Copyright
by
Ivan Jibaja
2015

The Dissertation Committee for Ivan Jibaja
certifies that this is the approved version of the following dissertation:

Exploiting Hardware Heterogeneity and Parallelism for Performance and Energy Efficiency of Managed Languages

Committee:

Kathryn S. McKinley, Supervisor

Stephen M. Blackburn, Co-Supervisor

Emmett Witchel, Supervisor

Don Batory

Calvin Lin

Exploiting Hardware Heterogeneity and Parallelism for Performance and Energy Efficiency of Managed Languages

by

Ivan Jibaja, B.S.; M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2015

To my family and friends for all of their love and support, especially my wife Shannon and my parents.

Acknowledgments

I would like to express my deepest appreciation for my advisors, Kathryn McKinley and Steve Blackburn, for supporting and mentoring me academically and personally. Their dedication, technical insights, interest, and consideration of me (and all of their students) made this PhD possible. Kathryn's optimism and enthusiasm encouraged me to follow my ideas, while Steve's perseverance and rigorousness inspired me to seek solutions for the seemingly impossible. Together, their confidence in me was the fuel to pursue different research interests and supported me through all the hard times. As I reflect back on my time during graduate school, I am inspired by Kathryn and Steve to always strive to be a better person.

I am thankful to have had a remarkable mentor, Mohammad Haghighat, as part of many internships. Moh has been a supporter and has devoted a great amount of time to giving me career and personal advice. He has always celebrated my successes enthusiastically and encouraged me to always keep moving forward during difficult times. During the last years of my PhD while I was working remotely, he provided me with a second home at Intel. There, I was surrounded by people doing exciting and novel work. I thank Moh for teaching me the value of balancing computer science and engineering. I'm also thankful to Peter Jensen, who always patiently listened to all of my complaints during the hardest part of graduate school and encouraged me to finish.

I am also incredibly grateful to my mentor, undergraduate advisor, and friend, Kelly Shaw, for her immense patience with me as an undergraduate

student researcher. Kelly's belief in me, advice, support, and encouragement to follow a career in computer science research are the reasons why I ended up at UT pursuing a PhD.

I would like to thank all of the great friends I made in graduate school for their support, collaboration, and for the pleasure of their company. I am thankful to Bryan Marker, Xi Yang, Alan Dunn, Mrinal Deo, Ting Cao, Na Meng, Suriya Subramanian, and Katie Coons for all of their patient listening in good and bad times. Whether I was physically in Austin, Canberra, or in the CA Bay Area, I always knew they were a phone (or Skype) call away if I ever needed support. I'm also thankful to my good friend, Kristof Zetenyi. Although in different fields and across the country, we started graduate school at the same time after undergrad and commiserated over the hardships as well as celebrated our progress.

I would like to thank all of the members of the UT Speedway and ANU Computer Systems research groups. Although our time together was short, I always received advice, encouragement, and support throughout my PhD when our paths would cross at conferences or other events.

I am thankful to Lindy Aleshire and Lydia Griffith for their guidance and help navigating UT. Lindy's responsiveness, quick wit, and positive attitude through all my unique circumstances will always be appreciated.

I would not be here today if it weren't for my family. Particularly, my parents, Ivan and Nelly, have provided me with unconditional love, selfless support, affection, and a great education throughout my life; they have seen me through thick and thin and always pushed me to reach for the sky. Additionally, my cousins, Diego and Amanda, celebrated each of my successes

through the last years of my PhD and helped me keep moving through the hardships.

Finally, I would like to thank my best friend and wife, Shannon, for her love and unwavering belief in me. She has been my rock and a source of unconditional support and encouragement, and tolerated me throughout this long journey.

P.S.: I would like to express my sincere 'gratitude' to my cats, Luna and Tigre, for laying down on my keyboard every single time that I sat down to write this thesis.

Exploiting Hardware Heterogeneity and Parallelism for Performance and Energy Efficiency of Managed Languages

Publication No. ____________________

Ivan Jibaja, Ph.D.
The University of Texas at Austin, 2015

Supervisors: Kathryn S. McKinley
Stephen M. Blackburn
Emmett Witchel

On the software side, managed languages and their workloads are ubiquitous, executing on mobile, desktop, and server hardware. Managed languages boost the productivity of programmers by abstracting away the hardware using virtual machine technology. *On the hardware side*, modern hardware increasingly exploits parallelism to boost energy efficiency and performance with homogeneous cores, heterogenous cores, graphics processing units (GPUs), and vector instructions. Two major forms of parallelism are: task parallelism on different cores and vector instructions for data parallelism. With task parallelism, the hardware allows simultaneous execution of multiple instruction pipelines through multiple cores. With data parallelism, one core can perform the same instruction on multiple pieces of data. Furthermore, we expect hardware parallelism to continue to evolve and provide more heterogeneity. Existing programming language runtimes must continuously evolve so

programmers and their workloads may efficiently utilize this evolving hardware for better performance and energy efficiency. However, efficiently exploiting hardware parallelism is at odds with programmer productivity, which seeks to abstract hardware details.

My thesis is that managed language systems should and can abstract hardware parallelism with modest to no burden on developers to achieve high performance, energy efficiency, and portability on ever evolving parallel hardware. In particular, this thesis explores how the runtime can optimize and abstract heterogenous parallel hardware and how the compiler can exploit data parallelism with new high-level languages abstractions with a minimal burden on developers.

We explore solutions from multiple levels of abstraction for different types of hardware parallelism. (1) For asymmetric multicore processors (AMP) which have been recently introduced, we design and implement an application scheduler in the Java virtual machine (JVM) that requires no changes to existing Java applications. The scheduler uses feedback from dynamic analyses that automatically identify critical threads and classifies application parallelism. Our scheduler automatically accelerates critical threads, honors thread priorities, considers core availability and thread sensitivity, and load balances scalable parallel threads on big and small cores to improve the average performance by 20% and energy efficiency by 9% on frequency-scaled AMP hardware for scalable, non-scalable, and sequential workloads over prior research and existing schedulers. (2) To exploit vector instructions, we design SIMD.js, a portable single instruction multiple data (SIMD) language extension for JavaScript (JS), and implement its compiler support that together add fine-grain data parallelism to JS. Our design principles seek portability,

scalable performance across various SIMD hardware implementations, performance neutral without SIMD hardware, and compiler simplicity to ease vendor adoption on multiple browsers. We introduce *type speculation*, compiler optimizations, and code generation that convert high-level JS SIMD operations into minimal numbers of SIMD native instructions. Finally, to accomplish wide adoption of our portable SIMD language extension, we explore, analyze, and discuss the trade-offs of four different approaches that provide the functionality of SIMD.js when vector instructions are not supported by the hardware. SIMD.js delivers an average performance improvement of $3.3\times$ on micro benchmarks and key graphic algorithms on various hardware platforms, browsers, and operating systems. These language extension and compiler technologies are in the final approval process to be included in the JavaScript standards.

This thesis shows using virtual machine technologies protects programmers from the underlying details of hardware parallelism, achieves portability, and improves performance and energy efficiency.

Table of Contents

Acknowledgments **v**

Abstract **viii**

List of Tables **xiv**

List of Figures **xv**

Chapter 1. Introduction **1**

1.1 Problem Statement 1
1.2 Contribution 3
1.2.1 AMP scheduler 4
1.2.2 SIMD.js 6
1.3 Thesis Outline 8

Chapter 2. Efficient Scheduling of Asymmetric Multicore Processors **9**

2.1 Motivation 9
2.2 Challenges 10
2.3 Related Work and Quantitative Analysis 10
2.4 Overview of a Managed Language AMP Runtime 14
2.5 Workload Analysis 16
2.6 Dynamic Workload and Bottleneck Analysis 20
2.7 Speedup and Progress Prediction 23
2.8 The WASH Scheduling Algorithm 26
2.8.1 Overview 26
2.8.2 Single-Threaded and Low Parallelism 29
2.8.3 Scalable multithreaded 30
2.8.4 Non-scalable multithreaded WASH 30

2.9 Methodology 32
2.10 Results 36
2.10.1 Single-Threaded Benchmarks 38
2.10.2 Scalable Multithreaded Benchmarks 39
2.10.3 Non-scalable Multithreaded Benchmarks 40
2.10.4 Sensitivity to speedup model 42
2.10.5 Thread & Bottleneck Prioritization 43
2.10.6 Multiprogrammed workloads 44
2.11 Conclusion 51

Chapter 3. Vector Parallelism in JavaScript **52**
3.1 Motivation 52
3.2 Related Work 54
3.3 Our Contribution 57
3.4 Design Rationale 61
3.5 Language Specification 62
3.5.1 Data Types 62
3.5.2 Operations 63
3.6 Compiler Implementations 67
3.6.1 Type Speculation 71
3.7 Methodology 75
3.7.1 Virtual Machines and Measurements 75
3.7.2 Hardware Platforms 75
3.7.3 Benchmarks 76
3.7.4 Measurement Methodology 79
3.8 Results 80
3.8.1 Time 80
3.8.2 Energy 83
3.9 Scalarlization 84
3.9.1 Polyfill in JavaScript 87
3.9.2 Polyfill in asm.js 89
3.9.3 JIT code generation of scalar code 91

3.9.4 asm.js code 93
3.9.5 Discussion 94
3.10 Future Work Discussion 96
3.11 Conclusion 97

Chapter 4. Conclusion **99**

List of Tables

2.1 Qualitative comparison of related work to the WASH AMP-aware runtime. . . . 13

2.2 Experimental processors. . . . 33

3.1 Experimental platforms 75

3.2 Benchmark characteristics. We measure the lines of code (LOC) for the kernel of each benchmark in both scalar and SIMD variants. †In the case of Sinex4, the table reports the LOC for the simple sine kernel, which makes calls to the sine function in the Math libary and the equivalent SIMD implementation respectively. The full first-principles implementation of SIMD sine takes 113 LOC and makes 74 SIMD calls. . . . 78

List of Figures

2.1 Linux OS scheduler (Oblivious) on homogenous configurations, normalized to one big core. We classify benchmarks as single threaded (ST), non-scalable multithreaded (MT), and scalable MT. Lower is better. 17

2.2 Quantitive time comparison of Proportional Fair Scheduling (PFS) [46, 12], Linux (Oblivious), and bindVM [10] on AMP configurations normalized to 1B5S Oblivious. Lower is better. No approach dominates. 19

2.3 Fraction of time spent waiting on locks / cycles, per thread in multithreaded benchmarks. Left benchmarks (purple) are scalable, right (pink) are not. Low ratios are highly predictive of scalability. 22

2.4 PCA selects best performance counters to predict core sensitivity of threads with linear regression. 23

2.5 Accurate prediction of thread core sensitivity. Y-axis is predicted. X-axis is actual speedup. 23

2.6 Geomean time, power and energy with Oblivious, bindVM, and WASH on all three hardware configs. 37

2.7 Single-threaded benchmarks. Normalized geomean time, power and energy for different benchmark groups. Lower is better. . 38

2.8 Scalable multithreaded. Normalized geomean time, power and energy for different benchmark groups. Lower is better. 39

2.9 Non-scalable multithreaded. Normalized geomean time, power and energy for different benchmark groups. Lower is better. . 40

2.10 WASH with best (default) model, second best model, and a bad model (inverse weights) on 1B5S. 42

2.11 WASH (green) out-performs our implementation of PFS [46, 12] (red) which lacks lock analysis and VM vs application priority on 1B5S. 44

2.12 Performance with eclipse adversary on 1B5S. 45

2.13 Running time on 1B5S 46

2.14 Energy on 1B5S 47

2.15 Power on 1B5S 47
2.16 Running time on 2B4S 48
2.17 Energy on 2B4S 48
2.18 Power on 2B4S 49
2.19 Running time on 3B3S 49
2.20 Energy on 3B3S 50
2.21 Power on 3B3S 50

3.1 SIMD Type Hierarchy 64
3.2 Visualization of `averagef32x4` summing in parallel. 68
3.3 V8 Engine Architecture. 69
3.4 Example V8 compiler generated code 73
3.5 SIMD performance with V8 and SpiderMonkey (SM). Normalized to scalar versions. Higher is better. 81
3.6 Comparison of scalar vs. SIMD versions of ShiftRows function 82
3.7 SIMD energy on select platforms with V8. Normalized to the scalar versions. Higher is better. 84
3.8 Performance comparison when naively replacing vector instructions for scalar instructions. Lower is better. 86
3.9 Snippet of polyfill support for SIMD.js 88
3.10 Snippet of asm.js polyfill support for SIMD.js 90
3.11 Phases of optimizing compiler (IonMonkey) from the Firefox web browser. 92
3.12 Examples JIT generated scalar code 92

Chapter 1

Introduction

This thesis explores how a programming language runtime can optimize and abstract heterogenous parallel hardware and how compilers can exploit data parallelism using new high-level languages abstractions with minimal burden on developers.

1.1 Problem Statement

Hardware is constantly evolving. As we reach the limit of Moore's law and power consumption has taken center stage, modern hardware has turned to parallelism as its primary solution. From mobile, desktop, to server devices, all modern devices use parallelism to drive performance and energy efficiency. Parallelism is where hardware computes multiple calculations simultaneously in order to solve a problem faster.

Contemporary hardware delivers parallelism in three major forms: instruction, task, and data parallelism. Instruction-level parallelism executes multiple instructions concurrently on a single processor. Task-level parallelism executes applications on multiple processors simultaneously. Data-parallelism uses vector instructions to execute the same calculation on multiple pieces of data simultaneously. For the last two decades, commodity hardware has had data parallel instruction sets available. Additionally, vendors are now intro-

ducing asymmetric multicore processors (AMPs) as they combine big high-performance cores and small energy-efficient cores to improve performance and energy. These changes in hardware demand software to adapt to take advantage of its new features for performance and energy efficiency, but forcing developers to constantly modify their applications to match evolving hardware is both impractical and costly.

Software demands are also changing. Programmer productivity, correctness, security, and portability have popularized programming languages with higher levels of abstractions over low-level compiled ahead-of-time languages. Managed languages abstract over hardware using virtual machines (VMs). Despite the power and performance overheads introduced by VMs, programmers choose high-level managed programming languages for the automatic memory management, portability across different hardware, lack of unsafe pointers, and large availability of standard libraries. Because this evolution occurred in concert with exponential increases in processor speed (the golden era of Moore's law), much of this cost was previously ignored.

Managed programming languages such as Java and C# have emerged in the server side while JavaScript is the language of choice in the new software landscape for web applications. Because these languages are portable (architecture-independent), the parallelism in the language does not target a particular hardware. Therefore, because hardware is evolving increasingly towards parallelism, the software stack – applications, compilers, runtime systems, and operating systems – must introduce new or leverage existing abstractions to exploit available hardware heterogeneity and parallelism for performance and energy efficiency. To ensure adoption of such abstractions, they must perform efficiently.

The abstraction provided by managed languages VMs offers both an opportunity and a challenge when addressing the constant evolution of hardware and software. The opportunity is that virtual machines (VMs) abstract over hardware complexity and profile, optimize, and schedule applications. The challenge is that applications written in managed languages are complex and *messy* which makes previous approaches unusable.

This thesis addresses the problem of matching the software abstractions and *messiness* provided by managed programming languages to the constantly evolving heterogeneity and parallelism that is in the hardware. Specifically, we tackle the problem of efficiently scheduling server applications written in Java onto asymmetric multicore processors, and allowing high-level web applications written in JavaScript to exploit the data parallel instructions available in the hardware.

1.2 Contribution

With with little or no work on the programmers side, we provide a VM abstraction layer over various machine-specific parallelism and heterogeneity details and add novel virtual machine technology that well exploits this parallelism and heterogeneity. We exploit the existing parallel language constructs in Java. We add new languages abstractions for data parallelism to JavaScript that are easy for programmers to use. We introduce new static and dynamic analyses, compile-time optimizations, and runtime optimizations that exploit the parallelism offered by the hardware. We show that with these changes managed languages can deliver significantly better performance and energy efficiency with modest programmer effort and in some cases, no effort.

In this thesis, we explore two particular types of hardware parallelism: asymmetric multicore and vector parallelism, both of which are already available today but not utilized by managed language applications. We leverage the opportunity provided in the managed language abstraction and attack this challenge in two ways. 1) For thread level parallelism on the server-side, we identify *messy* parallel, but non-scalable managed applications as an important and unaddressed workload. We introduce new analysis to match this workload to the capabilities of heterogenous multicore hardware. 2) For data level parallelism in the client-side, we design and implement a new high-level vector language extension and show how to map it down to commonly available low level data parallel hardware. This thesis focuses on these two major projects, which we overview in more detail below.

1.2.1 AMP scheduler

Asymmetric Multicore Processors (AMPs) combine big and small cores to optimize performance and energy efficiency. Big cores are intended to optimize latency, while small cores are intended to deliver efficient performance for throughput on parallel workloads and workloads that are not non-latency sensitive. Heterogeneous multicores are emerging because they offer substantial potential performance improvements over homogeneous multicores given the static and dynamic power constraints on current systems [21, 31, 33, 20, 42]. The challenge for AMP is to match the workload to the capabilities of the heterogenous cores.

To exploit this hardware, one must consider sensitivity of each thread to core selection, how to balance the load, and how to identify and schedule the critical path. Applying these criteria effectively is challenging, especially

for the complex and *messy* workloads from managed languages. Expecting programmers to take all of these variables into account is unrealistic. However, because managed languages virtual machines (VMs) provide a higher level of abstraction, they already profile, optimize, and schedule applications. We enhance these capabilities to make these AMP scheduling decisions automatically, invisible to the developer.

Because extracting performance from AMPs is hardware specific and further complicated by multiprogramming, application programmers should not be required to manage this complexity. Existing scheduler solutions require programmer hints and/or new hardware [16, 26, 27, 48]. We present the design and implementation of the WASH scheduler which improves over prior approaches by removing the need for programmer involvement or the introduction of new hardware. This approach is applicable to other managed languages with builtin parallel programming constructs, such as C#.

Key technical contributions of this work are:

1. We identify and characterize the inherent *messy* parallelism in managed languages workloads.
2. We demonstrate the power of information already available within a managed language VM for scheduling threads on AMP hardware.
3. We present automatic, accurate, and low-overhead dynamic analysis that: (a) classifies parallelism, (b) predicts core capabilities, (c) prioritizes threads holding contended locks, and (d) monitors thread progress.
4. In particular, we introduce the first fully automatic software mechanism to identify threads that hold critical locks that cause bottlenecks in multithreaded software. We then show how to priortize the most critical

of these threads based on the time other threads wait on them. Thus, we automatically prioritize multiple threads that hold multiple distinct locks.

5. We exploit this information in the WASH scheduler to customize its optimization strategy to the workload, significantly improving over previous approaches.

1.2.2 SIMD.js

Fine-grain vector parallel instructions – Single-Instruction, Multiple-Data (SIMD) have multiple processing units that simultaneously perform the same operation on multiple data values. Although many instruction set architectures have had SIMD instructions for a long time, high-level managed programming languages only offered access to them through their native interfaces. However, a native interface significantly limits the programmers productivity and portability across hardware because of the need for specific code for each architecture.

The JavaScript language specification does not include parallel constructs, such as parallel loops, threads, or locks, because the standards committee viewed the potential for concurrency errors from shared memory parallel constructs to out weigh the performance advantages of these constructs. However, more and more JavaScript workloads are compute intensive and amenable to parallelism, such as graphics workloads. We thus took a pragmatic choice to addressing this problem that side steps the concurrency errors in shared memory constructs. We add high-level language abstractions together with new compile-time optimizations to deliver a portable SIMD language extension for JavaScript (JS). To use these instructions from JavaScript or other dynamic

languages in the past, applications had to perform cross-language library calls, which are inefficient. This thesis introduces novel language extensions and new Just-in-Time (JIT) compiler optimizations that directly deliver the efficiency of SIMD assembly instructions from the high-level JavaScript SIMD operations. Applications that use these JavaScript SIMD language extensions achieve the full portability benefits of JavaScript, whether or not the hardware supports vector instructions, executing on any browser without the need for external plugins. When the hardware has vector instructions, the applications achieve vector performance. When the hardware lacks vector instructions, we explore and analyze four different approaches for providing SIMD.js functionality. At one end, we show how a simple library for SIMD.js adds correctness without any changes to browsers. At the other end, we add scalarizing compiler support for SIMD.js and show that it delivers performance and energy efficiency similar to the original scalar code, making SIMD.js performance neutral when hardware does not contain vector instructions. The language, compiler specification, and runtime support in this thesis is in the last stages of becoming part of the JavaScript language standard. This work thus has already had significant industry impact.

Key technical contributions of this work are:

1. A language design justified on the basis of portability and performance.
2. Compiler type speculation without profiling in a dynamic language.
3. The first dynamic language with SIMD instructions that deliver their performance and energy benefits to applications.
4. Support for performance and functionality of this language extension when hardware support is not available.

We use the VMs as an abstraction layer between the programmer and the new forms of parallelism offered by the hardware while still providing portability and improved performance and energy efficiency.

1.3 Thesis Outline

The body of this thesis is structured around the two key contributions outlined above, each one contains its own related work. The rest of this document is organized as follows. Chapter 2 details our scheduling work for asymmetric multicore processors. It presents the design and implementation of an asymmetric multicore processor (AMP) application scheduled in the Java Virtual Machine (JVM) that uses feedback from dynamic analyses to improve the performance and energy efficiency of Java applications on a frequency-scaled AMP. This work is accepted to appear at the International Symposium on Code Generation and Optimization (CGO) 2016. Chapter 3 presents the design for our portable SIMD language extension for JavaScript, SIMD.js, and its compiler support. This language extension allows JavaScript applications to exploit the already widely available hardware heterogeneity in the form of vector instructions. This work is currently in final stages of approval [49] by the TC39 ECMAScript standardization committee and has been published in the proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT 2015). Chapters 2, and 3 each covers the related work and background material in detail. Finally, Chapter 4 concludes the thesis, summarizing how the contributions presented in this thesis abstract hardware parallelism with little or no burden on developers to achieve high performance, energy efficiency, and portability on ever evolving parallel hardware.

