individual benchmark results on 1B5S. Lower is better.



(a) Time on 2B4S.

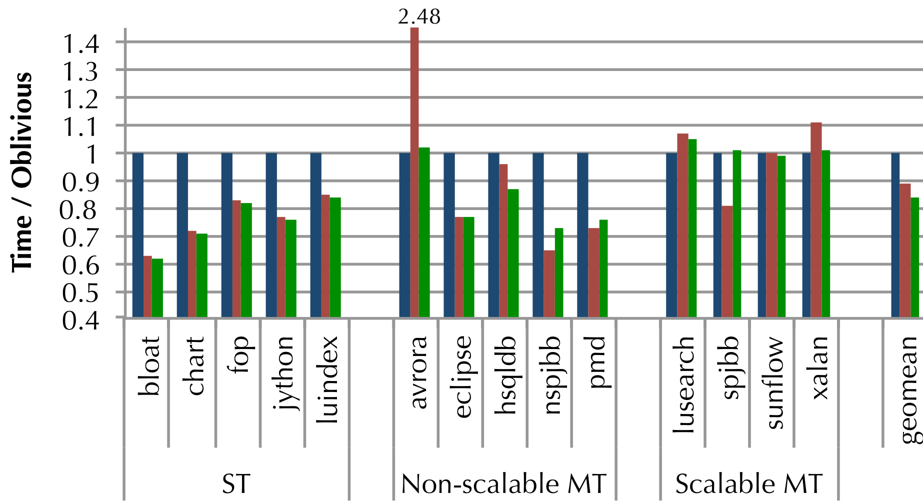


(b) Power on 2B4S.

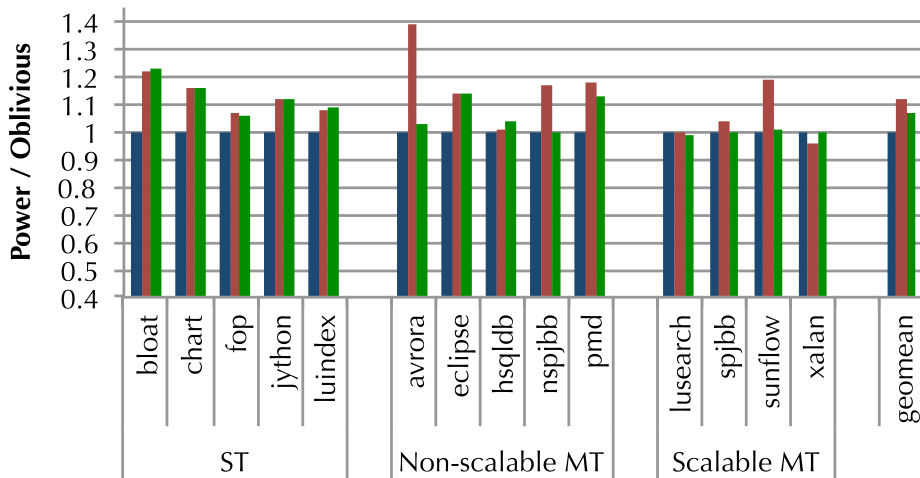


(c) Energy on 2B4S.

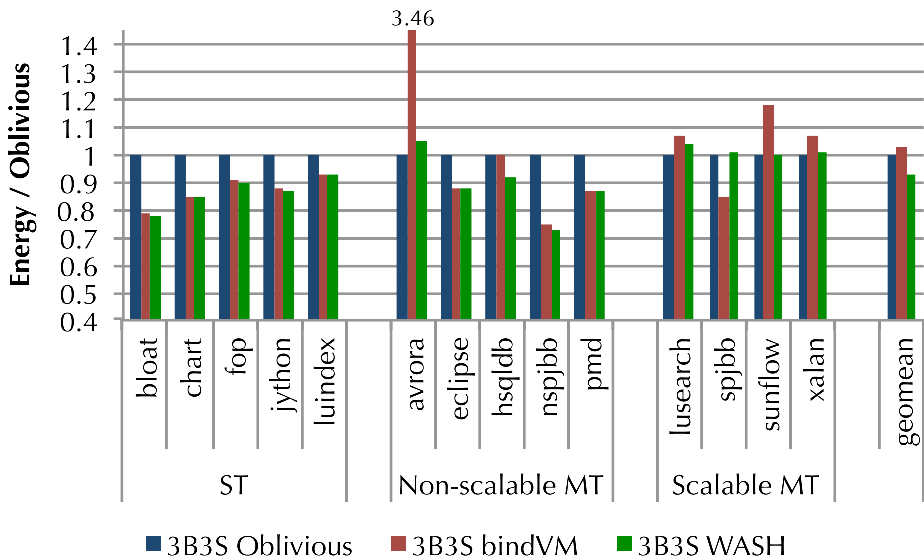
Figure 5.8: Individual benchmark results on 2B4S. Lower is better.



(a) Time on 3B3S.



(b) Power on 3B3S.



(c) Energy on 3B3S.

Figure 5.9: Individual benchmark results on 3B3S. Lower is better.

fails is that it simply binds all application threads to the big cores, leaving the small cores under-utilized. Scalable benchmarks with homogeneous threads benefit from round-robin scheduling policies as long as the scheduler migrates threads among the fast and slow cores frequently enough. Even though Oblivious does not reason explicitly about the relative core speeds, because it migrates threads frequently enough, it works well. However, WASH reasons explicitly about relative core speeds using historical execution data to migrate threads fairly between slow and fast cores. As the number of big cores increases, the difference between bindVM and Oblivious reduces.

5.7.3 Non-scalable Multithreaded Benchmarks

Figure 5.6(c) shows that WASH consistently performs best and neither Oblivious or bindVM is consistently second best in the more complex setting of non-scalable multithreaded benchmarks. For example, eclipse has about 20 threads and hsqldb has about 80. They each have high degrees of contention. In eclipse, the *Javaindexing* thread takes 56% of the total application thread cycles while the three *Worker* threads consume just 1%. Furthermore, avrora threads spend around 60-70% of their cycles waiting on contended locks, while pmd threads spend around 40-60% of their cycles waiting on locks. These messy workloads make scheduling challenging.

For eclipse and hsqldb, Figure 5.7, 5.8, and 5.9 show that the results for WASH and bindVM are very similar with respect to time, power, and energy in almost all hardware settings. The reason is that even though eclipse has about 20 and hsqldb has 80 threads, in both cases only one or two of them are dominant. In eclipse, the threads *Javaindexing* and *Main* are responsible for more than 80% of the application's cycles. In hsqldb, *Main* is responsible for 65% of the cycles. WASH will correctly place the dominant threads on big cores, since they have higher priority. Most other threads are very short lived, shorter than our 40 ms scheduling quantum. Since before profiling information is gathered, WASH binds application threads to big cores, the short-lived threads will just stay on big cores. Since bindVM will put all application threads on big cores too, the results for the two policies are similar.

For avrora, nspjbb, and pmd, Figure 5.7, 5.8, and 5.9 are good examples of WASH choosing to execute application threads that will not benefit as much from a big core on a small core. Particularly, for 1B5S, in Figure 5.7, compared to bindVM, WASH time is lower; however, because of its better use of the small cores, the power measurements are much lower. Since bindVM does not reason about core sensitivity, it does not make a performance/power trade-off. WASH makes better use of small cores resulting in improved energy efficiency for avrora, nspjbb, and on a smaller scale for pmd.

In summary, all the above results show that WASH improves performance and energy consistently. WASH is better at utilizing both small and big cores as appropriate to attain better performance at higher power than Oblivious, which under utilizes the big cores and over utilizes small cores because it does not reason about them. On the other hand, WASH, uses its understanding of the workload and core

sensitivity analysis to obtain both better performance and lower power than bindVM which under utilizes the small cores for the scalable and non-scalable multithreaded benchmarks. WASH is robust to other OS scheduling choices.

5.8 Summary

This chapter shows how to exploit the VM abstraction for managed applications to achieve transparency, performance, and efficiency on AMP architectures. We introduce a new fully automatic dynamic analysis that classifies parallelism in workloads and identifies critical threads. We introduce a new scheduler called WASH that uses this analysis and predicts performance gains among heterogeneous cores. WASH sets thread affinities for application and VM threads by assessing thread criticality, expected benefit, and core capacity to balance performance and energy efficiency. The OS and/or the C standard libraries could implement similar analyses to improve the OS scheduler, although the VM would still need to differentiate and communicate a priori knowledge of low priority VM service threads to the OS.

We show that this system delivers substantial improvements in performance and energy efficiency on frequency scaled processors. Future heterogeneous hardware should benefit even more. For instance, Qualcomm big/little ARM cores include more energy-efficient little cores. With better heterogeneous hardware, our approach should deliver better energy efficiency.

Combined with the findings in the last chapter, we further show that the challenges and opportunities of AMP architectures and managed software are complementary. The heterogeneity of AMP eliminates the overhead of managed software, and the abstraction of managed software makes transparent the complexity of AMP. The conjunction of AMP and managed software can highlight each other's advantages and provide a win-win opportunity for hardware and software communities now confronted with performance and power challenges in an increasingly complex software and hardware landscape.

Chapter 6

Conclusion

The past decade has seen both hardware and software undergo disruptive revolutions. The way software is built, sold, and deployed has changed radically, while the principle constraint in CPU design has shifted from transistor count to power. On one hand, single-ISA Asymmetric Multicore Processors (AMP) use general purpose cores that make different tradeoffs in the power and performance design space to resolve power constraints in computer architecture design. However, they expose hardware complexity to software developers. On the other hand, managed software is increasingly prevalent in client applications, business software, mobile devices and the cloud. However, while managed languages abstract over hardware complexity, they do so at a significant cost.

This thesis seeks to find synergy between two computing revolutions and use a software and hardware co-design approach to solve the challenges of hardware complexity without forcing developers to explicitly manage this complexity and to exploit the opportunities of managed software and AMP architectures.

We present a quantitative analysis of the performance, power and energy characteristics of managed and native workloads on a range of hardware features. The two major findings that (1) native workloads do not approximate managed workloads, and (2) each hardware feature elicits a huge variety of power, performance and energy response, motivate exploring the opportunities of managed software optimization and the utilization of AMP architectures.

We identify four key software properties that make a task amenable to running on small cores: (1) parallelism, (2) asynchrony, (3) non-criticality, and (4) hardware sensitivity. These properties form a guide for selecting tasks that can be performed on small cores.

Virtual Machine (VM) services (GC, interpretation and JIT), together imposing substantial energy and performance costs, are shown to manifest a differentiated performance and power workload. To differing degrees, they are parallel, asynchronous, communicate infrequently and are not on the application's critical path, fulfilling the small cores requirements of AMP designs. By modelling AMP processors with cores differing in microarchitecture, using a separate small core to run VM services can improve total performance by 13% on average and reduce energy consumption by 7% at the cost of 5% higher power.