Chapter 2

Efficient Scheduling of Asymmetric Multicore Processors

2.1 Motivation

By combining *big* high-performance cores and *small* energy-efficient cores, single-ISA Asymmetric Multicore Processors (AMPs) hold promise for improving performance and energy efficiency significantly, while meeting power and area constraints faced by architects [21, 31, 33, 20, 42]. Bigger cores accelerate the critical latency-sensitive parts of the workload, whereas smaller cores deliver throughput by efficiently executing parallel workloads. The first device using Qualcomm's 4 big and 4 little ARM processor was delivered in November 2014 and more are announced for 2015 [42], showing this design is gaining momentum in industry. Unfortunately, even when cores use the same ISA, forcing each application to directly manage heterogeneity will greatly inhibit portability across hardware configurations with various numbers and types of cores, and across hardware generations. Adding to this complexity, asymmetry may also be dynamic, due to core frequency and voltage scaling, core defeaturing, simultaneous multithreading (SMT) resource contention, and competing applications. Relying on application programmers to realize the potential of AMPs seems unwise.

2.2 Challenges

To achieve the performance and energy efficiency promises of AMP is challenging because the runtime must reason about *core sensitivity*, which thread's efficiency benefits most from which core; *priorities*, executing non-critical work on small cores and prioritizing the critical path to big cores; and *load balancing*, effectively utilizing available hardware resources. Prior work addresses some but not all of these challenges. For instance, prior work accelerates the critical path on big cores [16, 26, 27, 48], but needs programmer hints and/or new hardware. Other work manages core sensitivity and load balancing with proportional fair scheduling [11, 12, 45, 46], but their evaluation is limited to scalable applications and multiprogrammed settings with equal numbers of threads and hardware contexts, an unrealistic assumption in the multiprogrammed context. Many real-world parallel applications also violate this assumption. For instance, the eclipse IDE manages logical asynchronous tasks with more threads than cores. Even if the application matches its threads to cores, runtimes for many languages add compiler and garbage collection helper threads. We experimentally show that the prior work does not handle such complex parallel workloads well.

2.3 Related Work and Quantitative Analysis

Table 2.1 qualitatively compares our approach to prior work with respect to algorithmic features and target workload. As far as we are aware, our approach is the only one to automatically identify bottlenecks in software and to comprehensively optimize *critical path*, *core sensitivity*, *priorities* and *load balancing*. Next, we overview how related work addresses these individual concerns and then present a quantitative analysis that motivates our approach.

Critical path Amdahl's law motivates accelerating the critical path by scheduling it on the fastest core [16, 26, 27, 48]. However, no prior work *automatically* identifies and prioritizes the critical path in software as we do. For instance, Joao et al.[27] use programmer hints and hardware to priorize threads that hold locks. Du Bois et al. [16] identify and accelerate critical thread(s) by measuring its useful work and the number of waiting threads with new hardware, but do not integrate into a scheduler. Our software approach automatically optimizes more complex workloads.

Priorities Scheduling low priority tasks, such as VM helper threads, OS background tasks, and I/O tasks, on small cores improves energy efficiency [10, 37], but these systems do not schedule the application threads on AMP cores. No prior AMP scheduler integrates priorities with application scheduling, as we do here.

Core sensitivity Prior work chooses the appropriate cores for competing threads using a cost benefit analysis based on speedup, i.e., how quickly each thread retires instructions on a fast core relative to a slow core [7, 11, 12, 30, 45, 46]. Systems model and measure speedup. Direct measurement executes threads on each core type and uses IPC to choose among competing threads [7, 12, 30]. To avoid migrating only for profiling and to detect phase changes, systems train predictive models offline with features such as ILP, pipeline-stalls, cache misses, and miss latencies collected from performance counters [11, 12, 45, 46]. At execution time, the models prioritize threads to big cores based on their relative speedup [11]. We combine profiling and a predictive model.

Load balancing *Static* schedulers do not migrate threads after choosing a core [30, 45], while *dynamic* schedulers adapt to thread behaviors [7, 12, 46] and load imbalance [33]. Li et al. schedule threads on fast cores first and ensure load is proportional to core capabilities and consider migration overhead, but do not consider core sensitivity nor complex workloads [33]. Saez et al. [46] and Craeynest et al. [12] both perform *proportional fair scheduling* (PFS) on scalable applications. The OS scheduler load balances by migrating threads between core types based on progress and core capabilities. They simplify the problem by assuming threads never exceed the number of cores: $|\text{threads}| \leq |\text{cores}|$ (pg. 20 [46]). Craeynest et al. compare to a scheduler that binds threads to cores, a poor baseline.

Quantitative Analysis We first explore the performance of PFS, the best software only approach [12, 46] and compare it to the default round-robin Linux (Oblivious) scheduler, which seeks to keep all cores busy, and avoids thread migration [38, 34], and to Cao et al. [10] (bindVM), which simply binds VM helper threads to small cores. We execute 14 DaCapo Java benchmarks [8]. (Section 2.9 describes methodology in detail.) Figure 2.2 shows the performance on AMP configurations organized by workload type: sequential (ST), scalable, non-scalable. Note that (a) no one approach dominates on every workload; (b) bindVM performs best on sequential and non-scalable; (c) Oblivious and PFS perform essentially the same on scalable workloads, because in these workloads, threads out-number cores, and thus both reduce to round-robin scheduling across all cores. Since Oblivious and bindVM dominate PFS, we use them as our baseline throughout the paper. While bindVM is best here on non-scalable workloads, we will show that WASH is better.

Approach	Algorithmic Features: core sensitivity	load balancing	critical path	priorities	requires hints	Workload: limited to $\lvert threads \rvert \leq \lvert cores \rvert$	sequential	scalable	non-scalable
Becchi and Crowley [7]	✓				no	no	✓		
Kumar et al. [30]	✓				no	yes	✓		
Shelepov et al. [47]	✓				no	yes	✓		
Craeynest et al. [11]	✓				yes	yes	✓		
Craeynest et al. [12]	✓	✓			no	yes	✓	✓	✓
Saez et al. [46]	✓	✓			no	yes	✓	✓	
Du Bois et al. [16]			✓		yes	yes		✓	
Suleman et al. [48]			✓		yes	yes		✓	✓
Joao et al. [26]			✓		yes	yes		✓	✓
Joao et al. [27]	✓		✓		yes	yes	✓	✓	✓
Li et al. [33]		✓			no	no	✓	✓	✓
Cao et al. [10]	limited			✓	no	no	✓	✓	✓
VM + WASH	**✓**	**✓**	**✓**	**✓**	**no**	**no**	**✓**	**✓**	**✓**

Table 2.1: Qualitative comparison of related work to the WASH AMP-aware runtime.

2.4 Overview of a Managed Language AMP Runtime

This chapter introduces an AMP aware runtime that includes: (1) a model that predicts thread sensitivity on frequency-scaled cores and same-ISA cores with different microarchitectures; (2) dynamic analysis that computes and assesses expected thread progress using the model and performance counter data; (3) dynamic parallelism analysis that determines scalability based on work, progress, resource contention, and time waiting on contended locks; (4) a priori tagging of helper threads as low priority; and (5) a new Workload Aware Scheduler for Heterogeneous systems (WASH). WASH first classifies application behavior as single threaded, non-scalable multi-threaded, and scalable multi-threaded and then customizes its scheduling decisions accordingly. For instance, it proportionally fair schedules scalable applications and accelerates the critical path in non-scalable applications.

A key contribution of our work is a new dynamic analysis that automatically identifies *bottleneck threads* that hold contended locks and prioritizes them by the cumulative time other threads wait on them. This analysis finds and accelerates the critical path, improving *messy* non-scalable workloads. For efficiency, we piggyback it on the VM's biased locking [6, 3]. Our VM profiling periodically monitors thread progress, thread core sensitivity, and communicates scheduling decisions to the OS with thread affinity settings.

Evaluation We implement our AMP runtime in a high performance Java VM for desktops and servers. Its mature compiler, runtime, and benchmarking ecosystems make it a better evaluation platform than immature mobile systems. We evaluate benchmarks from active open source projects on an AMD Phenom II x86 with core-independent frequency scaling (DVFS) configured as

an AMP. Prior work establishes this methodology [10, 46], which dramatically increases experiments compared to simulation, which is slow and less accurate. This methodology understates the benefits of AMP on energy efficiency since DVFS is less effective than different microarchitectures are at delivering performance at low power. We compare to (a) the default round-robin Linux scheduler [38, 34], (b) Cao et al. [10], which simply binds VM helper threads to small cores, and (c) proportional fair scheduling (PFS) [46, 12], the closest related work.

WASH improves energy and performance over these schedulers in various AMP configurations: 9% average energy and 20% performance, and up to 27% as hardware asymmetry increases. Although, simply binding helper threads to small cores works well for sequential workloads, and round-robin and PFS [46, 12] work well for scalable workloads, no prior work performs well on all workloads. In particular, we improve over PFS and the others because WASH efficiently identifies and prioritizes bottleneck threads that hold locks in *messy* non-scalable workloads. These results understate the benefits on an AMPs with an optimized microarchitecture.

Our VM scheduler is just as effective in a multiprogrammed workload consisting of a complex multithreaded adversary scheduled by the OS. Our VM approach adjusts even when the OS is applying an independent scheduling algorithm. A sensitivity analysis of our core model shows that we need a good predictor, but not a perfect one. In summary,

1. We demonstrate the power of information available within the VM for scheduling threads on AMP hardware.

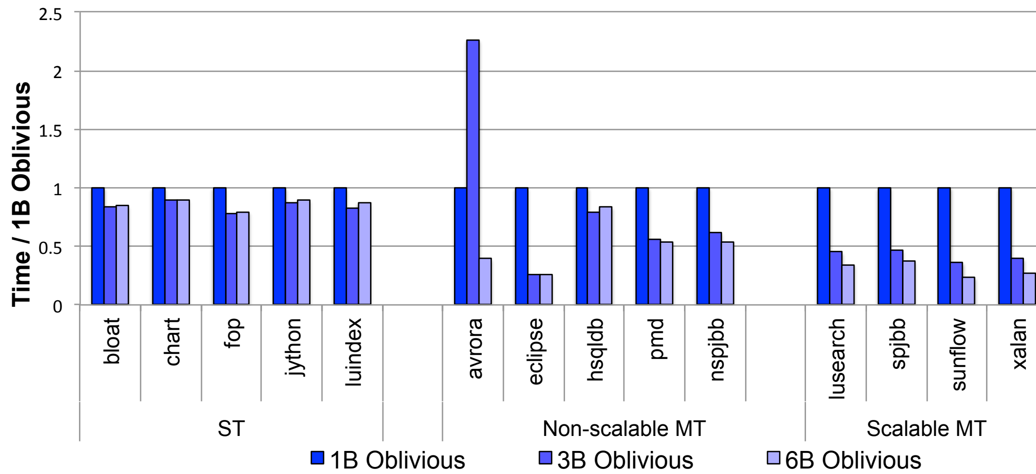
2. We present automatic, accurate, and low-overhead VM dynamic analysis that: (a) classifies parallelism, (b) predicts core capabilities, (c) prioritizes threads holding contended locks, and (d) monitors thread progress.
3. We exploit this information in the WASH scheduler to customize its optimization strategy to the workload, significantly improving over other approaches.
4. We will open source our system upon publication.

2.5 Workload Analysis

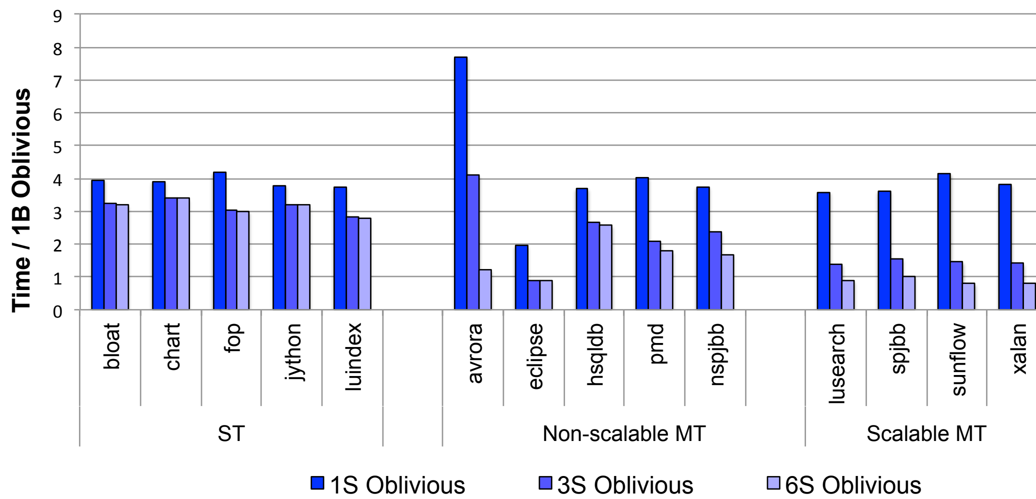
This section analyzes the strengths and weakness of two simple schedulers to motivate our approach. Each scheduler is very effective for some, but not all, workloads. We first explore scalability on small numbers of homogeneous cores by configuring a 6-core Phenom II to execute with 1, 3, and 6 cores at two speeds: big (B: 2.8 GHz) and small (S: 0.8 GHz) and execute 14 DaCapo Java benchmarks [8] with Jikes RVM and Linux kernel (3.8.0). Section 2.9 describes our methodology.

Workload characteristics Figure 2.1(a) shows workload execute time on a big homogeneous multicore configuration and Figure 2.1(b) shows a small homogeneous multicore configuration, both normalized to one big core. Lower is better. We normalize to the default Linux scheduler (Oblivious). Linux schedules threads round robin on each core, seeks to keep all cores busy, and avoids thread migration [38, 34]. It is oblivious to core capabilities, but this deficit is not exposed on these homogeneous hardware configurations.

Based on these results, we classify four of nine multithreaded benchmarks (lusearch, sunflow, spjbb and xalan) as scalable because they improve both



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



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

Figure 2.1: Linux OS scheduler (Oblivious) on homogenous configurations, normalized to one big core. We classify benchmarks as single threaded (ST), non-scalable multithreaded (MT), and scalable MT. Lower is better.

from 1 to 3 cores, and from 3 to 6 cores. Five other multithreaded benchmarks respond well to additional cores, but do not improve consistently. For instance, avrora performs worse on 3 big cores than on 1, and eclipse performs the same on 3 and 6 cores. pjbb2005 does not scale in its default configuration. We increased the size of its workload to produce a scalable variant (spjbb). The original pjbb2005 is labeled nspjbb. The number of application threads and these results yield our single threaded (ST), non-scalable multithreaded (Non-scalable MT), and scalable multithreaded (scalable MT) classifications.

Note single threaded applications in Figure 2.1 improve slightly as a function of core count. Because managed runtimes include VM helper threads, such as garbage collection, compilation, profiling, and scheduling, the VM process itself is multithreaded. Note just observing speedup as a function of cores in the OS *cannot* differentiate single-threaded from multithreaded applications in managed workloads. For example, fop and hsqldb have similar responses to the number of cores, but fop is single threaded and hsqldb is multithreaded.

AMP scheduling insights We next measure AMP performance of Oblivious and the VM-aware scheduler from Cao et al. [10] (bindVM)—it schedules VM services on small cores and application threads on big cores. Cao et al. show that bindVM improves over Oblivious on one big and five small (1B5S) and Figure 2.2 confirms this result. Regardless of the hardware configuration (1B5S, 2B4S, or 3B3S), bindVM performs best for single-threaded benchmarks 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 big cores.

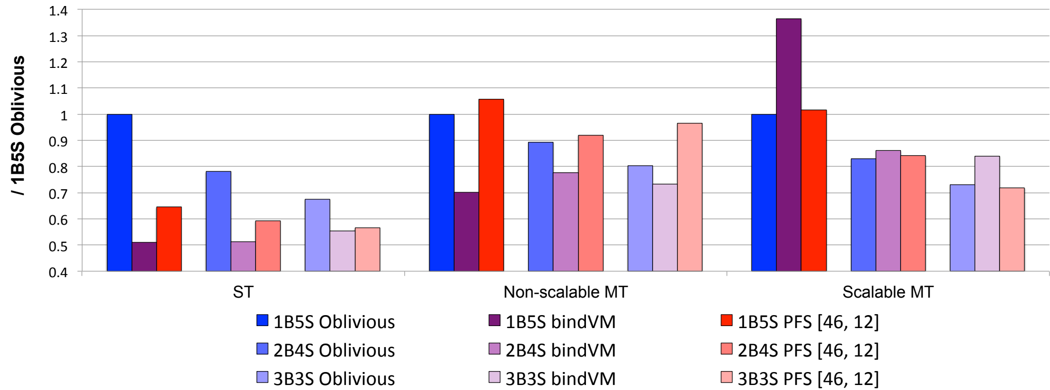


Figure 2.2: Quantitive time comparison of Proportional Fair Scheduling (PFS) [46, 12], Linux (Oblivious), and bindVM [10] on AMP configurations normalized to 1B5S Oblivious. Lower is better. No approach dominates.

On the other hand, Oblivious performs best on scalable applications and much better than bindVM on 1B5S because bindVM restricts the application to the one big core, leaving small cores underutilized. On this Phenom II, one big core and five small cores has 41.6% more instruction execution capacity than six small cores. Ideally, a scalable parallel program will see this improvement on 1B5S. For scalable benchmarks, exchanging one small core for a big core boosts performance by 33%, short of the ideal 41.6%. PFS also performs well on scalable applications, but it performs worse than bindVM on both ST and non-scalable MT because it does not prioritize application threads over VM helper threads, nor does it analyze threads that hold locks.

For non-scalable multithreaded workloads, bindVM performs best on 1B5S, but improvements on 2B4S and 3B3S are limited. Intuitively, with only one big core, binding the VM threads gives application threads more access to the big core. With more big cores, round robin does a better job of load balancing on big cores. Each scheduler performs well for some workloads, but

no scheduler is best on all workloads.

2.6 Dynamic Workload and Bottleneck Analysis

This section describes a new dynamic analysis that automatically classifies application parallelism and prioritizes bottleneck threads that hold locks. It is straightforward for the VM to count application threads separately from those it creates for GC, compilation, profiling, and other VM services. We further analyze multithreaded applications to classify them as scalable or non-scalable, exploiting the Java Language Specification and lock implementation.

To priortize bottlenecks among the threads that hold locks, we modify the VM to compute the ratio between the 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 a threshold of execution time as a function of the total available hardware resources (e.g., 1% with 2 cores, 0.5% with 4 cores, and so on), we categorize the benchmark as non-scalable. We set this threshold based on the number of cores and threads. The highest priority bottleneck thread is the lock-holding thread that is delaying the most work.

To priortize among threads that hold locks, the VM piggybacks on the lock implementation and thread scheduler. When a thread tries to acquire a lock and fails, the VM scheduler puts 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. At each scheduling quanta, the VM computes the waiting time for each thread waiting on a lock, then sums them, and then priortizes the thread(s) with the longest waiting time to big cores. The VM implements biased lock-

ing, 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 [6, 44]. Many lock implementations are similar. We place our instrumentation on this less frequent code path of a contended lock, resulting in negligible overhead.

Our critical thread analysis is more general than prior work because it automatically identifies bottlenecks in multithreaded applications with many ready threads and low priority VM threads, versus requiring new hardware [11] or developer annotations [26, 27]). Our analysis may be adopted in any system that uses biased locking or a similar optimization. Modern JVMs such as Jikes RVM and HotSpot already implement it. Although by default Windows OS and Linux do not implement biased locking, it is in principle possible. For example, Android implements biased locking in its Bionic implementation of the pthread library [6, 15, 3].

Figure 2.3 shows the results for 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. 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.