To shield the hardware complexity and fully expose the potential efficiency of

AMP designs, only exploiting AMP architectures for VM services is insufficient. This thesis develops a comprehensive dynamic VM scheduling policy—WASH, which dynamically monitors and characterizes both managed applications and VM services' core sensitivity, criticality, parallelism and CPU load balance. WASH effectively uses thread contention information to classify workloads into three groups and applies different scheduling policy for each group. On a 1B5S AMP with cores only differing in frequency, WASH improves performance by 40% and energy efficiency by 13% compared to default OS scheduling. Compared to only scheduling VM services to small cores, WASH improves performance by 7% and energy efficiency by 15%.

In combination, these contributions demonstrate that there exists a synergy between managed software and AMP architectures that can be automatically exploited to reduce VM overhead and deliver the efficiency promise of AMP architectures while abstracting hardware complexity from developers.

6.1 Future Work

There are many directions for future work. The work presented was constrained to 32-bit ISA in order to fairly include the Pentium 4 in the analysis. This work could be redone using 64-bit ISA although the conclusions should be unchanged. More interesting would be to extend the work to include the evaluation of power and energy of the memory, and in particular contrast the behaviour of native workloads with that of managed languages. The measurement of memory power and energy would have presented greater implementation difficulties, and is likely to have little bearing on examining the synergy between AMP and managed software. The following two sections focus on future directions that may have significant impact on how to coordinate managed software and AMP architectures for efficiency.

6.1.1 WASH-assisted OS Scheduling

The WASH scheduler takes advantages of VMs to dynamically gain insight into the different kinds of threads executing (both application and VM threads), to efficiently monitor the locking behaviours and to profile the threads for core sensitivity and criticality. However, the information that VMs cannot gather is the global workload information on the processor for all applications. The current WASH algorithm uses the POSIX pthread affinity settings to schedule threads within one managed workload and assumes no other workloads are competing the resources.

A promising approach to solve this problem is WASH-assisted OS scheduling. After WASH makes a thread scheduling decision, instead of scheduling the thread to the core directly, it will communicate the decision to the OS, and let the OS make a global decision according to global workload distribution on the processor. This way, the VM and OS can complement each other for AMP scheduling.

6.1.2 Heterogeneous-ISA AMP and Managed Software

This thesis focuses on abstracting over hardware complexity for single-ISA AMP architectures. Heterogeneous-ISA AMP is also a possible direction for future architecture design, which provides even greater challenges for application programmers to exploit their potential efficiency. While completely-different-ISA AMP architectures are unlikely, already cores within a sophisticated AMP device may have ISAs that differ, such as TI's OMAP4470 [Texas Instruments, 2013]. Pruning of the ISA allows for simpler, more efficient cores, at the cost of heterogeneity.

To expose the potential efficiency of heterogeneous-ISA AMP architectures for managed software, running VM services on the small core should be straightforward. To schedule arbitrary application threads via the VM requires several areas to be addressed: (1) Some proportion of the code may need to be compiled for each heterogeneous core type—code bloat may be a significant issue mostly for small or embedded AMP devices although it is less likely to be an issue for big servers. (2) To effectively schedule threads between different core type, the VM will take an even greater responsibility for scheduling. A possible approach is to use thread models and schedulers where the VM maintains an $M : N$ mapping of M software threads to N hardware threads. This mechanism allows a software thread to be mapped to a hardware thread with one core type then, perhaps at a method call point, have its execution transferred to another hardware thread with a different core type. This transfer would be intra-process and should be comparatively cheaper than an OS level thread migration. (3) A VM will need to integrate compilers with optimizations that take full advantage of the differentiated cores, which is not significantly different from the collection of compilers or compiler back ends already implemented for the hardware.

Bibliography

ALPERN, B.; AUGART, S.; BLACKBURN, S. M.; BUTRICO, M.; COCCHI, A.; CHENG, P.; DOLBY, J.; FINK, S.; GROVE, D.; HIND, M.; MCKINLEY, K. S.; MERGEN, M.; MOSS, J.; NGO, T.; SARKAR, V.; AND TRAPP, M., 2005. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44, 2 (2005), 399–418. doi:10.1147/sj.442.0399. (cited on page 8)

ANDROID, 2014. Bionic platform. https://github.com/android/platform_bionic. (cited on pages 75 and 79)

ARNOLD, M.; FINK, S.; GROVE, D.; HIND, M.; AND SWEENEY, P., 2000. Adaptive optimization in the Jalapeño JVM (poster session). In OOPSLA '00: *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (Minneapolis, Minnesota, USA, Oct. 2000), 125–126. ACM, New York, NY, USA. doi:10.1145/353171.353175. (cited on pages 9, 10, and 83)

AZIZI, O.; MAHESRI, A.; LEE, B. C.; PATEL, S. J.; AND HOROWITZ, M., 2010. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *ISCA '10: Proceedings of the 37th International Symposium on Computer Architecture* (Saint-Malo, France, Jun. 2010), 26–36. ACM, New York, NY, USA. doi:10.1145/1815961.1815967. (cited on pages 31 and 33)

BACON, D. F.; CHENG, P.; AND SHUKLA, S., 2012. And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. In PLDI '12: *Proceedings of the 2012 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China, Jun. 2012), 23–34. ACM, New York, NY, USA. doi:10.1145/2254064.2254068. (cited on pages 12 and 13)

BACON, D. F.; KONURU, R. B.; MURTHY, C.; AND SERRANO, M. J., 1998. Thin locks: Featherweight synchronization for Java. In PLDI '98: *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Montreal, Canada, Jun. 1998), 258–268. ACM, New York, NY, USA. doi:10.1145/277650.277734. (cited on pages 75 and 79)

BECCHI, M. AND CROWLEY, P., 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In CF '06: *Proceedings of the Third Conference on Computing Frontiers* (Ischia, Italy, May 2006), 29–40. ACM, New York, NY, USA. doi:10.1145/1128022.1128029. (cited on pages 6, 7, and 73)

BIENIA, C.; KUMAR, S.; SINGH, J. P.; AND LI, K., 2008. The PARSEC benchmark

suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University. (cited on pages 18 and 19)

Bircher, W. L. and John, L. K., 2004. Analysis of dynamic power management on multi-core processors. In ICS '08: *Proceedings of the 22nd Annual International Conference on Supercomputing* (Munich, Germany, Jun. 2004), 327–338. ACM, New York, NY, USA. doi:10.1145/1375527.1375575. (cited on page 23)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004. Myths and realities: The performance impact of garbage collection. In SIGMETRICS '04: *Proceedings of the 2004 International Conference on Measurements and Modeling of Computer Systems* (New York, NY, USA, Jun. 2004), 25–36. ACM, New York, NY, USA. doi:10.1145/1005686.1005693. (cited on pages 11 and 57)

Blackburn, S. M.; Garner, R.; Hoffman, C.; Khan, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In OOPSLA '06: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (Portland, Oregon, USA, Oct. 2006), 169–190. ACM, New York, NY, USA. doi:10.1145/1167515.1167488. (cited on pages 16, 19, 52, 56, 57, 58, and 88)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator locality. In PLDI '08: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA, Jun. 2008), 22–32. ACM, New York, NY, USA. doi:10.1234/12345678. (cited on pages 8 and 57)

Blackburn, S. M.; McKinley, K. S.; Garner, R.; Hoffman, C.; Khan, A. M.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2008. Wake up and smell the coffee: Evaluation methodologies for the 21st century. *Communications of the ACM*, 51, 8 (Aug. 2008), 83–89. doi:10.1145/1378704.1378723. (cited on page 20)

Cao, T.; Blackburn, S. M.; Gao, T.; and McKinley, K. S., 2012. The yin and yang of power and performance for asymmetric hardware and managed software. In ISCA '12: *Proceedings of the 39th International Symposium on Computer Architecture* (Portland, OR, USA, Jun. 2012), 225–236. IEEE Computer Society, Washington, DC, USA. doi:10.1145/2366231.2337185. (cited on page 49)

Chakraborty, K., 2008. *Over-provisioned Multicore Systems*. Ph.D. thesis, University of Wisconsin-Madison. (cited on page 26)

Chen, G.; Shetty, R.; Kandemir, M.; Vijaykrishnan, N.; Irwin, M.; and Wolczko, M., 2002. Tuning garbage collection in an embedded Java environment. In HPCA '02: *Proceedings of the 8th International Symposium on High-Performance Computer Architecture* (Boston, Massachusettes, USA, Feb. 2002), 92–103. IEEE Computer Society, Washington, DC, USA. doi:10.1109/HPCA.2002.995701. (cited on page 11)

Cher, C.-Y. and Gschwind, M., 2008. Cell GC: using the Cell synergistic processor as a garbage collection coprocessor. In VEE '08: *Proceedings of the 4th International Conference on Virtual Execution Environments* (Seattle, WA, USA, Mar. 2008), 141–150. ACM, New York, NY, USA. doi:10.1145/1346256.1346276. (cited on pages 12 and 13)

Chitlur, N.; Srinivasa, G.; Hahn, S.; Gupta, P. K.; Reddy, D.; Koufaty, D. A.; Brett, P.; Prabhakaran, A.; Zhao, L.; Ijih, N.; Subhaschandra, S.; Grover, S.; Jiang, X.; and Iyer, R., 2012. QuickIA: Exploring heterogeneous architectures on real prototypes. In HPCA '12: *Proceedings of the 18th International Symposium on High-Performance Computer Architecture* (New Orleans, LA, USA, Feb. 2012), 433–440. IEEE Computer Society, Washington, DC, USA. doi:10.1109/HPCA.2012.6169046. (cited on pages 1 and 5)

Click, C., 2009. Azul's experiences with hardware/software co-design. Keynote presentation. In VEE '09: *Proceedings of the 5th International Conference on Virtual Execution Environments* (Washington, DC, USA, Mar. 2009). ACM, New York, NY, USA. (cited on pages 12 and 13)

Click, C.; Tene, G.; and Wolf, M., 2005. The pauseless GC algorithm. In VEE '05: *Proceedings of the 1st International Conference on Virtual Execution Environments* (Chicago, IL, USA, Jun. 2005), 46–56. ACM, New York, NY, USA. doi:10.1145/1064979.1064988. (cited on pages 12 and 13)

Craeynest, K. V.; Jaleel, A.; Eeckhout, L.; Narváez, P.; and Emer, J. S., 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In ISCA '12: *Proceedings of the 39th International Symposium on Computer Architecture* (Portland, OR, USA, Jun. 2012), 213–224. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ISCA.2012.6237019. (cited on pages 6, 73, and 83)