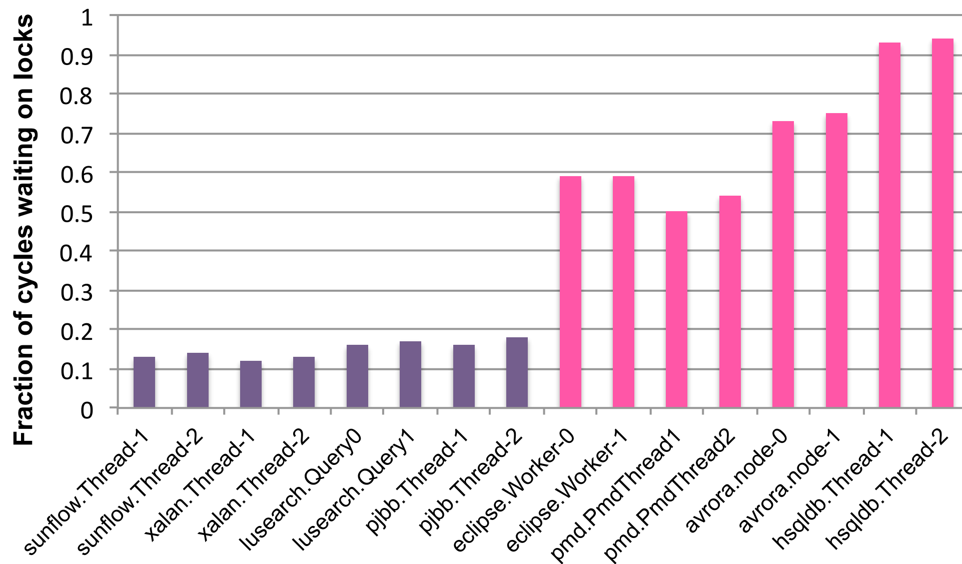
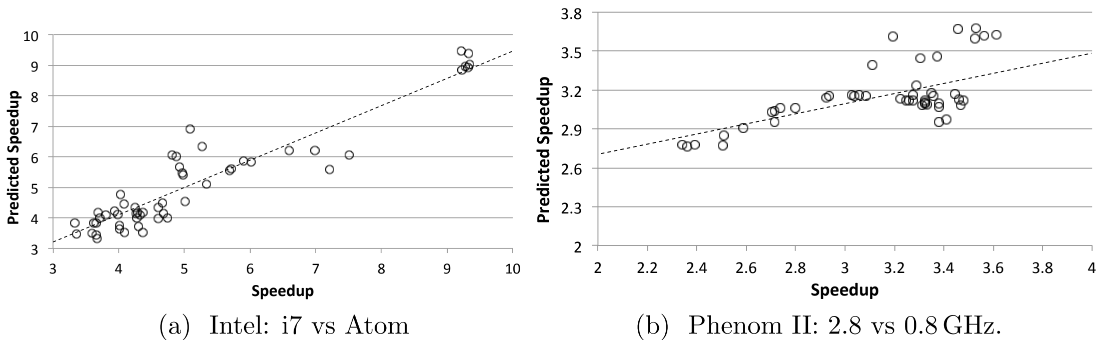


Figure 2.3: Fraction of time spent waiting on locks / cycles, per thread in multithreaded benchmarks. Left benchmarks (purple) are scalable, right (pink) are not. Low ratios are highly predictive of scalability.

Performance counters

Intel	AMD
A: INSTRUCTIONS_RETIRED	X: RETIRED_INSTRUCTIONS
B: UNHALTED_REFERENCE_CYCLES	Y: RETIRED_UOPS
C: UNHALTED_CORE_CYCLES	Z: CPU_CLK_UNHALTED
D: UOPS_RETIRED:STALL_CYCLES	W: REQUESTS_TO_L2:ALL
E: L1D_ALL_REF:ANY	
F: L2_RQSTS:REFERENCES	
G: UOPS_RETIRED:ACTIVE_CYCLES	
linear prediction model	
(-608B+609C+D+17E+27F-14G)/A	1.49+(1.87Y-1.08Z+27.08W)/X

Figure 2.4: PCA selects best performance counters to predict core sensitivity of threads with linear regression.



(a) Intel: i7 vs Atom

(b) Phenom II: 2.8 vs 0.8 GHz.

Figure 2.5: Accurate prediction of thread core sensitivity. Y-axis is predicted. X-axis is actual speedup.

2.7 Speedup and Progress Prediction

To effectively schedule AMPs, the system must consider thread sensitivity to core features. When multiple threads compete for big cores, ideally the scheduler will execute the threads on big cores that benefit the most. For example, memory bound threads will not benefit much from a higher instruction issue rate. Our methodology for modeling the core sensitivity of a thread is similar to Saez et al. [46] and Craeynest et al. [12].

Offline, we create a predictive model that we use at run time. The model takes as input performance counters while the thread is executing and predicts slow down and speedup on different core types, as appropriate. We

use linear regression and Principal Component Analysis (PCA) to learn the most significant performance monitoring events and their weights. Since each processor may use only a limited number of performance counters, PCA analysis selects the most predictive ones.

Predicting speedup from little to big when the microarchitectures differ is often not possible. For example, with a single issue little core and a multiway issue big core, if the single issue core is always stalled, it is easy to predict that the thread will not benefit from more issue slots. However, if the single issue core is operating at its peak issue rate, no performance counter on it will reveal how much potential speedup will come from multi-issue. With a frequency-scaled AMPs, our model can and does predict both speedups and slow downs because the microarchitecture does not change.

We explore the generality of our methodology and models using the frequency-scaled Phenom II and a *hypothetical* big/little design composed of an Intel Atom and i7. 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. We use PCA to compute a weight for each component (performance event) on a big core to learn the relative performance on a small core. Based on the weights, we incrementally eliminate performance events with the same weights (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, four on the AMD Phenom II and seven on the Intel Sandy Bridge.

This analysis results in the performance events and linear models listed in Figure 2.4.

Figures 2.5(a) and 2.5(b) show the results of the learned models for predicting relative performance between the Intel processors and the frequency-scaled AMD Phenom II. These resulting model predicts for each application thread running on a big core its performance on a small core (and vice versa for frequency scaling) from a model trained on all the other applications threads, using leave-one-out validation. When executing benchmarks, we use the model trained on the other benchmarks in *all* experiments. These results show that linear regression and PCA have good predictive power.

Progress monitoring 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 [5], 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 (Figure 2.5(a)). Our model predicts both slow-to-fast and fast-to-slow with frequency scaled AMD cores (Figure 2.5(b)). Thread criticality is decided based on predicted gain if it stays on or migrates to a big core. We present a sensitivity analysis to the accuracy of the model in Section 2.10.4

2.8 The WASH Scheduling Algorithm

This section describes how we use core sensitivity, thread criticality, and workload to schedule threads when application *and* runtime threads exhibit varying degrees of heterogeneity and parallelism. We implement the WASH algorithm by setting thread affinities with the standard POSIX interface, which directs the OS to bind thread execution to one or more nominated cores. The VM assesses thread progress periodically (a 40 ms quantum) by sampling per-thread performance counter data and adjusts core affinity as needed. We require no changes to the OS. Because the VM monitors the threads and adjusts the schedule accordingly, even when the OS shares VM-scheduled cores with other competing complex multiprogrammed workloads (see Section 2.10.6), the VM scheduler adapts to OS scheduling choices.

2.8.1 Overview

The scheduler starts with a default policy that assigns application threads to big cores and VM threads to small cores when they are created, following prior work [10]. 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 assesses thread sensitivity, criticality, and progress and then adjusts the affinity between threads and cores accordingly.

We add to the VM the parallelism classification and the core sensitivity models described in Sections 2.6 and 2.7. The core sensitivity model takes as input performance counters 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 dynamic waiting time to classify applications as single threaded or multithreaded scalable or multithreaded non-scalable.

The VM stores a log of the execution history for each thread using performance counters and uses it: (1) to detect resource contention among application threads by comparing expected progress of each thread on its assigned core with its actual progress; and (2) to ensure that application threads have equal access to the big cores when there exist more ready application threads than big cores.

Algorithm 1 WASH

```
1: function WASH(T_A,T_V,C_B,C_S,t)
2:   T_A: Set of application threads
3:   T_V: Set of VM services threads, where T_A ∩ T_V = ∅
4:   C_B: Set of big cores, where C_B ∩ C_S = ∅
5:   C_S: Set of small cores, where C_B ∩ C_S = ∅
6:   t: Thread to schedule, where t ∈ T_A ∪ T_V
7:   if |T_A| ≤ |C_B| then
8:     if t ∈ T_A then
9:       Set Affinity of t to C_B
10:      return
11:    else
12:      Set Affinity of t to C_B ∪ C_S
13:      return
14:  else
15:    if t ∈ T_A then
16:      if ∀τ ∈ T_A(Lock%(τ) < Lock_Thresh) then
17:        Set Affinity of t to C_B ∪ C_S
18:        return
19:      else
20:        T_Active ← {τ ∈ T_A : IsActive(τ)}
21:        T_Contd ← {τ ∈ T_A : Lock%(τ) > Lock_Thresh}
22:        if ExecRank(t, T_Active) < r_iThresh|C_B| or
23:            LockRank(t, T_Contd) < r_lThresh|C_B| then
24:          Set Affinity of t to C_B
25:          return
26:        else
27:          Set Affinity of t to C_B ∪ C_S
28:          return
29:    else
30:      Set Affinity of t to C_S
31:      return
```

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 (lines 7-9). (2) WASH classifies a workload as scalable

when no thread is a bottleneck and then ensures that all threads have equal access to the big cores (lines 16-18). This strategy often follows the default round robin OS scheduler, but is robust to changes in the OS scheduler and when other applications compete for resources (line 20-28). (3) For non-scalable workloads, the algorithm identifies application threads whose instruction retirement rates on big cores match the rate at which the big cores can retire instructions. WASH uses historical thread execution time to determine if competing applications (scheduled by the OS) are limiting the execution rate of any of the big cores (line 22). WASH also uses the accrued lock waiting time to prioritize the bottleneck threads on which other threads wait the most (line 23). WASH priortizes a thread to the big cores in both cases. VM service threads are scheduled to the small cores, unless the number of application threads is less than the number of big cores (line 29). The next 3 subsections discuss each case in more detail.

2.8.2 Single-Threaded and Low Parallelism

When the application creates a thread, WASH's fallback policy sets the thread's affinity to big cores. At each time quantum, WASH assesses the thread schedule. When WASH dynamically detects 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 $|T_A|$ of the big cores (line 7). WASH also sets the affinity for the VM threads such that they may execute on the remaining big cores or on the small cores. It sets the affinity of VM threads to all cores (line 12), 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 Cao et al. [10]. Single-threaded applications are the most common example of this scenario.

2.8.3 Scalable multithreaded

When the VM detects a scalable $|T_A| > |C_B|$ and homogenous workload, then the analysis in Section 2.5 shows that Linux's default CFS scheduler does an excellent job of scheduling application threads. We use our efficient lock contention analysis to establish scalability (line 16). We empirically established a threshold of 0.5 contention level (time spent contended / time executing) beyond which a thread is classified as contended (see Figure 2.3). WASH monitors the execution schedule from the OS scheduler and ensures that all of the homogeneous application threads make similar progress. If any thread falls behind, for example, by spending a lot of time on a small core or because the OS has other competing threads to schedule, WASH boosts its priority and binds it to a big core. It thus reprioritizes the threads based on their expected progress. Below we describe this process in more detail.

2.8.4 Non-scalable multithreaded WASH

The most challenging case is how to prioritize non-scalable application threads when the number of threads outstrips the number of cores and all threads compete for both big and small cores. Our algorithm is based on two main considerations: a) how critical the thread is to the overall progress of the application (lock information), and b) the relative capacity of big cores compared to small cores to retire instructions for each thread.

We rank threads based on their relative capacity to retire instructions, seeking to accelerate threads that dominate in terms of productive work (line

22). For each active thread (non-blocked for the last two scheduling quantum), we compute *ExecRank*: a rank based on the running total of retired instructions, corrected for core capability. If a thread runs on a big core, we accumulate its retired instructions from the dynamic performance counter. When the thread runs on a small core, we increase its retired instructions by multipling it by predicted speedup from executing on the big core. Thus we assess importance on a level playing field — judging each thread's progress as if it had access to large cores. Then, we compose this amount with the predicted speedup for all threads. 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 also compute a *LockRank*, which prioritizes bottlenecks based on the amount of time other threads have been waiting for it.

We next select a set of the highest execution-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 ($B_{RI}/(B_{RI}+S_{RI})$). 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, the size of the set would be $B_{RI}/(B_{RI}+S_{RI}) = 0.5$. In that case, we will assign to the big cores the top N most important threads such that the adjusted retired instructions rate of those N threads is 0.5 of the total in this example (line 22-23). We also select a set of the highest lock-ranked threads to execute on the big cores. We

size this set according to the fraction of cores that are big ($B/(B+S)$).

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. VM helper threads actually benefit from the big core in some cases [10] (see Section 2.10.5), but WASH ensures application threads get priority over them. Furthermore, WASH 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.

Summary The WASH VM application scheduler customizes its strategy to the workload, applying targeted heuristics (Algorithm 1), accelerating the critical path, prioritizing application threads over low-priority helper threads to fast cores, and proportionally scheduling parallelism among cores with different capabilities and effectiveness based on the thread progress.

2.9 Methodology

We report hardware power and performance measures, leveraging prior tools and methodology [10, 17, 8].

Hardware configuration methodology We measure and report performance, power, and energy on the AMD Phenom II because it supports independently clocking each core with DVFS. Prior work establishes this methodology for evaluating AMP hardware [10, 46]. Concurrently with our

	i7	**Atom**	**Phenom II**
Processor	Core i7-2600	AtomD510	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	2.8 GHz & 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 2.2: Experimental processors.

work, Qualcomm announced AMP Snapdragon hardware with 4 big and 4 little cores, but it was not available when we started our work and it uses the ARM ISA, whereas our tools target x86. Since DFVS is not as energy efficient as designing cores with different power and performance characteristics, this methodology understates the energy efficiency improvements. Compared to simulation, there are no accuracy questions, but we explore fewer hardware configurations. However, we measure an existing system orders of magnitude faster than simulation and consequently explore more software configurations.

Table 2.2 lists characteristics of the experimental machines. We only use the Sandy Bridge and Atom to show the core prediction model generalizes (see Section 2.6)—*this AMP architecture does not exist.* We use Cao et al.'s Hall effect sensor methodology to measure power and energy on the Phenom II [10]. Section 2.10.4 analyzes WASH's sensitivity to model accuracy, showing good prediction is important. All the performance, power, and energy results in this section use the Phenom II.

Operating System We perform all the experiments using Ubuntu 12.04 with the recent default 3.8.0 Linux kernel. The default Linux CFS scheduler 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 [38, 34].

Workload We use thirteen Java benchmarks taken from DaCapo: bloat, eclipse, fop, chart, jython and hsqldb (DaCapo-2006); avrora, luindex, lusearch, pmd, sunflow, and xalan (DaCapo-9.12); and from SPEC, pjbb2005 [8]. These are all the DaCapo 2009 benchmarks that executed on the base VM; the others depend on class libraries that are not supported by the GNU classpath which Jikes RVM is dependent upon. These benchmarks are non-trivial real-world open source Java programs under active development [8]. Finding that pjbb2005 does not scale well, we increased its parallelism by increasing transactions from 10,000 to 100,000, yielding spjbb, which scales on our six core machine. We use the workload classification from Section 2.5 to organize our analysis and results.

Virtual machine configuration We add our analysis and scheduler to Jikes RVM [2, 50]. The VM scheduler executes periodically. We choose a 40 ms time quantum following prior work [11], which shows no discernible migration overhead on shared-LLC AMP processors with 25 ms. All measurements follow Blackburn et al.'s best practices for Java performance analysis with the following modifications of note [8]. We measure first iteration, since we explore scheduling JIT threads. We use concurrent Mark-Sweep collection, and collect every 8 MB of allocation for avrora, fop and luindex, which have the

highest rates of allocation, and 128 MB for the others. We configure the number of collection threads to be the same as available cores. We use default JIT settings in Jikes RVM, which intermixes compilation with execution. Jikes RVM does not interpret: a baseline compiler JITs code upon first execution and then the compiler optimizes at higher levels when its cost model predicts the optimized code will amortize compilation costs [5].

Measurement Methodology We execute each configuration 20 or more times, reporting first iteration results, which mix compilation and execution. We omit confidence intervals from graphs. For the WASH scheduling algorithm, the largest 95% confidence interval for time measurements with 20 invocations is 5.22% and the average is 1.7%. For bindVM, the 95% confidence interval for time measurements with 20 invocations is 1.64% and the average is 0.72%. Oblivious has the largest 95% confidence interval; with 20 invocations, it is 15% and the average is 5.4%. Thus, we run the benchmarks with Oblivious for 60 invocations, lowering average error to 3.7%. The 95% confidence intervals are a good indicator of performance predictability of the algorithms.

Comparisons We compare WASH to two baselines: the default OS scheduler (Oblivious) with no VM direction and bindVM [10], which simply binds VM helper threads to the small cores using the OS set affinity interface. We use the unmodified bindVM implementation from the Jikes VM repository. These schedulers are the only ones that handle general workloads automatically, e.g., messy non-scalable MT workloads with a mix of application and VM threads. Furthermore, they require no programmer intervention to identify bottleneck locks and/or no new hardware. Section 2.10.5 shows that an approximation

of the closest prior work [46, 12] performs much worse than WASH, Linux, and bindVM because it does not consider thread priorities (VM versus application threads) nor prioritize threads that create bottlenecks by holding locks, contributions of our work.

2.10 Results

Figure 2.6 summarizes the performance, power, and energy results on three AMD hardware configurations: 1B5S (1 Big core and 5 Small cores), 2B4S and 3B3S. We weigh each benchmark group (single threaded, scalable, and non-scalable) equally. We normalize to Oblivious, lower is better.

Figure 2.6 shows that WASH improves performance and energy on all three hardware configurations on average. Oblivious has the worst average time on all of the configurations. Even though it has the lowest power cost, up to 16% 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.

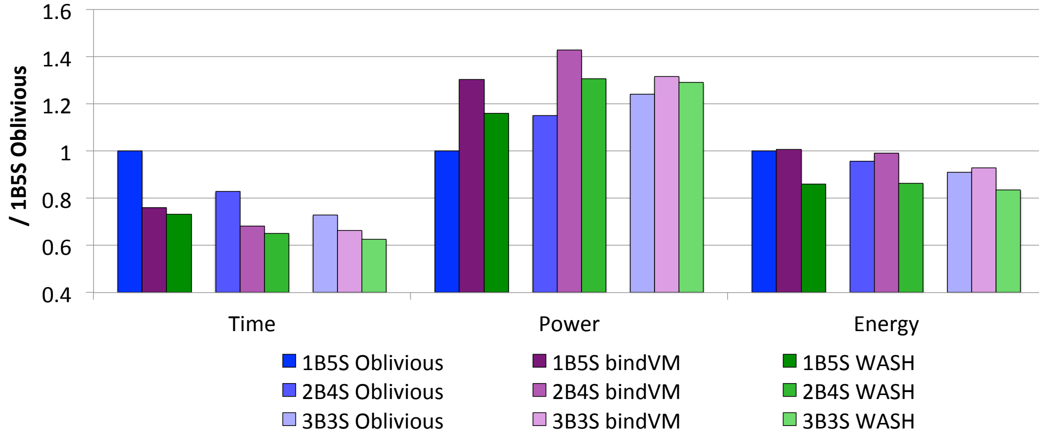


Figure 2.6: Geomean time, power and energy with Oblivious, bindVM, and WASH on all three hardware configs.

WASH attains its performance improvement by using more power than Oblivious, but at less additional power than bindVM. The bindVM scheduler has lower average time compared to Oblivious, but it has the worst energy and power cost in all hardware settings, especially on 2B4S. bindVM uses up to 18% more energy than WASH. The bindVM scheduler overloads big cores with work that can be more efficiently performed by small cores, leading to higher power and underutilization of small cores.

WASH and bindVM are especially effective compared to Oblivious on 1B5S, where the importance of correct scheduling decisions is most exposed. On 1B5S, WASH reduces the geomean time by 27% compared to Oblivious, and by about 5% comparing to bindVM. For energy, WASH saves more than 14% compared to bindVM and Oblivious. WASH consistently improves over bindVM on power. In the following subsections, we structure detailed analysis based on workload categories.

2.10.1 Single-Threaded Benchmarks

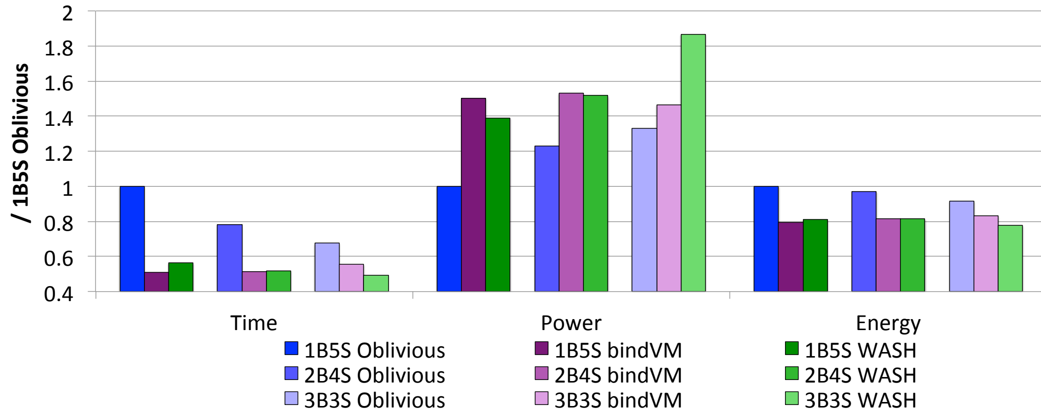


Figure 2.7: Single-threaded benchmarks. Normalized geomean time, power and energy for different benchmark groups. Lower is better.

Figure 2.7 shows that for single-threaded benchmarks, WASH performs very well in terms of total time and energy on all hardware settings, while Oblivious performs poorly. WASH consumes the least energy on all hardware settings. Compared to Oblivious scheduling, WASH reduces execution time by as much as 44%. WASH lowers energy by 19% but increases power by as much as 39% compared to Oblivious. Figures 2.13 show results for individual benchmarks in the 1B5S hardware scenario. 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 GC threads to the small cores. When the number of big cores increases, as in Figure 2.16 and Figure 2.19, 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. Note that power consumption is higher for bindVM and WASH than for Oblivious. When the application thread migrates to the small cores, it consumes less power compared to bindVM and WASH, but the loss in performance more than outweighs the decrease in power. Thus total energy is reduced by WASH. In the single-threaded scenario, WASH occasionally adds some overhead over bindVM for its analysis, but in general,WASH and bindVM perform very similar to each other on all AMP configurations.

2.10.2 Scalable Multithreaded Benchmarks

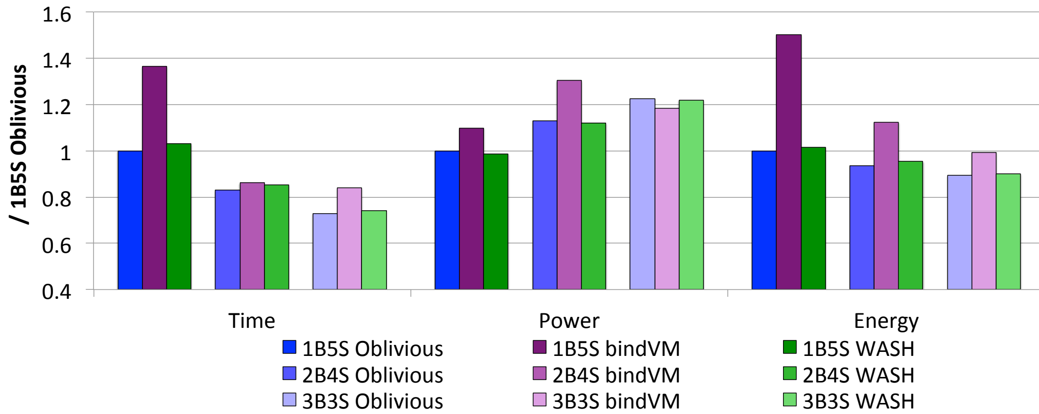


Figure 2.8: Scalable multithreaded. Normalized geomean time, power and energy for different benchmark groups. Lower is better.

Figure 2.8 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. Compared to WASH and Oblivious scheduling, bindVM increases time by as much as 36%, increases energy by as much as 50%, and power by as much as 15%. By using the contention information we gather online, WASH detects scalable benchmarks. Since WASH

correctly identifies the workload, WASH and Oblivious generate similar results, although WASH sometimes adds overhead due to its dynamic workload and lock analysis. The reason bindVM performs poorly 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, these benchmarks all have more threads than cores and thus Oblivious works well because it migrates threads among all the cores frequently enough. However, WASH reasons explicitly about relative core speeds using historical execution data to migrate threads fairly between slow and fast cores. Figures 2.13, 2.16, and 2.19 show that as the number of large cores increases, the difference between bindVM and Oblivious reduces.

2.10.3 Non-scalable Multithreaded Benchmarks

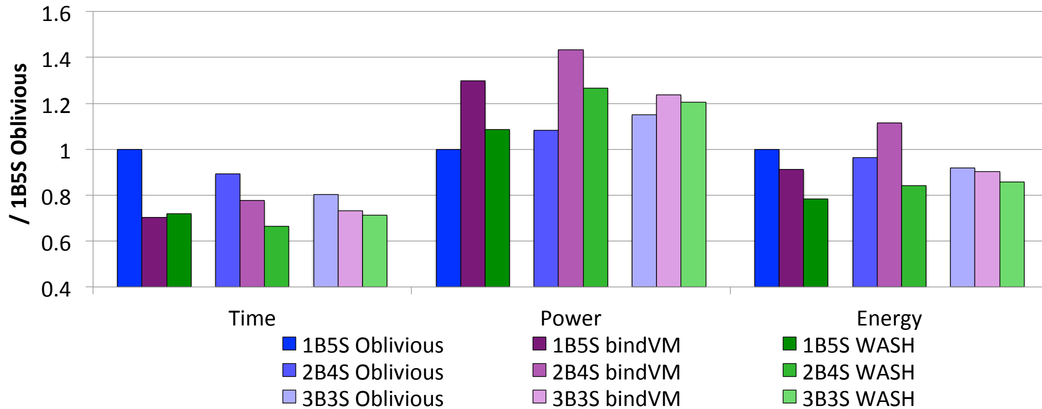


Figure 2.9: Non-scalable multithreaded. Normalized geomean time, power and energy for different benchmark groups. Lower is better.

Figure 2.9 shows that WASH on average 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 consumes 56% of all of the threads' cycles while the three *Worker* threads consume just 1%. In 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, Figures 2.13, 2.16, and 2.19 show that the results for WASH and bindVM are 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 the profile 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 and these benchmarks mostly use just one application thread, the results for the two policies are similar.

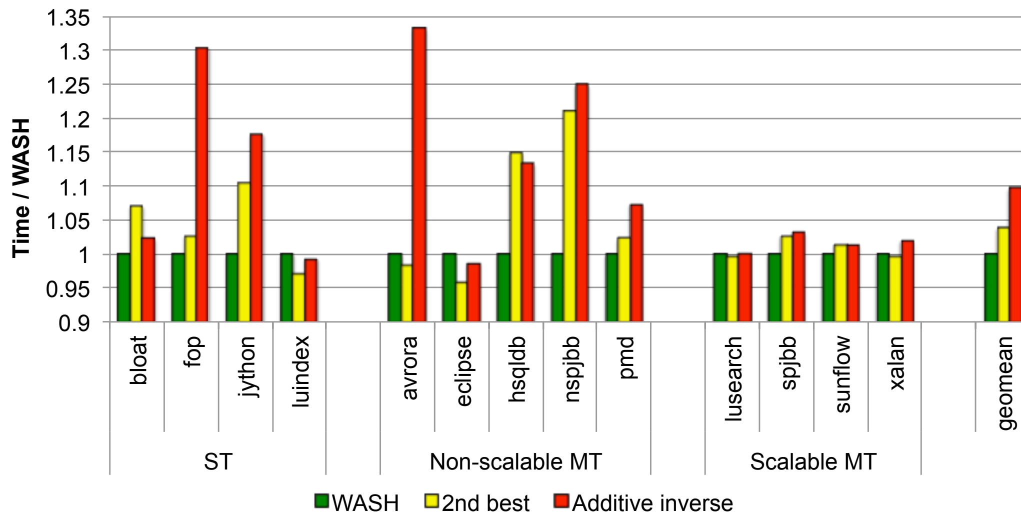


Figure 2.10: WASH with best (default) model, second best model, and a bad model (inverse weights) on 1B5S.

On the other hand, avrora, nspjbb, and pmd are more complex. They benefit from WASH's ability to priortize among the application threads and choose to execute some threads on the slow cores, because they do not require a fast core. Particularly, for 1B5S, compared to WASH, bindVM time is lower; however, because of WASH makes better use of the small cores for some application threads, it delivers performance with much less power. Since bindVM does not reason about core sensitivity, it does not make a performance/power trade-off. WASH makes better use of small cores, improving energy efficiency for avrora, nspjbb, and pmd.

2.10.4 Sensitivity to speedup model

Figure 2.10 shows the sensitivity of WASH to its speedup model with a) the default best model from Section 2.7, b) a model using the next best

4 different hardware performance counters from the same training set, and c) a model with the additive inverse weights of the default one. WASH only degrades by 2-3% when we change the model to slightly worse one. However, a bad model for the speedup prediction, degrades performance by 9%.

2.10.5 Thread & Bottleneck Prioritization

For completeness, this section compares directly to our implementation of the Proportional Fair Scheduling algorithm proposed in prior work [46, 12]. We simply remove the features that are unique to WASH: a priori knowledge to preferentially schedule application threads to big cores and automatically prioritizing among contended locks. The resulting scheduler performs proportional fair scheduling for homogeneous workloads on AMP and prioritizes threads based on core sensitivity. Figure 2.11 shows that as expected, these schedulers (red) perform well on scalable benchmarks, consistent with prior results [46, 12] that test on workloads with total |threads| = |cores| and with parallelism that, for the most part, is homogeneous and scalable. However, the prior work performs poorly on sequential and non-scalable parallel benchmarks. The orange bars show just disabling prioritization of application over VM threads, revealing that this feature is critical to good performance for single-threaded and non-scalable workloads. We find that the JIT thread benefits from big cores more than most application threads, so core sensitivity scheduling alone will mistakenly schedule it on the big core even though it is often not critical to application performance (as revealed in the single-threaded results). WASH correctly deprioritizes JIT threads to small cores. Highly concurrent messy workloads, such as avrora and eclipse, suffer greatly when the scheduler does not prioritize contended locks. Avrora uses threads

as an abstraction to model high degrees of concurrency for architectural simulation. Eclipse is a widely used Integrated Development Environment (IDE) with many threads that communicate asynchronously. Both are important examples of concurrency not explored, and consequently not addressed, by the prior work. WASH's comprehensive approach correctly prioritizes bottleneck threads in these programs, handling more general and challenging settings.

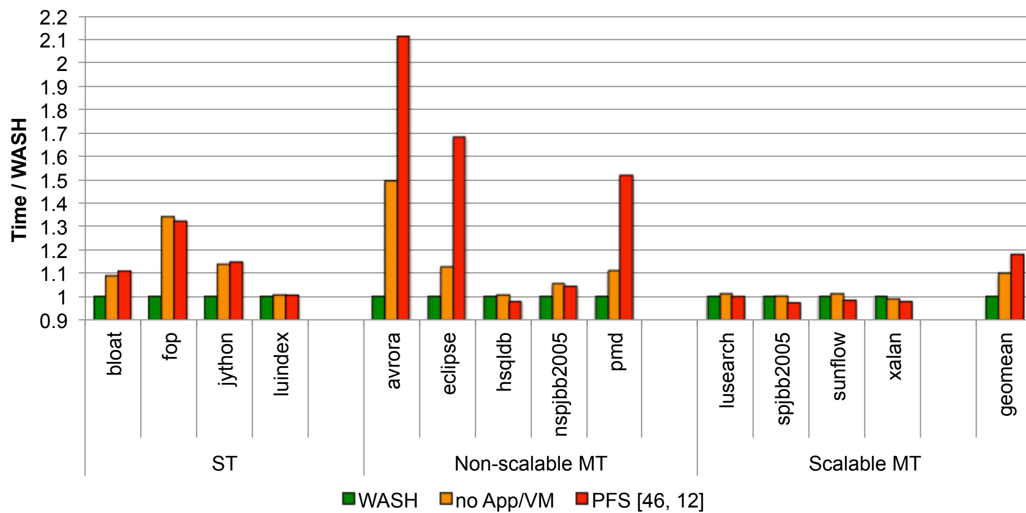


Figure 2.11: WASH (green) out-performs our implementation of PFS [46, 12] (red) which lacks lock analysis and VM vs application priority on 1B5S.

2.10.6 Multiprogrammed workloads

This section presents an experiment with multiprogrammed workload in the system, of which our VM is unaware. We use eclipse as the OS scheduled workload across all cores and WASH scheduling on each of our benchmarks. Eclipse has twenty threads with diverse needs and is demanding both computationally and in terms of memory consumption.

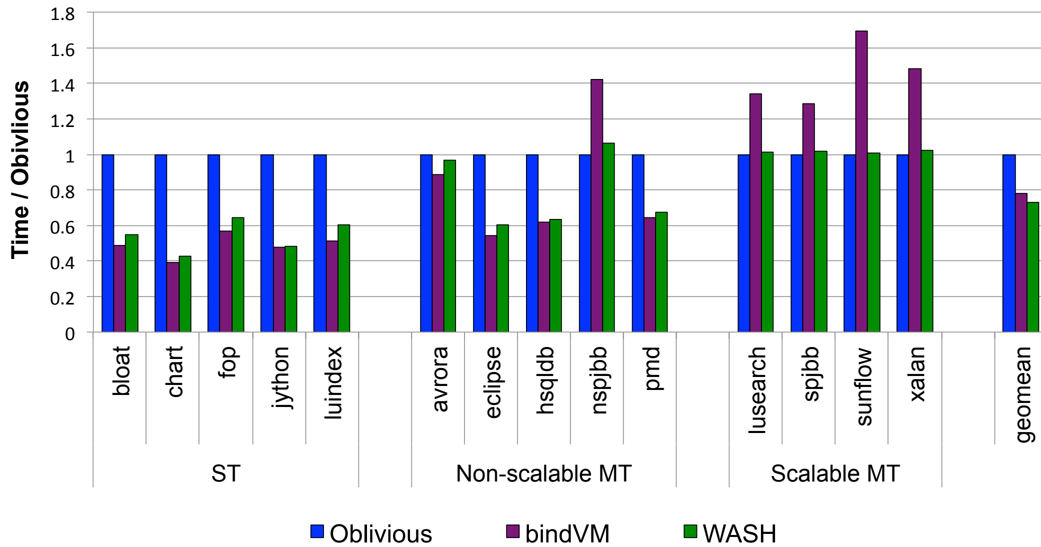


Figure 2.12: Performance with eclipse adversary on 1B5S.

Figure 2.12 shows the performance of WASH and bindVM compared to Oblivious in the presence of the eclipse adversary. Although bindVM's overall performance is largely unchanged compared to execution with no adversary, it degrades on both nspjbb and spjbb. WASH performs very well in the face of the adversary, with average performance 27% better than Oblivious and a number of benchmarks performing twice as well as Oblivious. WASH's worst case result (nspjbb) is only 7% worse than Oblivious.

Summary The results show that WASH improves performance and energy on average over all workloads, each component of its algorithm is necessary and effective, and it is robust to the introduction of a non-trivial adversary. WASH utilizes both small and big cores as appropriate to improve performance at higher power compared to Oblivious, which under-utilizes the big cores and over-

utilizes little cores because it does not reason about them. On the other hand, WASH, uses its workload and core sensitivity analysis to improve performance and lower power compared to bindVM which under utilizes the little cores for the scalable and non-scalable multithreaded benchmarks.