David, H.; Gorbatov, E.; Hanebutte, U. R.; Khanaa, R.; and Le, C., 2010. RAPL: Memory power estimation and capping. In ISLPED '10: *Proceedings of the 2010 International Symposium on Low Power Electronics and Design* (Austin, Texas, USA, Aug. 2010), 189–194. IEEE Computer Society, Washington, DC, USA. doi:10.1145/1840845.1840883. (cited on page 53)

Deutsch, L. P. and Schiffman, A. M., 1984. Efficient implementation of the Smalltalk-80 system. In POPL '84: *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages* (Salt Lake City, Utah, USA, Jan 1984), 297–302. ACM Press. doi:10.1145/800017.800542. (cited on page 10)

Dice, D.; Moir, M.; and Scherer, W., 2010. Quickly reacquirable locks, patent 7,814,488. (cited on pages 75 and 79)

Du Bois, K.; Eyerman, S.; Sartor, J. B.; and Eeckhout, L., 2013. Criticality stacks: identifying critical threads in parallel programs using synchronization behavior. In ISCA '13: *Proceedings of the 40th International Symposium on Computer Architecture* (Tel-Aviv, Israel, Jun. 2013), 511–522. ACM, New York, NY, USA. doi:10.1145/2485922.2485966. (cited on pages 7, 73, and 74)

Emer, J. S. and Clark, D. W., 1984. A characterization of processor performance in the VAX-11/780. In ISCA '84: *Proceedings of the 11st International Symposium on Computer Architecture* (Ann Arbor, USA, Jun. 1984), 301–310. ACM, New York, NY, USA. doi:10.1145/285930.285986. (cited on page 15)

Emer, J. S. and Clark, D. W., 1998. Retrospective: A characterization of processor performance in the VAX-11/780. *ACM 25 Years ISCA: Retrospectives and Reprints 1998*, (1998), 274–283. doi:10.1145/285930.285946. (cited on page 16)

Esmaeilzadeh, H.; Blem, E. R.; Amant, R. S.; Sankaralingam, K.; and Burger, D., 2011a. Dark silicon and the end of multicore scaling. In ISCA '11: *Proceedings of the 38th International Symposium on Computer Architecture* (San Jose, CA, USA, Jun. 2011), 365–376. ACM, New York, NY, USA. doi:10.1145/2000064.2000108. (cited on page 1)

Esmaeilzadeh, H.; Cao, T.; Yang, X.; Blackburn, S.; and McKinley, K. S., 2012a. What is happening to power, performance, and software? *IEEE Micro*, 32, 3 (2012), 110–121. doi:10.1109/MM.2012.20. (cited on page 15)

Esmaeilzadeh, H.; Cao, T.; Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2011b. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In ASPLOS '11: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, CA, USA, Mar. 2011), 319–332. ACM, New York, NY, USA. doi:10.1145/1950365.1950402. (cited on page 15)

Esmaeilzadeh, H.; Cao, T.; Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2011c. Source materials in ACM Digital Library for: Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems*, 319–332. http://doi.acm.org/10.1145/1950365.1950402. (cited on pages 15 and 24)

Esmaeilzadeh, H.; Cao, T.; Yang, X.; Blackburn, S. M.; and McKinley, K. S., 2012b. Looking back and looking forward: power, performance, and upheaval. *Communications of the ACM*, 55, 7 (2012), 105–114. doi:10.1145/2209249.2209272. (cited on page 15)

Georges, A.; Buytaert, D.; and Eeckhout, L., 2007. Statistically rigorous Java performance evaluation. In OOPSLA '07: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (Montreal, Quebec, Canada, Oct. 2007), 57–76. ACM, New York, NY, USA. doi:10.1145/1297027.1297033. (cited on page 20)

Greenhalgh, P., 2011. Big.LITTLE processing with ARM Cortex^{TM}-A15 & Cortex-A7. (cited on pages 1 and 5)

Ha, J.; Arnold, M.; Blackburn, S.; and McKinley, K., 2011. A concurrent dynamic analysis framework for multicore hardware. In OOPSLA '09: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (Orlando, Florida, USA, Oct. 2011), 155–174. ACM, New York, NY, USA. doi:10.1145/1639949.1640101. (cited on page 52)

Ha, J.; Gustafsson, M.; Blackburn, S. M.; and McKinley, K. S., 2008. Microarchitectural characterization of production JVMs and Java workloads. In *IBM CAS Workshop* (Austin, Texas, 2008). (cited on page 51)

Ha, J.; Haghighat, M. R.; Cong, S.; and McKinley, K. S., 2009. A concurrent trace-based Just-In-Time compiler for single-threaded JavaScript. In PESPMA 2009: *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*. (cited on page 10)

Hempstead, M.; Wei, G.-Y.; and Brooks, D., 2009. Navigo: An early-stage model to study power-constrained architectures and specialization. In *Workshop on Modeling, Benchmarking, and Simulations*. (cited on page 26)

Horowitz, M.; Alon, E.; Patil, D.; Naffziger, S.; Kumar, R.; and Bernstein, K., 2005. Scaling, power, and the future of CMOS. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, 7–15. ACM/IEEE. doi:10.1109/IEDM.2005.1609253. (cited on page 26)

Horvath, O. and Meyer, M., 2010. Fine-grained parallel compacting garbage collection through hardware-supported synchronization. In ICPPW '10: *Proceedings of the 39th International Conference on Parallel Processing Workshops* (San Diego, California, USA, Sept. 2010), 118–126. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ICPPW.2010.28. (cited on pages 12 and 13)

Hu, S. and John, L. K., 2006. Impact of virtual execution environments on processor energy consumption and hardware adaptation. In VEE '06: *Proceedings of the 2st International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada, Jun. 2006), 100–110. ACM, New York, NY, USA. doi:10.1145/1134760.1134775. (cited on page 11)