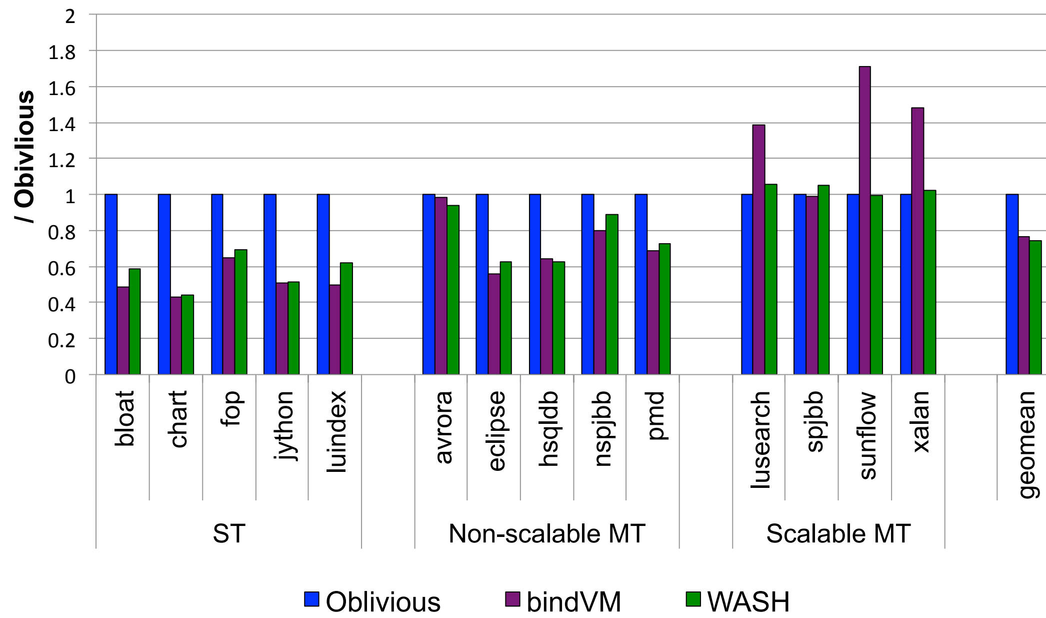


Figure 2.13: Running time on 1B5S

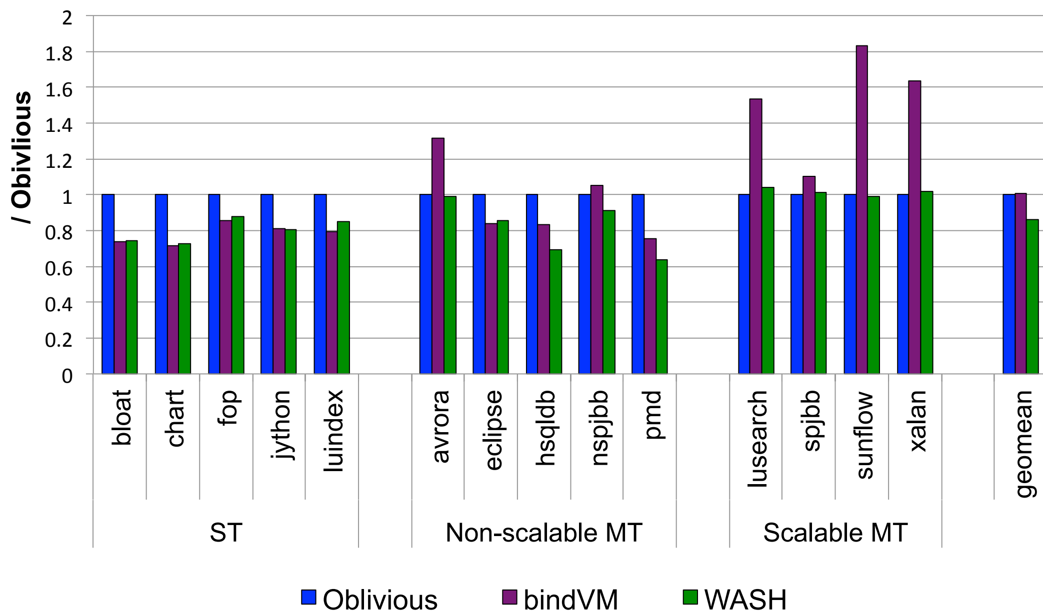


Figure 2.14: Energy on 1B5S

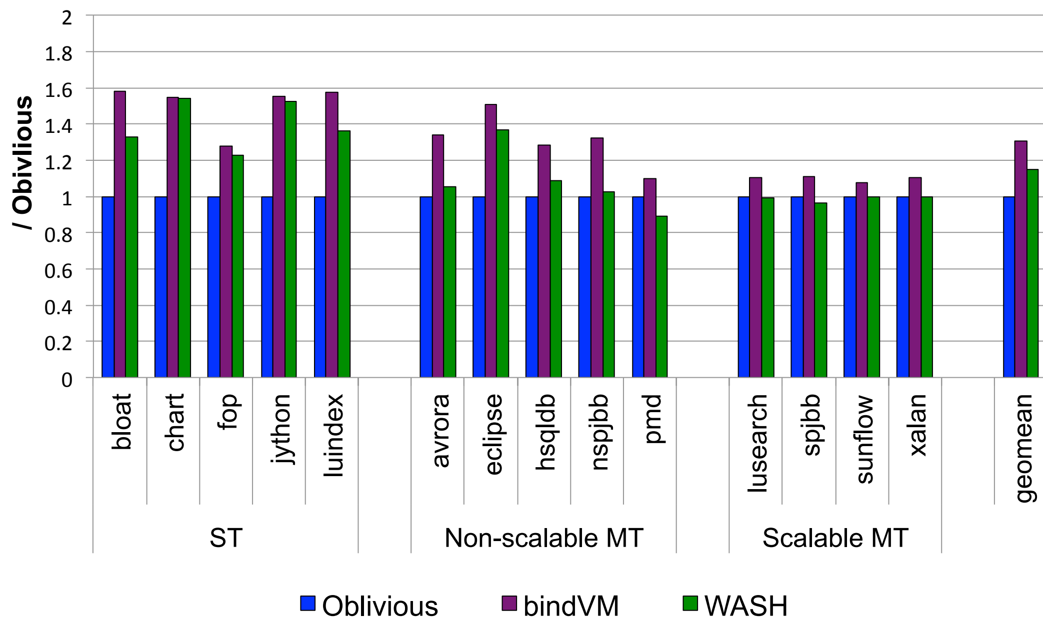


Figure 2.15: Power on 1B5S

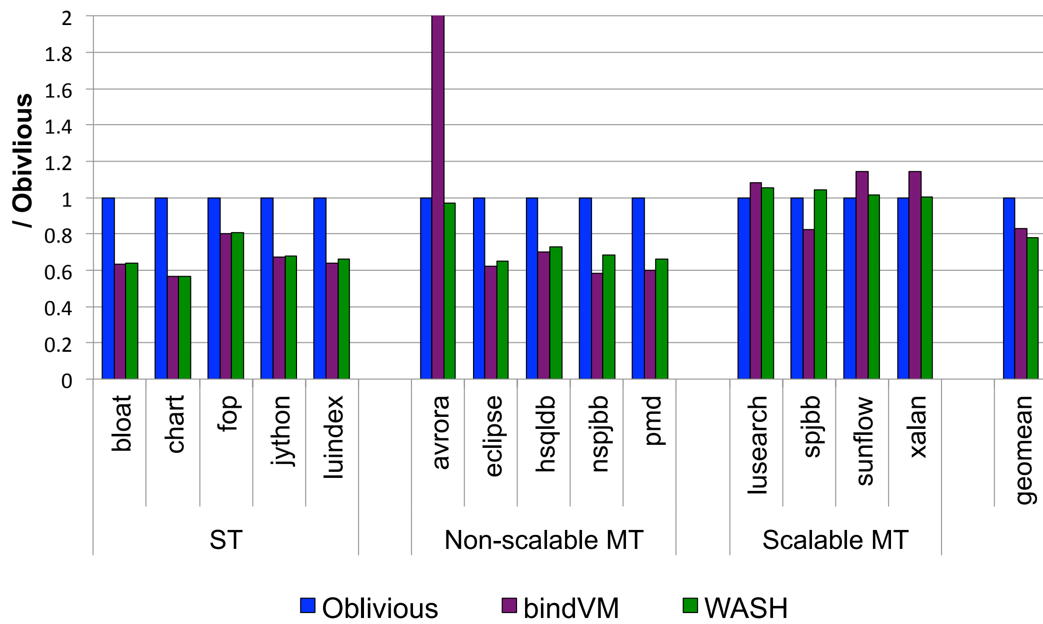


Figure 2.16: Running time on 2B4S

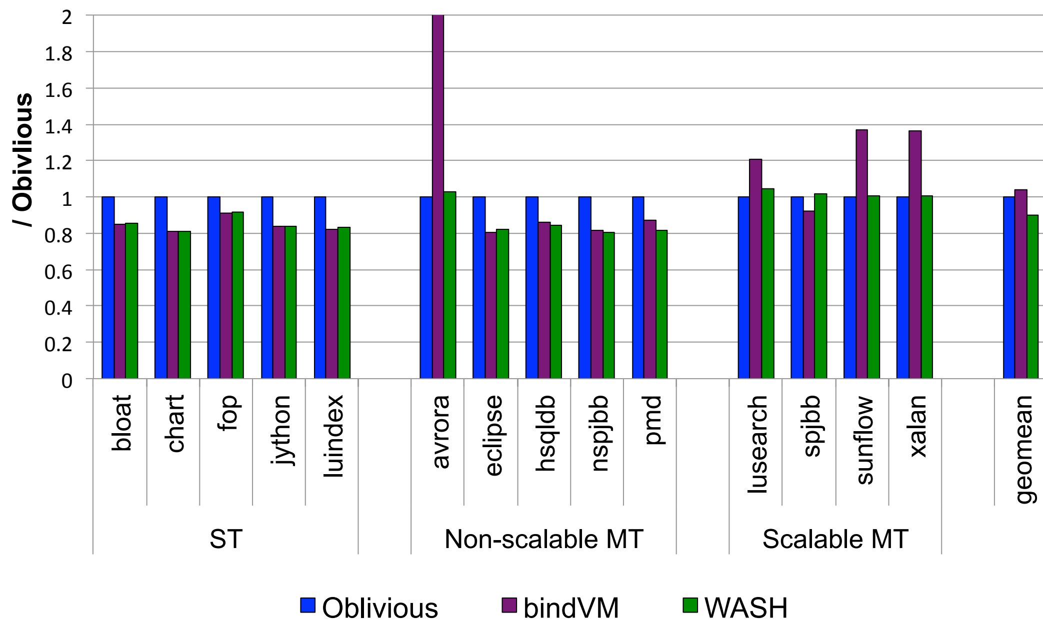


Figure 2.17: Energy on 2B4S

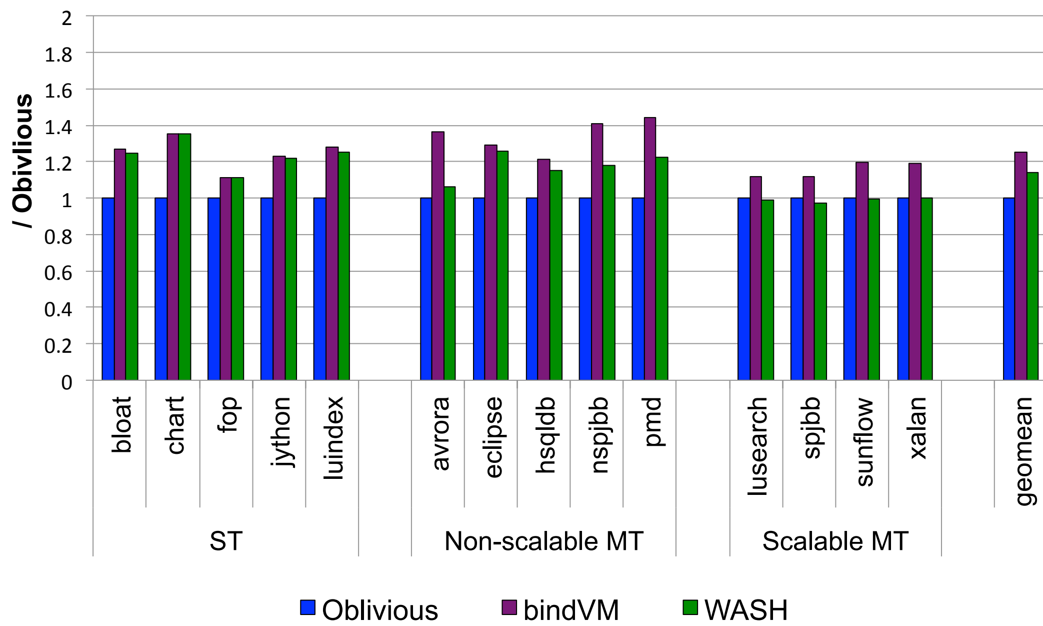


Figure 2.18: Power on 2B4S

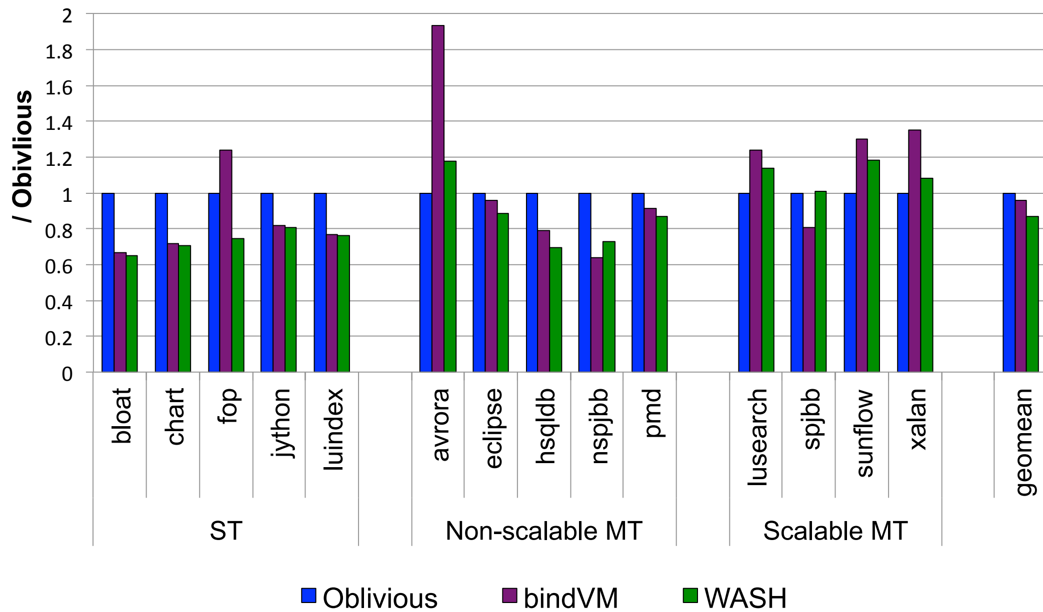


Figure 2.19: Running time on 3B3S

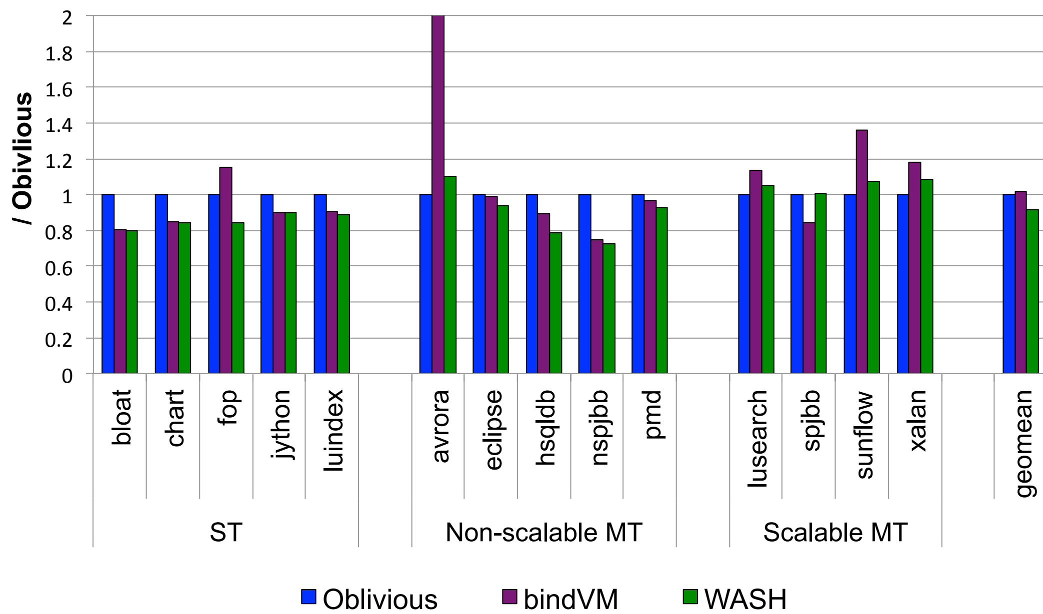


Figure 2.20: Energy on 3B3S

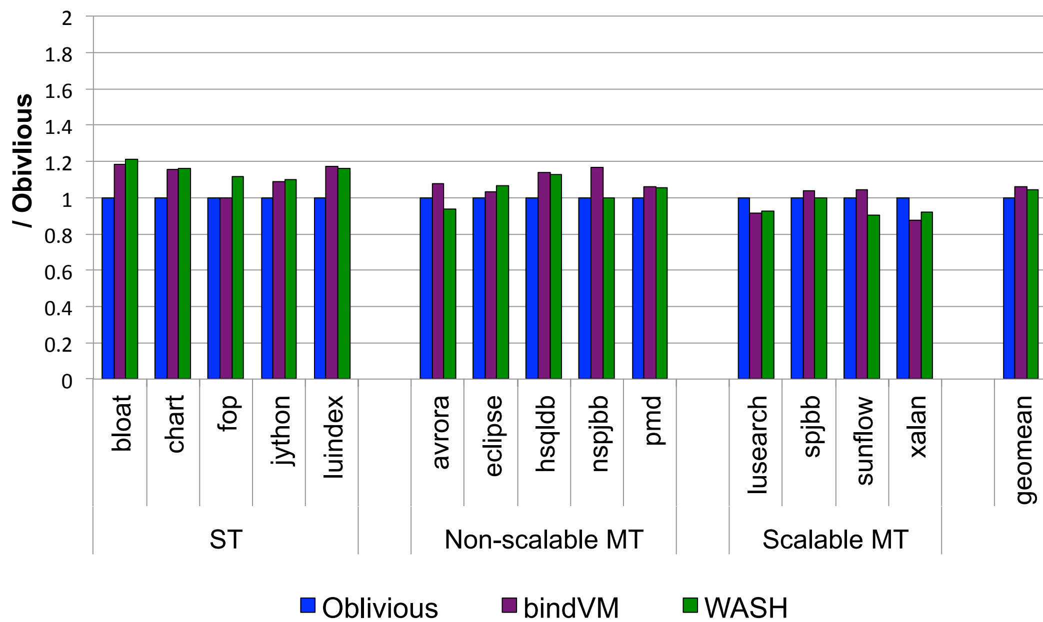


Figure 2.21: Power on 3B3S

2.11 Conclusion

A promising approach to improving performance and energy in power constrained devices is heterogeneity. The more transparent hardware complexity is to applications, the more applications are likely to benefit from it. Furthermore, a layer of abstraction that protects developers from this complexity will enhance portability and innovation over generations of heterogeneous hardware. This chapter shows how to modify a VM to monitor bottlenecks, AMP sensitivity, and progress to deliver transparent, portable performance, and energy efficiency to managed applications on AMP. In particular, we introduce new fully automatic dynamic analyses that identify and monitor scalable parallelism, non-scalable parallelism, bottleneck threads, and thread progress. Our dynamic software analysis automatically identifies and prioritizes bottleneck threads that hold locks, whereas prior work requires developer annotations on locks and/or new hardware. This analysis is useful for performance debugging as well. We show that this system delivers substantial improvements in performance and energy efficiency on frequency-scaled processors over prior software approaches. These results likely understate the potential advantages from more highly optimized commercial AMP systems that vendors are beginning to deliver.

Chapter 3

Vector Parallelism in JavaScript

This chapter presents the design and implementation of SIMD language extensions and compiler support for JavaScript:

1. a language design justified on the basis of portability and performance;
2. compiler type speculation without profiling in a dynamic language; and
3. the first dynamic language with SIMD instructions that deliver their performance and energy benefits.

The result is the first high level language design and implementation that delivers direct access to SIMD performance in an architecture-independent manner.

3.1 Motivation

Increasingly more computing is performed in web browsers. Since JavaScript is the dominant web language, sophisticated and demanding applications, such as games, multimedia, finance, and cryptography, are increasingly implemented in JavaScript. Many of these applications benefit from hardware parallelism, both at a coarse and fine grain. Because of the complexities and potential for concurrency errors in coarse grain (task level) parallel

programing, JavaScript has limited its parallelism to asynchronous activities that do not communicate through shared memory [51]. However, fine-grain vector parallel instructions — Single-Instruction, Multiple-Data (SIMD) — do not manifest these correctness issues and yet they still offer significant performance advantages by exploiting parallelism.

SIMD instructions are now standard on modern ARM and x86 hardware from mobile to servers because they are both high performance and energy efficient. Intel announced that all future machines will contain SIMD instructions. SIMD standards include the SSE4 x86 SIMD and the NEON ARM SIMD, and are already widely implemented in modern x86 processors such as Intel Sandybridge, Intel IvyBridge, and AMD K10, and in ARM processors such as the popular ARM Cortex-A8 and Cortex-A9. Both standards implement 128 bits and x86 processors now include larger widths in the AVX instruction set. However, ARM NEON's largest width is 128 bits, with no apparent plans to grow. These instruction set architectures include vector parallelism because it is very effective at improving performance and energy efficiency in many application domains, for example, image, audio, and video processing, perceptual computing, physics engines, fluid dynamics, rendering, finance, and cryptography. Such applications also increasingly dominate client-side and server-side web applications. Exploiting vector parallelism in JavaScript should therefore improve performance and energy efficiency on mobile, desktop, and server, as well as hybrid HTML5 mobile JavaScript applications.

3.2 Related Work

Westinghouse was the first to investigate vector parallelism in the early 1960s, envisioning a co-processor for mathematics, but cancelled the effort. The principal investigator, Daniel Slotnick, then left and joined University of Illinois, where he lead the design of the ILLIAC IV, the first supercomputer and vector machine [9]. In 1972, it was 13 times faster than any other machine at 250 MFLOPS and cost $31 million to build. CRAY Research went on to build commercial vector machines [13] and researchers at Illinois, CRAY, IBM, and Rice University pioneered compiler technologies that correctly transformed scalar programs into vector form to exploit vector parallelism.

Today, Intel, AMD, and ARM processors for servers, desktop, and mobile offer coarse-grain multicore and fine-grain vector parallelism with Single Instruction Multiple Data (SIMD) instructions. For instance, Intel introduced MMX instructions in 1997, the original SSE (Streaming SIMD Extensions) instructions in 1999, and its latest extension, SSE4, in 2007 [23, 24]. All of the latest AMD and Intel machines implement SSE4.2. ARM implements vector parallelism with its NEON SIMD instructions, which are optional in Cortex-A9 processors, but standard in all Cortex-A8 processors.

The biggest difference between vector instruction sets in x86 and ARM is vector length. The AVX-512 instruction set in Intel processors defines vector lengths up to 512 bits. However, NEON defines vector lengths from 64-bit up to 128-bit. To be compatible with both x86 and ARM vector architectures and thus attain vector-performance portability, we choose one fixed-size 128-bit vector length, since it is the largest size that both platforms support. Choosing a larger or variable size than all platforms support is problematic when executing on machines that only implement a shorter vector size because

some long SIMD instructions can only be correctly implemented with scalar instructions on shorter vector machines. See our discussion below for additional details. By choosing the largest size all platforms support, we avoid exposing developers to unpleasant and hard to debug performance degradations on vector hardware. We choose a fixed size that all architectures support to deliver performance portability on all vector hardware.

Future compiler analysis could generate code for wider vector instructions to further improve performance, although the dynamically typed JavaScript setting makes this task more challenging than, for example, in Fortran compilers. We avoided a design choice that would require significant compiler support because of the diversity in JavaScript compiler targets, from embedded devices to servers. Our choice of a fixed-size vector simplifies the programming interface, compiler implementation, and guarantees vector performance on vector hardware. Supporting larger vector sizes could be done in a library with machine-specific hooks to attain machine-specific benefits on streaming SIMD operations, but applications would suffer machine-specific performance degradations for non-streaming operations, such as shuffle, because the compiler must generate scalar code when an architecture supports only the smaller vector sizes.

Both SSE4 and NEON define a plethora of SIMD instructions, many more than we currently propose to include in JavaScript. We choose a subset for simplicity, selecting operations based on an examination of demanding JavaScript applications, such as games. Most of our proposed JavaScript SIMD operations map directly to a hardware SIMD instruction. We include a few operations, such as swizzle and shuffle, that are not directly implemented in both architectures, but are important for the JavaScript applications we

studied. For these operations, the compiler generates two or three SIMD instructions, rather than just one. The current set is easily extended.

Intel and ARM provide header files which define SIMD intrinsics for use in C/C++ programs (`xmmintrin.h` and `arm_neon.h`, respectively). These intrinsics directly map to each SIMD instruction in the hardware, thus there are currently over 400 intrinsic functions [24, 4]. These platform-specific intrinsics result in architecture-dependent code, thus using either one directly in JavaScript is not desirable nor an option for portable JavaScript code.

Managed languages, such as Java and C#, historically only provide access to SIMD instructions through their native interfaces, JNI (Java Native Interface) and C library in the case of Java and C# respectively, which use the SIMD intrinsics. However, recently Microsoft and the C# Mono project announced a preliminary API for SIMD programming for .NET [36, 39]. This API is currently only limited to streaming operations (e.g., arithmetic) and it introduces hardware-dependent size vectors. In C#, the application can query the hardware to learn the maximum vector length. This API results in hardware-dependent types embedded in the application which will not allow its expansion to non-streaming operations (e.g., shuffle) in a portable fashion. The lack of non-streaming operations will limit performance in very common algorithms (e.g., matrix operations) and is thus not in line with our performance goals for JavaScript.

Until now, dynamic scripting languages, such as PHP, Python, Ruby, and JavaScript, have not included SIMD support in their language specification. We analyzed the application space and chose the operations based on their popularity in the applications and their portability across the SSE3, SSE4.2, AVX, and NEON SIMD instruction sets. We observed a few addi-

tional SIMD patterns that we standardize as methods, which the JIT compiler translates into multiple SIMD instructions.

3.3 Our Contribution

Design This chapter presents the design, implementation, and evaluation of SIMD language extensions for JavaScript. We have two design goals for these extensions. (1) Portable vector performance on vector hardware. (2) A compiler implementation that does not require automatic vectorization technology to attain vector performance. The first goal helps developers improve the performance of their applications without unpleasant and unexplainable performance surprises on different vector hardware. The second goal simplifies the job of realizing vector performance in existing and new JavaScript Virtual Machines (VMs) and compilers. Adding dependence testing and loop transformation vectorizing technology is possible, but our design and implementation do not require it to deliver vector performance.

This chapter defines SIMD language extensions and new compiler support for JavaScript. The language extensions consist of fixed-size immutable vectors and vector operators, which correspond to hardware instructions and vector sizes common to ARM and x86. The largest size common to both is 128 bits. Although an API with variable or larger sizes (e.g., Intel's AVX 512-bit vectors) may seem appealing, correctly generating code that targets shorter vector instructions from longer ones violates our vector performance portability design goal. For example, correctly shortening non-streaming vector instructions, such as shuffle/swizzle, requires generating scalar code that reads all values out and then stores them back, resulting in scalar performance instead of vector performance on vector hardware.

We define new SIMD JavaScript data types (e.g., Int32x4, Float32x4), constructors, lane accessors, operators (arithmetic, bitwise operations, comparisons, and swizzle/shuffle), and typed array accessors and mutators for these types. To ease the compiler implementation, most of these SIMD operations correspond directly to SIMD instructions common to the SSE4 and NEON extensions. We choose a subset that improve a wide selection of sophisticated JavaScript applications, but this set could be expanded in the future. This JavaScript language specification was developed in collaboration with Google for the Dart programming language, reported in a workshop paper [35]. The Dart and JavaScript SIMD language specifications are similar in spirit. The language extensions we present are in the final stages of approval by the ECMAScript standardization committee (Ecma TC39) [49].

Type Speculation We introduce *type speculation*, a modest twist on type inference and specialization for implementing these SIMD language extensions. For every method containing SIMD operations, the Virtual Machine's Just-in-Time (JIT) compiler immediately produces SIMD instructions for those operations. The JIT compiler speculates that every high-level SIMD instruction operates on the specified SIMD type. It translates the code into an intermediate form, optimizes it, and generates SIMD assembly. In most cases, it produces optimal code with one or two SIMD assembly instructions for each high-level SIMD JavaScript instruction. The code includes guards that check for non-conforming types that reverts to unoptimized code when needed. In contrast, modern JIT compilers for dynamic languages typically perform some mix of *type inference*, which uses static analysis to *prove* type values and eliminate any dynamic checks, and *type feedback*, which *observes* common types

over multiple executions of a method and then optimizes for these cases, generating guards for non-conforming types [1, 28]. Our JIT compilers instead will use the assumed types of the high-level SIMD instructions as hints and generate code accordingly.

Implementation We implement and evaluate this compiler support in two JavaScript Virtual Machines (V8 and SpiderMonkey) and generate JIT-optimized SIMD instructions for x86 and ARM. Initially, V8 uses a simple JIT compiler (full codegen) to directly emit executable code [19], whereas SpiderMonkey uses an interpreter [41, 18]. Both will detect hot sections and later JIT compilation stages will perform additional optimizations. We add JIT type speculation and SIMD optimizations to both Virtual Machines (VMs). Our JIT compiler implementations include type speculation, SIMD method inlining, SIMD type unboxing, and SIMD code generation to directly invoke the SIMD assembly instructions. When the target architecture does not contain SIMD instructions or the dynamic type changes from the SIMD class to some other type, SpiderMonkey currently falls back on interpretation and V8 generates deoptimized (boxed) code.

Benchmarks For any new language features, we must create benchmarks for evaluation. We create microbenchmarks by extracting ten kernels from common application domains. These kernels are hot code in these algorithms that benefit from vector parallelism. In addition, we report results for one application, skinning, a key graphics algorithm that associates the skin over the skeleton of characters from a very popular game engine. We measure the benchmarks on five different Intel CPUs (ranging from an Atom to an i7), and

four operating systems (Windows, Unix, OS X, Android). The results show that SIMD instructions improve performance by a factor of 3.4× on average and improve energy by 2.9× on average. SIMD achieves super-linear speed ups in some benchmarks because the vector versions of the code eliminate intermediate operations, values, and copies. On the skinning graphics kernel, we obtain a speedup of 1.8×.

Artifact The implementations described in this chapter are in Mozilla Firefox Nightly builds and in submission to Chromium as of December 2015. This work shows that V8 and SpiderMonkey can support SIMD language extensions without performing sophisticated dependence testing or other parallelism analysis or transformations, i.e., they do not *require* automatic vectorization compiler technology. However, our choice does not *preclude* such sophisticated compiler support, or preprocessor/developer-side vectorization support in tools such as Emscripten [40], or higher level software abstractions that target larger or variable size vectors, as applicable, to further improve performance. By adding portable SIMD language features to JavaScript, developers can exploit vector parallelism to make demanding applications accessible from the browser. We expect that these substantial performance and energy benefits will inspire a next generation of JIT compiler technology to further exploit vector parallelism.

Contributions This chapter presents the design and implementation of SIMD language extensions and compiler support for JavaScript. No other high level language has provided direct access to SIMD performance in an architecture-independent manner. The contributions of this chapter are as follows:

1. a language design justified on the basis of portability and performance;
2. compiler type speculation without profiling in a dynamic language; and
3. the first dynamic language with SIMD instructions that deliver their performance and energy benefits.

3.4 Design Rationale

Our design is based on fixed-width 128-bit vectors. A number of considerations influenced this decision, including the programmer and the compiler writer.

A fixed vector width offers simplicity in the form of consistent performance and consistent semantics across vector architectures. For example, the number of times a loop iterates is not affected by a change in the underlying hardware. A variable-width vector or a vector width larger than the hardware supports places significant requirements on the JIT compiler. Given the variety of JavaScript JIT VMs and the diverse platforms they target, requiring support for variable-width vectors was considered unviable. Additionally, variable width vectors cannot efficiently implement some important algorithms (e.g., matrix multiplication, matrix inversion, vector transform). On the other hand, developers are free to add more aggressive JIT compiler functionality that exploits wider vectors if the hardware provides them. Another consideration is that JavaScript is heavily used as a compiler target. For example, Emscripten compiles from C/C++ [25], and compatibility with `mmintrin.h` offered by our fixed width vectors is a bonus.

Finally, given the decision to support fixed width vectors, we selected 128 bits because it is the widest vector supported by all major architectures

today. Not all instructions can be decomposed to run on a narrower vector instruction. For example, non-streaming operations, such as the shuffle instruction, in general cannot utilize the SIMD hardware at all when the hardware is narrower than the software. For this reason, we chose the largest common denominator. Furthermore, 128 bits is a good match to many important algorithms, such as single-precision transformations over homogeneous coordinates in computer graphics (XYZW) and algorithms that manipulate the RGBA color space.

3.5 Language Specification

This section presents the SIMD data types, operations, and JavaScript code samples. The SIMD language extensions give direct control to the programmer and require very simple compiler support, but still guarantees vector performance when the hardware supports SIMD instructions. Consequently, most of the JavaScript SIMD operations have a one-to-one mapping to common hardware SIMD instructions. This section includes code samples for the most common data types. The full specification is available on line [49].

3.5.1 Data Types

We add the following new fixed-width 128-bit numeric value types to JavaScript.

Float32x4 Vector with four 32-bit single-precision floats

Int32x4 Vector with four 32-bit signed integers

Int16x8 Vector with 8 16-bit signed integers

Int8x16 Vector with 16 8-bit signed integers

Uint32x4 Vector with 4 32-bit unsigned integers

Uint16x8 Vector with 8 16-bit unsigned integers

Uint8x16 Vector with 16 8-bit unsigned integers

Bool32x4 Vector with 4 boolean values

Bool16x8 Vector with 8 boolean values

Bool8x16 Vector with 16 boolean values

Figure 3.1 shows the simple SIMD type hierarchy. The SIMD types has four to sixteen *lanes*, which correspond to degrees of SIMD parallelism. Each element of a SIMD vector is a lane. Indices are required to access the lanes of vectors. For instance, the following code declares and initializes a SIMD single-precision float and assigns `3.0` to `a`.

```
var v1 = SIMD.Float32x4 (1.0, 2.0, 3.0, 4.0);
var a  = SIMD.Float32x4.extractLane(v1,3);
```

3.5.2 Operations

Constructors The type defines the following constructors for all of the SIMD types. The default constructor initializes each of the two or four lanes to the specified values, the `splat` constructor creates a constant-initialized SIMD vector, as follows.

```
var c = SIMD.Float32x4(1.1, 2.2, 3.3, 4.4);
// Float32x4(1.1, 2.2, 3.3, 4.4)
var b = SIMD.Float32x4.splat(5.0);
// Float32x4(5.0, 5.0, 5.0, 5.0)
```

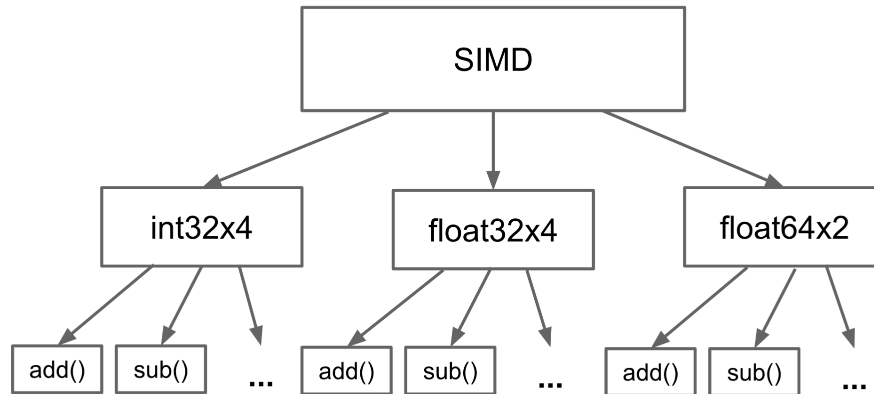


Figure 3.1: SIMD Type Hierarchy

Accessors and Mutators The proposed SIMD standard provides operations for accessing and mutating SIMD values, and for creating new SIMD values from variations on existing values.

extractLane Access one of the lanes of a SIMD value.

replaceLane Create a new instance with the value change for the specified lane.

select Create a new instance with selected lanes from two SIMD values.

swizzle Create a new instance from another SIMD value, shuffling lanes.

shuffle Create a new instance by selecting lane values from a specified mix of lane values from the two input operands.

These operations are straightforward and below we show a few examples.

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = a.x; // 1.0
```

```
var c = SIMD.Float32x4.replaceLane(1, 5.0);
// Float32x4(5.0, 2.0, 3.0, 4.0)
var d = SIMD.Float32x4.swizzle(a, 3, 2, 1, 0);
// Float32x4(4.0, 3.0, 2.0, 1.0)
var f = SIMD.Float32x4(5.0, 6.0, 7.0, 8.0);
var g = SIMD.Float32x4.shuffle(a, f, 1, 0, 6, 7);
// Float32x4(2.0, 1.0, 7.0, 8.0)
```

Arithmetic The language extension supports the following thirteen arithmetic operations over SIMD values: **add, sub, mul, div, abs, max, min, sqrt, reciprocalApproximation, reciprocalSqrtApproximation, neg, clamp, scale, minNum, maxNum**

We show a few examples below.

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.Float32x4(4.0, 8.0, 12.0, 16.0);
var c = SIMD.Float32x4.add(a,b);
// Float32x4(5.0, 10.0, 15.0, 20.0)
var e = SIMD.reciprocalSqrtApproximation(d);
// Float32x4(0.5, 0.5, 0.5, 0.5);
var f = SIMD.scale(a, 2);
// Float32x4(2.0, 4.0, 6.0, 8.0);
var lower = SIMD.Float32x4(-2.0, 5.0, 1.0, -4.0);
var upper = SIMD.Float32x4(-1.0, 10.0, 8.0, 4.0);
var g = SIMD.Float32x4.clamp(a, lower, upper);
// Float32x4(-1.0, 5.0, 3.0, 4.0)
```

Bitwise Operators The language supports the following four SIMD bitwise operators: **and, or, xor, not**

Bit Shifts We define the following logical and arithmetic shift operations and then show some examples: **shiftLeftByScalar, shiftRightLogicalByScalar,**

shiftRightArithmeticByScalar

```
var a = SIMD.Int32x4(6, 8, 16, 1);
var b = SIMD.Int32x4.shiftLeftByScalar(a,1);
// Int32x4(12, 16, 32, 2)
var c = SIMD.Int32x4.shiftRightLogicalByScalar(a, 1);
// Int32x4(3, 4, 8, 0)
```

Comparison We define three SIMD comparison operators that yield SIMD Boolean vectors, e.g., Bool32x4: **equal, notEqual, greaterThan, lessThan, lessThanOrEqual, greaterThanOrEqual**

```
var a = SIMD.Float32x4(1.0, 2.0, 3.0, 4.0);
var b = SIMD.Float32x4(0.0, 3.0, 5.0, 2.0);
var gT = SIMD.Float32x4.greaterThan(a, b);
// Float32x4(0xF, 0x0, 0x0, 0xF);
```

Type Conversion We define type conversion from floating point to integer and bit-wise conversion (i.e., producing an integer value from the floating point bit representation): **fromInt32x4, fromFloat32x4, fromInt32x4Bits, fromFloat32x4Bits**

```
var a = SIMD.Float32x4(1.1, 2.2, 3.3, 4.4)
var b = SIMD.Int32x4.fromFloat32x4(a)
// Int32x4(1, 2, 3, 4)
var c = SIMD.Int32x4.fromFloat32x4Bits(a)
// Int32x4(1066192077, 1074580685, 1079194419, 1082969293)
```

Arrays We introduce load and store operations for JavaScript typed arrays for each base SIMD data type that operates with the expected semantics. An example is to pass in a `Uint8Array` regardless of SIMD type, which is useful because it allows the compiler to eliminate the shift in going from the index

to the pointer offset. The extracted SIMD type is determined by the type of the load operation.

```
var a   = new Float32Array(100);
for (var i = 0, l = a.length; ++i) {
  a[i] = i;
}
for (var j = 0; j < a.length; j += 4) {
  sum4 = SIMD.Float32x4.add(sum4,
              SIMD.Float32x4.load(a, j));
}
var result = SIMD.Float32x4.extractLane(sum4, 0) +
             SIMD.Float32x4.extractLane(sum4, 1) +
             SIMD.Float32x4.extractLane(sum4, 2) +
             SIMD.Float32x4.extractLane(sum4, 3);
```

Figure 3.2 depicts how summing in parallel reduces the number of sum instructions by a factor of the width of the SIMD vector, in this case four, plus the instructions needed to sum the resulting vector. Given a sufficiently long array and appropriate JIT compiler technology, the SIMD version reduces the number of loads and stores by about 75%. This reduction in instructions has the potential to improve performance significantly in many applications.

3.6 Compiler Implementations

We add compiler optimizations for SIMD instructions to Firefox's SpiderMonkey VM [41, 18] and Chromium's V8 VM [19]. We first briefly describe both VM implementations and then describe our type speculation, followed by unboxing, inlining, and code generation that produce SIMD assembly instructions.

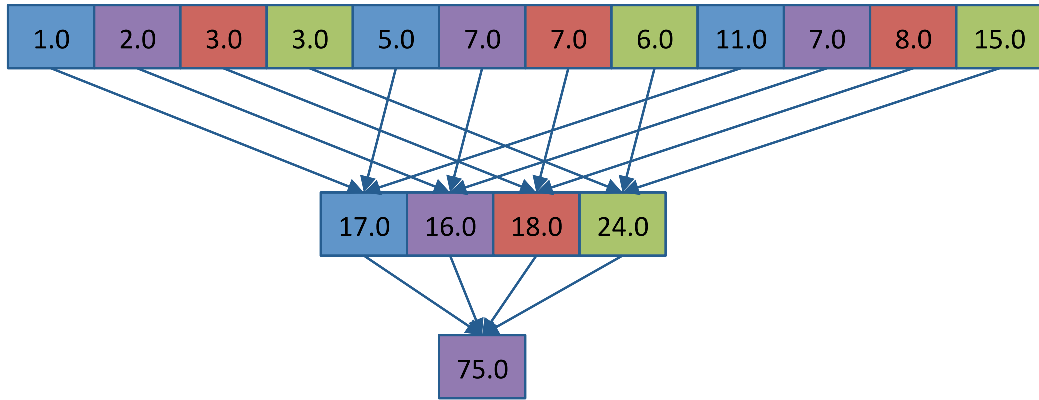


Figure 3.2: Visualization of `averagef32x4` summing in parallel.

SpiderMonkey We modified the open-source SpiderMonkey VM, used by the Mozilla Firefox browser. SpiderMonkey contains an interpreter that executes unoptimized JavaScript bytecodes and a Baseline compiler that generates machine code. The interpreter collects execution profiles and type information [41]. Frequently executed JS functions, as determined by the profiles collected by the interpreter, are compiled into executable instructions by the Baseline compiler. The Baseline compiler mostly generates calls into the runtime and relies on inline caches and hidden classes for operations such as property access, function calls, array indexing operations, etc. The Baseline compiler also inserts code to collect execution profiles for function invocations and to collect type information. If a function is found to be hot, the second compiler is invoked. This second compiler, called IonMonkey, is an SSA-based optimizing compiler, which uses the type information collected from the inline cache mechanism to inline the expected operations, thereby avoiding the call overhead into the runtime. IonMonkey then emits fast native code translations of JavaScript. We added SIMD support in the runtime. Instead of waiting for the runtime to determine whether the method is hot, we perform type

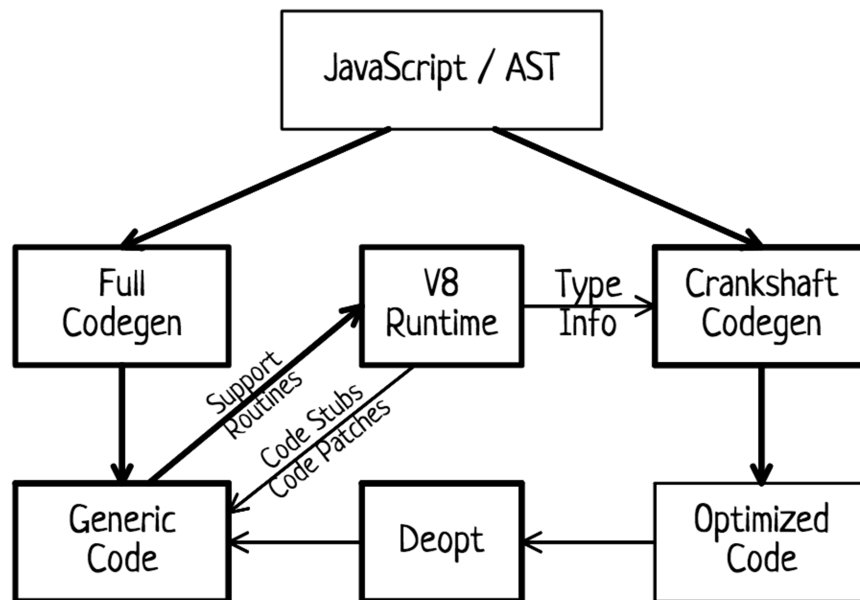


Figure 3.3: V8 Engine Architecture.

speculation and optimizations for all methods that contain SIMD operations. We modify the interpreter, the Baseline compiler code generator, and the IonMonkey JIT compiler. We add inlining of SIMD operations to the IonMonkey compiler. We modify the register allocator and bailout mechanism to support 128-bit values.

V8 We also modified the open-source V8 VM, used by the Google Chromium browser. Figure 3.3 shows the V8 Engine Architecture. V8 does not interpret. It translates the JavaScript AST (abstract syntax tree) into executable instructions and calls into the runtime when code is first executed, using the Full Codegen compiler. Figure 3.4 shows an example of the non-optimized code from this compiler. This compiler also inserts profiling and calls to runtime support routines. When V8 detects a hot method, the Crankshaft compiler translates into an SSA form. It uses the frequency and type profile information to perform global optimizations across basic blocks such as type specialization, inlining, global value numbering, code hoisting, and register allocation. We modify the runtime to perform type speculation when it detects methods with SIMD instructions, which invokes the SIMD compiler support. We added SIMD support to both the Full Codegen compiler and the Crankshaft compiler. For the Full Codegen compiler, the SIMD support is provided via calls to runtime functions implemented in C++, as depicted in Figure 3.4. We modified the Crankshaft compiler, adding support for inlining, SIMD register allocation, and code generation which produces optimal SIMD code sequence and vector performance in many cases.

Both compilers use frequency and type profiling to inline and then perform

type specialization on other types, other optimizations, register allocation, and finally generate code.

3.6.1 Type Speculation

Optimizing dynamic languages requires type specialization [32, 22], which emits code for the common type. This code must include an initial type tests and a branch to deoptimized generic code or jumps back to the interpreter or deoptimized code when the types do not match [29]. Both optimizing compilers perform similar forms of type specialization. In some cases, the compiler can use type inference to prove that the type will never change and can eliminate this fallback [1, 14]. For example, unboxing an object and its fields generates code that operates directly on floats, integers, or doubles, rather than generating code that looks up the type of every field on each access, loads the value from the heap, operates on them, and then stores them back into the heap. While a value is unboxed, the compiler assigns them to registers and local variables, rather than emitting code that operates on them in the heap, to improve performance.

The particular context of a SIMD library allows us to be more agressive than typical type specialization. We *speculate* types based on a number of simple assumptions. We consider it a design bug on the part of the programmer to override methods of the SIMD API, and thus we produce code speculatively for the common case of SIMD methods operating on SIMD types specified by our language extension. If programs override the SIMD methods, type guards that the compiler inserts in the program correctly detects this case, but performance suffers significantly. Likewise, we speculate that SIMD API arguments are of the correct type and optimize accordingly. If they are not,

the compiler correctly detects this case, but performance will suffer.

These assumptions are predicated on the fact that the SIMD instructions have well established types and semantics, and that developers who use the API are expected to write their code accordingly. Because we expect developers to use SIMD instructions in performance-sensitive settings, we have the opportunity to aggressively optimize methods that contain them more eagerly, rather than waiting for these methods to become hot. We expect the net result to be a performance win in a dynamic optimization setting. Each of these simple optimizations is a modest twist on conventional JIT optimization of dynamic code that we tailor for the performance critical SIMD setting.

Inlining To achieve high performance with the proposed SIMD language extension, we modified the optimizing compilers to always replace method calls to SIMD operations on SIMD types with inlined lower level instructions (IR or machine-level) that operate on unboxed values. The compilers thus eliminates the need for unboxing by keeping the values in registers. The compiler identifies all the SIMD methods and inlines them. These methods are always invoked on the same SIMD type and with the same parameters with the appropriate SIMD or other type. Thus, inlining is performed when the system observes the dynamic types of each SIMD method call and predicts they are monomorphic.

Value Boxing and Unboxing Both baseline compilers will box SIMD objects, arguments, and return values like any regular JavaScript object. Both of the JavaScript VM JIT compilers optimize by converting boxed types to unboxed values [32]. As discussed above, boxed values are allocated on the heap,

Original JavaScript Code

```
sum = SIMD.float32x4.add(sum, SIMD.Float32x4.load(a, j));
```

V8 Full Codegen code (not optimized)

```
3DB5BCBE   222  ff3424        push [esp]
3DB5BCC1   225  89442404      mov [esp+0x4],eax
3DB5BCC5   229  ff75e8        push [ebp-0x18]
3DB5BCC8   232  ba02000000    mov edx,00000002
3DB5BCCD   237  8b7c2408      mov edi,[esp+0x8]
;; call .getAt()
3DB5BCD1   241  e84a24fcff    call 3DB1E120
3DB5BCD6   246  8b75fc        mov esi,[ebp-0x4]
3DB5BCD9   249  890424        mov [esp],eax
3DB5BCDC   252  ba04000000    mov edx,00000004
3DB5BCE1   257  8b7c240c      mov edi,[esp+0xc]
;; call .add()
3DB5BCE5   261  e876fdffff    call 3DB5BA60
3DB5BCEA   266  8b75fc        mov esi,[ebp-0x4]
3DB5BCED   269  83c404        add esp,0x4
3DB5BCF0   272  8945ec        mov [ebp-0x14],eax
```

V8 CrankShaft Code (optimized)

```
3DB5E306   358  0f101cc6      movups xmm3,[esi+eax*8]
3DB5E30A   362  0f58d3        addps xmm2,xmm3
```

Figure 3.4: Example V8 compiler generated code

garbage collected, and must be loaded and stored to the heap on each use and definition, respectively. To improve performance, the optimizing compilers put unboxed values in registers, operate on them directly, and then stores modified values back to the heap as necessary for correctness and deoptimization paths. We modified this mechanism in both compilers to operate over SIMD methods.