Hu, S.; Valluri, M. G.; and John, L. K., 2005. Effective adaptive computing environment management via dynamic optimization. In CGO '05: *Proceedings of the*

3nd IEEE / ACM International Symposium on Code Generation and Optimization (San Jose, CA, USA, Mar. 2005), 63–73. IEEE Computer Society, Washington, DC, USA. doi:10.1109/CGO.2005.17. (cited on page 11)

Huang, X.; Moss, J. E. B.; McKinley, K. S.; Blackburn, S.; and Burger, D., 2003. Dynamic SimpleScalar: Simulating Java virtual machines. Technical Report TR-03-03, University of Texas at Austin, Department of Computer Sciences. (cited on page 11)

Intel, 2008. Intel Turbo Boost technology in Intel core microarchitecture (Nehalem) based processors. (cited on pages 1, 26, and 45)

Intel, 2011. Intel 64 and IA-32 architectures software developer's manual volume 3A: System programming guide, part 1. (cited on page 54)

Intel Corporation, 2011. Intel Hyper-Threading Technology. http://www.intel.com/technology/platform-technology/hyper-threading. (cited on page 26)

Isci, C. and Martonosi, M., 2003. Runtime power monitoring in high-end processors: Methodology and empirical data. In MICRO '03: *Proceedings of the 36th Annual International Symposium on Microarchitecture* (San Diego, CA, USA, Dec. 2003), 93–104. ACM, New York, NY, USA / IEEE Computer Society, Washington, DC, USA. doi:10.1145/956417.956567. (cited on page 23)

ITRS Working Group, 2011. International technology roadmap for semiconductors. http://www.itrs.net. (cited on page 41)

Jibaja, I.; Cao, T.; Blackburn, S. M.; and McKinley, K. S., 2014. WASH: Dynamic analysis and scheduling managed software on asymmetric multicore processors. In *Under review*. (cited on page 73)

Joao, J. A.; Suleman, M. A.; Mutlu, O.; and Patt, Y. N., 2012. Bottleneck identification and scheduling in multithreaded applications. In ASPLOS '12: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems* (London, UK, Mar. 2012), 223–234. ACM, New York, NY, USA. doi:10.1145/2150976.2151001. (cited on pages 7, 73, 74, and 75)

Joao, J. A.; Suleman, M. A.; Mutlu, O.; and Patt, Y. N., 2013. Utility-based acceleration of multithreaded applications on asymmetric CMPs. In ISCA '13: *Proceedings of the 40th International Symposium on Computer Architecture* (Tel-Aviv, Israel, Jun. 2013), 154–165. ACM, New York, NY, USA. doi:10.1145/2485922.2485936. (cited on pages 7, 73, 74, 75, 78, 79, and 86)

Kahng, A. B.; Li, B.; Peh, L.-S.; and Samadi, K., 2009. ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration. In DATE '09: *Proceedings of the Conference on Design, Automation and Test in Europe* (Nice, France, Apr. 2009), 423–428. IEEE Computer Society, Washington, DC, USA. (cited on page 31)

Kumar, R.; Farkas, K. I.; Jouppi, N. P.; Ranganathan, P.; and Tullsen, D. M., 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In MICRO '03: *Proceedings of the 36th Annual International Symposium on Microarchitecture* (San Diego, CA, USA, Dec. 2003), 81–92. ACM, New York, NY, USA / IEEE Computer Society, Washington, DC, USA. doi:10.1145/1028176.1006707. (cited on pages 1 and 5)

Kumar, R.; Tullsen, D. M.; Ranganathan, P.; Jouppi, N. P.; and Farkas, K. I., 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In ISCA '04: *Proceedings of the 31st International Symposium on Computer Architecture* (Island of Kos, Greece, Jun. 2004), 64–75. IEEE Computer Society, Washington, DC, USA. doi:10.1109/ISCA.2004.1310764. (cited on pages 6, 7, and 73)

Laros III, J. H.; Pedretti, K.; Kelly, S. M.; Shu, W.; Ferreira, K.; Vandyke, J.; and Vaughan, C., 2013. Energy-efficient high performance computing. (2013), 51–55. doi:10.1007/978-1-4471-4492-2_8. (cited on page 51)

Le Sueur, E., 2011. *An analysis of the effectiveness of energy management on modern computer processors*. Master's thesis, School of Computer Science and Engineering, University of NSW, Sydney. (cited on page 55)

Le Sueur, E. and Heiser, G., 2010. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Workshop on Power Aware Computing and Systems* (Vancouver, Canada, Oct. 2010). (cited on page 23)

Li, S. H., 2011. Linux kernel bug 5471. https://bugzilla.kernel.org/show_bug.cgi?id=5471. (cited on page 26)

Li, T.; Baumberger, D. P.; Koufaty, D. A.; and Hahn, S., 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In SC '07: *Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing* (Reno, Nevada, USA, Nov. 2007), 53. ACM, New York, NY, USA. doi:10.1145/1362622.1362694. (cited on pages 6 and 7)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. (cited on page 8)

Maas, M.; Reames, P.; Morlan, J.; Asanovic, K.; Joseph, A. D.; and Kubiatowicz, J., 2012. GPUs as an opportunity for offloading garbage collection. In ISMM '12: *Proceedings of the 11th International Symposium on Memory Management* (Beijng, China, Jun. 2012), 25–36. ACM, New York, NY, USA. doi:10.1145/2258996.2259002. (cited on pages 12 and 13)

McCarthy, J., 1978. History of Lisp. *ACM SIGPLAN Notices*, 13 (Aug. 1978), 217?223. (cited on page 10)

Meyer, M., 2004. A novel processor architecture with exact tag-free pointers. *IEEE Micro*, 24, 3 (2004), 46–55. doi:10.1109/MM.2004.2. (cited on page 12)

Meyer, M., 2005. An on-chip garbage collection coprocessor for embedded real-time systems. In RTCSA '05: *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications* (Hong Kong, China, Aug. 2005), 517–524. IEEE Computer Society, Washington, DC, USA. doi:10.1109/RTCSA.2005.25. (cited on page 12)

Meyer, M., 2006. A true hardware read barrier. In ISMM '06: *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada, Jun. 2006), 3–16. ACM, New York, NY, USA. doi:10.1145/1133956.1133959. (cited on page 12)

Mogul, J. C.; Mudigonda, J.; Binkert, N. L.; Ranganathan, P.; and Talwar, V., 2008. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28, 3 (2008), 26–41. doi:10.1109/MM.2008.47. (cited on page 6)

Moon, D. A., 1984. Garbage collection in a large LISP system. In LFP '84: *Proceedings of the 1984 ACM Conference on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 235–246. ACM, New York, NY, USA. doi:10.1145/800055.802040. (cited on page 8)

Moon, D. A., 1985. Architecture of the Symbolics 3600. In ISCA '85: *Proceedings of the 12nd International Symposium on Computer Architecture* (Boston, MA, USA, Jun. 1985), 76–83. IEEE Computer Society, Washington, DC, USA. doi:10.1145/327010.327133. (cited on page 12)

Morad, T. Y.; Weiser, U. C.; Kolodnyt, A.; Valero, M.; and Ayguadé, E., 2006. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 5, 1 (2006), 14–17. doi:10.1109/L-CA.2006.6. (cited on pages 1 and 5)

Nilsen, K. D. and Schmidt, W. J., 1994. A high-performance architecture for real-time garbage collection. *Journal of Programming Language*, 2 (Jan. 1994), 1–40. (cited on page 12)

NVIDIA, 2013. NVIDIA Tegra 4 family CPU architecture. (cited on page 1)

Ossia, Y.; Ben-Yitzhak, O.; Goft, I.; Kolodner, E. K.; Leikehman, V.; and Owshanko, A., 2002. A parallel, incremental and concurrent GC for servers. In PLDI '02: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, Jun. 2002), 129–140. ACM, New York, NY, USA. doi:10.1145/512529.512546. (cited on page 9)

O'Toole, J. and Nettles, S., 1984. Concurrent replicating garbage collection. In LFP '94: *Proceedings of the 1994 ACM Conference on LISP and Functional Programming* (Orlando, Florida, USA, Aug. 1984), 34–42. ACM, New York, NY, USA. doi:10.1145/182409.182425. (cited on page 9)

Pallipadi, V. and Starikovskiy, A., 2006. The ondemand governor: Past, present and future. In *Proceedings of Linux Symposium*, vol. 2, 223–238. (cited on page 23)

Piszczalski, M., 2012. Locating data centers in an energy-constrained world. http://pro.gigaom.com/2012/05/locating-data-centers-in-an-energy-constrained-world/. (cited on page 1)

Pizlo, F.; Frampton, D.; and Hosking, A. L., 2011. Fine-grained adaptive biased locking. In PPPJ '11: *PPPJ* (Kongens Lyngby, Denmark, Aug 2011), 171–181. ACM, New York, NY, USA. doi:10.1145/2093157.2093184. (cited on page 77)

Printezis, T. and Detlefs, D., 2000. A generational mostly-concurrent garbage collector. In ISMM '00: *Proceedings of the 2000 International Symposium on Memory Management* (Minneapolis, Minnesota, USA, Oct. 2000), 143–154. ACM, New York, NY, USA. doi:10.1145/362422.362480. (cited on page 9)

Russell, K. and Detlefs, D., 2006. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In OOPSLA '06: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (Portland, Oregon, USA, Oct. 2006), 263–272. ACM, New York, NY, USA. doi:10.1145/1167473.1167496. (cited on page 79)

Saez, J. C.; Fedorova, A.; Koufaty, D.; and Prieto, M., 2012. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Transactions on Computer Systems*, 30, 2 (2012), 6. doi:10.1145/2166879.2166880. (cited on pages 6, 7, 73, 80, 81, and 86)

Saez, J. C.; Fedorova, A.; Prieto, M.; and Vegas, H., 2010. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In CF '10: *Proceedings of the 7th Conference on Computing Frontiers* (Bertinoro, Italy, May 2010), 31–40. ACM, New York, NY, USA. doi:10.1145/1787275.1787281. (cited on pages 6 and 73)

Saez, J. C.; Shelepov, D.; Fedorova, A.; and Prieto, M., 2011. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *Journal of Parallel and Distributed Computing*, 71, 1 (2011), 114–131. doi:10.1016/j.jpdc.2010.08.020. (cited on pages 6, 7, and 73)

Sarkar, C., 2010. Scheduling in linux. http://www.cs.montana.edu/~chandrima.sarkar/AdvancedOS/CSCI560_Proj_main/index.html. (cited on pages 75 and 88)