Example Consider again the `averagef32x4()` method from Section 3.5. In V8, the Full Codegen compiler generates the code in Figure 3.4, which is a straight forward sequence of calls into the V8 runtime. The parameters reside in heap objects. Note below that the parameters reside in heap objects and pointers to those heap objects are passed on the stack. The two calls invoke the .add() operator and the .getAt operator, respectively. The runtime has its own established calling convention using registers. However, all user visible values are passed on the stack.

The V8 Crankshaft compiler generates an SSA IR, directly represents SIMD values in registers, and uses SIMD instructions directly instead of runtime calls. The final code produced by the V8 Crankshaft optimizing compiler; after inlining, unboxing, and register allocation is the optimal sequence of just two instructions, as illustrated at the bottom of Figure 3.4. SpiderMonkey generates the same code.

The code has just two instructions; one for fetching the value out of the array and one for adding the two float32x4 values. The compiler puts `sum` variable in the xmm2 register for the entire loop execution!

3.7 Methodology

This section describes our hardware and software, measurements, and workload configurations. All of our code, workloads, and performance measurement methodologies are publicly available.

3.7.1 Virtual Machines and Measurements

We use the M37, branch 3.27.34.6, version of V8 and version JavaScript-C34.0a1 of SpiderMonkey for these experiments.

3.7.2 Hardware Platforms

We measure the performance and energy of SIMD language extension on multiple different architectures and operating system combinations. Table 3.1 lists characteristics of our experimental platforms. We report results on x86 hardware. Among the hardware systems we measure are an in-order Atom processor, a recent 22 nm low-power dual-core Haswell processor (i5-4200U) and a high-performance quad-core counterpart (i7-4770K).

Processor	Architecture	Frequency	Operating System
i5-4200U	Haswell	1.60 GHz	Ubuntu Linux 12.04.4x64
i7-3720QM	Sandy Bridge	2.60 GHz	Mac OS X 10.9.4
i5-2520M	Sandy Bridge	2.50 GHz	Windows 7 64-bit
i7-4770K	Haswell	3.50 GHz	Ubuntu Linux 12.04.4
Atom Z2580	Cloverview	2.00 GHz	Android 4.2.2

Table 3.1: Experimental platforms

3.7.3 Benchmarks

To evaluate our language extensions, we developed a set of benchmarks from various popular JavaScript application domains including 3D graphic code, cryptography, arithmetic, higher order mathematical operations, and visualization. Table 3.2 lists their names and the number of lines of code in the original, the SIMD version, and the number of SIMD operations. For this initial SIMD benchmark suite, we select benchmarks that reflect operations typical of SIMD programming in other languages such as C++, and that are sufficiently self-contained to allow JavaScript VM implementers to use them as a guide for testing the correctness and performance of their system. These benchmarks are now publicly available.

Although JavaScript only supports double-precision numeric types, we take advantage of recent optimizations in JavaScript JIT compilers that optimize the scalar code to use single-precision instructions when using variables that are obtained from Float32x4Arrays. All of our scalar benchmarks perform float32 operations (single precision). The scalar codes thus have the advantage of single precision data sizes and optimizations, which makes the comparison to their vector counterparts an apples-to-apples comparison, where the only change is the addition of SIMD vector instructions.

3D graphics As noted in Section 3.4, float32x4 operations are particularly useful for graphics code. Because most of the compute intensive work on the CPU side (versus the GPU) involves computing projection and views of matrices that feed into WebGL, we collected the most common 4x4 matrix operations and a vertex transformation of a 4 element vector for four of our benchmarks:

MatrixMultiplication 4x4 Matrix Multiplication

Transpose4x4 4x4 Matrix Transpose

Matrix4x4Inverse 4x4 Matrix Inversion

VertexTransform 4 element vector transform

Cryptography While cryptography is not a common domain for SIMD, we find that the hottest function in Rijnadel cipher should benefit from SIMD instructions. We extracted this function into the following kernel.

ShiftRows Rotation of row values in 4x4 matrix

Higher Level Math Operations Mathematical operations such as trigonometric functions, logarithm, exponential, and power, typically involve complicated use of SIMD instructions. We hand-coded a representative implementation of the sinx4() function. We believe such operations will become important in emerging JavaScript applications that implement physics engines and shading. For example, the AOBench shading (Ambient Occlusion benchmark) benefits from 4-wide cosine and sine functions.

Sine Four element vector sine function.

Math, Mandelbrot, and more graphics In addition, we modified the following JavaScript codes to use SIMD optimizations.

Average32x4 Basic math arithmetic (addition) on arrays of float32 items.

Benchmark	LOC Scalar	LOC SIMD	SIMD calls
Transpose4x4	17	26	8
Matrix4x4Inverse	83	122	86
VertexTransform	26	12	13
MatrixMultiplication	54	41	45
ShiftRows	12	18	3
AverageFloat32x4	9	9	2
Sinex4†	14	5	1
Mandelbrot	25	36	13
Aobench	120	201	119
Skinning	77	90	66

Table 3.2: Benchmark characteristics. We measure the lines of code (LOC) for the kernel of each benchmark in both scalar and SIMD variants. †In the case of Sinex4, the table reports the LOC for the simple sine kernel, which makes calls to the sine function in the Math libary and the equivalent SIMD implementation respectively. The full first-principles implementation of SIMD sine takes 113 LOC and makes 74 SIMD calls.

Mandelbrot Visualization of the calculation of the Mandelbrot set. It has a static number of iterations per pixel.

AOBench Ambient Occlusion Renderer. Calculates how exposed each point in a scene is to ambient lighting.

Skinning Graphics kernel from a game engine to attach a renderable skin to an underlying articulated skeleton.

Developers are porting many other domains to JavaScript and they are likely to benefit from SIMD operations, for example, physics engines; 2D graphics, e.g., filters and rendering; computational fluid dynamics; audio and video processing; and finance, e.g., Black-Scholes.

3.7.4 Measurement Methodology

We first measure each non-SIMD version of each benchmark and configure the number of iterations such that it executes for about 1 second in steady state. This step ensures the code is hot and the JIT compilers will be invoked on it. We measure the SIMD and non-SIMD benchmark configurations executing multiple iterations 10 times. We invoke each JavaScript VM on the benchmark using their command line JavaScript shells. Our benchmark harness wraps each benchmark, measuring the time and energy using performance counters. This methodology results in statistically significant results comparing SIMD to non-SIMD results.

Time We measure the execution time using the low overhead real time clock. We perform twenty measurements, interleaving SIMD and scalar systems, and report the mean.

Energy We use the Running Average Limit Power (RAPL) Machine Specific Registers (MSRs) [43] to obtain the energy measurements for the JavaScript Virtual Machine running the benchmark. We perform event-based sampling through CPU performance counters. We sample PACKAGE_ENERGY_STATUS which is the energy consumed by the entire package, which for the single-die packages we use, means the entire die. Two platforms, the Android Atom and the Windows Sandy Bridge, do not support RAPL and thus we report energy results only for the other three systems.

The performance measurement overheads are very low (less than 2% for both time and energy). We execute both version of the benchmarks using the above iteration counts.

3.8 Results

This section reports our evaluation of the impact of SIMD extensions on time and energy.

3.8.1 Time

Figure 3.5 shows the speedup due to the SIMD extensions. The graphs show scalar time divided by SIMD time, so any value higher than one reflects a speedup due to SIMD. All benchmarks show substantial speedups and unsurprisingly, the micro benchmarks (left five) see greater improvement than the more complex kernels (right five).

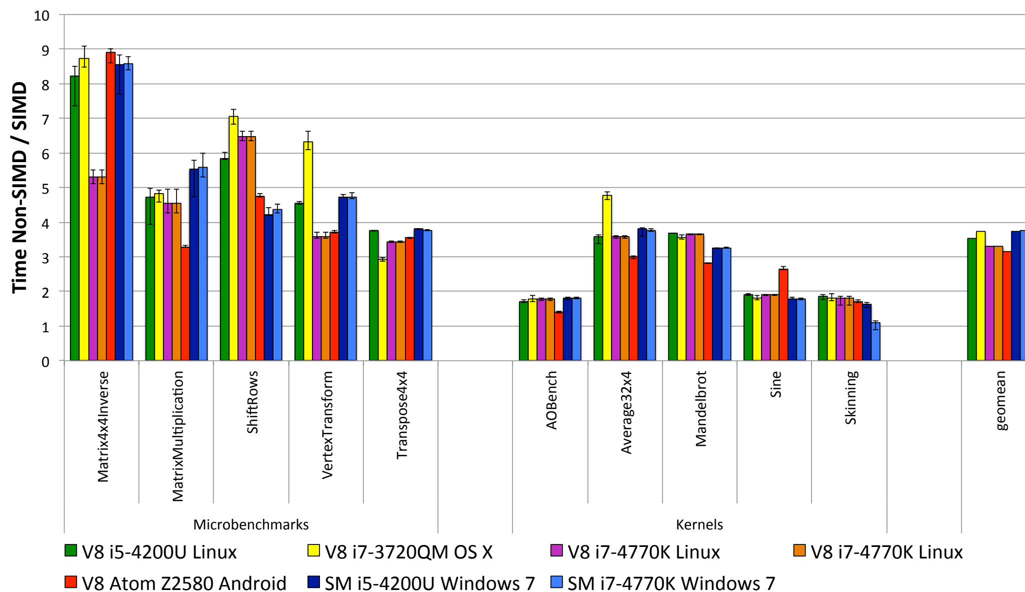


Figure 3.5: SIMD performance with V8 and SpiderMonkey (SM). Normalized to scalar versions. Higher is better.

It may seem surprising that many of the benchmarks improve by more than 4×, yet our SIMD vectors are only four-wide. Indeed, the matrix shift rows benchmark improves by as much as 7× over the scalar version. This super-linear speed up is due to the use of our SIMD operations in an optimized manner that changes the algorithm. For example, the code below shows the SIMD and nonSIMD implementations of the shift row hot methods. Note how we eliminate the need to have temporary variables because we do the shifting of the rows by using SIMD swizzle operations. Eliminating temporary variables and intermediate operations deliver the super-linear speed ups. In summary, the kernels improved by 2× to 9× due to the SIMD extension.

```
// Typical implementation of the shiftRows function
function shiftRows(state, Nc) {
  for (var r = 1; r < 4; ++r) {
    var ri = r*Nc; // get the starting index of row 'r'
    var c;
    for (c = 0; c < Nc; ++c)
        temp[c] = state[ri + ((c + r) % Nc)];
    for (c = 0; c < Nc; ++c)
        state[ri + c] = temp[c];
} }
// The SIMD optimized version of the shiftRows function
// Function special cased for 4 column setting (Nc==4)
// This is the value used for AES blocks
function simdShiftRows(state, Nc) {
  if (Nc !== 4) {
    shiftRows(state, Nc);
  }
  for (var r = 1; r < 4; ++r) {
    var rx4 = SIMD.Int32x4.load(state, r << 2);
    if (r == 1) {
      SIMD.Int32x4.store(state, 4,
            SIMD.Int32x4.swizzle(rx4, 1, 2, 3, 0));
    } else if (r == 2) {
      SIMD.Int32x4.store(state, 8,
            SIMD.Int32x4.swizzle(rx4, 2, 3, 0, 1));
    } else { // r == 3
      SIMD.Int32x4.store(state, 12,
            SIMD.Int32x4.swizzle(rx4, 3, 0, 1, 2));
} } }
```

Figure 3.6: Comparison of scalar vs. SIMD versions of ShiftRows function

Of course the impact of the SIMD instructions is dampened in the richer workloads for which SIMD instructions are only one part of the instruction mix. Nonetheless, it is encouraging to see that the skinning benchmark, which

is based on an important real-world commercial JavaScript workload, enjoys a 1.8× performance improvement due to the addition of SIMD instructions.

3.8.2 Energy

Figure 3.7 shows the energy improvement due to the SIMD extensions. The graph shows scalar energy divided by SIMD energy using the hardware performance counters. Any value higher than one reflects an energy improvement due to SIMD. The results are consistent with those in Figure 3.5, with the improvements dampened slightly. The dampening is a result of measurement methodology. Whereas performance is measured on one CPU, package energy is measured for the entire chip. The energy draw is affected both by ‘uncore’ demands such as the last level cache and memory controller, as well other elements of the core such as L1 and L2 caches, the branch predictor, etc., none of which are not be directly affected by the SIMD extensions.

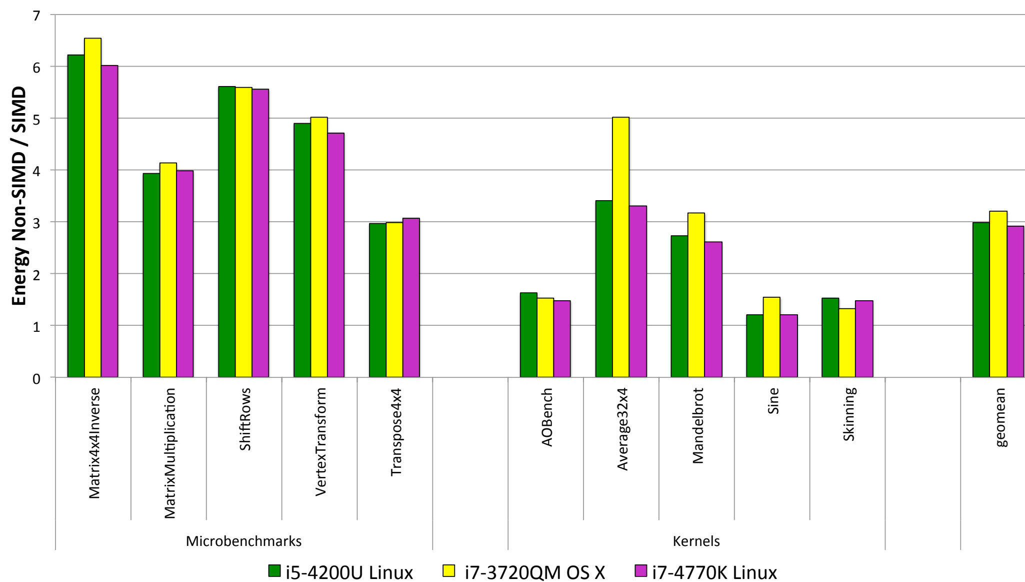


Figure 3.7: SIMD energy on select platforms with V8. Normalized to the scalar versions. Higher is better.

Nonetheless, the energy improvements are substantial. For the real-world skinning workload the improvements are between 25% and 55%, which is significant, particularly in the power-sensitive context of a mobile device.

3.9 Scalarlization

Programming language abstractions that introduce new features must be able to execute on diverse hardware and software (browsers) without sacrificing performance portability. The programming language extension that we presented in this chapter for SIMD.JS guarantees performance when correctly used on supported hardware. However, in order to have a complete solution – one that is practical to standardize, we must analyze and optimize the use of this abstraction in cases where the hardware does not support it. Ideally, when

a programmer changes an application to exploit the performance benefits of the SIMD hardware, the given application should perform at least as well as the original version when executed on hardware that does not support SIMD.

In order to have browser and hardware portability (not yet performance), a polyfill for SIMD.js, which implements the specification of this language extension in JavaScript, is provided. In web development, a polyfill is a piece of code which provides facilities that are not built into a particular web browser. It implements technology that a developer expects the browser to provide natively, providing a more uniform API landscape.

In this section, we explore the performance trade-offs by implementing the following four different approaches to providing the functionality of SIMD.js when vector instructions are not supported by the hardware – ordered from the worst to the most performant: a) a polyfill approach using standard JavaScript, b) a polyfill approach using the asm.js subset of JavaScript, c) using the JIT compiler to perform scalar code generation from the SIMD.js operations directly, and d) using an asm.js version of the SIMD.js functions inlined in the JavaScript application directly (automated in a pre-pass).The results expose the tradeoffs between compiler implementation effort, programmer involvement, and performance. We prototype these approaches in Spidermonkey, the JavaScript engine in the Mozilla Firefox browser.

Figure 3.8 presents the performance results of the four different approaches we explore to provide SIMD.js functionality when vector instructions are not supported by the hardware. The next four subsections go into more detail for each. For each of the proposed approaches, we give examples, and discuss the trade-offs between performance, implementation effort, and programmer involvement.

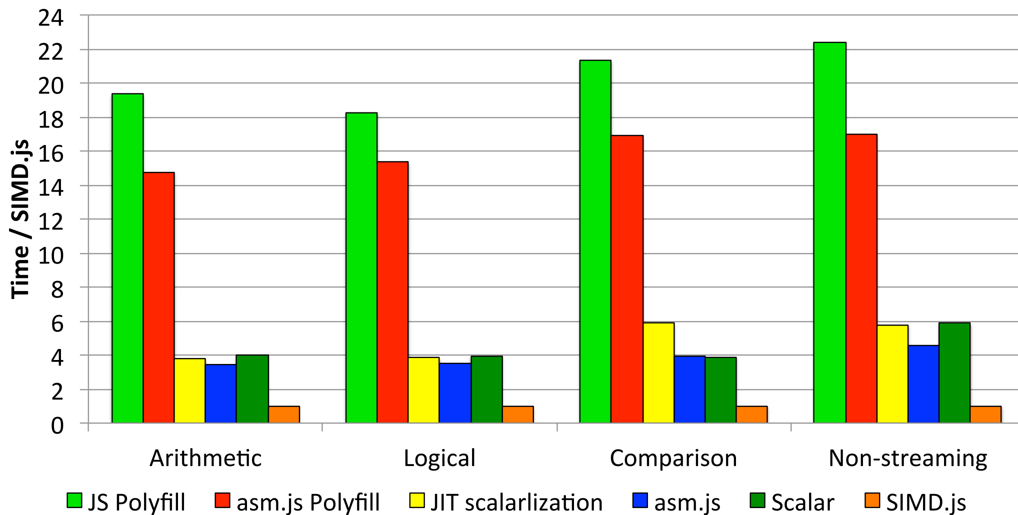


Figure 3.8: Performance comparison when naively replacing vector instructions for scalar instructions. Lower is better.

Each of the subsections below give examples that show how each approach would implement support for SIMD.js functionality when vector instructions are not supported by the hardware. The examples will refer to the code below to show how the functionality of this sum function is provided:

```
var a = new Int32Array(1000);
function sum(n) {
  var a_length = a.length;
  var sum4 = SIMD.Int32x4.splat(0.0);
  for (var j = 0; j < a_length; j += 4) {
    sum4 = SIMD.Int32x4.add(sum4, SIMD.Int32x4.load(a, j));
  }
  return (SIMD.Int32x4.extractLane(sum4, 0) +
      SIMD.Int32x4.extractLane(sum4, 1) +
      SIMD.Int32x4.extractLane(sum4, 2) +
      SIMD.Int32x4.extractLane(sum4, 3));
}
```

This code implements a single function to add one thousand 32-bit integers from an array and return the sum.

3.9.1 Polyfill in JavaScript

When a new feature is being added to the JavaScript language, some browsers will implement it natively while others will not. For the browsers that don't support this functionality yet, developers must provide a library that ensures the functionality of that feature (a polyfill). This is a common approach in web development to provide functionality of new features that are being added natively to the browser (e.g., JSON 2). Once features are standardized and browsers support the latest version of the language, a polyfill is no longer needed.

We introduce a new use for polyfills: in addition to providing cover for a transient lapse in software support, the polyfill covers for a lack of hardware support. A polyfill provides the functionality without needing to have multiple versions of the JavaScript application depending on the hardware support for vector instructions. Because the polyfill can be thought of as a library that is developed by whoever is adding the new feature, this approach does not burden the programmer with any changes other than to include the polyfill in their application. Nonetheless, because the polyfill is written in JavaScript itself, we expect it to perform poorly while still ensuring functionality.

Here we present a snippet of the simplified polyfill that supports the functionality of the average function that we presented above. The JIT compiler would return undefined when an application checks its definition and the compiler knows that the hardware doesn't support vector instructions.

```
var _i32x4 = new Int32Array(4);

if (typeof SIMD.Int32x4 === "undefined") {
  SIMD.Int32x4 = function(x, y, z, w) {
    if (!(this instanceof SIMD.Int32x4))
      return new SIMD.Int32x4(x, y, z, w);
    this.x_ = x;
    this.y_ = y;
    this.z_ = z;
    this.w_ = w;
}}
if (typeof SIMD.Int32x4.splat === "undefined") {
  SIMD.Int32x4.splat = function(s)
    return SIMD.Int32x4(s, s, s, s);
}
if (typeof SIMD.Int32x4.extractLane === "undefined") {
  SIMD.Int32x4.extractLane = function(t, i) {
    switch(i) {
      case 0: return t.x_;
      case 1: return t.y_;
      case 2: return t.z_;
      case 3: return t.w_;
}}}
if (typeof SIMD.Int32x4.add === "undefined") {
  SIMD.Int32x4.add = function(a, b) {
    return SIMD.Int32x4(
        SIMD.Int32x4.extractLane(a, 0) +
                SIMD.Int32x4.extractLane(b, 0),
        SIMD.Int32x4.extractLane(a, 1) +
                SIMD.Int32x4.extractLane(b, 1),
        SIMD.Int32x4.extractLane(a, 2) +
                SIMD.Int32x4.extractLane(b, 2),
        SIMD.Int32x4.extractLane(a, 3) +
                SIMD.Int32x4.extractLane(b, 3));
}}
```

Figure 3.9: Snippet of polyfill support for SIMD.js

The use of SIMD.js operations has typing information embedded as presented in Section 3.5. However, the polyfill must use base types in JavaScript and thus implements all the SIMD functions using scalar JavaScript operations. Because of JavaScript's weak typing for numbers (e.g., int32 vs float32), the polyfill loses the typing information. Thus, Figure 3.8 shows that this approach performs very poorly. This heavy performance penalty is because of the extra type checks that have to be performed at each basic block entry point, the overhead of function calls, and the boxing/unboxing of arguments and return values.

3.9.2 Polyfill in asm.js

The *asm.js* language consists of a strict JavaScript subset. The primary purpose of asm.js is a target into which code written in statically-typed languages with manual memory management (such as C) is translated by a source-to-source compiler such as Emscripten (based on LLVM) [40]. *asm.js* performs better than regular JavaScript because it limits language features to those amenable to ahead-of-time optimization (e.g., type consistency and virtually no garbage collection). Generally, a preprocessor will generate asm.js code (e.g., source-to-source compiler) because its specification makes it very difficult to write and debug by hand. However, we expect to have better performance numbers by using the asm.js subset to implement the polyfill that provides the functionality of SIMD.js.

Here we present a snippet of the asm.js polyfill that supports the functionality of the average function that we present above. For conciseness, we only present the implementation of the constructor and extractLane functions to understand the difference of *asm.js* and regular JavaScript.

```
if (typeof SIMD.Int32x4 === "undefined") {
  /**
   * Construct a new instance of Int32x4 number.
   * @param {integer} 32-bit value used for x lane.
   * @param {integer} 32-bit value used for y lane.
   * @param {integer} 32-bit value used for z lane.
   * @param {integer} 32-bit value used for w lane.
   * @constructor
   */
  SIMD.Int32x4 = function(x, y, z, w) {
    "use␣asm";
    this.x_ = x | 0;
    this.y_ = y | 0;
    this.z_ = z | 0;
    this.w_ = w | 0;
  }
}
if (typeof SIMD.Int32x4.extractLane === "undefined") {
  /**
   * @param {Int32x4} t An instance of Int32x4.
   * @param {integer} i Index in concatenation of t for lane i
   * @return {integer} The value in lane i of t.
   */
  SIMD.Int32x4.extractLane = function(t, i) {
    "use␣asm";
    i = i | 0;
    switch(i) {
      case 0: return t.x_ | 0;
      case 1: return t.y_ | 0;
      case 2: return t.z_ | 0 ;
      case 3: return t.w_ | 0;
    }
  }
}
```

Figure 3.10: Snippet of asm.js polyfill support for SIMD.js

In this case, the ahead-of-time compiler OdinMonkey – part of the JavaScript engine in the Mozilla Firefox browser – will generate optimized code that doesn't include all the unnecessary type checks that the standard JavaScript polyfill did. However, because of specification of asm.js, OdinMonkey would not be able to compile the full SIMD.js module because there is a dynamic check that the SIMD.js functions haven't been patched. Figure 3.8 shows that this approach will also suffer in performance because of the overhead of function calls as well as the boxing/unboxing of arguments and return values. However, this version does combine better performance and portability without compiler or hardware support for SIMD.js.

3.9.3 JIT code generation of scalar code

JIT code generation and optimization of scalar instructions from the SIMD.js instructions has the opportunity to further improve performance, but requires an implementation within each JavaScript runtime. Figure 3.11 shows the different phases of IonMonkey, Firefox's optimizing compiler. In the JIT compiler, we can and do exploit the implementation that optimizes for vector hardware when we target a platform without vector hardware. We use the suppport for inlining and unboxing of SIMD.js values to registers in the JIT compiler. Therefore, the compiler generates optimized scalar code by exploiting the type information in the SIMD.js instructions. Since the previous approaches do not modify the compiler, they do not exploit the type information and therefore their performance suffers.

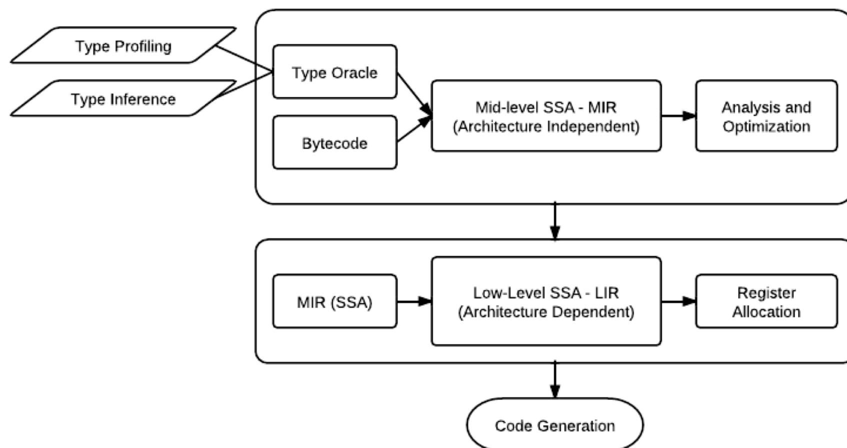


Figure 3.11: Phases of optimizing compiler (IonMonkey) from the Firefox web browser.

Original JavaScript Code

```
sum4 = SIMD.Int32x4.add(sum, SIMD.Int32x4.load(a, j));
```

Optimized Vector code

```
movq mm3,[esi+eax*8]
paddd mm2,mm3
```

Optimized Scalar code

```
movq mm3,[esi+eax*8]
add mm2,mm3
add eax,4
movq mm3,[esi+eax*8]
add mm2,mm3
add eax,4
movq mm3,[esi+eax*8]
add mm2,mm3
add eax,4
movq mm3,[esi+eax*8]
add mm2,mm3
```

Figure 3.12: Examples JIT generated scalar code

Figure 3.8 shows that this approach performs even better than the original scalar code for arithmetic and logical operations. However, in the case of comparisons we see a performance degradation of 1.5X. This slowdown is due to the JIT compiler having to maintain the abstraction that is expected from the SIMD.js operations when SIMD hardware is available: When a comparison operation is performed on two vectors, the result is another vector whose lanes represent the results of the comparisons; the resulting vector is considered a single value that can be unboxed into a single register to be used by a future operation. Therefore, even though we can unroll the comparison operation into four scalar versions, there is still an extra cost of putting the results into a single variable that can be unboxed into a register.

3.9.4 asm.js code

This approach requires programmer involvement (or the use of a pre-processor tool) at the time of the JavaScript application development. In this case, the developer would manually inline a scalar version of the application using the *asm.js* subset. As mentioned above, *asm.js* code is expected to be generated by a pre-processor which can automatically generate this version of the code. Therefore, the developer can develop a single version of the application in a language that can be converted to LLVM (e.g., C/C++) and then use Emscripten [40] to generate two versions of the JavaScript application in asm.js – SIMD.js and scalar.

Figure 3.8 shows that this approach is the one that provides the best performance compared to all of the solutions presented above and even the original scalar code. This performance improvement comes at a cost of programmer involvement. For comparison operations, the main advantage of this

approach compared to the JIT code generation of scalar code is the fact that the return values from the SIMD operations can be kept in separate variables (and after unboxing separate registers) because the programmer understands the semantics of the application and can use those separate variables directly afterwards. For non-streaming operations, this approach performs significantly better because the shuffle and select operations can be replaced for simple no-ops because the asm.js code just changes the name/order of the variables that used in the subsequent scalar code.

3.9.5 Discussion

While SIMD.js takes the journey to standardization through the ECMAScript standardization committee, the existing specification of SIMD.js hasn't yet specified how SIMD.js will provide functionality when there is no hardware support available. It is possible that the JavaScript standards could leave this up to the browser vendor. We have shown four different ways to provide SIMD.js functionality when there is no hardware support available. The trade-offs between performance, implementation effort, and programmer involvement are in the results. Currently, all applications written using the SIMD.js language extension are shipping with the standard polyfill discussed in Subsection 3.9.1 as a way to guarantee functionality when the support is not available natively in the software or in the hardware. However, from the results presented above, it is clear that this is far from the most performant solution. Since SIMD.js is a language extension designed to exploit the vector parallelism already available in the hardware to obtain better performance, it would then follow that any developer that cares to use SIMD.js are likely to care about performance on hardware that doesn't support vector instructions.

Because SIMD.js targets performance-sensitive developers, it makes sense that programmer effort will be traded in favor of performance. We now explore how great the programmer effort is likely to be. This is dependent on the development path taken by the application that uses the SIMD.js language extension. Section 3.4 describes SIMD.js as a compile target from C/C++ as part of the design rationale. We expect that a significant majority of the applications that use the SIMD.js language extension will come through Emscripten which uses JavaScript as a compile target. In this scenario, the programmer involvement is minimal as Emscripten can be modified to generate two versions of the JavaScript application from the C/C++ code: 1) One that uses the SIMD.js extension when vector code is found in the native version and 2) One that generates regular asm.js code by hot patching the native vector code. Once both versions of the JavaScript code are generated, the web application (HTML) could pick between the two implementations that are automatically generated by Emscripten depending on whether hardware support is available. This means that the programmer involvement for this development path is limited to the time of generating the web application and it is thus an acceptable solution. Unfortunately, this is not the case when a developer uses SIMD.js directly as part of his JavaScript application. To obtain similar performance, the developer would need to: a) Use asm.js to develop the part of the application that uses SIMD.js and b) Write a scalar version of that part using asm.js as well. Developing in asm.js is not an easy task and takes away a lot of the productivity of JavaScript as a language, making this approach an unfeasible solution.

Because of the availability of multiple browsers that implement the JavaScript standard, the implementation effort for producing scalar code from

SIMD.js operations plays a role in deciding which approach is used when we lack the hardware support. This trade-off between performance, compiler implementation effort, and programmer involvement will have to be decided based on whether SIMD.js will be expected to be used mostly/solely as a compiler target from other languages that already have vector representation or whether it is expected that JavaScript developers will use it to optimize their existing applications. Stage four of standardization through the ECMAScript committee requests developer feedback to understand whether any changes will need to be made to the specification before it is final. We expect to see more applications that use SIMD.js after this step and hope to have a clear choice for scalarlization based on the development paths of these applications.

3.10 Future Work Discussion

Our design goal of portability is intended to be consistent with the existing JavaScript language specification. However, this constraint precludes platform-specific optimizations which are not currently accessible from JS that would benefit performance and energy efficiency.

First, the opportunity to use wider vector lengths, such as the AVX 512 vector instruction set that Intel is already shipping, will deliver additional performance improvements, particularly on the server side. Stream processors (in the form of an API) can and will be built in software on top of the current SIMD.js specification to utilize this hardware.

Second, a large number of hardware operations are currently not supported, including platform-specific ones. One approach may be to provide an extended API for SIMD.js that accesses platform-specific instructions and optimizations. This API would sit on top of and complement the base API

described in this chapter. The extension API could offer opportunities for performance tuning, specialized code sequences, and support porting of code from other platforms. One can classify the remaining SIMD operations in two groups: those that are portable but have semantic differences (SIMD.Relaxed) and those that are only available on some platforms (SIMD.Universe). Functions in SIMD.Relaxed would mimic functions in the base API with corresponding names, and provide weaker portability with greater potential for performance (e.g., with unspecified results if NaN were to appear in any part of the computation, treating zero as interchangeable with negative zero, and unspecified results if an overflow occurs). Functions in the SIMD.Universe namespace could adhere to well defined semantics but their availability would result in various code paths depending on the architecture. For example, SIMD.isFast would need to check whether the JIT compiler can generate a fast code sequence from each operation for the current hardware.

3.11 Conclusion

This chapter describes the design and implementation of a portable SIMD language extension for JavaScript. This specification is in the final stages of adoption by the JavaScript standards committee, and our implementations are available in the V8 and SpiderMonkey open-source JavaScript VMs. The contributions of this chapter include a principled design philosophy, a fixed-size SIMD vector language specification, a type-speculation optimization, and performance and energy evaluations on critical kernels. We describe the language extension, its implementation in two current JavaScript JIT compilers, and evaluate the impact in both execution time and energy efficiency. Programability, portability, ease of implementation, and popular use-cases all

influenced our decision to choose a fixed-width 128-bit vector design. Our evaluation demonstrates that the SIMD extensions deliver substantial improvements in execution time and energy efficiency for vector workloads. Our design and implementation choices do not preclude adding more SIMD operations in the future, high-level JavaScript libraries to implement larger vector sizes, or adding automatic vectorizing compiler support. Another potential avenue for future work is agressive machine-specific JIT optimizations to utilize wider vectors when available in the underling hardware. Our results indicate that these avenues would likely be fruitful.

Chapter 4

Conclusion

Software and hardware platforms have and will continue to keep evolving over time in order to satisfy ever increasing demands for productivity, performance, and energy efficiency. On one hand, programmers choose managed languages to boost productivity and increase agility of software development by leveraging virtual machine services. On the other hand, hardware design continues to evolve and introduce many types of parallelism as the principle constraint on hardware shifts from transistor count to power. This shift is introducing specialized hardware (e.g., FPGAs) as means for better performance and energy efficiency for general computing. Changing the software for every new version of the hardware is unfeasible and too expensive. Particularly, as hardware vendors make different design decisions to provide better efficiency, attaining portable performance is going to become even more difficult.

The increasingly prevalence of managed languages in mobile, client, and server workloads makes virtual machine technologies an ideal abstraction over hardware complexity because they already profile, optimize, and schedule applications. However, they do so at a significant cost and do not yet target all forms of parallelism.

This thesis seeks to allow managed programming languages to efficiently exploit the underlying parallelism available in the hardware. Leveraging virtual machine technology as an efficient abstraction limits the developers exposure to the increasingly complex parallelism of evolving hardware.

We use the virtual machine abstraction layer to leverage two particular forms of heterogenous parallelism available in the hardware today: task level parallelism available in asymmetric multicore processors and data parallelism available as instructions sets in the hardware. This thesis demonstrates novel approaches to exploiting parallelism and heterogeneity in managed programming languages in two distinct ways. 1) We present virtual machine technologies that analyze applications and matches them to the heterogenous parallelism in asymmetric multicore processors to significantly improve performance and energy efficiency of server Java applications. We identify non-scalable parallelism (i.e., messiness) of managed programming languages workloads as a key problem in these workloads. In particular, we present the first fully automated software approach to identify, prioritize, and accelerate threads that hold locks and are thus performance bottlenecks. 2) We present a language extension for JavaScript that allows web applications to exploit the data parallel instruction sets available in the hardware in a portable manner. The impact of this thesis is already being felt because the SIMD.js JavaScript language extension is making it to the language standard.

In combination, these contributions demonstrate the opportunities and tackle the challenges that arise from using the managed languages abstractions over the complex parallelism in the hardware. We deliver solutions to efficiently exploit the evolving hardware with little or no burden to the application developers by 1) designing and implementing a new language abstraction for data parallelism for JavaScript that is easy for programmers to use and 2) introducing new static and dynamic analyses, compile-time optimizations, and runtime optimizations that exploit the parallelism in asymmetric multicore processors.

Bibliography

[1] Ole Agesen and Urs Hölzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *ACM Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 91–107, 1995.

[2] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes RVM project: Building an open source research community. *IBM Systems Journal*, 44 (2):399–418, 2005.

[3] Android. Bionic platform, 2014. URL `https://github.com/android/platform_bionic`.

[4] ARM. NEON and VFP intel (SSE4) programming reference. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204j/Chdehgeh.html`, retrieved 2014.

[5] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 47–65, 2000. doi: 10.1145/353171.353175.

[6] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for Java. In *ACM Conference*

on Programming Language Design and Implementation, pages 258–268, 1998. doi: 10.1145/277650.277734.

[7] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. pages 29–40, 2006. doi: 10.1145/1128022.1128029.

[8] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 169–190, October 2006. doi: 10.1145/1167515.1167488.

[9] W.J. Bouknight, S.A. Denenberg, D.E. McIntyre, J.M. Randell, A.H. Sameh, and D.L. Slotnick. The Illiac IV System. *Proceedings of the IEEE*, 60(4):369–388, 1972.

[10] Ting Cao, Stephen M. Blackburn, Tiejun Goa, and Kathryn S. McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *ACM/IEEE International Symposium on Computer Architecture*, 2012.

[11] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narváez, and Joel S. Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *ACM/IEEE International Symposium on Computer Architecture*, pages 213–224, 2012. doi: 10.1109/ISCA.2012.6237019.

[12] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. *IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 177–187, 2013. doi: 10.1109/PACT.2013.6618815.

[13] CRAY Research, Inc. The CRAY-I Computer System, 1976.

[14] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programmming (ECOOP)*, pages 77–101, 1995.

[15] D. Dice, M. Moir, and W. Scherer. Quickly reacquirable locks, patent 7,814,488, 2010.

[16] Kristof Du Bois, Stijn Eyerman, Jennifer B. Sartor, and Lieven Eeckhout. Criticality stacks: identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th International Symposium on Computer Architecture*, pages 511–522, New York, NY, USA, Jun. 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485966.

[17] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. 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*, pages 319–332, 2011. doi: 10.1145/1950365.1950402.

[18] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan. G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. R. Haghighat, M. Bebenita, M. Chang, and M Franz. Trace-based just-in-time type specialization for dynamic languages. In *Programming Language Design and Implementation (PLDI 2009)*, pages 465–478, 2009.

[19] Google. V8 JavaScript Engine, 2014. `http://code.google.com/p/v8`.

[20] Peter Greenhalgh. Big. little processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper*, 2011.

[21] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[22] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 326–336, 1994.

[23] Intel. Pentium Processor with MMX Technology. `http://download.intel.com/support/processors/pentiummmx/sb/24318504.pdf`, 1997.

[24] Intel. Intel (SSE4) programming reference, 2007.

[25] P. Jensen, I. Jibaja, N. Hu, D. Gohman, and J. McCutchan. SIMD in javascript via C++ and emscripten. In *Workshop on Programming Models for SIMD/Vector Processing*, 2015.

[26] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Bottleneck identification and scheduling in multithreaded applications. In

ACM International Conference on Architectural Support for Programming Languages and Operation Systems, pages 223–234, 2012. doi: 10.1145/2150976.2151001.

[27] Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-based acceleration of multithreaded applciations on asymmetric cmps. In *ACM/IEEE International Symposium on Computer Architecture*, 2013.

[28] Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. In *Dynamic Languages Symposium (DLS)*, pages 37–48, 2013.

[29] Madhukar N. Kedlaya, Behnam Robatmili, Cġlin Caşcaval, and Ben Hardekopf. Deoptimization for dynamic language jits on typed, stack-based virtual machines. In *ACM International Conference on Virtual Execution Environments (VEE)*, pages 103–114, 2014.

[30] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. pages 64–75. doi: 10.1109/ISCA.2004.1310764.

[31] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 81–92, 2003.

[32] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of the 19th ACM SIGPLAN-SIGACT*, ACM Symposium on Principles of Programming Languages (POPL), pages 177–188, 1992.

[33] Tong Li, Dan P. Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. page 53. doi: 10.1145/1362622.1362694.

[34] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 65–74, New York, NY, USA, 2009. ACM. doi: 10.1145/1504176.1504188.

[35] J. McCutchan, H. Feng, N. D. Matsakis, Z. Anderson, and P. Jensen. A SIMD programming model for Dart, JavaScript, and other dynamically typed scripting languages. In *Workshop on Programming Models for SIMD/Vector Processing (WPMVP)*, 2014.

[36] Microsoft. The JIT finally proposed. JIT and SIMD are getting married. `http://blogs.msdn.com/b/dotnet/archive/2014/04/07/the-jit-finally-proposed-jit-and-simd-are-getting-married.aspx`, 2014.

[37] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan L. Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3):26–41, 2008. doi: 10.1109/MM.2008.47.

[38] Ingo Molnar. Modular Scheduler Core and Completely Fair Scheduler [CFS]. http://lwn.net/Articles/230501/, April 2007.

[39] Mono Project. Mono open-source .NET implementation for C#. http://docs.go-mono.com/?link=N%3aMono.Simd, 2007.

[40] Mozilla Corporation. Emscripten. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten, 2012.

[41] Mozilla Foundation. SpiderMonkey, 2014. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[42] Qualcomm. Snapdragon 810 processors, 2014. URL https://www.qualcomm.com/products/snapdragon/processors/810.

[43] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March 2012. ISSN 0272-1732. doi: 10.1109/MM.2012.12. URL http://dx.doi.org/10.1109/MM.2012.12.

[44] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *ACM Conference on Object–Oriented Programming Systems, Languages, and Applications*, pages 263–272, 2006. doi: 10.1145/1167473.1167496.

[45] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *Jour-*

nal of Parallel and Distributed Computing, 71(1):114–131, 2011. doi: 10.1016/j.jpdc.2010.08.020.

[46] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems. *ACM Transactions on Computer Systems*, 30(2):6, 2012. doi: 10.1145/2166879.2166880.

[47] Daniel Shelepov, Juan Carlos Saez, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: a scheduler for heterogeneous multicore systems. *Operating Systems Review*, 43(2):66–75, 2009. doi: 10.1145/1531793.1531804.

[48] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, 2009. doi: 10.1145/1508244.1508274.

[49] TC39 - ECMAScript. SIMD.js specification v0.9, retrieved October 2015. `http://tc39.github.io/ecmascript_simd/`.

[50] The Jikes RVM Research Group. Jikes Open-Source Research Virtual Machine, 2011. URL `http://www.jikesrvm.org`.

[51] WHATWG. HTML living standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html`.