Sasanka, R.; Adve, S. V.; Chen, Y.-K.; and Debes, E., 2004. The energy efficiency of CMP vs. SMT for multimedia workloads. In ICS '04: *Proceedings of the 18th Annual International Conference on Supercomputing* (Saint Malo, France, Jun. 2004), 196–206. ACM, New York, NY, USA. doi:10.1145/1006209.1006238. (cited on page 38)

Shelepov, D.; Saez, J. C.; Jeffery, S.; Fedorova, A.; Perez, N.; Huang, Z. F.; Blagodurov, S.; and Kumar, V., 2009. HASS: a scheduler for heterogeneous multicore systems. *Operating Systems Review*, 43, 2 (2009), 66–75. doi:10.1145/1531793.1531804. (cited on pages 6 and 7)

Singhal, R., 2011. Inside Intel next generation Nehalem microarchitecture. Intel Developer Forum (IDF) presentation (August 2008). http://software.intel.com/file/18976. (cited on page 36)

Srisa-an, W.; Lo, C.-T. D.; and Chang, J. M., 2003. Active memory processor: A hardware garbage collector for real-time Java embedded devices. *IEEE Transaction of Mobile Computing.*, 2, 2 (2003), 89–101. doi:10.1109/TMC.2003.1217230. (cited on page 12)

Stanchina, S. and Meyer, M., 2007a. Exploiting the efficiency of generational algorithms for hardware-supported real-time garbage collection. In SAC '07: *Proceedings of the 2007 ACM Symposium on Applied Computing* (Seoul, Korea, Mar. 2007), 713–718. ACM, New York, NY, USA. doi:10.1145/1244002.1244161. (cited on pages 12 and 13)

Stanchina, S. and Meyer, M., 2007b. Mark-sweep or copying?: a "best of both worlds" algorithm and a hardware-supported real-time implementation. In ISMM '07: *Proceedings of the 6th International Symposium on Memory Management* (Montreal, Quebec, Canada, Oct. 2007), 173–182. ACM, New York, NY, USA. doi:10.1145/1296907.1296928. (cited on page 12)

Standard Performance Evaluation Corporation, 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 34, 4 (Sept. 2006), 1–17. (cited on page 18)

Standard Performance Evaluation Corporation, 2010. SPEC Benchmarks. http://www.spec.org. (cited on page 19)

Suleman, M. A.; Mutlu, O.; Qureshi, M. K.; and Patt, Y. N., 2010. Accelerating critical section execution with asymmetric multicore architectures. *IEEE MICRO*, 30, 1 (2010), 60–70. doi:10.1145/1508244.1508274. (cited on page 73)

Texas Instruments, 2013. OMAP4470 mobile applications processor. (cited on page 99)

The DaCapo Research Group, 2006. The DaCapo Benchmarks, beta-2006-08. http://www.dacapobench.org. (cited on page 19)

The Jikes RVM Research Group, 2011. Jikes Open-Source Research Virtual Machine. http://www.jikesrvm.org. (cited on page 8)

Tullsen, D. M.; Eggers, S. J.; and Levy, H. M., 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In ISCA '95: *Proceedings of the 22nd International*

Symposium on Computer Architecture (Santa Margherita Ligure, Italy, Jun. 1995), 392–403. ACM, New York, NY, USA. doi:10.1145/200912.201006. (cited on page 36)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In SDE '84: *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, USA, Apr. 1984), 157–167. ACM, New York, NY, USA. doi:10.1145/800020.808261. (cited on page 8)

Ungar, D., 1987. *The design and evaluation of a high performance Smalltalk system*. Ph.D. thesis, University of California, Berkeley. (cited on page 12)

Velasco, J.; Atienza, D.; Olcoz, K.; Catthoor, F.; Tirado, F.; and Mendias, J., 2005. Energy characterization of garbage collectors for dynamic applications on embedded systems. In PATMOS '05: *Proceedings of Integrated Circuit and System Design, Power and Timing Modeling, Optimization and Simulation, 15th International Workshop* (Leuven, Belgium, Sept. 2005), 908–915. Springer, New York, NY, USA. doi:10.1007/11556930_8. (cited on page 11)

Venkatesh, G.; Sampson, J.; Goulding, N.; Garcia, S.; Bryksin, V.; Lugo-Martinez, J.; Swanson, S.; and Taylor, M. B., 2010. Conservation cores: Reducing the energy of mature computations. In ASPLOS '10: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems* (Pittsburgh, PA, USA, Mar. 2010), 205–218. ACM, New York, NY, USA. doi:10.1145/1736020.1736044. (cited on page 31)

Wise, D. S.; Heck, B. C.; Hess, C.; Hunt, W.; and Ost, E., 1997. Research demonstration of a hardware reference-counting heap. *Lisp and Symbolic Computation*, 10, 2 (1997), 159–181. (cited on page 12)

Wolczko, M. and Williams, I., 1992. Multi-level garbage collection in a high-performance persistent object system. In POS '92: *Proceedings of the Fifth International Workshop on Persistent Object Systems* (San Miniato (Pisa), Italy, Sept. 1992), 396–418. Springer, New York, NY, USA. (cited on page 12)

Yang, X.; Blackburn, S.; Frampton, D.; Sartor, J.; and McKinley, K., 2011. Why nothing matters: The impact of zeroing. In OOPSLA '11: *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications* (Portland, OR, USA, Oct. 2011), 307–324. ACM, New York, NY, USA. doi:10.1145/2048066.2048092. (cited on page 52)

Yuasa, T., 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11 (Mar. 1990), 181–198. doi:10.1016/0164-1212(90)90084-Y. (cited on page 9)